Feature Interaction Analysis of the Feature-Oriented Requirements-Modelling Language Using Alloy

David Dietrich, Pourya Shaker, Joanne M. Atlee, and Derek Rayside
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{d4dietri, p2shaker, jmatlee, drayside}@uwaterloo.ca

Jan Gorzny
University of Toronto
Toronto, Canada
jgorzny@cs.toronto.edu

ABSTRACT
Feature-oriented software development (FOSD) has several benefits over traditional development practices, but also introduces the problem of feature interactions. We have defined a method for detecting semantic feature interactions in models expressed in the feature-oriented requirements modelling language (FORML) by translating a FORML model into an Alloy model and using the Alloy analyzer to detect interactions. We have implemented a tool that handles the translation and interaction analysis. In this paper we present our method for detecting feature interactions, show that it scales well and discuss how it can detect feature interactions in partial models.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms
Verification

Keywords
Feature-oriented requirements, Feature interactions

1. INTRODUCTION
Feature-oriented software development (FOSD) favours the treatment of features as first-class entities during all development phases. A major benefit of FOSD is the ability to decompose a software project into features such that multiple products can be derived from different combinations of features. However, features may not be separate concerns: they may conflict over the values of shared variables or may indirectly interact with each other over their effects on the environment [2]. The feature-oriented requirements-modelling language (FORML) [16] is a modelling language created to support FOSD concepts. In this paper we present a method for detecting unintended feature interactions in FORML models.

In the FORML, the environment is modelled for all features, and each feature's behaviour is modelled as a separate UML-like state-machine [15] that can be composed with other feature's models to create a model of a software product line (SPL). Features interact in their actions on a shared environment. We have chosen to use the FORML because it defines the composition of features as well as the meaning of a feature interaction. This has allowed us to focus on the problem of detecting feature interactions.

We use the Alloy analyzer to detect feature interactions automatically [10]. A feature interaction occurs when the actions performed by two separate features conflict, or when an action violates a specified constraint. During the translation from FORML to Alloy, an Alloy assertion is generated for every pair of transitions that can execute concurrently. If such an assertion does not hold, then the pair of transitions may interact (false positives may occur due to the static nature of our analysis). The Alloy visualizer can be used to view the environmental state exactly before the interaction occurs. By using the Alloy visualizer, we are able to display interactions to users in an intuitive manner.

The contributions of our paper are:

1. Interaction-detecting assertions are generated automatically, based on the feature models being analyzed. Potential interactions do not need to be known in advance.
2. The analysis works on partial models of feature behaviour.
3. Intended feature interactions are not reported – only unintended interactions.
4. We have addressed the scalability issues that arise by reusing the set of Alloy-generated model instances.

The rest of this paper is organized as follows. Section 2 provides detail about the translation from FORML to Alloy. A short introduction to the FORML is given in Section 2.1. This paper assumes that the reader is familiar with the Alloy language. Sections 3 and 4 discuss our method of detecting interactions and the visualization of counter-examples when an interaction is detected. Section 5 presents a small case study, and Section 6 discusses the results of our work. Section 7 presents related work, and Section 8 presents some concluding remarks and future work.

2. TRANSLATING FORML TO ALLOY

We have created a tool, *FORML2Alloy*, that automatically translates a FORML model into Alloy. The input provided to *FORML2Alloy* is a textual FORML model (FORML was created as a graphical language, but a textual grammar has also been defined). The translator is written in the Turing eXtender Language (TXL) [7].

The remainder of this Section provides a brief introduction to the FORML language (Section 2.1) and describes the translation process in detail (Section 2.2). A detailed description of the FORML can be found in a recent paper by Shaker et al. [16]. Throughout the rest of this paper we will use a running example of a feature interaction between a BDS and a Cruise Control (CC) feature. BDS handles the basic driving functionality of a vehicle (acceleration, deceleration, steering). CC maintains the vehicle speed at a driver set limit. The behaviour model for CC will not be shown due to space constraints, but can be found in the paper by Shaker et al.

