Towards Collaborative Modeling Using a Concern-Driven Version Control System

Omar Alam
Trent University
Peterborough, ON, Canada
omaralam@trentu.ca

Vasco Sousa and Eugene Syriani
University of Montreal
Montreal, QC, Canada
{dasilvav, syriani}@iro.umontreal.ca

***Abstract*— This paper presents an approach for collaboration in modeling that builds on ideas of Concern-Oriented Reuse (CORE). CORE is a novel reuse paradigm that supports broad-based model reuse. We leverage the ideas of CORE and version control systems to support collaborative modeling of reusable concerns. We capture our ideas in a metamodel and demonstrate our approach by collaboratively modeling a drone system concern.**

I. Introduction

Nowadays, software developers extensively use version control systems (VCS), such as the Git [1] repositories, to collaborate and track development activities throughout their projects. These systems allow developers to organize and distribute the development effort among themselves, keep track of issues and bugs, and schedule the delivery of releases. In addition, the success of reuse in software development as exemplified by, e.g., class libraries, services, and components facilitated collaboration and coordination during software development activities. A developer now can reuse the artifacts developed by other developers through the reuse interfaces of these artifacts. Together with VCS and repositories, advances in reuse allowed software development activities to be more organized and collaborative. For example, a group of developers can use a Git repository to collaborate in coding the classes of a software program, and another developer can reuse these coded classes.

However, despite these advances in the programming world, application of VCS in model-driven engineering (MDE) is still in its infancy. For MDE practitioners, a VCS is mostly used to keep track and collaborate in the development of models [2]. However, unlike VCS repositories for programs, reuse of models in repositories through VCS is still a challenge in MDE [3]. Enforcing consistent reuse is necessary to cope with the growing complexity of software systems. Nevertheless, modelers usually create models from scratch because modeling languages offer limited support to reuse existing models and modeling tools in general are not shipped with a library of reusable models. Lack of support for reuse limits the potential of VCS in MDE, as checking out, committing, and updating models that cannot be reused will not be very useful for collaborative environments.

This paper tackles the aforementioned issues by leveraging the ideas of Concern-Oriented Reuse (CORE) [4], [5] and VCS to support collaborative modeling. CORE defines a new reuse unit, called *concern*, that enables broad-scale model reuse. Whereas developers in existing collaborative environments collaborate to develop classes of a code/model base, in our approach, modelers collaborate to incrementally model the features belonging to a CORE concern. We capture our ideas in a metamodel and demonstrate our approach through modeling an autonomous drone system collaboratively. We believe that our reuse-focused, feature-driven, collaborative and incremental modeling approach allows modelers to better distribute development effort among themselves, manage development life-cycle activities, and increase productivity and time to market.

In the following section, we provide background information on CORE and introduces our running example. In Section III, we present how we support versioning in CORE. In Section IV, we discuss the lessons learnt in modeling the drone concern. In Section V, we present related work and conclude in Section VI.

II. Background on CORE

CORE [4], [6] is a new software development paradigm inspired by the ideas of multi-dimensional separation of concerns [7]. CORE builds on the disciplines of MDE, software product lines (SPL) [8], goal modeling [9], and advanced modularization techniques offered by aspect-orientation [10] to define flexible software modules that enable broad-scale model-based software reuse. CORE introduces a modular unit of reuse called concern which encapsulates a set of software development artifacts, i.e., models and code, describing all properties of a domain of interest during software development in a versatile, generic way [4]. The models within a concern can span multiple phases of software development and levels of abstraction (from requirements, analysis, architecture, and design models to code). Concerns decompose software into reusable units according to some points of interest [11], [12] and may have varying scopes, e.g., encapsulating several authentication choices, communication protocols, or design patterns. The main premise of CORE is that recurring development concerns are made available in a concern library, which eventually should cover most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as alternative

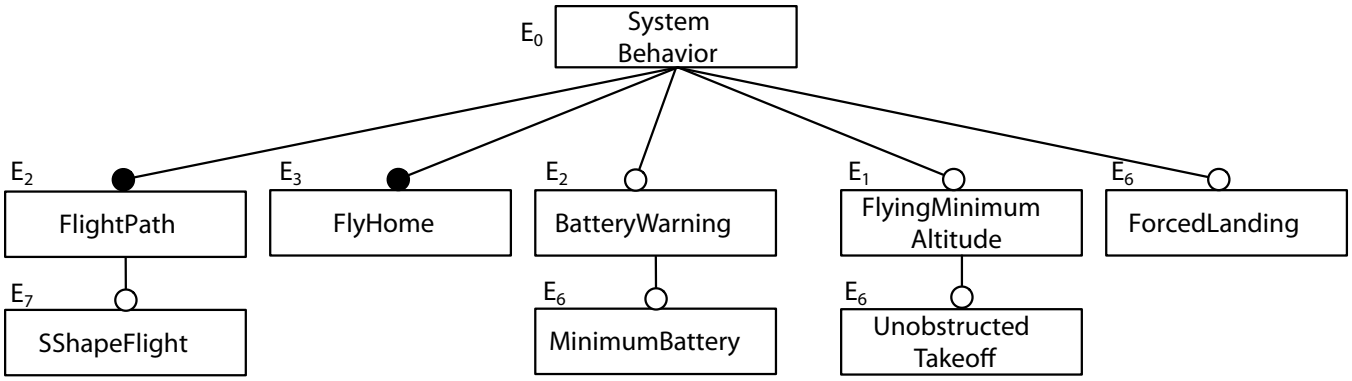

Fig. 1. Drone product line feature model

architectural, algorithmic, and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces. The same idea is applied to the development of concerns as well: high-level/more specific concerns can reuse low-level/more generic concerns to realize the functionalities they encapsulate. In the end, the architecture of a software developed with CORE takes the form of a concern hierarchy (directed, acyclic graph), thus supporting hierarchical modularity [13].

