Merging of Use Case Models: Semantic Foundations

Stephen Barrett, Daniel Sinnig, Patrice Chalin, Greg Butler
Faculty of Engineering and Computer Science
Concordia University, Montreal, Quebec, Canada
Email: {ste_barr, d_sinnig, chalin, gregb}@encs.concordia.ca

***Abstract*—Use case models are the artifact of choice for capturing functional requirements. This typically collaborative activity makes merging a necessity. Use cases however, are often neglected when it comes to model merging, since they are commonly treated as text only items. By defining a formal syntax and semantics for use case models, manipulated within a generic metamodel for operation-based merging, we show how use case models can be effectively merged. This formal foundation allows for the modeling of use cases; defining meaningful change operations on them; and for detecting modeling inconsistencies, inconformities, and conflicts. Several practical examples validate the concepts presented: existing and planned tool support is introduced.**

***Index Terms*—Model merging, use case model, operation-based merging, inconsistency, inconformity, conflict, finite state machine, model transformation, change plane.**

I. Introduction

Since their introduction by Jacobson [1] in the early 90s, creating use cases has become a key software development activity. Meanwhile, model-driven engineering (MDE) has been shifting the focus of the development process from code to models. Thus in a collaborative MDE setting, the group goal, instead of one of refining code, becomes one of moving incomplete or abstract models to more complete or concrete models. Growing evidence suggests that the adoption of use case modeling leads to significant benefits for customers and developers alike, through clearer requirements understanding, and through greater programming automation [2]–[4].

With developers making contributions in parallel models naturally diverge. It then becomes the task of model merging to bring these evolved replicas into agreement. This is a concern for all models, which to our knowledge has not been addressed for use cases in any formal way. Too often use cases are viewed as textual artifacts to be merged using traditional line-by-line approaches, or by forming the union of all contributions. By providing a formalization for use case models we are able to map them into finite state machines, and with this semantic underpinning, able to merge them under the detailed metamodel. As well as analyze them for modeling and merging errors, with in some cases, automatic resolution.

In this paper we define a semantic operation-based merge process for use case models. Section II touches on merge types and approaches before presenting our metamodel for merging, formed around model transformations. In Section III the syntactic and semantic rules and definitions for formalizing use case models as finite state machines are provided, and a sample use case mapping is presented. Using this example, Section IV demonstrates the application of our formalisms for detecting model and merge errors, while highlighting the differences between error types. Existing and planned tool support is discussed in Section V, related work in Section VI, and our conclusions and future plans in Section VII.

II. Model Merging Metamodel

A simple, intuitive, and yet mathematically founded merging metamodel is highly useful as a means of understanding model merging in general, and when coupled with the semantics of Section III, use case model merging in particular.

A. Merge Types and Approaches

The merge of two **replica** models (i.e., initially identical copies (M_L and M_R) of some source model) that have evolved from a common ancestor (M_A) is shown in Fig. 1, in which the diverged replicas are combined into a final merged model (M_F). Since **three-way merging** as it is known, can automatically avoid inconsistencies by rearranging changes, as well as resolve some conflicts by referencing the common source, merge tools make extensive use of it [5], as do we.

The need for merging is brought about by changes being made to an artifact's replicas. Since it follows an **operation-based merging** approach, our metamodel takes these changes into account, whereas the other broad approach, state-based merging, does not [6]. By analyzing the dynamic traces of model-altering user actions gathered during replica modification, operations may be selected, interleaved, reordered, and simplified in order to create a merged trace of model changes.

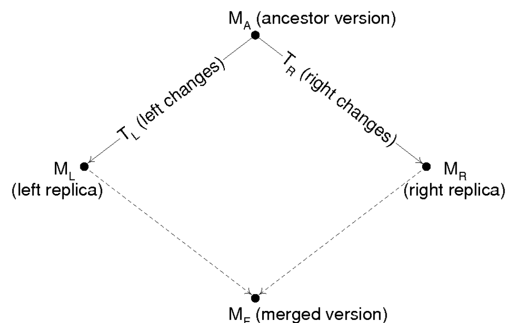

Fig. 1. Merging two replicas derived from a common ancestor model.

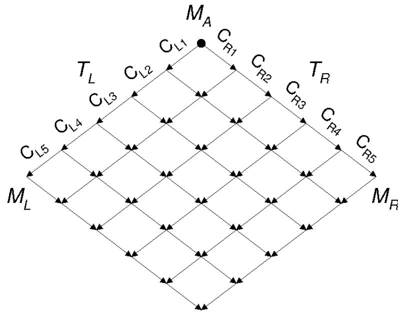

Fig. 2. Model change operations projected onto the merge change plane.

B. The Change Plane

Each change made to a model replica potentially transforms it into a new model. In this way model changes act as atomic operations which can be treated as functions on models. Doing so allows us to use functional composition to define an endogenous **transformation** T_e that takes as its input some initial model M_a conforming to the replica's metamodel, and produces as its output a final model M_f conforming to the same: $T_e M_a = (T_{e_n} \circ T_{e_{n-1}} \circ \ldots \circ T_{e_1}) M_a = M_f$ where T_{e_j} is a **subtransformation** of T_e.