2.1 FORML

The FORML is composed of two sub-models: a world model and a behaviour model. The syntax and semantics of the world model is similar to a UML 2.0 class diagram. The world model defines all of the environmental phenomena that are relevant to the requirements of a set of features. The world model is composed of a number of *concepts*, which represent types of environmental phenomena, relationships between them, and the features in the software product line. A feature concept (bordered by dashed lines) describes the variables, and input and output messages of a specific feature. An example of a world model is given in Figure 1.

The corresponding FORML behaviour model specifies how each feature reacts to input events from the environment and generates output actions that change the environment. The FORML behaviour model is expressed as a set of modified UML 2.0 state-machine fragments; with one or more fragments describing the behaviour of each feature. Individual fragments are composed together to form one large state machine that describes the behaviour of a SPL. Transitions in a state-machine have labels:

```
transitionName: trigger [guard] / actions
```

For example, in the BDS feature (Figure 2) transition `t3` states that when the Accelerate signal occurs, the AutoSoftCar's acceleration is set to the value returned by the accelerate() function. The behaviour model contains support for transition priorities (the priority on transition `t4` states that if `t4` and `t3` are simultaneously enabled then only `t4` executes) and transition overrides (i.e., a transition can override and pre-empt all of the actions of a simultaneously occurring transition). Priorities and overrides allow the developer to explicitly model cases where one feature is intended to change the behaviour of another feature (what we call *intended interactions*).

2.2 Translation Process

The entire FORML world model is translated into Alloy. Every concept in the world model (e.g., *RoadObject* in Figure 1) is translated into an Alloy type signature. Abstract concepts and inheritance of concepts are also translated into

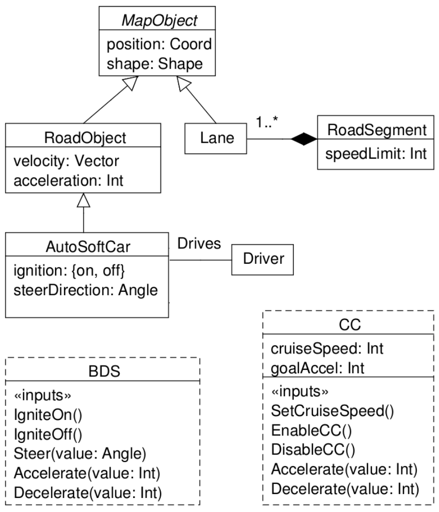

Figure 1: A FORML world model.

their appropriate Alloy counterparts. Multiple inheritance of concepts is not supported by the translation [1].

In the translated model, there is a single type signature that defines the state of the world at a single point in time (referred to as the *world state*). The world-state signature includes sets of all the concepts in the world model, relationships between concepts, and each concept's attributes. An instance of the world-state signature is a single instance of the corresponding FORML world model. For example, the *RoadObject* concept would be translated as:

```
sig RoadObject extends MapObject{}
sig WS { // World State
  RoadObjects: set RoadObject,
  RoadObject_velocity: RoadObjects -> Vector,
  RoadObject_acceleration: RoadObjects -> Int,
  // additional concepts
}
```

Constraints over the world model are translated into two Alloy predicates. The world-state constraints (WSC) predicate contains constraints on a single world state, such as cardinality constraints on relations and constraints related to attribute values. The world-state-transition constraints (WSTC) predicate specifies constraints on how a world state can change; they are constraints over consecutive world states. For example, in Figure 1, a RoadSegment must contain `1..*` Lanes, and if a RoadSegment is removed, then its Lane objects are also removed.

