A Feature Interaction Resolution Scheme Based on Controlled Phenomena

by

Cecylia Bocovich

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Systems that are assembled from independently developed features suffer from *feature interactions*, in which features affect one another's behaviour in surprising ways. To ensure that a system behaves as intended, developers need to analyze all potential interactions – and many of the identified interactions need to be fixed and their fixes verified. The *feature-interaction problem* states that the number of potential interactions to be considered is exponential in the number of features in a system. Resolution strategies combat the feature-interaction problem by offering general strategies that resolve entire classes of interactions, thereby reducing the work of the developer who is charged with the task of resolving interactions. In this thesis, we focus on resolving interactions due to conflict. We present an approach, language, and implementation based on resolver modules modelled in the situation calculus in which the developer can specify an appropriate resolution for each variable under conflict. We performed a case study involving 24 automotive features, and found that the number of resolutions to be specified was much smaller than the number of possible feature interactions (6 resolutions for 24 features), that what constitutes an appropriate resolution strategy is different for different variables, and that the subset of situation calculus we used was sufficient to construct nontrivial resolution strategies for six distinct output variables.

Acknowledgements

Thank you to my supervisor, Jo Atlee, for the time she spent, not only on supervision and guidance on research, but also on providing me with opportunities and teaching me how to be a graduate student. I would also like to thank my readers, Nancy Day and Krzysztof Czarnecki.

Thank you to my labmates: Pourya Shaker, Sandy Beidu, David Dietrich, Shoham Ben-David, and Amirhossein Vakili for guidance, models, and support.

A huge thanks to all of the friends I made at the University of Waterloo. From the first day of orientation to the present, you guys made my experience here one I will never forget. I will always treasure CS Club 7 (its beginnings and the dozens of people it grew to encompass), the D. R. C. School of Super Friends intramural hockey team (and our fans), and the PMath people.

Finally, thank you to my parents, Mike and Joanne Bocovich, my siblings, Carolyne and Nick, and Nana, for their confidence and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the size and complexity of software systems grows, large-scale systems are measured not only in lines of code, but also in terms of variability. The demand for variety, customization, and portability across multiple platforms leads to increasingly larger and more complicated requirements. The complexity and variability of these systems is addressed by decomposing the system into units of functionality that may be optional. In **feature-oriented software development**, a system's functionality is decomposed into features, where each **feature** is an identifiable unit of functionality or variation [8]. Users view features as system capabilities. For example, telephony users may subscribe to features such as Call Waiting or Caller ID, which extend the basic functionality provided to them through the base call service. An automobile may have Anti-Lock Brakes or Cruise Control features, which enhance driver safety or vehicle performance. Each of these features are optional and may be selected or billed according to the users' preferences. Feature orientation also has the added benefit that features can serve as a shared vocabulary among diverse stakeholders (e.g., marketers, customers, other engineers) in a way that other types of software fragments — such as modules, objects, or components — cannot.

Decomposition of a system into features also allows for incremental software development. Since features encapsulate separate units of functionality, they can be easily added or updated in new releases of the system. Additionally, this decomposition enables the parallel development of features by developers on the same team or by third parties.

Feature-oriented software development also helps to manage variability between instances of a system. Software product lines group families of similar software products (e.g., smartphones, cars). These families are managed and evolved in terms of their features. Each individual product in a product line comprises a basic service and a subset of

optional features [36].

Although features are considered individually, they are often not truly separate concerns and problems arise when developers try to integrate them into a coherent product. Consider, for example, a user in a telephony system who subscribes to Call Waiting (CW) and Call Forwarding on Busy (CFB). Separately, the behaviours of these features are well-defined. In the event a subscriber to CW is currently involved in a call, she will receive notification of any new incoming calls and have the option of putting either the incoming call or the current call on hold. If a subscriber to CFB receives an incoming call in this situation, the call will be rerouted to a predefined user-set forwarding number. When the same user subscribes to both of these features, there is a conflict between the two features. When the subscriber is busy, CW will inform the user of an incoming call, while CFB will attempt to forward the incoming call to another number. In this case, the behaviour of the system is not well-defined. It is unclear whether an incoming call should be forwarded automatically, or whether the user should be notified and given the opportunity to place one of the calls on hold. This is an example of a **feature interaction** — where the presence of multiple features in the system causes the behaviour of individual features to deviate from their respective specifications.