CORE advocates a three-part interface [5] to describe each concern and enable its reuse: the *variation interface* for exposing decisions (using a feature model [14]) and their impact on system qualities (using an impact model [15]); the *customization interface* to adapt the chosen variation to a specific reuse context (customization is here used with a different meaning than in the SPL [14] paradigm); and the *usage interface* to trigger the functionality encapsulated by the customized concern. In CORE, models are built for the root phase and all follow-up phases using the most appropriate modeling formalisms to express the properties of the concern that are relevant during each phase. Consequently, a concern is typically described by many modeling notations, (e.g., based on the Unified Modeling Language (UML) [16]), but can also offer other language mechanisms (e.g., aspect-oriented features).

A. Running example: drone system

To illustrate the ideas of CORE, we use the example of developing a drone concern. This concern is part of a larger surveillance system. It describes the behavior of different variants of a drone system with surveillance purposes. We modeled this concern following the ideas of incremental modeling in CORE. A concern is built by adding small model increments to some base models. In other words, instead of creating large monolithic models of a concern for each variation, the model is decomposed into many small *realization models*. The feature model [14] of this concern is shown in Figure 1 which presents the common and varying features to make the drone system work properly. This feature model is part of the concern interface and is modeled incrementally. The root feature DroneSystemBehavior in Figure 1 is the starting point of the incremental modeling effort. After that, we modeled other features incrementally in a process detailed in the next section. Each feature is realized by a *realization model* modeled as a state machine. When the user of a concern makes a feature selection (i.e. configures the concern), a customized concern is generated by composing its realization models [5].

The state machine shown in Figure 2 realizes the root feature by modeling the basic behavior of a drone system. This state machine consists of a sequence of states that define the different stages of a drone's operation. For example, the state TakeOff provides the minimum behavior required to take off, which will be further refined to include additional behavior. A realization model can *extend* another realization model [4] to have a complete excess to the structure and the behavior of the extended model.

The realization model for the feature FlyHome shown in Figure 3 refines the base realization model of the root feature (Figure 2) to define the behavior of the Descend state. When we specify that FlyHome *extends* SystemBehavior, FlyHome will have complete access to the states and transitions defined in SystemBehavior, hence, there is no need to re-define them again. Realization models in CORE often extend each other and produce a complex extension hierarchy [17]. Realization models in the upper levels of the extension hierarchy often also designate the concern-partial elements [4] that must be concretized within the concern. These elements are noted with a disjoint bar symbol preceding the element name, such as states Fly and Descend as shown in Figure 3. Their role is to define generic properties shared by a set of realization models, but require to be completed with additional properties in order to function correctly.

Breaking down the concern into small realization models that are modeled incrementally allows adding the increment when the knowledge required to model it becomes available. It also allows multiple modelers to collaborate when developing the concern, with each modeler being responsible to model a particular set of increments. Furthermore, incremental modeling in CORE allows the concerns to continuously evolve by adding features as needed by the reusing concern/application.