The behaviour model of a feature changes the environment through world-change actions (WCAs), of which there are four kinds: (1) adding an object to the world state, (2) adding a message to the world state, (3) removing an object from the world state, and (4) changing an attribute of an object in the world state. Each WCA is modelled as an Alloy predicate that expresses the post conditions for that action. For example, the WCA predicate for changing the *AutoSoftCar's* acceleration is:

```
pred change_AutoSoftCar_accel
  (postState: WS, car: AutoSoftCar, value: Int) {
  car.(postState.acceleration) = value
}
```

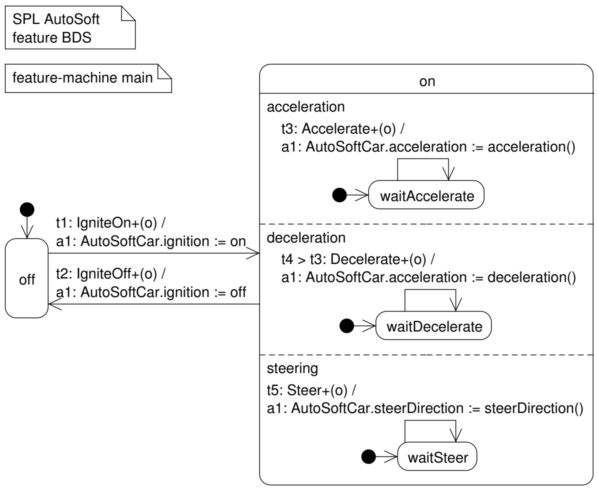
SPL AutoSoft
feature BDS
feature-machine main
on
acceleration
t3: Accelerate+(o) /
a1: AutoSoftCar.acceleration := acceleration()
waitAccelerate
t1: IgniteOn+(o) /
a1: AutoSoftCar.ignition := on
off
t2: IgniteOff+(o) /
a1: AutoSoftCar.ignition := off
deceleration
t4 > t3: Decelerate+(o) /
a1: AutoSoftCar.acceleration := deceleration()
waitDecelerate
steering
t5: Steer+(o) /
a1: AutoSoftCar.steerDirection := steerDirection()
waitSteer

Figure 2: A behaviour model for the Basic Driving Service (BDS) feature.

The *change_AutoSoftCar_accel* predicate will be true even if the *acceleration* attribute is changed to its previous value, but this does not affect the analysis. The generated Alloy model may contain hundreds (or thousands) of WCA predicates, but a majority of these may never be used and the presence of unused predicates does not affect the performance of the interaction analysis.

Every transition is translated into an Alloy predicate that is a conjunction of the predicates of the WCAs that the transition executes. For example, transition `t3` is translated into a predicate that is true when the *change_AutoSoftCar_accel* predicate is true. Note that only a transitions' actions are translated into Alloy, and not their enabling conditions (i.e., triggers, guard conditions). Thus our analysis can be performed on early iterations of models that capture a feature's actions but not the conditions under which the actions are performed.

3. INTERACTION ANALYSIS

The FORML defines an interaction as: two transitions from different features simultaneously attempting to change the world in inconsistent ways. For example, the actions of two concurrently executing transitions may modify the same attribute of the same object or may simultaneously modify and remove an object. Those are obvious interactions. Less obvious interactions include actions that together violate the world-state constraints; or one transition whose actions remove one object, resulting in the removal of related objects, one of which is being modified by a second transition. This definition of interaction can be extended to transitions executing in different concurrent regions within the same feature, or interactions on a single transition.

Following the definition of an interaction in FORML, we define an interaction in Alloy as: given some world state, applying the conjunction of the actions of an individual transition (or pair of transitions) to that state *never* results in a new, valid world state. This definition is used to create the Alloy assertions that detect interactions. *FORML2Alloy* creates such an assertion for every transition and every pair of transitions that can execute concurrently (and is not an *intended interaction*).

```
1  assert transition1_transition2 {
2   distinct_valid_WSs implies
3    all wsPre:WS |
4     all [transition arguments] |
5      some wsPost : {ws:WS - wsPre | WSTC[wsPre, ws]} |
6       [transition invocations]
7  }
```

Figure 3: General structure of the Alloy assertion for detecting interactions. A concrete example is given in Figure 4.