There are many types of feature interactions. Formal definitions for each type of interaction often vary across different kinds of modelling languages used to express feature behaviour. In this thesis, feature behaviour is expressed as the transitions of a state machine and the realization in a world model of world-change actions output by the collection of composed features. We are primarily concerned with conflict interactions. A **conflict interaction** is a type of interaction that occurs when the next world state cannot be computed because the set of world-change actions cannot be executed simultaneously [32]. This definition is seen in many previous works on feature interaction detection [1, 15, 38]. More specifically, a conflict arises when multiple features attempt to assign different values to the same instance of an output variable in a single execution step [24].

The consequences of feature interactions range from unexpected or unpredictable behaviour to potentially dangerous situations. For example, the software controllers for the braking features on the 2010 Toyota Prius interacted badly, reducing drivers' overall ability to brake and leading to multiple crashes and injuries [35]. To be safe, software developers must consider how features interact and must resolve undesired interactions. However, because features are optional in many products, or can be turned on and off dynamically, developers must look for potential interactions in different combinations of features. For each identified interaction, the developer must first determine if it is problematic. If it is problematic, she must devise a **resolution** for the interaction.

A resolution defines appropriate behaviour for the system in the event that a feature interaction occurs. The details of a resolution depend on the domain, the developer's original intention in specifying feature behaviour, and the nature of the interaction. For example, a resolution strategy to resolve the conflict between Call Waiting and Call Forwarding on Busy may be to seek user input: notifying the subscriber if an incoming call arrives when the subscriber is already on the phone and giving her the choice of putting one of the calls on hold or forwarding the new call to a predefined number. Whereas this resolution strategy would not be appropriate for an interaction between cruise-control features in an automotive system. Performing multiple acceleration changes in quick succession could lead to jolting behaviour and interfere with the driver's safety.

Resolving interactions is a lot of work, because appropriate resolutions may vary, even within the same domain. The developer must find all interactions, and fix the problematic ones, among all possible combinations of features; and the number of feature combinations to consider is exponential in the number of possible features in the system. Thus, as the size of the system and the number of features grows, a software team finds that the development of new features is eventually dominated by the **Feature Interaction Problem**: the need to analyze, resolve, and verify interactions [5].

One approach to the Feature Interaction Problem is to employ a default strategy for resolving interactions, thereby reducing the number of interactions that need to be individually addressed by the developers. Default resolution strategies include resolution by feature priority [18, 22, 29], feature precedence [3, 9, 21], negotiating compromises [16], setting interaction thresholds [12], involving the user [13], rolling back conflicting actions [29], disabling feature activation [20], terminating features [29], and terminating the application. However, most of these strategies are coarse grained (e.g., based on the priority or precedence of the features themselves rather than the features' interacting actions); they provide suboptimal win/lose resolutions in which some features' actions are sacrificed in favour of other features' actions; and they often require an upfront total or partial ordering on features [40].

Thesis Statement:

Rather than trying to resolve every feature interaction due to conflict individually or devising one resolution strategy to handle all feature interactions, many aspects of the Feature Interaction Problem can be addressed by focusing on the outputs of the system and specializing default resolution strategies to apply to these outputs. This allows developers to specify appropriate win/win resolutions that do not require a total or partial ordering on features. It is

possible to reduce the work done by the developer to be linear in the number of types[1] of output variables modified by multiple features, rather than exponential in the number of features. It also preserves the advantages of feature-oriented software development; it is agnostic to the number of features in the system and maintains feature obliviousness, allowing for the incremental and parallel development of features.

1.1 Contributions

The contributions of this thesis are as follows:

- We introduce a new approach to resolving features' interactions, in which the resolution strategies are specific to the variables being acted on.
- We present an implementation of the approach, in which feature actions and developer-provided resolution strategies are encoded in the situation calculus [30] and are executed by a GOLOG interpreter [28]. We identify sufficient and necessary conditions on the developer-provided resolutions that ensure that a resolution has desired properties (e.g., is deterministic, is conflict-free, terminates).
- We performed a case study in which we used our approach and implementation to model the actions of 24 automotive features and to specify appropriate resolutions for 6 distinct output variables. The results of the case study demonstrate that different output variables require different resolution strategies. The case study also assessed the expressiveness of situation calculus as a suitable language for encoding feature actions and resolution strategies.