In the next section, we discuss how we leverage the CORE

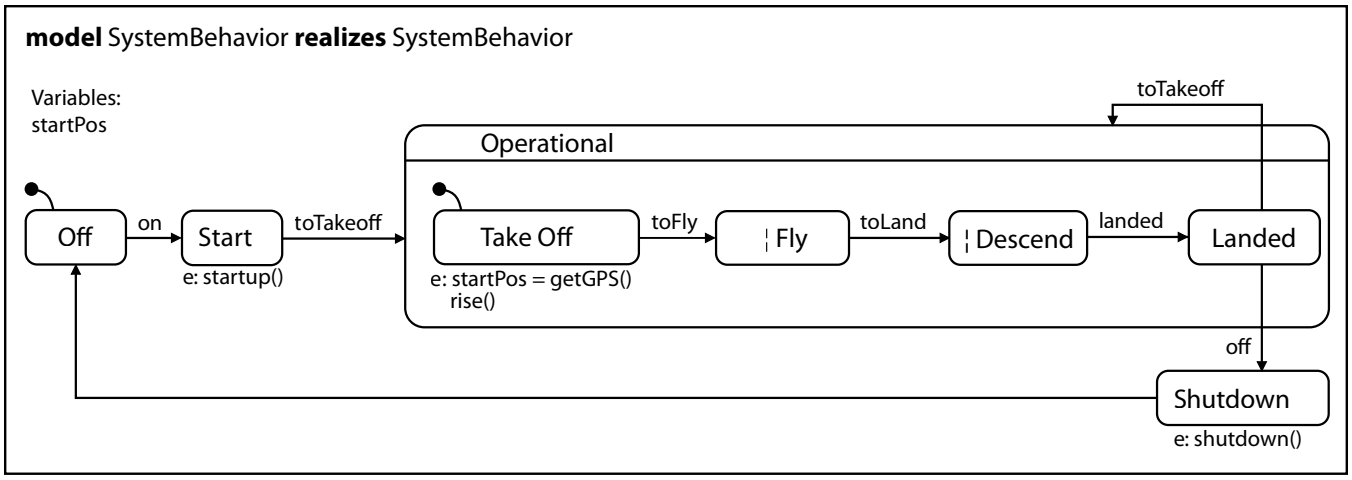

Fig. 2. Base drone behavior Statechart

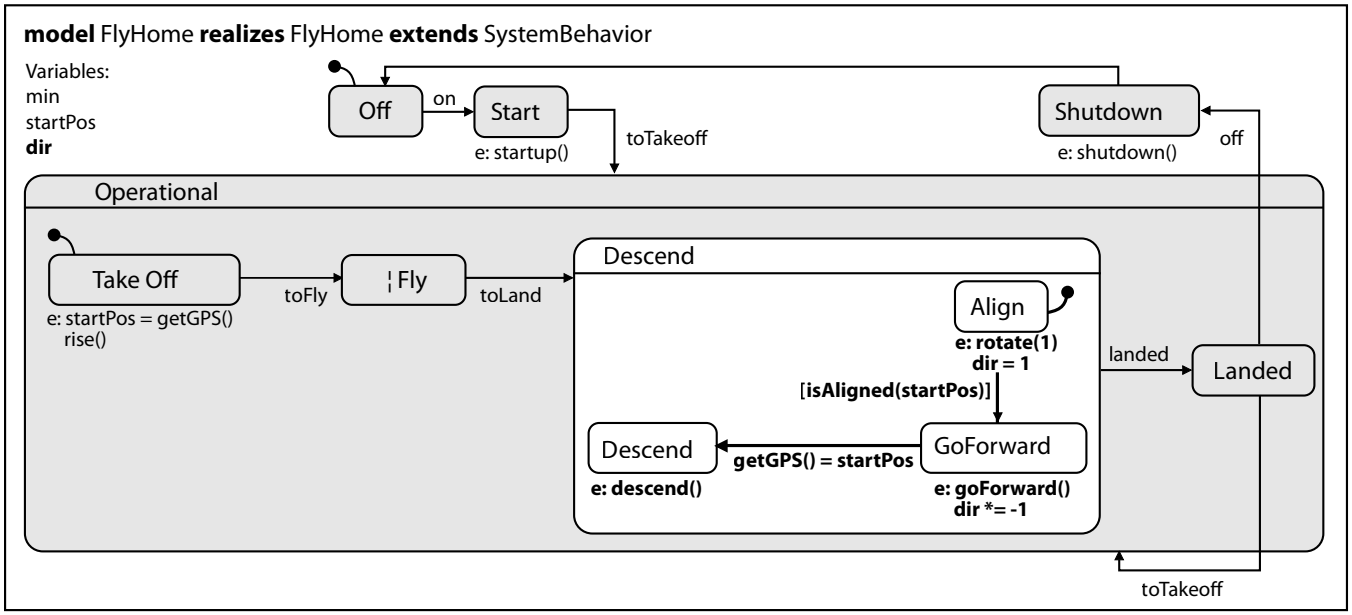

Fig. 3. Realization model of the FlyHome feature

concepts to support collaborative modeling. We show how multiple modelers participate in incrementally modeling an example drone concern.

III. Support for Versioning in CORE

In this section, we present how we leverage the concepts of CORE, such as reuse, incremental modeling, interfaces (especially the variation interface) to support collaborative modeling. We present an approach that allows multiple modelers to participate in developing a concern through the support of a concern-driven VCS. In the example discussed later in this section, each modeler is responsible to model a particular feature of a concern after discussing with the development team. The feature tree of the concern is built incrementally by extending the realization models, and collaboratively by assigning features to different modelers. The CORE composition algorithms, along with the composition/refinement mechanisms of the modeling language are responsible to derive the *common model* for a particular feature selection [17]. Since a concern is designed in a modular way, the relationships between features and the dependencies between realization models set the course for collaboration: easing the process of assigning tasks to collaborators.

A. Extension of CORE metamodel

The concepts of CORE are captured by a metamodel and supported by a reference implementation [4] that allow different modeling languages to support concern-orientation. We extend this metamodel with new concepts to support versioning and collaborative modeling. Figure 4 presents a simplified view of the extended CORE metamodel. The shaded classes depict the new concepts that are added to CORE to support versioning, while the others are the existing CORE concepts related to incremental modeling, reuse, and