Two transformations are involved in a three-way merge: one that it is convenient to think of as taking place on the left, T_L that transforms a replica of the ancestor model M_A into model M_L; and one on the right, T_R that transforms another replica into model M_R. The essence of merging is to blend T_L and T_R into one transformation that will transform M_A into a final merged model M_F. In the absence of conflicts the transformations may be combined tip-to-tail fashion—as is done with vectors—otherwise they must be cobbled together piecemeal, subtransformation by subtransformation.

We orient the **transformation grid** of [7] in Fig. 2 to mimic the classic merge diamond. With model M_A situated at the origin, its axes represent the left and right transformations; each made up of subtransformations (directed segments). To traverse a path from the origin to another node is to execute the sequence of elementary change operations that lie on the path, transforming M_A in the process. Hence, every grid point hosts the set of models produced by all paths that reach that point. We term the resulting two-dimensional surface of model subtransformations and potential models, the **change plane**.

From a transformational point of view, to move forward (i.e., out from the origin) over an edge is to *accept* the change that the edge represents, while to avoid traversing an edge is to *reject* its change. The sequence of changes represented by a path of subtransformations in the plane produces a new version of the ancestor model. An **inconsistency** occurs if any operation in the path fails (e.g., referencing a nonexistent element). A syntactic **inconformity** results when an operation produces a model that is not well-formed (e.g., a nonunique ID). A semantic **conflict** is a contradictory pair of operations (e.g., naming the same element differently). Inconsistencies can often be avoided, and inconformities at times resolved by reordering the change sequences, while some conflicts can be eliminated by referring to the common ancestor. Conflicts that remain are resolved by arbitrarily selecting one change over the other, employing heuristics, or by appealing to the user.

C. Commutation and Conflict Resolution

Paths that respect the original ordering of their operations relative to a given change plane (that is, T_{L_i} comes before $T_{L_{i+1}}$, even if interleaved by T_{R_j}), *and* result in a consistent model are known as **weaves**. A path with an operation whose specification cannot be met will fail, and thus is not a weave: The path is prevented from reaching its target node due to an inconsistency. Weaves that terminate at the same node, but produce different outcomes are in conflict: A choice as to which weave to follow must be made, generally with knowledge a tool will not possess.

Noncommutative operations across merge transformations are at the root of conflicts, because a node's weaves differ only in the order of their subtransformations. If operations T_1 and T_2 commute globally, i.e., $T_1 \circ T_2 = T_2 \circ T_1$, then they will not conflict. Additionally, transformations that do not commute globally *may* commute locally when operating on mutually removed sections of a model, i.e., $(T_1 \circ T_2)M = (T_2 \circ T_1)M$, and consequently do not conflict when transforming M [7]. Checking for operation commutation then is an appropriate way of determining if transformations are capable of being merged, and for isolating the causes of conflicts for purposes of deletion or modification. Similarly, inconsistencies can often take advantage of commutativity to find another path in which the circumstance that causes the inconsistency does not occur.

A node that hosts the same model for every one of its weaves is said to be **single-valued**. If any of the subtransformations are not at least locally commutative, then the node hosts more than one model and is said to be **multiple-valued**. The multiple-valued points that border the plane's single-valued nodes is called the **frontier set** [7]. It represents the boundary at which conflicts occur, and at which operation-based merging cannot automatically acquire more transformations. It is clear that the further the frontier set can be pushed out from the origin, the fewer decisions the user will need to be confronted with.

Consider a simple class model consisting of classes A and B. In part *a* of Fig. 3 the ancestor model is modified by the left transformation to initialize a member of A and B to 0, while the right transformation sets $A.x$ to 1 before deleting B. All of the points along the axes are single-valued since there is only

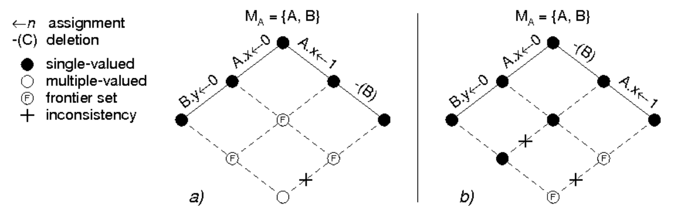

Fig. 3. Using operation commutation to push back the frontier set.

one path that reaches any one of them (disallowing backward travel). Because $A.x \leftarrow 0$ and $A.x \leftarrow 1$ do not commute the two weaves that terminate at the center node produce different models, making the node multiple-valued. The two mid-border points are multiple-valued for similar reasons—modifying $B.y$ and deleting the class do not commute. Some of the paths going to the outermost node cause an inconsistency, because attempting to access B after it has been deleted violates a precondition of assignment, namely that the target exist. Thus for this structuring of transformations, the frontier set consists of only those nodes that lie on the axes.

Swapping the position of the right hand subtransformations as is done in part *b*, eliminates the conflict, thereby turning two more nodes into single-valued points. The frontier set has thus been expanded slightly: Whereas the first structuring could only incorporate two changes that came from the same side (i.e., no merge), the second can incorporate one entire side and half of the other. This comes at the negligible expense of another inconsistency.