1.2 Organization

This thesis is organized as follows. In the next chapter, we give an overview of feature-oriented requirements modelling and the feature-interaction problem. In Chapter 3, we describe an approach to resolving feature interactions in terms of resolution strategies per output variable, including how to encode feature actions and resolution strategies

[1] In the case of multiple instantiations of the same type of output variable, the developer only needs to specify one resolution to apply to all variable instances.

in situation calculus. In Chapter 4, we identify necessary and sufficient conditions for resolutions to satisfy desired properties (e.g., are deterministic, are conflict-free, terminate), and we present the results of our case study. Chapter 5 summarizes related work, and Chapter 6 concludes the thesis.

Chapter 2

Preliminaries

Throughout this thesis, we use examples from the automotive domain. Each feature extends a Basic Driving Service (BDS), which serves as a base system whose functionality is entirely self-contained [8]. An automotive system comprises BDS and a subset of optional features.

2.1 Feature-Oriented Requirements

We are primarily concerned with the requirements stage of feature-oriented software development. Behavioural requirements for each feature are modelled independently and then composed into a system. There are many ways to model feature behaviour, but we focus on state-machine approaches [17]. The language we describe here is based on the Feature-Oriented Requirements Modelling Language (FORML) [31] as it provides a rich syntax for expressing feature behaviour.

A system's behaviour is expressed in terms of its reactions to changes and conditions in its environment, and its actions on environmental variables. **Monitored variables** such as *car.speed* represent environmental phenomena that are sensed by or act as inputs to the system. Changes to monitored variables prompt reactions in the system behaviour. **Controlled variables** represent environmental phenomena that are controlled or affected by system outputs.

A **world model** is a conceptual model of the system's environment. Figure 2.1 shows a partial world model for an automotive product with the features Cruise Control, Lane-Change Control, Headway Control, and Speed-Limit Control. A world model shows as-

