Modelling Feature Interactions in the Automotive Domain

Alma L. Juarez-Dominguez
School of Computer Science
University of Waterloo
Waterloo, Canada
aljuarez@cs.uwaterloo.ca

Nancy A. Day
School of Computer Science
University of Waterloo
Waterloo, Canada
nday@cs.uwaterloo.ca

Jeffrey J. Joyce
Critical Systems Labs Inc.
Vancouver, Canada
jeff.joyce@cslabs.com

ABSTRACT

We propose to use model checking to detect feature interactions in a set of features under design for an automotive embedded system. In this paper, we present (1) the characteristics of the feature interaction problem in the automotive domain that make model checking an appropriate detection technique; (2) our proposal for a general, systematic definition of feature interactions for this domain based on the set of actuators in the vehicle influenced by the features; and (3) our solutions to two modelling issues that arise when creating a description in SMV of the behaviour of an integrated set of automotive features designed in MATLAB's STATEFLOW.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Program Verification

General Terms

Design, Verification

Keywords

Feature interaction, automotive systems, model checking, composition

1. INTRODUCTION

Automotive embedded systems consist of software, and electrical and mechanical processes monitored by the software. In the automotive domain, a feature is a bundle of system functionality recognized by the driver; it is normally implemented in software, and provides advanced functionality to the vehicle. For instance, an automotive feature is Cruise Control (CC). These features have a degree of control over the mechanical components that operate the dynamics of the vehicle. Examples of these mechanical components are brakes, throttle, and steering.

MiSE'08, May 10–11, 2008, Leipzig, Germany.

Ideally, features should be able to be integrated into or removed from an existing embedded system without problems. However, when features that have been developed and tested in isolation are integrated into a system, the combination of features can cause unexpected and undesired system behaviour. In automotive systems, these feature interaction problems arise from the activation of two or more features sending requests to the mechanical processes that cause contradictory physical forces, possibly at distinct times. An example is having simultaneous requests to apply the brakes and the throttle. A feature interaction can cause problems ranging from driver discontent to a fatal accident. Traditionally, in automotive systems, reliability and correctness checks are performed using extensive testing and techniques such as probabilistic reliability modelling [20] and Failure Modes and Effects Analysis (FMEA) [12]. These techniques focus on individual feature failures and do not help identify feature interactions during integration. The feature interaction problem becomes more prevalent with the increased complexity in systems. Given that features often evolve as a product line, feature interaction is an ongoing problem.

After identifying the unique characteristics of the automotive domain (described in Section 2), we recognized that model checking is an appropriate technique to detect feature interactions between automotive feature designs. We use the SMV model checker [14] because of its powerful features and general-purpose, flexible input language. MATLAB's STATEFLOW language [6] is used extensively for designing embedded components in the automotive and avionics industries, so we are creating a translator from STATEFLOW to SMV [8]. This paper presents three contributions:

1. A discussion of the characteristics of the feature interaction problem in the automotive domain contrasted with the well-studied version of the problem in telecommunications. (Section 2)
2. A proposal for a general, systematic definition of feature interactions for this domain that is independent of the features. Our definition is based on the actuators in the mechanical processes controlled by the features, and results in a set of temporal logic properties. (Section 3)
3. Our solutions for how to model in SMV: (a) STATEFLOW's AND-state composition (whose meaning is distinct from AND-states in Statecharts[7]) used within feature design models, and (b) the concurrent behaviour of an integrated set of features. (Section 4).

We are currently working on case studies with our own features [8] and features designed by domain experts to validate our approach experimentally.

2. CHARACTERISTICS OF AUTOMOTIVE EMBEDDED SYSTEMS

An automotive embedded system is composed of a "cyber" part, which is software components running on digital hardware that have a degree of control over the mechanical processes, and a "physical" part, which is the mechanical processes.[1] The cyber and physical components of an automotive embedded system are illustrated in Figure 1, where the main mechanical components that operate the dynamics of the vehicle are throttle, braking and steering. The cyber components receive information from the mechanical processes through sensors, and send requests, which can take values as parameters, to be executed by the mechanical processes through actuators. In the automotive domain, a *feature* is a service recognized by the driver and it is implemented in software, for example, Collision Warning (CW).