III. Use Case Modeling

A use case model captures the "complete" set of use cases for an application. Each use case specifies possible usage scenarios for some particular functionality offered by the system. Every use case of a model starts with a header section consisting of various properties (e.g., primary actor, goal, goal-level, etc.). The core of a use case is its **main success scenario**, which lays out the most common way for the primary actor to reach the goal when using the system. Use case **extensions** define alternative scenarios which may, or may not lead to fulfillment of the use case goal.

A. Syntax of a Use Case Model

Notations of varying degrees of formality have been suggested for expressing use cases. They range from purely textual constructs [2], to entirely formal specifications in Z [8], to abstract state machines [9], [10], to graph structures [11]. While modeling in prose facilitates communication among stakeholders, its informal nature is prone to ambiguities, leaving little room for tool support. In this article we adopt an intermediate solution, named—after our research group—the DSRG-style use case model, which enforces a formal structure, but also preserves the intuitive nature of use cases: We formalize the sequencing of use case steps, but leave the respective actions and associated conditions to be specified informally. Except for the discrete *goal-level*, use case properties are specified using narrative language.

We define a DSRG-style use case model (UCM) as a set of interrelated use cases, with one acting as the root. As shown in Fig. 4, a use case consists of: a property section, a main success scenario, and a set of extensions. The main success scenario, as well as each extension consists of a sequence of **use case steps**, which are one of seven different kinds: **Atomic** steps are performed either by the primary actor or the system, and contain no sub-steps; **Choice** steps provide the primary actor with a choice between several interactions, each interaction being in turn defined by a sequence of steps; **Concurrent** steps define a set of steps the primary actor may perform in any order; **Goto** steps denote jumps to other steps within the same use case; **Include** steps define the inclusion of a sub-use case; and **Success** and **Failure** denote respectively, the successful or unsuccessful termination of the use case. We note that each use case, use case step, and extension is assigned an ID for use as a reference (e.g., for Goto or Include steps).

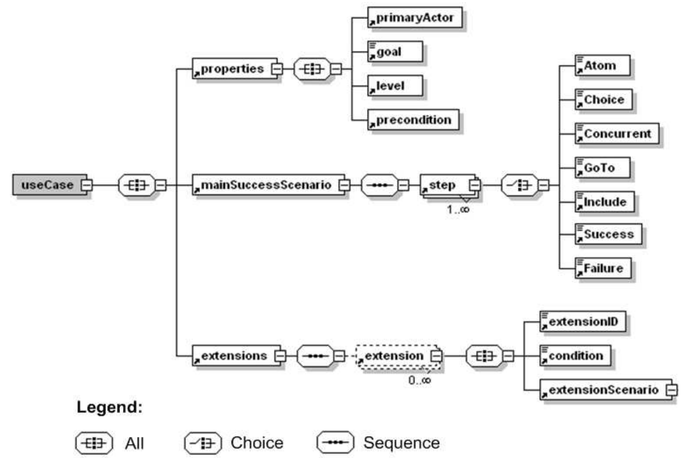

Fig. 4. DSRG-style use case structure.

A DSRG-style use case model is well-formed iff:

1) All use case, use case step, and extension IDs are unique.
2) For every Goto step or extension reference there exists respectively, a corresponding use case step or use case extension within the same use case.
3) Every use case in the use case model, except for the designated root, must be directly or indirectly included in the root use case.
4) For every Include step, the included sub-use case must exist, and the inclusion must not be circular.
5) The last element of every use case step sequence is either a Goto, Success, or Failure step.

Consult [12] for the full list of well-formedness rules.

The abbreviated use case of Fig. 5 captures interactions for the "Order Product" functionality of an invoicing system. The main success scenario describes the situation in which the primary actor directly accomplishes their goal of ordering a product. The reduced collection of extensions specify alternative scenarios which, in this case lead to the abandonment of the use case goal. For convenience, each use case step has been further attributed with a mnemonic label.

B. Formal Semantics

By establishing a formal semantics for use cases in terms of finite state machines (FSMs), we are able to define an equivalence relation for use case models with which we can determine whether two model changes are in conflict.

Def. 1. *(Finite State Machine):* A finite state machine is defined by the tuple: $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set of events, $\delta : Q \times \Sigma \rightarrow Q$ the

Use case: Order Product

Properties

Primary Actor: Customer
Goal: Primary Actor places an order for a specific product.
Goal Level: User goal
Precondition: Primary Actor is logged into the system.

Main Success Scenario

1. Primary Actor specifies desired product category. [spCA]
2. System displays products matching category criteria. [diSR]
3. Primary Actor selects a product in desired quantity. [slPQ]
4. System verifies that product is available in requested amount. [vaPQ]
5. System displays the purchase summary. [diPS]
6. Primary Actor submits payment information. [sbPI]
7. System carries out payment. [caPA]
8. System provides a confirmation number. [prCN]
   Use case ends successfully.

Extensions

4a. **Desired product is not available.**
  4a1. System informs Primary Actor of unavailability. [inPU]
    Use case ends unsuccessfully.

7a. **The Payment was not authorized.**
  7a1. System informs Primary Actor payment is not authorized. [inPD]
    Use case ends unsuccessfully.