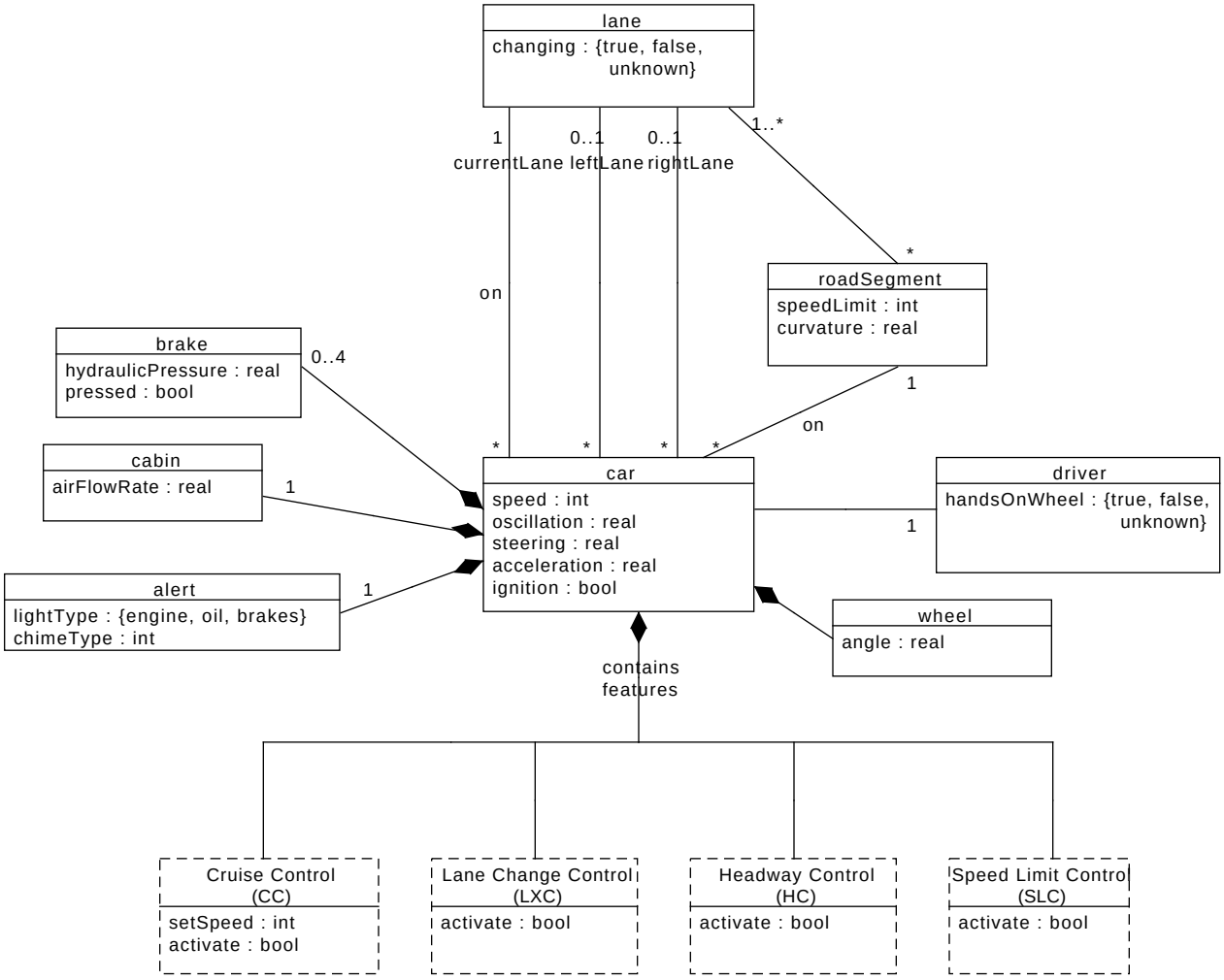


Figure 2.1: Partial world model for a car with several features

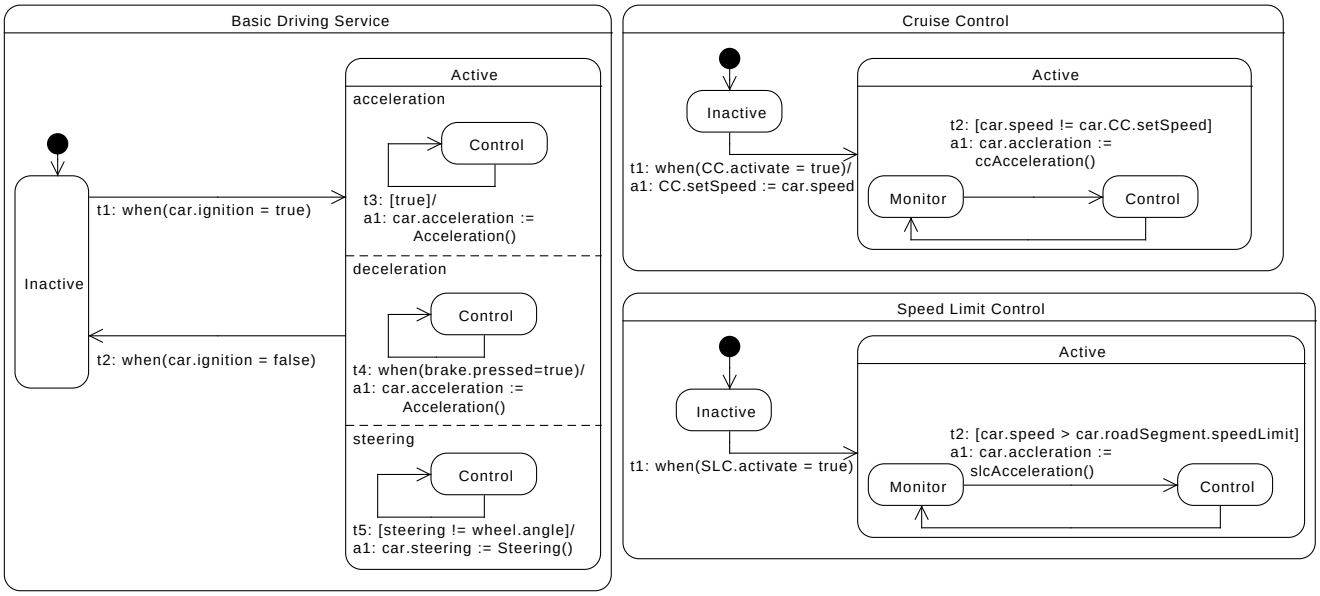


Figure 2.2: Behaviour model of BDS and some automotive features

sociations between concepts in the world. For example, a *car* always has one *driver* and may be one of many other *car* objects on a *roadSegment*. This model also shows concept attributes; for example, a car has the attributes *speed*, *oscillation*, *steering*, and *acceleration*. The attributes of the world model comprise the monitored and controlled variables in the system's environment.