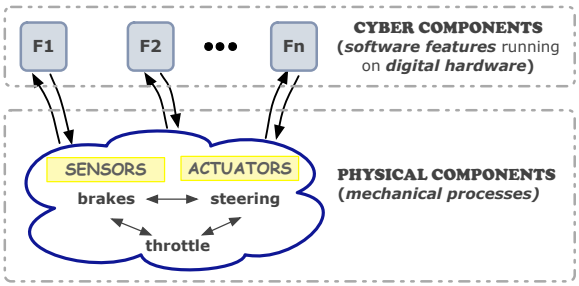

Figure 1: Components of an Automotive System

In our models, we are concerned only with the features. The environment of our models consists of the mechanical processes, which perform actions that create physical forces. Inputs from this environment are normally the discrete approximations of the values read from the sensors (*e.g.,* speed value at time t), and outputs are action requests to the actuators (*e.g.,* set throttle to 10).

In automotive embedded systems, features usually run simultaneously and independently, possibly on different hardware, with no direct communication between features. Indirect communication occurs through the environment of the mechanical processes. Features and their environment form a closed loop: the output of the features change the environment, which is read through the sensors and becomes input to the features at a later time. For these systems, the single input assumption is not valid. Feature interactions arise from the activation of two or more features whose output requests to the actuators cause contradictory physical forces on the mechanical components, potentially at distinct times.

The feature interaction problem has been studied extensively in the last couple of decades in telecommunication systems, but it is not yet completely solved [18]. The problem has been found more recently in other domains, wherever software components are required to work together, such as web services, networked appliances, and embedded systems. However, the characteristics and challenges of the problem are slightly different between the telecommunication domain and the automotive domain.

In a telecommunication system, the feature interactions appear in the cyber part (*e.g.,* call blocking compromised by call forwarding when the later forwards a call to a number in the subscriber's blocking list). In automotive systems, the interactions may be in the physical part, creating contradictory physical forces in the environment (*e.g.,* actuators for brakes and throttle conflict in their influence on speed in the environment).

In an automotive system, the set of features is fixed at retail or release time making an off-line solution based on analysis of the designs reasonable. A telecommunication system is more likely to need an on-line feature detection and resolution solution because of the incremental addition of features by a user.

In a telecommunication system, multiple copies of the same feature may be invoked dynamically by different users during a call (*e.g.,* multiple use of call waiting). Automotive systems have the advantage that only one copy of a feature can be active in a vehicle, making it possible to analyze these systems as if the features are invoked statically.

Finally, in an automotive system, the set of actuators and their range of possible values are known. In contrast, in a telecommunication system we do not know the values that the route can take since subscription information can change dynamically and features can be customized by the users.

In summary, the feature interaction problem in automotive systems is bounded as compared to telecommunication systems, which makes off-line symbolic model checking of feature designs a promising detection technique. However, the set of potential feature interactions cannot be described solely in terms of explicit feature behaviour; they may depend on environmental behaviour between forces unrelated in name.

3. DEFINITION OF FEATURE INTERACTIONS

While classifications of feature interactions in telecommunications exist (*e.g.,* [3, 10]), these classifications do not systematically lead to a set of properties to detect feature interactions. We believe that the unique characteristics of the automotive domain make it possible to create a general definition of feature interactions independent from the set of features. Our proposed definition provides a systematic method for using model checking to detect interactions.

Across all domains, a feature interaction is when a feature in an environment of other features does not perform as it does when the feature operates in isolation. An interaction arises when the features have conflicting effects on a system and its environment. While the behaviour of both features may be correct according to their intended behaviours, their interaction is undesired. A prerequisite for a feature interaction is that the two features both influence (*i.e.,* modify) the same element of the system or its environment.