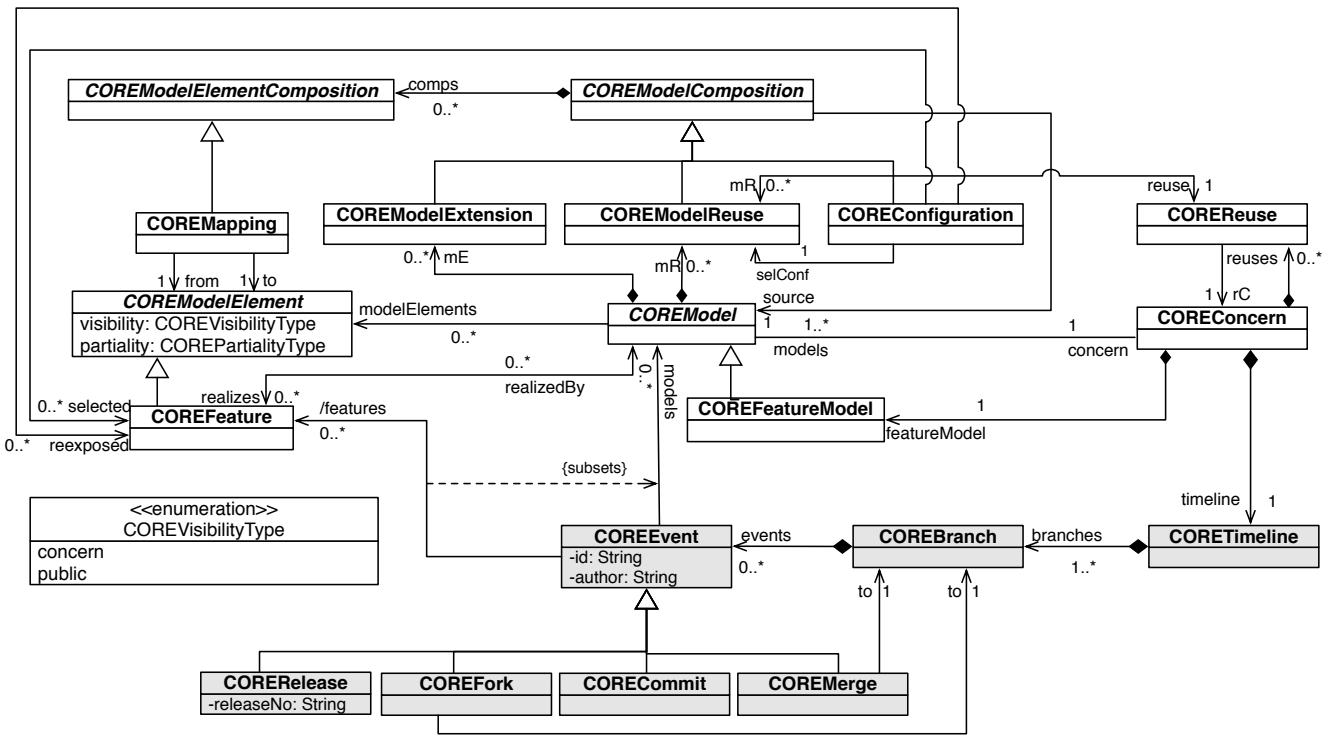

Fig. 4. A simplified CORE metamodel.

composition. As shown in Figure 4, a COREConcern contains a timeline (CORETimeline) which is the development timeline maintained by the VCS. The timeline has branches (COREBranch) which contain events (COREEvent). An event can be a merge (COREMerge), fork (COREFork), commit (CORECommit), or release (CORERelease). These concepts are closely related to the concepts of VCS, such as Git [1].

Each versioning event is linked with the set of features of the concern that the modeler modified or added. COREEvent has a link to COREFeature to allow the versioning event to be associated with the concern's features. This link is derived from the link between COREEvent and COREModel, which allows an event can be associated with any kind of model in CORE (including feature models, impact models, and models of the CORE-based modeling languages). These events follow the standard semantics as in common VCS. A developer can create a release if (1) all generated customized concerns correctly satisfy the constraints of the feature model and (2) all concern-partial elements are concretized in at least one generated customized concern.

As discussed in [4], [17], COREFeatures realize a set of realization models. The realization models are linked with directed extension dependencies (COREModelExtension). The association realizedBy/realizes between COREFeature and COREModel encodes which realization model belongs to which feature. A CORE-based modeling language will subclass from COREModel, and in our case, CORE Statechart-Model (not shown in Figure 4) subclasses COREModel so that all the fine-grained changes in the statecharts are still in CORE.

B. Development timeline

The extension to the CORE metamodel allows us to develop our *DroneSystem* concern incrementally and retain a trace of its development. Figure 5 illustrates the development timeline of the concern that we, the three authors of this paper, followed. It shows the branches used to organize and divide the development effort. Figure 5 also includes the sequence of events we created. The numbers in Figure 1 illustrate the order of the events in which the features were modeled during the development timeline.

In the beginning, we discussed and collaborated to model the root feature DroneSystemBehavior (i.e., the base realization model shown in Figure 2), which depicts the minimum behavior of the drone system. After the feature DroneSystemBehavior was developed, we did the first commit in the concern's timeline at *event* E_0 as shown in Figure 5. After that, we added features that add details to the behavior of the root feature. The modelers collaborated in adding an optional feature FlyingMinimumAltitude, which guarantees that a drone only operates at a minimum safety altitude. This feature refines the TakeOff state in DroneSystemBehavior.

At this point, we decided to work separately in different branches to speed up the development time. Therefore, we *forked* two additional branches were created at event E_1, and the modelers decided to model a separate feature in each branch. In one branch, the BatteryWarning feature was modeled, which monitors whether the drone has enough battery to operate, and issues a warning if it reaches bellow 20% of capacity. In another branch, FlightPath was modeled as mandatory feature that defines the flight path procedure

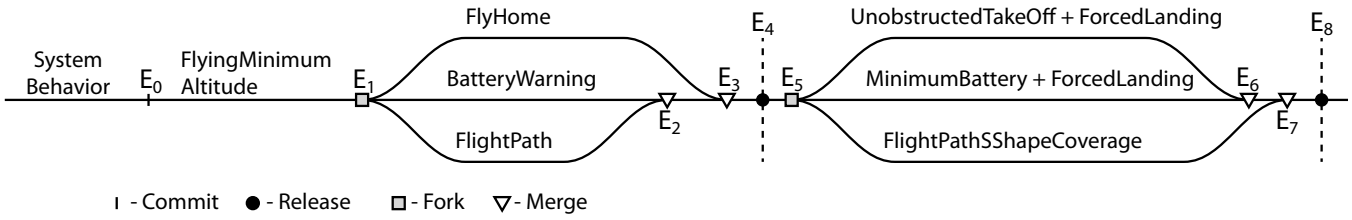

Fig. 5. Development timeline

to cover a particular area. In the third branch, FlyHome was also modeled as a mandatory feature to specify the behavior of returning back to the starting coordinates when the drone is asked to land.

When modeling of these features were completed, they were merged with each other. First, the FlightPath branch was merged onto BatteryWarning branch at event E_2. Then, the FlightHome branch was merged onto the BatteryWarning branch at event E_3. It is important to note that we could designate one of the three branches as the main branch and the two other as secondary branches as the way it is done in Git. In that case the BatteryWarning branch could be designated as the main branch that other branches merge onto. After the merge at E_3, the modelers decided that there the current features of the concern are sufficient to make a release.

The concern is now released at E_4 with five features: DroneSystemBehavior, FlyingMinimumAltitude, FlightHome, BatteryWarning and FlightPath. After that, we *forked* branches again to model three additional features, UnobstructedTakeOff, MinimumBattery and SShapeFlight. UnobstructedTakeOff ensures the drone only operates if it can reach its minimum operation altitude without obstructions. The MinimumBattery feature ensures the drone only stays airborne if it has enough battery to operate and return to its origin point. Finally, SShapeFlight directs the drone to fly repeatedly in a specific *S* shape to cover an area.