A **world state** is an instantiation of the world model that reflects the current values of all the monitored and controlled variables. A world state contains objects, which are instantiations of concepts in the world model. The objects in a world state are dynamic: they may be added, removed, or modified as part of the behaviour of the system. Dot notation is used to refer to monitored and controlled variables in a world state. For example, the attribute *acceleration* of the object *car* is denoted *car.acceleration*. The value of the variable *car.acceleration* in the world state ws_i is denoted $ws_i :: car.acceleration$.

A feature's behaviour is modelled as a state machine, called a **feature machine**. Figure 2.2 shows feature machines for the Basic Driving Service (BDS) and two optional features, Cruise Control (CC) and Speed-Limit Control (SLC). A state in a feature machine reflects the current state of a feature's execution. States may be hierarchical, containing several sub-states that more finely describe feature behaviour. Superstates may also contain concurrent regions that execute in parallel. For example, the CC feature has Active and

Inactive superstates that reflect different aspects of the feature's behaviour. The Active state of BDS contains three concurrent regions responsible for monitoring and controlling different environmental phenomena. A world state reflects the current execution states of the feature machines for all features in the system, as well as the values of all monitored and controlled variables as described above.

Transitions between states are triggered by events, or changes to monitored variables. These triggering events are represented as expressions over variables in the previous and current world states, ws_p and ws_c, respectively. In Figure 2.2, the transition from the Inactive to Active superstate in the CC feature machine is triggered by the event of the feature CC's activate attribute becoming true:

$$ws_p :: car.CC.activate = false \wedge ws_c :: car.CC.activate = true$$

As shorthand, we use the UML construct $when(c)$ to denote the condition c becoming true. So, the triggering event above is expressed in Figure 2.2 as

$$when(car.CC.activate = true)$$

A transition may also be guarded by an expression over the current values of monitored variables. The transition will be executed only if the guard condition evaluates to true. For example, the transition between the Monitor and Control sub-states in the SLC feature machine is guarded by the condition $car.speed > speedLimit$. This evaluates to true if and only if the current speed of the vehicle is greater than the speed limit of the current segment of road.

In total, a transition between states may be labelled with an identifier id, a triggering event te, a guard condition gc, and one or more actions $a_1, \ldots, a_n$.

$$id : te\ [gc]/a_1 \ldots a_n$$

A transition from state s_1 to s_2 is enabled if the machine is currently in state s_1, the guard condition evaluates to true, and the triggering event occurs. An action corresponds to a prescribed change to controlled variables in the current world state. There are three types of actions, sometimes referred to as **world-change actions**:

- $+o(\text{list}(a = \langle expr \rangle))$ Adds object o to the (next) world state. Attributes of o are initialized with the evaluations of the given expressions.
- $-o$ Removes object o from the (next) world state.

- $o.a := \langle expr \rangle$ Assigns the evaluation of the expression $expr$ to the controlled variable $o.a$.

For example, many automotive features modify vehicle acceleration to maintain driver preferences or respond to safety concerns. This is modelled by assigning appropriate values to the controlled variable $car.acceleration$. Often, the details of calculating of these values are abstracted as uninterpreted functions, such as the action in transition $t3$ of the BDS feature machine:

$$a1: \ car.acceleration := CCAcceleration()$$

An execution step of a feature machine consists of the execution of at most one transition for every concurrent region of the feature's current execution states. An execution step of a system is the concurrent execution of each feature's execution step. In each execution step, the features' executing transitions and actions result in a (new) current world state, determined by the new execution states in each feature machine, the effects of the transitions' actions on the controlled variables, and changes in the values of noncontrolled variables made by the environment [32].

Figures 2.3 and 2.4 show the result of performing world change actions on a world state. Each world state in the figures is an instantiation of the world model shown in Figure 2.1. Transitions between these world states are the result of actions performed by the Lane Change Control feature (LXC).