Our goal is to create a general definition of the potential feature interactions in automotive systems by considering the elements influenced by multiple features. Figure 2 illustrates that in this domain an element with multiple feature influences might be directly modified by the system and therefore referred to by both features using the same name (*e.g.,* throttle); or it might exist outside the embedded system as a conflict manifested in the environment caused by outputs of the system (*e.g.,* speed in the environment affected by brakes and throttle). Simply searching the data flow for multiple feature influences would be too conservative

[1] In this section, we follow the terminology of cyber-physical systems (CPS), in which embedded computers and networks monitor and control physical processes [11].

		Immediate	*Temporal*
Same Actuator	***Boolean***	$\Diamond(X_1 \neq X_2)$	$\Diamond((X_1 \neq X_{\texttt{last\_set}}) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}X}) < \texttt{time\_threshold}))$
	Degrees	$\Diamond(\mid X_1 - X_2 \mid > \texttt{value\_threshold})$	$\Diamond((\mid X_1 - X_{\texttt{last\_set}} \mid > \texttt{value\_threshold}) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}X}) < \texttt{time\_threshold}))$
Conflicting Actuators	***Boolean***	$\Diamond(X \neq Y)$	$\Diamond((X \neq Y_{\texttt{last\_set}}) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}Y}) < \texttt{time\_threshold}))$
	Degrees	$\Diamond((X > \texttt{value\_threshold}_X) \wedge (Y > \texttt{value\_threshold}_Y))$	$\Diamond((X > \texttt{value\_threshold}_X) \wedge (Y_{\texttt{last\_set}} > \texttt{value\_threshold}_Y) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}Y}) < \texttt{time\_threshold}))$

Table 1: Definition of Feature Interactions

approach alone because the two features might never influence the element at the same time. In addition, it would not uncover conflicts that appear outside the system of features. Therefore, we use the elements of multiple feature influences to construct temporal logic properties that check the operational behaviour of the feature to detect whether a feature interaction can occur.

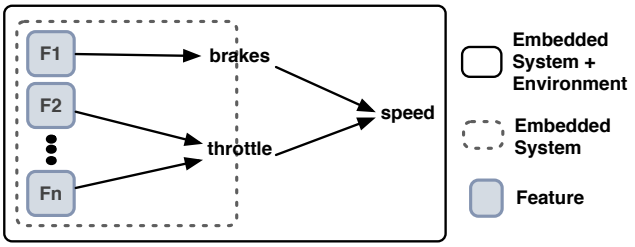

Figure 2: Example of Multiple Feature Influences

In automotive systems, the elements of multiple feature influence are either (1) the actuators in the mechanical processes that receive requests from the features, or (2) elements of the environment influenced by the actuators' actions. The set of actuators is a relatively small, stable set known to all features designers, and the elements of multiple actuator influence in the environment (*e.g.,* speed) are usually well-known. Our definition of feature interactions takes as input (a) the names of the actuators and (b) sets of actuators' names that all influence the same element in the environment. This information can be easily determined by domain experts and is unlikely to change across projects.

Table 1 presents our definition of the set of feature interactions in automotive embedded systems. Our definition contains two main parts: (1) *Same Actuator* for direct actuator conflicts, and (2) *Conflicting Actuators* for actuators that cause feature interactions in the environment. We differentiate between (a) Boolean-valued actuators (*e.g.,* `brakes` = `true` means "apply brakes"), and (b) degree-valued actuators where the degree of force requested is relevant (*e.g.,* `throttle` = 10). In the table, we use X and Y to represent actuator values for distinct actuators that are making requests. A feature interaction may be *Immediate*, in that the conflict is caused by two features making requests in the same step, or *Temporal*, in that the conflict is caused by two requests happening within a certain time threshold of each other. In the table, we describe the interaction as a formula in linear temporal logic (LTL) [13]. All the properties are eventually ($\Diamond$) properties to detect a feature interaction. If the property fails then a feature interaction has not been detected. We abbreviate the phrase "feature interaction detection property" to FIDP.