When we merged the branches that modeled UnobstructedTakeOff with MinimumBattery, we discovered that there is a conflict that occurred as a result of a particular behavior (ForcedLanding) in both models. We discuss how we resolved this conflict in the next section. After all branches were merged, we had a feature model that contains all the features that we intended the drone system to have. Therefore, we released a second release for this concern at E_8.

IV. Discussion

We discuss some of the benefits and reasons for collaborative modeling using our approach.

A. Reasons for branching

The example in the previous section assigns one modeler per branch to model one feature. This is not a constraint in our approach. Depending on the context and the nature of the concern, any number of modelers can be assigned to a branch to model any number of features. There are several factors that should be taken into account when creating a new branch or assigning modelers to a branch. Usually, a branch is created to distribute the modeling effort by assigning modelers to branches, hence, speeding up the development. Furthermore, a new branch can be created to assign the expert(s) for modeling particular feature(s) to that branch. Therefore, branches can be used to organize the development team, i.e., modelers can be assigned according to their specialization and expertise, and their authorship of the changes such as commit and merge can be traced (COREEvent has an author attribute as shown in Fig. 4).

Modelers can also be assigned based on their experience in the concerns that are going to be reused by features of a branch. For example, if a feature or group of features should reuse a security concern, then a branch can be created to model those features and modelers who are expert in security can be assigned to that branch.

In addition, branches can be created to partially model a feature, i.e., a branch can be created to model some realization models of a feature, but not all of them. This will be particularly useful for concerns that encapsulate models created using different modeling notations. For example, in a concern that encapsulates requirement and design models, a branch can be created to model the requirements models of the concern and can be merged with other branches that are also modeling requirements. Then new branches can be created to model the design models. In this situation, branching is helpful for planning the development activities according to software process that is being followed by the development team. However, modelers need to careful not to branch or merge when there are invalid feature or realization models. Automatic verification and validation methods can be used verify the consistency and correctness of the models in a branch [18].

B. Evolution of the feature model

In the previous section, the features of the drone system are added incrementally through the collaboration of three modelers. Initially, all three modelers participated in modeling the root feature, which was committed at E_0. Then, each modeler modeled a separate subfeature of the root in the branches created at E_1. Since concerns are modeled incrementally, either by adding realization models or by reusing other concerns [17], [4], it is not unusual that more than one modeler participates in adding the increments. In some cases, all participants can collaborate on modeling a single feature as in the case of modeling the root feature

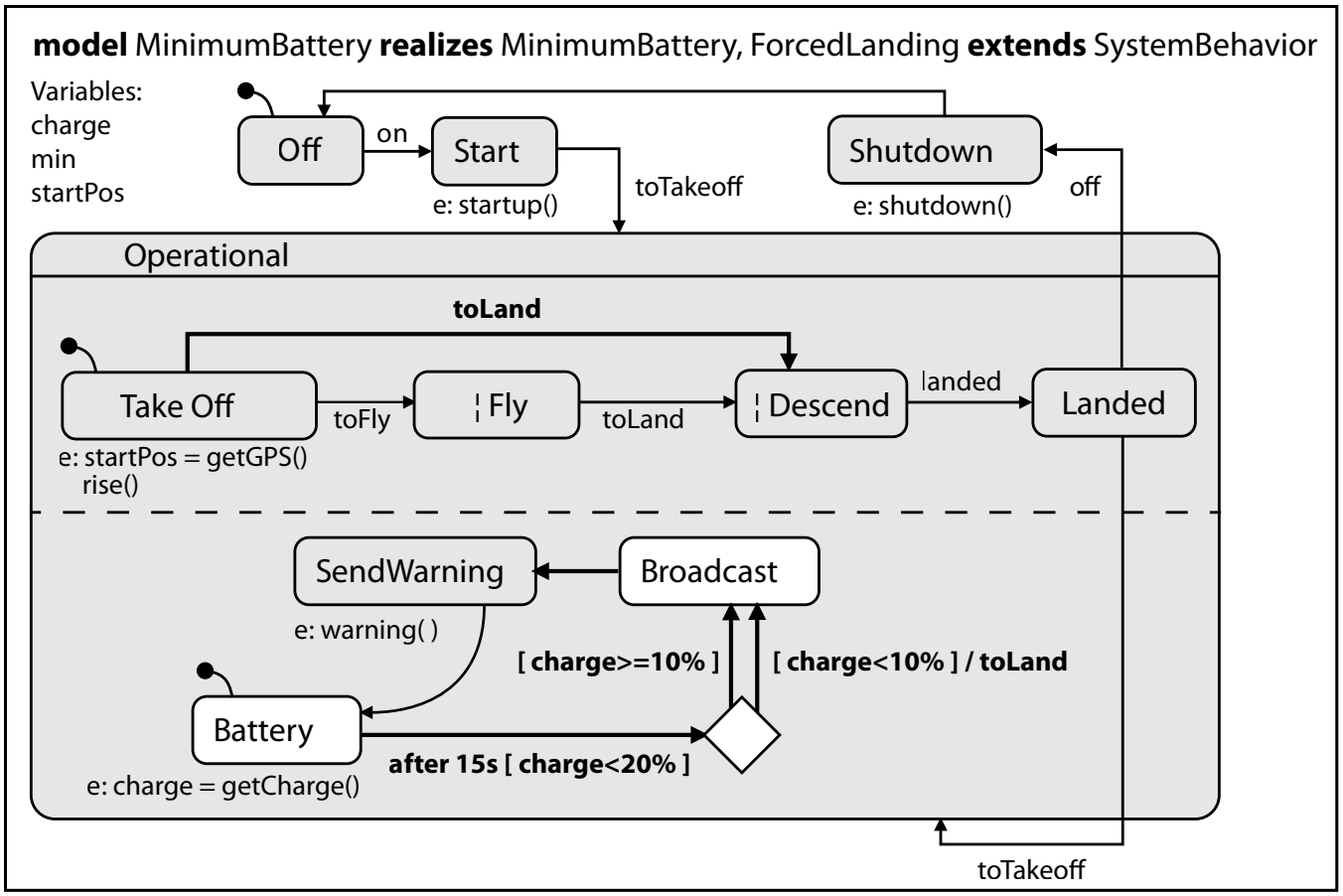

Fig. 6. Realization model of the MinimumBattery feature. The transition toLand is removed after the merge to resolve a conflict.

as explained in the previous section. By doing so, they are able to discuss, plan, and set the course for adding additional features. All participants share a common understanding of the root feature before adding additional variants. The feature model grows incrementally by adding features through collaboration and branching. Modelers discuss before branching which features they are going to model and the consistency of the feature model can be checked during merging.