Fig. 5. "Order Product" use case at an early stage of development.

transition function, q_0 the initial state with $q_0 \in Q$, and $F \subseteq Q$ the set of final states. We also define the extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$ in the standard way as $\delta^*(q_i, w) = Q_j$, where Q_j is the state of the FSM, having started in state q_i after accepting the sequence of events w.

Def. 2. *(Accepted Language of an FSM):* Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FSM. We define the *accepted language* of M as follows: $L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$

The semantic mapping from a DSRG-style use case model into an FSM is defined in a bottom-up manner, starting with the mapping of individual use case steps (see [12] for details). Each of the seven kinds of use case steps enumerated in Fig. 4 has its own specific mapping to an FSM. Atomic steps map to elementary FSMs consisting of only an initial state and a set of final states, connected by a transition that represents the use case step. A Choice step maps to a composite FSM consisting of the initial states of each choice's FSM. A Concurrent step is the product machine of its constituent FSMs. Goto steps map to an FSM with a single state defined to be equivalent to the initial state of the FSM representing the target of the jump. The complement FSM of an Include step consists of two states: one identified with the initial state of the FSM of the main success scenario of the invoked sub-use case, and the other identified with all final states of the sub-use case's FSM. For both Success and Failure, the coinciding FSMs consist of only a single final state.

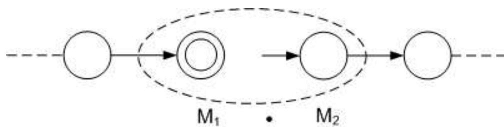

Fig. 6. Sequential composition of FSMs.

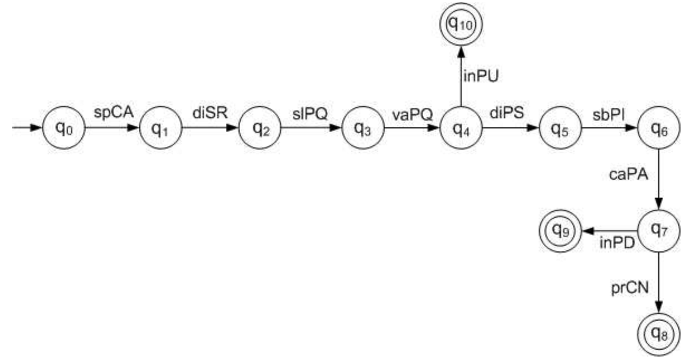

Fig. 7. FSM representation of "Order Product" use case.

Now that individual use case steps can be formally represented by FSMs, we can link arbitrary sequences of steps with the sequential composition operator (•), as schematically portrayed in Fig. 6. The operator unifies the final state, or states of its first operand with the initial state of its second. The FSMs thus obtained—representing the use case's main success scenario and its extensions—are then related together under a single FSM. Similarly, an entire use case model is mapped to an all-encompassing FSM that relates the individual use case FSMs via inclusion relationships, in accordance with well-formedness rule 3.

Semantic mapping of the "Order Product" use case (Fig. 5) produces the FSM depicted in Fig. 7. The resulting FSM has three final states: q_8 denoting the successful outcome of the use case (i.e., acceptance of the placed order), q_{10} denoting the unavailability of a product, and q_9 denoting that the payment was not authorized.

We now define an equivalence relation for use case models based on FSM language equivalence—since nondeterminism is not a possibility, the criterion is sufficiently strong. The relation will be used in Section IV for uncovering conflicts between model change operations. Note that the equivalence relation only considers the behavioral parts of a use case (i.e., main success scenario and extensions), and abstracts away the static property section.

Def. 3. *(Equivalence Relation for Use Case Models):* Let UCM_1 and UCM_2 be well formed DSRG-style use case models, and M_1 and M_2 their respective FSM representations. Then the equivalence relation ($\equiv$) between use case models is defined as:

$$UCM_1 \equiv UCM_2 \Longleftrightarrow L(M_1) = L(M_2)$$

IV. Merging of Use Case Models

As mentioned in the introduction, use cases are usually treated as text-only work products to be merged via line-based tools. What attention that is given to structure is usually confined to simple figures having perhaps some cumbersome linkage to text blocks. Merging is limited to visual details such as element placement, size, and labels. But as Section III makes clear, use cases are *highly* structured, and as such are amenable to more sophisticated techniques than line-by-line text merging [5], [13].

With an ability to map use cases into a formal domain, and a means for expressing model merging, we are now in a position to consider how the merging of use case models as they are refined over the course of a project might proceed.

A. An Example of Text Merge Shortcomings

Consider a use case consisting of only a main success scenario. Suppose modeler Left adds an extension that ends with, *"Use case resumes with step 9."* Now suppose that in the same modeling cycle modeler Right inserts a step in the use case's main scenario, say after step 2. In a modest modeling tool this has the effect of, let us assume, automatically bumping up the count of all the steps that follow, thereby breaking Left's change.

As new functionality, both changes need to be accepted by the merger, but one is now in error. Line-based merging will not make the link between the changes explicit, and with enough changes the resulting "cognitive noise" is apt to bury the merger's ability to divine one. By assigning the use case steps of the UCMs unique IDs in accordance with the DSRG style, the merge's **matching strategy** will be able to properly relate the changes, thereby avoiding the error.