For each actuator, X, we create FIDPs for the *Immediate* and *Temporal* cases of the *Same Actuator* row in Table 1. An *Immediate* interaction is simply a race condition on an actuator. The goal of these FIDPs is to detect situations where two features request sufficiently different values for an actuator that it is considered an interaction, *e.g.,* one feature requests brakes and the other requests no brakes. To avoid logical inconsistencies during the integration of features (*e.g.,* $\texttt{brakes} \wedge \neg\texttt{brakes}$), in our models each feature has its own name for each actuator. The *Immediate Same Actuator* set of FIDPs includes a FIDP for every pair of features that share an actuator. For Boolean actuators, the FIDP detects when a shared actuator has been set to different values by two features, *e.g.,* $\Diamond(\texttt{brakes\_1} \neq \texttt{brakes\_2})$. For degree-valued actuators, we detect if they have sufficiently different values based on a value threshold, *e.g.,* $\mid \texttt{throttle\_1} - \texttt{throttle\_2} \mid > 10$. The value thresholds would be determined by a domain expert and could be zero to detect if the requests to the actuators are different.

Because the actuators are requests to the mechanical processes, such as "start braking", the effects of these requests are unlikely to be instantaneous, so a feature interaction can occur between requests of the features at distinct times. For example, a feature interaction would occur if a feature requests full throttle force at time t, and while the throttle is still increasing, another feature requests a small amount of throttle at a time $t+1$. We call this case a *Temporal* interaction. A domain expert determines the threshold of time within which contradictory requests to an actuator constitute a feature interaction. To describe a temporal feature interaction in LTL, in our integrated set of feature models we create and maintain two history variables: one to capture the last time an actuator was assigned a value (*e.g.,* $t_{\texttt{last\_brakes}}$), and one to hold the last value that was assigned (*e.g.,* $\texttt{brakes}_{\texttt{last\_set}}$). For degree-valued actuators, the FIDP detects whether the difference between the last and current values exceeds the value threshold (*e.g.,* $(\mid \texttt{throttle\_1} - \texttt{throttle}_{\texttt{last\_set}} \mid > \texttt{value\_threshold})$), and whether sufficient time has passed for the previous output to take effect, *e.g.,* $\mid t_{\texttt{now}} - t_{\texttt{last\_throttle}} \mid > \texttt{time\_threshold}$. It might be the case that both immediate and temporal feature interactions exist with different value thresholds. In model checking, we plan to explore optimizations to handle time, *e.g.,* only storing the differences in time, rather than absolute times.

The *Conflicting Actuators* row of Table 1 creates FIDPs for pairs of different actuators (X and Y) that both influence an element of the environment. Because each feature has its own name for its actuators, for each actuator, we create a macro to indicate when any copy of the actuator is receiving a request. The *Temporal Boolean* case includes one FIDP for Y having been requested previously and X being requested now with a different value (shown in the table) and one FIDP for the symmetric case. For degree-valued actuators, in the *Immediate* case, we detect if the requests to the

conflicting actuators are both greater than each actuator's value threshold (defined by domain experts). For degree-valued actuators, the *Temporal* FIDP detects whether the current value of X exceeds its threshold of conflict and the last value of Y exceeds its threshold of conflict, and that the time between now and when Y was last set exceeds a time threshold. An FIDP for the symmetric case would also be created.

Our proposed definition of feature interactions based on the actuators and environmental elements of multiple feature influence is systematic and does not rely on examining the behaviour of the individual features. Because the set of actuators is small (10-20), we believe it is reasonable to ask a domain expert to provide the elements of shared influence as input to the process of creating the FIDPs. The FIDPs for a set of features can be created automatically. However, our definition of potential feature interactions is only complete with respect to the expert knowledge provided.

In contrast, a definition similar to the one proposed here would not work in telecommunications. Almost all feature interactions are conflicts on the creation of the one actuator the route, which has an unbounded set of possible values.