It is also possible to model the entire feature model only (without the realization models) before creating any branch in the timeline, apart from the main branch. In this case, modelers have the opportunity to collaborate in planning all the possible variants that they want to model upfront. In the branches, they model the realization model for the planned features.

In addition, since a CORE concern groups models of different modeling notations belonging to multiple development phases, our approach allows to use heterogeneous modeling languages collaboratively in a single project, allowing for better development life-cycle management.

In summary, our approach allows for different ways to plan the feature model, modelers can collaborate in modeling the feature model before realizing the individual features, or they can opt for starting with a root feature and branching to collaboratively add features and realizing them.

C. Conflict management

Similarly to VCS such as Git [1], our approach maintains a remote copy of the concern and local copies for each branch. When creating a new branch, a copy of the concern will be created for local use in that branch. The user can push her changes to the remote copy and conflicts can be resolved when they arise during commit or merge, similarly to how systems such as Git do. However, in our approach, only features that are changed need be checked for conflicts when merging branches. Furthermore, since CORE models are added incrementally (preserving the consistency of existing models [4]), only changes in the realization models need to be examined for conflicts during merge operations. This saves time as feature and realization models tend to grow in size when reusing other concerns [17], [19].

Features and models that are not changed and are not in conflict with the changed features/models can be automatically merged. However, in case of a conflict, the extension hierarchy of the realization models along with the constraints of the feature model will allow for dependency analysis to resolve the conflict [17]. When a conflict occurs between two or more realization models, a *conflict resolution model* can be introduced to resolve this conflict. For example, during the merge operation at E_6, we found that both *MinimumBattery* (Fig. 6) and *UnobstructedTakeoff* (Fig. 7) has a transition

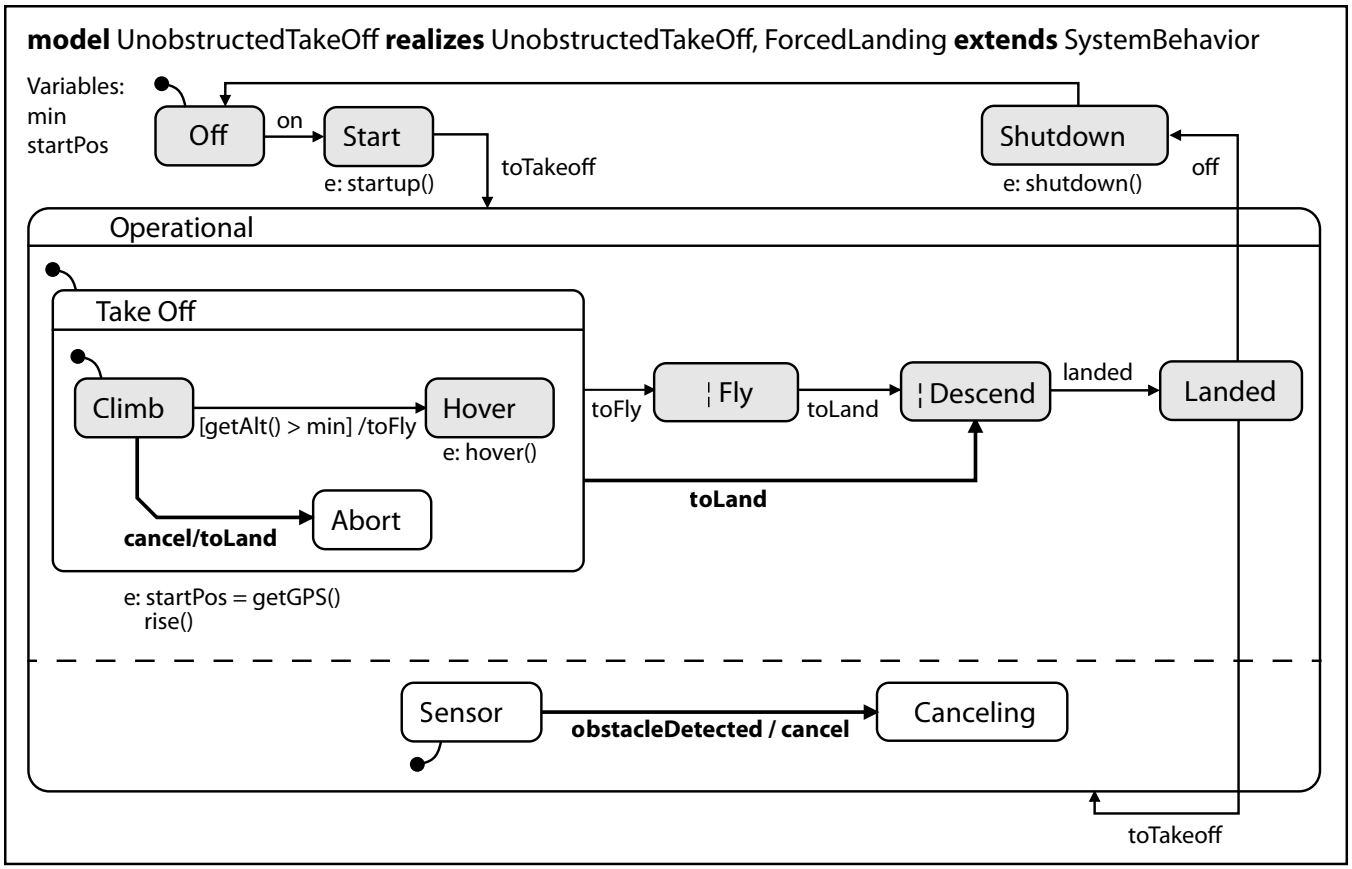

Fig. 7. Realization model of the UnobstructedTakeOff feature. The transition toLand is removed after the merge to resolve a conflict.