The remainder of this section details how the merging metamodel can take advantage of the formal use case semantics to detect and resolve the errors of Section II-B as encountered during a model merge session.

B. Use Case Model Change Operations

Having defined a formal structure for use case models (Section III) we are now able to identify the operations that can be executed against its elements to effect changes. Operations have different scope, ranging from those that affect only a single use case step, to those that affect an entire use case. We distinguish between three kinds of operations:

- Addition. Creates a new entity in the use case model with a unique ID, e.g., add a step.
- Deletion. Removes an entity and its identity from the use case model, e.g., delete a use case.
- Modification. Alters an entity while preserving its identity, e.g., modify an extension's condition.

The granularity of the operations is in stark contrast to those that have to be defined if use cases are regarded as text-only documents (e.g., the addition or deletion of words and characters).

C. Inconsistency

On the change plane, an inconsistency is manifest when a weave contains a failing operation, preventing further progress along the path. An example of how this could come to be is when modeler Left deletes a use case step that modeler Right modifies. A merged sequence that places Left's change first will introduce an inconsistency when executed, because it is not possible to alter a step that has been removed from the model.

Rearranging the operations of a merged sequence, or in effect, following a different path on the change plane, can often bypass an inconsistency; however, only at the expense of losing a change—in this case Right's step modification. In truth, depending on if Left's deletion is legitimate, it might not be an important loss. Something a tool might be able to acertain by examining the operations that come afterwords.

LEFT:
- 4a. **Desired product is not available in sufficient quantity:**
  - 4a1. System informs Primary Actor of unavailability. [inPU]
    Use case resumes with step 1.

RIGHT:
- 4a. **Desired product is not available in ordered amount:**
  - 4a1. System informs Primary Actor of unavailability. [inPU]
  - 4a2. System sends notify request to Customer Manager. [noCM]
    Use case resumes with step 3.

Fig. 8. Parallel changes made to the "Order Product" use case.

D. Inconformity

Models that do not conform to the use case well-formedness rules of Section III are in violation of its syntax, and will exhibit what we have termed inconformities. A model that is not well-formed cannot be mapped into an FSM, a transformation we rely on for conflict detection between use case models.

Our merging framework does not presume completeness: in a cooperative environment it is normal for inconformities to exist temporarily in isolation. However, upon merging all inconformities must be resolved. For instance, if modeler Left inserts an Include step that depends on the existence of a sub-use case that modeler Right is supposed to supply, then both models will run afoul of well-formedness rules: rule 4 for the left model because it references a nonexistent use case, and rule 3 on the right because it contains an unreferenced "dead" use case. Presumably, when a tool like our Syntactic Analyzer (Section V) points out these inconformities, the modelers will know that they are only temporary, to be resolved at merge time. If modeler Right for some reason fails to carry out his task, the merge will suffer from a "permanent" inconformity, thereby halting further conflict analysis which we discuss next.

E. Conflict

Consider two developers working on the "Product Unavailable" extension of Fig. 5 who make the changes underlined in Fig. 8. The left modeler qualifies the extension's condition to make an instock product unavailable if there are too few to satisfy the order. Realizing that product unavailability is not a sufficient reason to abort the entire use case, he also replaces the extension's termination with a jump back to the start of the use case. Making the same observations, the right modeler also alters the condition, but decides that branching back to the list of products at step 3 is best. In addition, she decides that the customer should be notified when the item becomes available, and so inserts a step into the extension to request that the external Customer Management system see to this.

Assume that the model changes are brought about by these two sequences of operations. Left: modify terminating step (∼step), modify extension condition (∼cond); and Right:

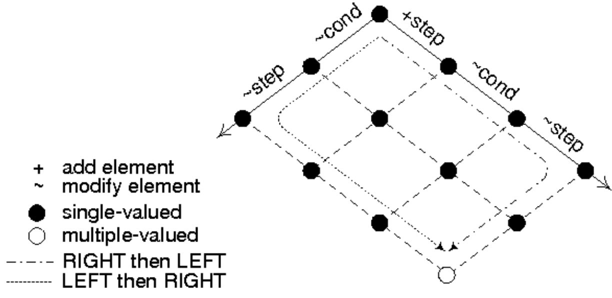

Fig. 9. Plane produced by merge of "Order Product" changes.

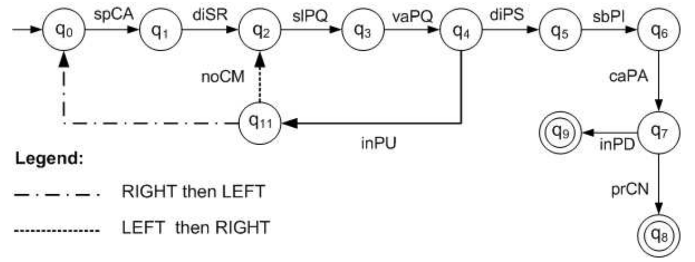

Fig. 10. Composite FSM of merged change operation sequences.

modify extension condition (∼cond), modify terminating step (∼step), add step 4a2 (+step). For reasons of rolling back the frontier set, we choose to distribute the operations along the axes of the change plane as shown in Fig. 9. This is possible because the operations on either side commute with all of the others on that *same* side.