4. MODELLING FEATURES IN SMV

We chose the symbolic model checker SMV to check for feature interactions because of its powerful features and simple, but flexible input language. In this section, we describe two interesting modelling aspects of the translation from feature designs in Stateflow into SMV. The SMV model must correctly describe the semantics of individual feature designs, as well as the semantics of features operating concurrently in a vehicle. Both of these aspects influence the way we map Stateflow "steps" to SMV "steps". The SMV step must match the step of the smallest granularity in Stateflow, but to match exactly the semantics of concurrent features and individual feature behaviour, we have to mimic the creation of larger steps in SMV.

A Stateflow model is a hierarchical state machine with transitions between states. The syntax of Stateflow is similar to that of Statecharts [7]. However, AND-states in Stateflow, which are frequently used within features by designers, have a very different meaning than AND-states in Statecharts. In Stateflow, individual features run in a single thread and there cannot be any concurrency between different parts of the feature. AND-state execution is sequential: each component of an AND-state is numbered and the sibling components execute one at a time in order. All siblings react to the same set of inputs. Figure 3 shows an example of the execution of an AND-state labelled **C** consisting of two components **A** (numbered 1) and **B** (numbered 2). (For the moment, please ignore state **D**, which will be used to explain concurrent execution of features later.) **A** is also an AND-state, consisting of two subcomponents, **A1** and **A2**. In this paper, we call Stateflow AND-states *ordered composition* to distinguish them from the common meaning of AND-state in Statecharts.

To model Stateflow in SMV, we first make an SMV step correspond to taking one transition. As is common in translators to SMV, control states are variables of enumerated type, and hierarchy is modelled using nested switch statements. Figure 4 shows the SMV module for the feature model **C** of Figure 3. One step in SMV chooses a case in the switch statement based on the enabling conditions on

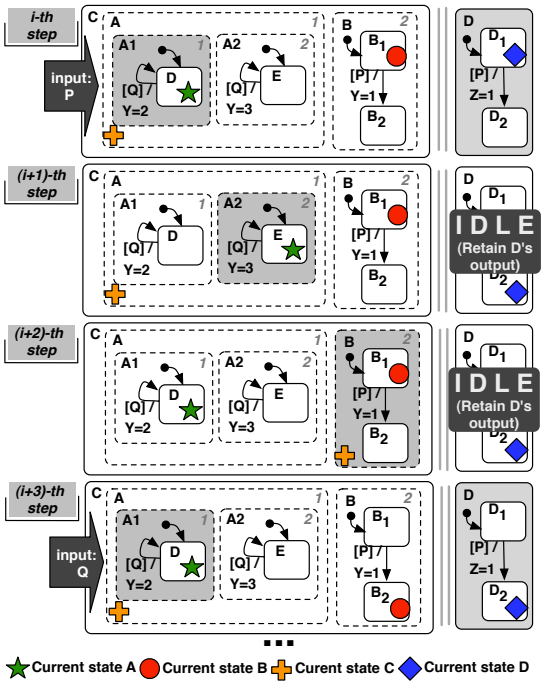

Figure 3: Illustration of Stateflow Feature Models

the transition and the current control states.[2] The control state is changed from the source state of the transition to its destination state, and changes are made that correspond to the actions on the transition.

To model the sequential behaviour of the components of an ordered composition operator in Stateflow appropriately in a sequence of SMV steps, each transition changes its parent state variable to the next component in the ordered composition (*e.g.,* line 27 in Figure 4). The last component transfers the token for execution back to the first component of the ordered execution (*e.g.,* lines 46 and 63). In addition, we semi-control the input variables. Inputs must stay the same for the execution of all components of the ordered composition. We allow the inputs to change only in the SMV step where the token is *about to be passed* to the first component of the ordered composition, *i.e.,* when all the siblings that are part of a feature's ordered composition are done executing. This approach does not introduce any extra steps in the computation. We model this behaviour in SMV using a macro on line 68–69, called `doneC`. The macro hierarchically considers whether the feature is going to transition to the component of the ordered composition with execution order 1. Because it is a macro, the number of variables in the SMV model does not increase. The if-then-else statement on lines 82–87 forces inputs to stay the same while components of an ordered composition are executing and allows inputs to change non-deterministically when the token is passed to the component with execution order 1. [3] The purpose of the macro `sys_ready` will be described next.