The structure of the generated assertions is given in Figure 3. The assertion states that within the world-state space (line 2), starting from any world-state `wsPre` (line 3), and for all possible parameter values of the transitions being analyzed (line 4), there exists a subsequent world-state `wsPost` that can be arrived at by performing the transitions. Specifically, the transition from `wsPre` to `wsPost` satisfies the world-state-transition constraints (line 5), and `wsPost` satisfies the conditions of all of the WCAs in the two transitions (line 6). The placeholder *[transition arguments]* refers to the WCA arguments of the two transitions. The placeholder *[transition invocations]* refers to the invocations of the transition predicates that were created for the transition(s) being tested.

An example of such an assertion is given in Figure 4. This assertion tests transitions `t3` in BDS (Figure 2) and `t6` in CC. Transition `t6` in CC states:

```
CC{t6}: after(t) /
     a1: AutoSoftCar.acceleration := CC_acceleration(),
     a2: CC.goalAccel := CC_acceleration()
     //goalAccel is an attribute of the CC feature
```

Line 4 of the assertion lists the parameters that are used to instantiate the transitions with concrete objects and values (the parameters are named according to the feature and action identifiers they are testing). Line 6 invokes the transition predicates of `t3` and `t6` with `wsPost` and the transitions' parameters. Both transitions are attempting to change the AutoSoftCar's acceleration attribute. This results in an interaction when both transitions are enabled at the same time, so the assertion is false. The visualization of the counter-example is given in Section 4.

Assertions of the type given in Figure 3 are tested using the `check` command in Alloy. The scope of the `check` command is the exact number of valid world-state instances, based on the scopes of all the concepts in the world model. This is due to the use of bounded quantification [5] in Alloy. Universal and existential quantifiers do not explore the entire set of world states, only the subset that was specified by the scope. If the scope of the world states is set lower than the number of possible world states, then the existential quantifier on line 4 of Figure 3 may fail to find a future world state (`wsPost`) where the conjunction of the transitions' actions hold. This will cause the assertion to be false even though an interaction did not occur. Likewise, if the scope of the world states is set higher than the number

```
1  assert BDS_t3_AND_CC_t6 {
2   distinct_valid_WSs implies
3    all wsPre:WS |
4     all cc_a2_v1:Int,cc_a2_o1:CC,cc_a1_v1:Int,cc_a1_o1:AutoSoftCar,bds_a1_v1:Int,bds_a1_o1:AutoSoftCar |
5      some wsPost : {ws:WS - wsPre | WSTC[wsPre, ws]} |
6       CC_t6[wsPost,cc_a2_v1,cc_a2_o1,cc_a1_v1,cc_a1_o1] and BDS_t3[wsPost,bds_a1_v1,bds_a1_o1]
7  }
```

Figure 4: The assertion that tests for an interaction between the BDS{t3} and CC{t6} transitions.

of possible world states, then there are not enough distinct world-state instances to satisfy the scope of the assertion on line 3, causing the assertion to be incomplete. Therefore, the scope of the assertion must be the exact number of world-state instances for the analysis to be complete.

3.1 Scalability of Analysis

A traditional Alloy analysis of our model suffers from scalability issues because Alloy regenerates the set of world-state instances with every assertion that is checked. However, the set of valid world states is constant with respect to the scopes of the concepts in the world model. To take advantage of this, we used a version of Alloy that has support for manually specifying the instances of a model [14]. Using this approach, the entire set of world-state instances needs to be computed only once, and then can be input to the Alloy analyzer and used in the check of every assertion. This speeds up the analysis significantly.

Specifying the instances of an Alloy model is a fairly simple process. In the Alloy visualizer, when viewing an instance of a model, there is an option to view a textual version of that model. We have created a second translator that automatically converts from this Alloy-instance syntax into the input language that is used to specify model instances for the version of Alloy we are using. To automate the creation of a model instance, the user (1) runs the Alloy analyzer (`run distinct_valid_WSs`) to generate all distinct and valid world states, (2) uses the textual output from the Alloy visualizer as input to this second translator, (3) copies the translator's output into an instance block (see [14]) in the Alloy model, and (4) checks the assertions using the new instance block as the assertion scope.