Recall from Section II that conflicts are revealed by weaves producing different models for the same change plane node. Accordingly, for each node we attempt to construct the DSRG-style use case model of every path that terminates at that node, by executing the path's sequence of change operations against the ancestor. Paths that fail (inconsistencies), or that produce nonwell-formed models (inconformities) are not weaves and are handled as described in the preceeding subsections.

What remains are weaves that produce models which may, or may not be equivalent. Formalizing these models by mapping them to finite state machines makes it possible to test their equivalence according to Def. 3 of Section III-B. Doing so identifies the change plane's set of single-valued nodes. The results for the example can be seen in Fig. 9, where a conflict is signified by the presence of a solitary multiple-valued node. This node actually hosts two models: one being generated by a weave that executes all of the changes on the right before those on the left (RtL), and the other by executing the left changes then the right changes (LtR). The node's other weaves can be ignored since they produce the same models as RtL or LtR.

The models produced by RtL and LtR are not equivalent. Though a textual comparison of the use cases might make the conflict apparent, in more involved cases this cannot be guaranteed. Mapping the UCMs to FSMs *guarantees* detection through equivalence checking.

A composite drawing of the two merged models as an FSM is given in Fig. 10. Here the solid lines represent elements resulting from merging together the nonconflicting operations of the change sequences. Only the jump targets of the extension terminators are in conflict: RtL returns to state q_0, while LtR returns to q_2. At this point a decision as to which change to accept must be made, either interactively with the merger, or by means of some heuristic. Maximizing single-valued nodes has resulted in a more complete merged model than if no operation rearrangement had taken place, providing a larger context on which to base merge decisions.

It must be pointed out that our Equivalence Checker is able to determine that the left and right changes affect the same elements only due to the well-formedness rule that requires unique IDs for every element. If a "Product Unavailable" extension had been newly added on each side, correspondence could not have been so easily established. Our merging meta-model uses a third dimension to address this issue of matching strategies, and we refer the reader to [6] for the details.

Another complication we wish to note is the textual conflict that arises due to both modelers changing the same extension's condition (see Fig. 8). The three nodes adjacent to the multiple-valued node of Fig. 9 all suffer from this "conflict." The nodes are single-valued because their generated models do not differ semantically with respect to the use case formalization structure, but only syntactically with respect to textual content. We consider this further in the conclusion.

V. Tool Support

The formal semantic use case foundation and merging meta-model presented in the previous sections form a theoretical basis for developing sophisticated support for the merging of use cases. We envision a "Use Case Merger" as having the following components:

- Syntactic Analyzer
- Equivalence Checker
- Operation Recorder
- Sequence Merger
- Conflict Visualizer

Thus far we have completed the development of the Syntactic Analyzer and the Equivalence Checker components. The others are still in the conception or architecting stages. Each component is overviewed below.

a) Syntactic Analyzer: Given a DSRG-style use case model expressed in XML, the Syntactic Analyzer processes the input to: 1) generate a print-friendly presentation of the use case model in HTML using XSLT technology, and 2) verify that the well-formedness properties summarized in Section III-A are satisfied.

b) Equivalence Checker: Two purposes are served by the Equivalence Checker: 1) mapping a well-formed DSRG-style use case model to an FSM representation, in accordance with the semantics of Section III-B; and 2) verifying whether two FSMs accept the same language and hence should be deemed equivalent, as stated in Def. 3 of Section III-B.

The stand-alone Java implementation of the algorithm for equivalence checking is similar to that used by the CSP model-checker FDR [14]. Equivalence is proved through pairwise comparison of the events accepted by "trace equivalent" states of the base and refining FSMs [15].

c) Operation Recorder: Model change operations provide the raw material for operation-based merging. The Operation Recorder will serve to capture the user actions (as outlined in Section IV-B) taken against a DSRG-style UCM during edit sessions. Tight coupling of the recorder to a particular modeling tool is to be avoided. Adherence to standards will help with this concern—the OMG's proposed distributed versioning model [16] for tracking model change history, and facilitating model element identity across versions holds some promise in this regard.

d) Sequence Merger: Given two DSRG-style UCMs, and the traces of operations by which they were derived from a common ancestor, the Sequence Merger will blend the traces into a single sequence of change operations. This sequence, when played against the ancestor model, will generate the merged UCM. Following the methods outlined in this paper, the Merger will rearrange operations to bypass inconsistencies, and put off conflicts as long as possible by pushing out the frontier set. Inconformities too, must be guarded against lest they be introduced by the merge. The Sequence Merger will depend on the Operations Recorder for its input of operational traces, the Equivalence Checker for conflict detection, and the Syntactic Analyzer for spotting inconformities.

This tool is likely to be built around one of the conflict detection algorithms proposed by Lippe and Oosterom [7]. We are still assessing the computational complexity involved, and looking for ways to shortcut the process. Heuristics to be employed for automatic conflict resolution will also be part of the core. Currently the architecture is taking form, and various platforms are being considered for the tool's implementation.

e) Conflict Visualizer: The last component slated for development is the Conflict Visualizer which will be used to elicit conflict resolution decisions from the user in those case where the heuristics of the Sequence Merger are found lacking. Since the Merger strives to maximize conflict-free merging, its use of the Visualizer will be minimized, requiring less user involvement. Still, the tool will need to present partially merged UCMs in which the conflicting sections are clearly demarcated, and in which the ramifications of merge decisions are made visible.