In our experience, feature designers for automotive systems work on individual features in Stateflow and do not

[2]This example is for illustration and could be optimized because **sA1** and **sA2** each have only one possible value.

[3]This aspect of our approach is similar to that used by Chan *et al.* [5] to model the semantics of RSML, except that the definition of when the inputs are allowed to change is quite different.

```
1  MODULE C (P,Q,X,Y)
2  {
3   sC : {A,B};    /*state C*/
4   sA : {A1, A2}; /*state A*/
5   sB : {B1,B2};  /*state B*/
6   sA1 : {D};    /*state A1*/
7   sA2 : {E};    /*state A2*/
8
9   init(sC) := A;
10  init(sA) := A1;
11  init(sB) := B1;
12  init(sA1) := D;
13  init(sA2) := E;
14
15  switch(sC) {
16   A:
17    switch(sA) {
18      A1:
19       if (sys_ready)
20       {
21        switch(sA1) {
22         D:
23          if( Q ) {
24           next(Y) := 2;
25           next(X) := X;
26           next(sA1) := D;
27           next(sA) := A2;
28           next(sC) := A; }
29          else {
30           next(X) := X;
31           next(Y) := Y;
32           next(sA1) := D;
33           next(sA) := A2;
34           next(sC) := A; }}
35       }
36       else {
37        next(X) := X;
38        next(Y) := Y;
39        next(sA1) := sA1;
40        next(sA) := sA;
41        next(sC) := sC; }
42      A2:
43        switch(sA2) {
44         E:
45          if( Q ) {
46           next(sA) := A1;
47           next(sC) := B; }
48           ...
49    /* Rest similar to A1 */
50
51   B:
52    switch(sB) {
53     B1:
54      if( P ) {
55       next(X) := X;
56       next(Y) := 1;
57       next(sB) := B2;
58       next(sC) := A; }
59      else {
60       next(X) := X;
61       next(Y) := Y;
62       next(sB) := B1;
63       next(sC) := A; } } }
64     B2: ...
65
66  DEFINE readyC :=
67   sC=A & sA=A11;
68  DEFINE doneC :=
69   next(sC)=A & next(sA)=A1;
70 }
71
72 MODULE main() {
73  P, Q : boolean;
74  X, Y : 0..10;
75  init(X):=0; init(Y):=0;
76  init(P):={true,false};
77  init(Q):={true,false};
78
79  DEFINE sys_ready:=readyC;
80  DEFINE sys_done:=doneC;
81
82  if (sys_done)
83  {next(P) := {true,false};
84   next(Q) := {true,false};}
85  else
86  {next(P) := P;
87   next(Q) := Q;}
88
89  C (P,Q,X,Y);
90 }
```

Figure 4: SMV Model of Feature C from Figure 3

use STATEFLOW to simulate their combined behaviour, unless they intend to create a new feature with the joint behaviours. There is no operator in STATEFLOW that corresponds to the concurrent behaviour of features integrated in a vehicle. In the integration, all features receive the same set of inputs simultaneously, execute concurrently without communicating with each other, and then produce outputs that are commands to the shared set of actuators. Even though all features receive their inputs synchronously, each feature must preserve its individual STATEFLOW semantics, such as sequentiality of execution when a feature contains ordered composition. Whenever a feature model with several components in an ordered composition is combined with a feature model with fewer or no ordered composition operators, the latter feature model must idle maintaining the values of its outputs while the former feature model finishes its sequential execution.