A second scalability enhancement that we use are reduced integer ranges. By default Alloy uses 16 integers to create model instances. It is possible to reduce that number because the values of objects' attributes are not important to our analysis – all that matters is that each attribute has a single value in a world state. The number of integers needed is only two because an interaction occurs when two or more unique values are applied to the same attribute. This enhancement reduces the number of possible model instances by a factor of 8.

4. VISUALIZING INTERACTIONS

When an assertion is false, the Alloy visualizer displays a counter-example that shows the world model just as the interaction is about to occur. The objects that are being changed are labelled by their variable names as they appear in the assertion. This makes it easy to spot the objects that are being changed despite the large number of objects and relations that may be present. By examining the changed objects, along with the knowledge of the assertions, it is possible to determine the cause of the interaction. An interaction can be resolved in several possible ways: changing a transition's actions, giving one transition priority over another, specifying that a pair of features cannot be included in the same SPL product, and so on. However, resolution of interactions is outside the scope of this paper.

Figure 5 shows the counter-example that is produced when the Alloy analyzer determines that the assertion in Figure 4 is false. To make the visualized model easier to read, we created a custom theme that omits information (e.g., objects, labels) not required to show the interaction.

The four objects that are parameters of the transitions' actions are marked by their parameter names in the assertion (line 4 of Figure 4). Labels `cc_a2_o1` on object `CC` and `cc_a2_v1` on object `0` indicate that the *goalAccel* attribute of `CC` is being set to `0` by transition t6 (t6's definition was given in Section 3). Labels `bds_a1_o1` on object `AutoSoftCar` and `bds_a1_v1` on object `-1` indicate that the acceleration attribute of `AutoSoftCar` is being set to `-1` by transition t3 in the BDS behaviour model (Figure 2). Lastly, labels `cc_a1_o1` on object `AutoSoftCar` and `cc_a1_v1` on object `0` indicate that the *acceleration* attribute of `AutoSoftCar` is being set to `0` by transition t6 in the *CC* behaviour model. The latter two labels identify an interaction when the two transitions change the acceleration attribute of `AutoSoftCar` to different values.

5. CASE STUDY

We have performed a small case study on several automotive features to demonstrate the utility of our approach. This case study was performed on a PC running Linux 3.2.0-27 on a quad-core 3.6 Ghz processor with 8GB of memory.

The case study was performed using the world model in Figure 1 as a base, that was extended with four additional features (for a total of six features to be tested). Using the lowest scope possible for the concepts, the number of distinct and valid world-state instances is 169. The composed behaviour model for all six features contains 25 transitions. The translator generates 25 assertions that check for interactions within singleton transitions, and 161 transitions that check pairs of transitions, for a total of 186 assertions.

Attempting to check a single assertion using the above scope, but not the precomputed set of world-state instances, results in an "Out of Memory exception" after several hours of execution – even when the memory limit is set to unlimited. In contrast, by specifying the set of world-state instances, the average run time for a single assertion was 6.6 seconds. And the total time to check all 186 assertions was only 20 minutes.

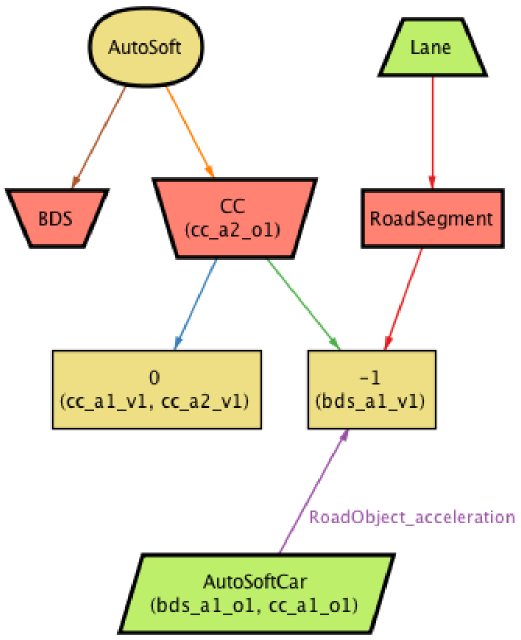

Figure 5: The visualized output of an interaction between the BDS and CC features.