What form the Visualizer's presentation will take is not known at this time. Rather than develop our own framework we would ideally like to build atop the existing visualization capabilities of a tool like IBM's Rational Software Architect, which in turn is built on the compare and visualization capabilities of Eclipse [17].

VI. Related Work

To our knowledge, this is the first work to propose a semantic foundation for the merging of *textual* use case models. In traditional use case analysis a great deal of what passes for "merging" happens when bringing together inherently ambiguous natural language descriptions of use cases, scenarios, steps, and goals. This situation results from a broad spectrum of stakeholders making use of natural language during the requirements elicitation phase.

The formal syntax and semantics for use case models presented in this paper are part of a larger project involving the formalization of use case and task models [18], [19]. The definition of formal semantics for UCMs has been attempted by various researchers. Fröehlich and Link [20] present a transformation algorithm that derives a UML state chart model from a given set of textual use cases. Rui, et al. suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactorings [21]. They represent scenarios as basic message sequence charts (MSC), partially adapted from the ITU MSC semantics [22].

Starting with use cases represented as UML sequence diagrams, Fernandes, et al. [23] demonstrate how to translate them into colored Petri net models. The translation is performed in a top down manner: first the use case model is mapped into a global Petri net containing a placeholder for each individual use case, then each placeholder is replaced by a sub-Petri net capturing the various scenarios of the use case. Probably the most comprehensive approach has been defined by Somé [24]. He proposes execution semantics for use cases by defining a set of mapping rules from well-formed use cases to basic Petri nets. Similar to our approach, a use case model is deemed well-formed if it conforms to a metamodel and satisfies a set of static well-formedness rules.

Difficulties with model merging tools are described in our evaluation of the state of the art [17]. Findings that have influenced our work here are the heavy demand for knowledgeable human input during the merge, and the level of "cognitive noise" associated with conflict presentation. Model merging has yet to be adequately defined; Brunet, et al. [25] attempt to put it on solid footing by treating a merge as an algebraic operator over models and model relationships. In [26] they describe a fully automated merge of behavioral models based on their model transition system formalism. Unlike our approach, however, it assumes that the replicas to be merged are fully compatible—conflicts, contradictions, and inconsistencies are not considered.

While reliance on unique element identifiers in the UCMs frees us from concerns over *other* matching strategies, it provides less flexibility when it comes to entity correspondence. More is offered by Xing and Stroulia [27] who heuristically measure name and structural-similarity in order to establish correspondence, or Kolovos, et al. [28] who furnish a general comparison language.

We make use of the work of Lippe and Oosterom [7] on operation-based merging. They describe a transformation grid in which merges of object-oriented database changes are weaves of primitive grid transformations. They relate operation commutation to conflict detection, and propose three merge algorithms. Ivkovic and Kontogiannis [29] define a graph metamodel for model synchronization which they apply to

UML class diagrams, establishing model equivalence through a set of dependency relations between model elements.

VII. Conclusion and Future Work

In this paper, a merging metamodel based on model change operations has been used to develop a framework for merging use case models. We have distinguished three types of modeling and merging errors: inconsistencies, inconformities, and conflicts, which we have consolidated under one theory. A means for detecting, and in some cases, automatically resolving these errors has also been provided.

A necessary precondition for the merging of use case models is their formalization. We accomplish this by imposing a formal syntactical structure on the UCM, complete with well-formedness rules, using finite state machines to give the model a formal semantics. Finally, an equivalence operator for FSMs is defined in order to aid in the detection of conflicts in accordance with the merging metamodel. The applicability of our approach is demonstrated through an example "Order Product" use case. Our formal theory has set the stage for development of the Use Case Merger outlined in Section V.

As for future work, besides continuing with implementation of the Merger tool, we are searching for efficient ways to incorporate textual conflict detection and merging into our framework. This is so as not to miss model changes similar to those made to the extension conditions of Fig. 8. Pushing this idea further, we hope to eventually add semantics to those textual areas that represent use case state information. A prerequisite for this is the extension of the semantic model in a way that provides the means for capturing the system state (e.g., using Kripke structures [30]), and also is able to map the occurrence of events (representing use case steps) to a set of allowed system states.

References

[1] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach. New York, NY, USA: Addison-Wesley, Jan 1992.

[2] A. Cockburn, Writing Effective Use Cases, ser. Agile Software Development. Boston, MA, USA: Addison-Wesley, 2001.

[3] C. Larman, Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[4] B. Selic, "The pragmatics of model-driven development," IEEE Softw., vol. 20, no. 5, pp. 19–25, Sep 2003.

[5] T. Mens, "A state-of-the-art survey on software merging," IEEE Trans. on Softw. Eng., vol. 28, no. 5, pp. 449–462, May 2002.

[6] S. Barrett, P. Chalin, and G. Butler, "Towards automated model merging using an operation-based metamodel," Concordia University, Montreal, QC, Canada, Technical Report DSRG-2009-02, May 2009, http://users.encs.concordia.ca/~chalin/papers/Barrett-MODELS09-preprint.pdf.