For example, in Figure 3, two features, **C** and **D**, execute concurrently. Feature **C** is an ordered composition of its two children **A** and **B** with execution order 1 and 2 respectively. Feature model **A** is also an ordered composition of its children, **A1** and **A2** with execution order 1 and 2 respectively. If at step i input P is received, **C** will take three steps to process the input since it has to check for transitions that can be taken in all of its children. **D** will only take one step to process the same input. **D** must keep its outputs constant while waiting for **C** to complete the steps of its ordered compositions. **D** can only begin executing again when **C** is back in state **A1** of **A**. At step $i + 3$, both feature models can process another input.

Figure 5 shows the SMV model for the STATEFLOW model of Figure 3. We define a macro **sys_ready** (line 42–43) to be true when all features are ready to begin with new inputs. The macro is the conjunction of a **ready** macro for each feature. The **ready** macro for each feature considers the complete hierarchy – it checks when all ordered compositions within a feature are in their components with execution order 1 (line 66–67 of Figure 4). The **sys_ready** macro is a condition on taking a transition in the first component of the innermost ordered composition operator in each feature (line 19 of Figure 4) or on taking any transition in a feature with no ordered composition (line 11 of Figure 5). It makes a feature idle when another feature is still processing an input following its ordered composition. If **sys_ready** is not true, the value of a feature's actuators is held constant.

Feature interactions should be checked when the **sys_ready** macro is true, which is the point when all features have produced their outputs for a set of inputs.

```
1  MODULE C (P,Q,X,Y)
2  {
3   /* Refer to Figure 4 */
4  }
5
6  MODULE D (P,Q,Z)
7  {
8   sD : {D1,D2}; /*state D*/
9   init(sD) := D1;
10
11  if (sys_ready)
12  {
13   switch(sD) {
14    D1:
15     if( P ) {
16      next(Z) := 1;
17      next(sD) := D2; }
18     else {
19      next(Z) := Z;
20      next(sD) := D1; }
21    D2: ... }
22  }
23  else
24  { next(Z) := Z;
25    next(sD) := sD;
26  }
27
28  DEFINE readyD := true;
29  DEFINE doneD := true;
30 }
31
32 MODULE main() {
33   P, Q : boolean;
34   X, Y, Z : 0..10;
35
36  init(X) := 0;
37  init(Y) := 0;
38  init(Z) := 0;
39  init(P) := {true,false};
40  init(Q) := {true,false};
41
42  DEFINE sys_ready :=
43   readyC & readyD;
44  DEFINE sys_done :=
45   doneC & doneD;;
46
47  if (sys_done)
48  {next(P):={true,false};
49   next(Q):={true,false};}
50  else
51  {next(P):=P;
52   next(Q):=Q;}
53
54  C (P,Q,X,Y);
55  D (P,Q,Z);
56 }
```

Figure 5: SMV Model from Figure 3

5. RELATED WORK

Wilson, Magill, and Kolberg consider feature interactions in embedded systems by investigating home automation systems [21, 22]. They use an on-line feature interaction manager (FIM), which determines if an interaction occurs when it detects shared access to environmental variables with different attributes. The FIM may have to be redesigned if new devices are added that influence new variables.

Metzger introduces a systematic approach for the automatic detection of feature interactions in embedded control systems, first giving examples on building control systems [16] and then applying the concepts to other kinds of embedded systems [15]. In this work, a feature interaction is defined as an element of multiple feature influence (us-

ing our terminology) on an object diagram. This approach is conservative because it does not consider the operational behaviour of each feature.

Banphawatthanarak and Krogh [2] translate STATEFLOW to SMV by creating an SMV module per OR- and AND-state, plus a module to coordinate the status of AND-states (*i.e.,* 'not-active', 'active-active', or 'active-wait'). The conditions for making transitions to each of these states are passed as parameters to the coordinator AND-state module. Our solution is much simpler. In addition, they only support Boolean variables, and do not support transition actions, needed in our features to model actuator request. In [2], each feature is analyzed only in isolation so they did not consider the semantics of the integration of features necessary to detect feature interactions.