called (*toLand*) from *TakeOff* to *Descend*. This transition allows the system to land before the drone starts flying, i.e., the drone system takes off and lands immediately. When features *MinimumBattery* and *UnobstructedTakeoff* are selected, two transitions with the name *toLand* will appear in the model, resulting a conflict. The system does not know which *toLand* transition to fire. We resolve this conflict by first removing the *toLand* transition from both *MinimumBattery* and *UnobstructedTakeoff*, then introducing a model that defines this transition, and make both *MinimumBattery* and *UnobstructedTakeoff* extend that new model. We call the new model *ForcedLanding*, shown in Fig. 8. *ForcedLanding* has a pointcut that detects the occurrence of *TakeOff* and *Descend* states and replaces them with the advice part which adds the toLand transition between those two states. There are other ways to resolve conflicts, some are discussed in [17]. In cases where conflicts occur because of reusing a particular concern (e.g., a feature from the reused concern conflicts with a feature from the reusing concern), then the conflict can be resolved either by using a conflict resolution model or by making a different feature selection from the reused concern. However, it is possible that some conflicts cannot be resolved using resolution models or by choosing an alternative variant from the reused concern (or the conflicting variant of the reused concern provides important functionality that cannot be replaced).

Despite the efforts from modelers to resolve conflicts as they arise, there are conflicts that are unresolvable or are too difficult/costly to resolve. When such conflicts occur during a merge or commit event, the concern designers can express them using *excludes* relationship in the feature model [14]. In some cases, a conflict can be resolved by selecting additional feature(s), as we discuss in the next subsection. In these cases, the concern designers can explicitly express these additional features to be selected. The advantage of using our approach over how concerns are currently designed, is that the concern designers can "revert" back when none of these solutions work, i.e., discard the features/models that introduced the conflict during the merge, and branch again to re-model them in a way that resolves the conflict.

D. Re-modeling of features

As discussed in the previous subsection, we resolved the conflict between the models in Fig. 6 and Fig. 7 by introducing a new model called *ForcedLanding* shown in Fig. 8. Here, we discuss a different way for resolving this conflict. We notice that even if the two *toLand* transitions are syntactically the same, they may get triggered by different situations. Therefore, these are domain-specific transitions that cause a semantic conflict. The merge operation could be parameterized by a semantic conflict detection function. We resolve the conflict by introducing a new feature in the feature model that realizes this shared behavior. During the

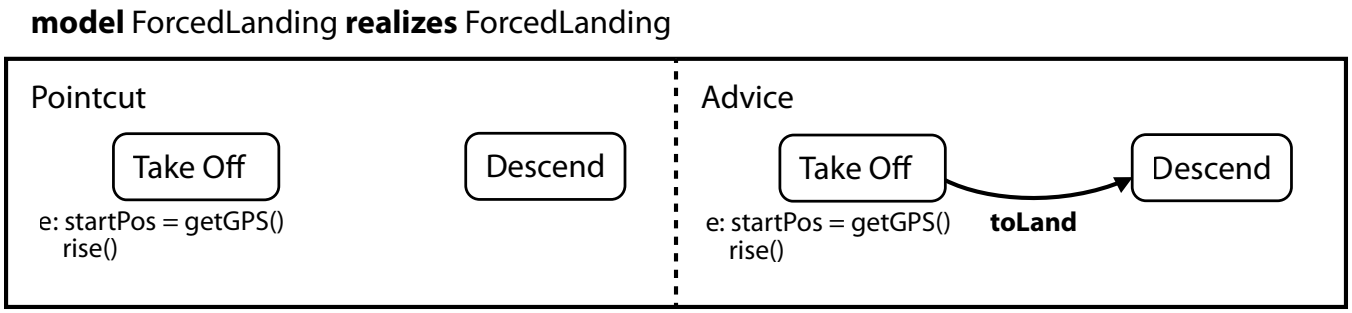

Fig. 8. Aspect pointcut and advice for ForcedLanding

merge operation, we decided to introduce a new optional feature called *ForcedLanding* (see Figure 1) that realizes the point-cut advice model that we initially introduced to resolve the conflict discussed previously. We make both realization models, shown in Figure 6 and Figure 7, realize the newly introduced feature *ForcedLanding* in addition to their respective features, *UnobstructedTakeoff* and *MinimumBattery*. Both features *require* the feature *ForcedLanding*, using the *require* constrain of feature models.

It is also possible to re-model a set of features as a separate concern. For example, during the merge operation at E_6, we can extract the battery features (*BatteryWarning* and *MinimumBattery*) from the *DroneSystem* and include them in a new concern which encapsulates all battery related features. For example, the new concern encapsulates additional features that allow the user to specify whether the battery could be recharged or should be disposed. When battery is separated from *DroneSystem* as a reusable concern of its own, we have to identify the features in *DroneSystem* that are affected by this re-modeling. In particular, we have to reuse the battery concern in those features and adapt their realization models accordingly. The same process should be followed when a new release of the battery concern comes out and we would like to reuse the new upgraded battery. We have to identify the features that reuse the old battery concern, remove the old reuse, and establish new reuse for the upgraded version of the battery concern. Finally, for successful reuse of the battery concern, we have to check for any conflict between the reused features in the battery concern and the reusing features of the DroneSystem concern [17].