[7] E. Lippe and N. van Oosterom, "Operation-based merging," ACM SIGSOFT Softw. Eng. Notes, vol. 17, no. 5, pp. 78–87, Dec 1992.

[8] G. Butler, P. Grogono, and F. Khendek, "A Z specification of use cases," in APSEC '97: Proc. 4th Asia-Pacific Software Eng. and Internat. Computer Science Conf. Washington, DC, USA: IEEE Computer Society, Dec 1997, pp. 94–101.

[9] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes, "Validating use-cases with the asml test tool," in QSIC '03: Proc. Third Internat. Conf. on Quality Software. Washington, DC, USA: IEEE Computer Society, Nov 2003, pp. 238–246.

[10] W. Grieskamp, M. Lepper, W. Schulte, and N. Tillmann, "Testable use cases in the abstract state machine language," in APAQS '01: Proc. Second Asia-Pacific Conf. on Quality Software. Washington, DC, USA: IEEE Computer Society, Dec 2001.

[11] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli, "Merging partial system behaviours: Composition of use-case automata," IET Softw., vol. 1, no. 4, pp. 143–160, Aug 2007.

[12] D. Sinnig, "Use case and task models: Formal unification and integrated development methodology," Ph.D. dissertation, Concordia University, Montreal, QC, Canada, Dec 2008.

[13] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," IEEE Comput., vol. 39, no. 2, pp. 25–31, Feb 2006.

[14] A. W. Roscoe, C. A. R. Hoare, and R. Bird, The Theory and Practice of Concurrency. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[15] D. Sinnig, P. Chalin, and F. Khendek, "LTS semantics for use case models," in SAC '08: Proc. 24th ACM Sympos. on Applied Computing, Requirements Eng. Track. New York, NY, USA: ACM, Mar 2009, pp. 365–370.

[16] P. Hnětynka and F. Plášil, "Distributed versioning model for MOF," in WISICT '04: Proc. Winter Internat. Sympos. on Information and Communication Technologies. Trinity College Dublin, May 2004, pp. 1–6.

[17] S. Barrett, P. Chalin, and G. Butler, "Model merging falls short of software engineering needs," in MoDSE '08: Proc. Workshop on Model-Driven Software Evolution, Apr 2008.

[18] D. Sinnig, P. Chalin, and F. Khendek, "Consistency between task models and use cases," in EHCI-HCSE-DSVIS '07: Proc. 14th Conf. on Design Specification and Verification of Interactive System. Berlin, Heidelberg: Springer-Verlag, Mar 2007, pp. 71–88.

[19] ——, "Common semantics for use cases and task models," in iFM '07: Proc. Integrated Formal Methods. Berlin, Heidelberg: Springer, Jul 2007, pp. 579–598.

[20] P. Fröehlich and J. Link, "Automated test case generation from dynamic models," in ECOOP '00: Proc. 14th European Conf. on Object-Oriented Programming. London, UK: Springer-Verlag, Jun 2000, pp. 472–492.

[21] K. Rui, "Refactoring use case models," Ph.D. dissertation, Concordia University, Montreal, QC, Canada, Apr 2007.

[22] ITU-T, "Recommendation z.120 - message sequence charts," Apr 2004, http://www.itu.int/ITU-T/studygroups/com17/languages/Z120.pdf.

[23] J. M. Fernandes, S. Tjell, J. B. Jorgensen, and O. Ribeiro, "Designing tool support for translating use cases and UML 2.0 sequence diagrams into a coloured Petri net," in SCESM '07: Proc. Sixth Internat. Workshop on Scenarios and State Machines. Washington, DC, USA: IEEE Computer Society, May 2007.

[24] S. S. Somé, "Petri nets based formalization of textual use cases," University of Ottawa, Ottawa, OT, Canada, Technical Report TR-2007-11, Nov 2007, http://www.site.uottawa.ca/eng/school/publications/techrep/2007/TR-2007-11.pdf.

[25] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in GaMMa '06: Proc. Internat. Workshop on Global Integrated Model Management. New York, NY, USA: ACM, May 2006, pp. 5–12.

[26] G. Brunet, M. Chechik, and S. Uchitel, "Properties of behavioural model merging," in FM '06: Proc. Internat. Conf. on Formal Methods. Springer, Aug 2006.

[27] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in ASE '05: Proc. 20th IEEE/ACM Internat. Conf. on Automated Software Eng. New York, NY, USA: ACM, Nov 2005, pp. 54–65.

[28] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model comparison: A foundation for model composition and model transformation testing," in GaMMa '06: Proc. Internat. Workshop on Global Integrated Model Management. New York, NY, USA: ACM, May 2006, pp. 13–20.

[29] I. Ivkovic and K. Kontogiannis, "Tracing evolution changes of software artifacts through model synchronization," in ICSM '04: Proc. 20th IEEE Internat. Conf. on Software Maintenance. Washington, DC, USA: IEEE Computer Society, Sep 2004, pp. 252–261.

[30] O. G. Edmund M. Clarke and D. A. Peled, Model Checking. Cambridge, MA: MIT Press, 1999.