There have been several previous efforts to translate STATEFLOW to the input languages of other model checkers. These efforts either (1) did not include a translation for STATEFLOW AND-states (*e.g.,* Camera [4] – STATEFLOW to VHDL, Pingree and Mikk [17] – STATEFLOW to SPIN); or (2) mapped ordered composition to the Statecharts semantics of AND-states (*e.g.,* Kalita and Khargonekar [9] – STATEFLOW to STeP); or (3) analyzed features only in isolation and did not consider the semantics of the integration of features necessary to detect feature interactions (*e.g.,* Scaife *et al.* [19] – STATEFLOW to Lustre). Agrawal *et al.* [1] use a graph rewriting approach to convert a subset of STATEFLOW to Hybrid Automata. Their approach involves flattening to a much more primitive state machine.

6. CONCLUSIONS

Feature interaction in the automotive domain is a relatively new area of study and will remain of interest in the future as vehicles continue to increase in complexity. To the best of our knowledge, we are the first to propose a general, systematic definition of feature interactions in the automotive domain. Our definition is based on the set of actuators used by the features and results in a set of LTL properties to detect feature interactions through model checking. We also discussed two modelling issues in creating SMV models of features designed in STATEFLOW. We are working on case studies with our own features and features designed by domain experts to validate our approach experimentally. During these case studies we will explore methods for handling state space explosion such as slicing and creating a library of domain-specific abstractions.

7. REFERENCES

[1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*. ENTCS, 2004.

[2] C. Banphawatthanarak and B. H. Krogh. Verification of stateflow diagrams using smv: sf2smv 2.0. Technical Report CMU-ECE-2000-020, Carnegie Mellon University, 2000.

[3] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[4] K. Camera. SF2VHD: A stateflow to VHDL translator. Master's thesis, University of California, Berkeley, 2001.

[5] W. Chan, R. J. Anderson, et al. Model checking large software specifications. *IEEE TSE*, 24(7):498–520, 1998.

[6] J. Dabney and T. L. Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Prog*, 8(3):231–274, June 1987.

[8] A. L. Juarez-Dominguez, N. A. Day, and R. T. Fanson. A preliminary report on tool support and methodology for feature interaction detection. Technical Report CS-2007-44, University of Waterloo, 2007.

[9] D. Kalita and P. P. Khargonekar. SF2STeP: A CAD tool for formal verification of timed stateflow diagrams. In *IEEE Int. Sym. on Comp.-Aided Control Sys. Design*, pages 156–162, 2000.

[10] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE TSE*, 24(10):779–796, 1998.

[11] E. A. Lee. Cyber-physical systems - are computing foundations adequate? *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, 2006.

[12] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, first edition, 2001.

[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, first edition, 1992.

[14] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, 1993.

[15] A. Metzger. Feature interactions in embedded control systems. *Computer Networks*, 45(5):625–644, 2004.

[16] A. Metzger and C. Webel. Feature interaction detection in building control systems by means of a formal product model. In *Feature Interactions in Telecom. and Soft. Sys.*, pages 105–121. IOS Press, 2003.

[17] P. J. Pingree and E. Mikk. The hivy tool set. In *CAV*, volume 3114, pages 466–469. Springer, 2004.

[18] S. Reiff-Marganiec and M. Ryan, editors. *Feature Interactions in Telecommunications and Software Systems VIII*. IOS Press, 2005.

[19] N. Scaife, C. Sofronis, P. Caspi, et al. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *ACM Int. Conf. on Embedded software*, pages 259–268. ACM Press, 2004.

[20] M. L. Shooman. *Probabilistic reliability : an engineering approach*. Brooklyn Polytechnic Institute series. McGraw-Hill, first edition, 1968.

[21] M. Wilson and E. H. Magill. An environmental model for service interaction in home networks. In *Postgraduate Research Conf. in Electronics, Photonics, Communications and Software.*, 2003.

[22] M. Wilson, E. H. Magill, and M. Kolberg. An online approach for the service interaction problem in home automation. In *IEEE Consumer Comm. and Networking Conf.*, pages 251– 256, 2005.