Nine feature interactions were detected by the analysis. All of these were confirmed by manual inspection, and all of them were caused by two transitions attempting to change the same attribute on the same object. Five of the reported interactions were false positives, that were due to not translating guard conditions on the transitions. The remaining four reported interactions were unintended interactions that were not handled during creation of the feature's behaviour models.

6. DISCUSSION

Our approach has several advantages. Firstly, the analysis is complete within the scope of the world-state instances. The analysis checks the actions of every pair of concurrent transitions (and every single transition) in every valid world state. If the actions of a pair of transitions (or within a transition) conflict, an interaction will be reported. Because we express interactions in a generalized manner we are able to detect multiple types of interaction without having to express them explicitly. Another advantage of the form of our assertions is that the analysis is open world – although the assertions check pairs of transitions, other transitions could be executing concurrently. Our results are open to being composed with other transitions. Moreover, our approach does not require frame conditions, which can become quite large and affect the performance of the analyzer. Lastly, although an interaction is an inconsistent world state and thus cannot be visualized, our Alloy assertions fail if there is no future state – leaving the preceding state and the interaction actions as a counter-example that can be visualized.

Our approach can be used to analyze partial models of feature behaviour, where the actions on transitions have been defined, but not the enabling conditions or order of execution. An expected but unproven advantage of this is the ability to perform feature-interaction analysis earlier in the development lifecycle when complete requirements models have not yet been created.

Of course the Alloy analyzer can do more than just detect feature interactions. Custom assertions could be added after the translation to test other properties of the model. It is also possible to use the analyzer to generate valid world states in order to visualize the context in which the features are executing.

On the downside, the analysis is not sound. Our analysis is conservative and checks all pairs of potentially concurrent transitions, without considering if they have mutually satisfiable enabling conditions. Thus, a pair of transitions may be labelled as interacting when in fact they never execute together. This limitation could be reduced by considering the transitions' guard conditions such that the analysis reports an interaction only if the enabling conditions are mutually satisfiable. However, without performing a full reachability analysis, it is not possible to completely resolve these false positives. If soundness is required, then a more heavyweight analysis method must be used – and complete models are required.

Another limitation is that because the scopes of our assertions are the set of all valid world states, the visualized output includes every world-state instance (the relevant world-state instance that shows the counter-example is labelled with the name of the violated assertion). This causes the visualizer to suffer from poor responsiveness when the model is large. We believe the best way to resolve this limitation is through the use of a theme file that displays only the relevant world-state instance that visualizes the interaction. However, this capability is currently not available in the Alloy visualizer.

7. RELATED WORK

Feature-interaction analysis of operational models: There are many approaches to feature-interaction analysis that apply to operational models of feature behaviour, such as process algebra models (e.g., [17]), state-machine models (e.g., [9, 4, 11]), and scenario models (e.g., [6]). Such approaches focus on different manifestations of feature interactions in the models including non-determinism ([17]), deadlock ([11]), unreachable states ([4]), and inconsistent actions ([9]). Our work adapts the inconsistent-actions manifestation to partial FORML models of feature behaviour; that is, models that contain only partial information on which actions can execute concurrently. The remaining manifestations rely on more complete information regarding the execution of actions, and therefore cannot be detected in such partial models. Furthermore, our analysis differs from most existing approaches in that it accounts for intended feature interactions, which are explictly specified in FORML, and does not report them. The approach by Hall [9] also considers intended interactions, but at the granularity of whole features; in contrast, our approach determines whether or not interactions are intended at the granularity of feature transitions.