V. Related Work

Although CORE comes with a library of reusable concerns, currently there is no support for VCS in CORE. Therefore, support for collaboration is limited in CORE. TouchCORE [20], which is a touch-based modeling tool for CORE-based design modeling, allows multiple modelers to collaborate in modeling through a touch-based screen. However, it does not support tracking development activities, branching, merging provided the VCS discussed in this paper.

Other modeling technologies provide some support for collaborative modeling. Commercial tools such as Rational Rhapsody [21], Visual Paradigm [22], MagicDraw [23], and Enterprise Architect [24] are modeling tools that are used in industry and support collaborative modeling. Some technologies such as MagicDraw, Eclipse CDO [25] and EMFStore [26] provide some support for VCS. Rocco et al. provide an overview these tools and discuss their potentials and shortcomings [2]. They acknowledge that support for reuse and discovering reusable artifacts is limited, increasing the upfront development cost for many model-based projects. Therefore, the potential benefits of collaborative modeling in these approaches is limited. In addition, their ad-hoc architectures make it difficult for domain-specific collaboration and version control, and their simplistic use of locking/conflict management slows down productivity [27]. However, a concern in our approach is modeled incrementally and collaboratively through reusing existing models. Since different modeling languages can be integrated under the CORE umbrella [4], our approach has the potential to be language-independent, allowing different modeling languages and tools (including domain-specific languages) to adopt our ideas. Furthermore, our approach proposes an explicit task-assignment mechanism based on the feature and realization models of a concern. Lastly, our approach allows for releasing versions of a concern, we are not aware of any modeling approach that supports gradual releasing of models.

VI. Conclusion

This paper introduces a novel approach for collaboration in modeling CORE concerns. A concern groups related heterogeneous models and provide interfaces to facilitate reuse. We build on ideas of CORE such as reuse and incremental modeling to support collaborative modeling using a VCS. We capture the ideas of our approach in a metamodel and demonstrate the effectiveness of our approach using a an example drone concern. In future, we plan to conduct larger case studies using multiple modeling languages to evaluate our approach. We also plan to integrate our approach with some existing based modeling tools.

We expect that this feature-driven, concern-oriented, collaborative modeling approach will make collaborative modeling easier, faster, and simpler with modular ways to resolve conflicts.

References

[1] "Git," Last accessed: 2017. [Online]. Available: https://git-scm.com/

[2] J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Collaborative repositories in model-driven engineering [software technology]," *IEEE Software*, vol. 32, no. 3, pp. 28–34, 2015.

[3] J. Whittle, "The truth about model-driven development in industry - and why researchers should care," 2012. [Online]. Available: http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/

[4] O. Alam, "Concern-oriented reuse: A software reuse paradigm," Ph.D. dissertation, McGill University, 2016.

[5] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013*, ser. Lecture Notes in Computer Science, vol. 8107. Springer Berlin Heidelberg, 2013, pp. 604–621.

[6] W. Al Abed and J. Kienzle, "Information Hiding and Aspect-Oriented Modeling," in *14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009*, October 2009, pp. 1–6.

[7] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, "N degrees of separation: Multi-dimensional separation of concerns," 1999, pp. 107–119.

[8] K. Pohl and A. Metzger, "Variability management in software product line engineering," in *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. ACM, 2006, pp. 1049–1050.

[9] International Telecommunication Union (ITU-T), "Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition," approved October 2012.

[10] R. Filman, T. Elrad, S. Clarke, M. Akşit, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[11] E. W. Dijkstra, *A Discipline of Programming*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[12] D. L. Parnas, "A technique for software module specification with examples," *Communications of the Association of Computing Machinery*, vol. 15, no. 5, pp. 330–336, May 1972.

[13] M. Blume and A. W. Appel, "Hierarchical modularity," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 813–847, Jul. 1999.

[14] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[15] G. Mussbacher, J. Araújo, A. Moreira, and D. Amyot, "Aourn-based modeling and analysis of software product lines," *Software Quality Journal*, vol. 20, no. 3-4, pp. 645–687, 2012.

[16] O. M. Group, *Unified Modeling Language: Superstructure (v 2.4.1)*.

[17] J. Kienzle, G. Mussbacher, P. Collet, and O. Alam, "Delaying decisions in variable concern hierarchies," in *GPCE 2016*, 2016, p. to be published.

[18] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature modelling and traceability for concern-driven software development with touchcore," in *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, 2015, pp. 11–14.

[19] O. Alam, J. Kienzle, and G. Mussbacher, "Modelling a family of systems for crisis management with concern-oriented reuse," *Softw., Pract. Exper.*, vol. 47, no. 7, pp. 985–999, 2017.

[20] W. Al Abed, M. Schöttle, A. Ayed, and J. Kienzle, "Concern-oriented behaviour modelling with sequence diagrams and protocol models," in *Behavior Modeling - Foundations and Applications*, ser. LNCS. Springer, 2015, vol. 6368, pp. 250 – 279.

[21] "Rational rhapsody designer manager," Last accessed: 2017, www-03.ibm.com /software/products/en /ibmratirhapdesimana.

[22] "Visual paradigm," Last accessed: 2017, https://www.visual-paradigm.com/.

[23] "Magicdraw," Last accessed: 2017, https://www.nomagic.com/.

[24] "Enterprise architect," Last accessed: 2017, http://www.sparxsystems.com/products/ea/.

[25] "Cdo model repository," Last accessed: 2017, http://www.eclipse.org/cdo/.

[26] "Emfstore," Last accessed: 2017, http://www.eclipse.org/emfstore/.

[27] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, ser. BigMDE '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:10.