Feature-interaction analysis using Alloy: The Alloy analyzer has been previously used for feature-interaction analysis. Apel et al. [3] introduce an extension of Alloy, called FeatureAlloy, in which an Alloy model is decomposed into feature modules. In such models, feature interactions manifest as inconsistencies between the correctness properties of different features, which are specified as Alloy assertions in feature modules. In contrast, a feature's correctness properties are not explicitly specified in a FORML model; instead, they are implicit in the postconditions of the feature's actions, which are then translated into Alloy assertions for consistency checking. Similar to our approach, Layouni et al. [13] check for conflicts between pairs of actions of call-control features expressed in the APPEL language, by translating the actions' postconditions into Alloy for consistency checking. However, our approach differs in that we consider conflicts between more than two actions, we account for intended interactions, and we improve the scalability of checking multiple Alloy assertions by storing the state-space under consideration as a partial instance. Scalability is particularly important in our work since we operate on state spaces that are potentially much larger than that considered in [13].

Reasoning about partial models: Famelis et al. [8] have developed a framework for expressing and reasoning over partial models. However, the notion of a partial model in their work differs from ours in that it refers to a model that encodes uncertainty about the presence of its elements. Such a partial model effectively specifies a family of models that are alternative resolutions of the encoded uncertainty. The reasoning over partial models in their work is to determines whether a property holds for all, some, or none of the alternative resolutions. However, they have not explored their reasoning in the context of the feature-interaction problem.

8. CONCLUSION

In this paper, we have presented an automated method for translating FORML models into Alloy and detecting feature interactions in the translated Alloy model. Our approach can accommodate partial FORML models in which features are expressed in terms of actions (without details about when actions might be performed). We have addressed the issue of providing visual feedback when an interaction leads to an inconsistent world state. We have resolved some issues of scalability by feeding the analyzer the set of world states to consider.

We are planning to perform a case study of telephony features [12] to assess the utility of this kind of static analysis. The utility will be measured using two criteria:

1. The ratio of non-obvious versus obvious interactions
2. The ratio of true positives to false positives

If the case study shows that a large number of false positives occur, then we plan to explore whether checking only pairs of transitions whose enabling conditions are mutually satisfiable (and including transitions' guard conditions in the Alloy model) reduces the number of false positives significantly. In the worst case, it may be that interaction detection requires reachability analysis to be effective. However, more complete analyses requires more complete models.

There is current work being done at the University of Waterloo on incrementally constructing the total set of instances of an Alloy model (e.g., from the model's signature). Most importantly, this work will automatically determine the exact scope of the set of world states. Additionally, this will generate the state space within Alloy, removing the need to use the translator described in Section 3.1 to compute and save the set of world states. We plan to wait until this work is complete before we begin the case study.

9. REFERENCES

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software System Models*, 9(1), 2010.

[2] S. Apel and C. Kastner. An overview of feature-oriented software development. *Journal of Object Technology*, 2009.

[3] S. Apel, W. Scholz, C. Lengauer, and C. Kastner. Detecting dependences and interactions in feature-oriented design. In *ISSRE*, 2010.

[4] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications and Software Systems*, 1997.

[5] J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. *Logic Programming and Automated Reasoning*, 1993.

[6] J. Blom. Formalisation of requirements with emphasis on feature interaction detection. In *Feature Interactions in Telecommunications and Software Systems*, 1997.

[7] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 1985.

[8] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, 2012.

[9] R. J. Hall. Feature combination and interaction detection via foreground/background models. In *Feature Interactions in Telecommunications and Software Systems*, 1998.

[10] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[11] A. Khoumsi. Detection and resolution of interactions between services of telephone networks. In *Feature Interactions in Telecommunications and Software Systems*, 1997.

[12] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff-Marganiec. Results of the second feature interaction contest. In *FIW*, 2000.

[13] A. F. Layouni, L. Logrippo, and K. J. Turner. Conflict detection in call control using first-order logic model checking. In *ICFI*, 2007.

[14] V. Montaghami and D. Rayside. Extending Alloy with partial instances. In *ABZ*, 2012.

[15] OMG. Documents associated with the UML version 2.4.1. Technical report, 2011.

[16] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *To appear in Requirements Engineering 2012*, 2012.

[17] M. Thomas. Modelling and analysing user views of telecommunications services. In *Feature Interactions in Telecommunications and Software Systems*, 1997.