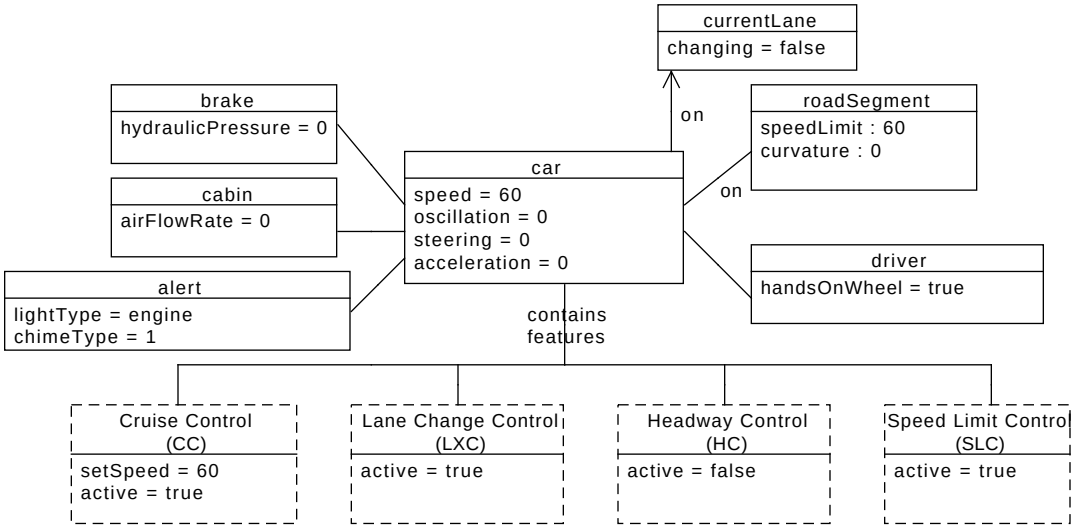
The first world state, ws_0, shows the current controlled objects in the world, along with the values of their attributes. The transition from ws_0 to ws_1 results from the addition of the object $leftLane$, shown in green. This change is the result of the following world change action, performed by a lane detection feature not described in this document:

$$+ car.leftLane(changing = false)$$

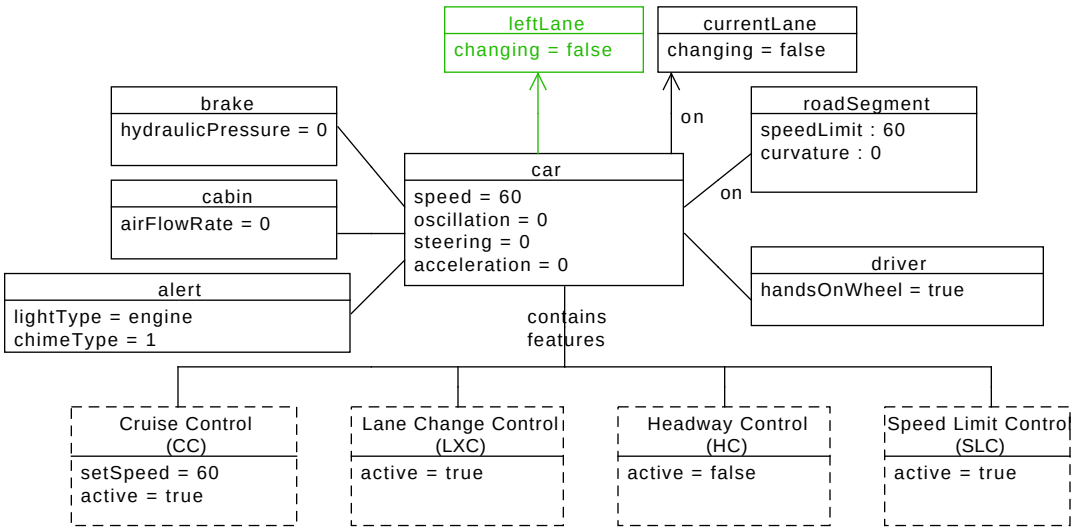
The next world state, $ws2$, results from a driver decision to change into the left lane. LXC sets the values of the variables $car.currentLane.changing$ and $car.leftLane.changing$ to $true$ with the world-change actions:

$$car.currentLane.changing := true$$
$$car.leftLane.changing := true$$

which represent the fact that the current lane of the vehicle will soon become the right lane and the left lane will become the current lane. This change is highlighted in yellow in the figure.

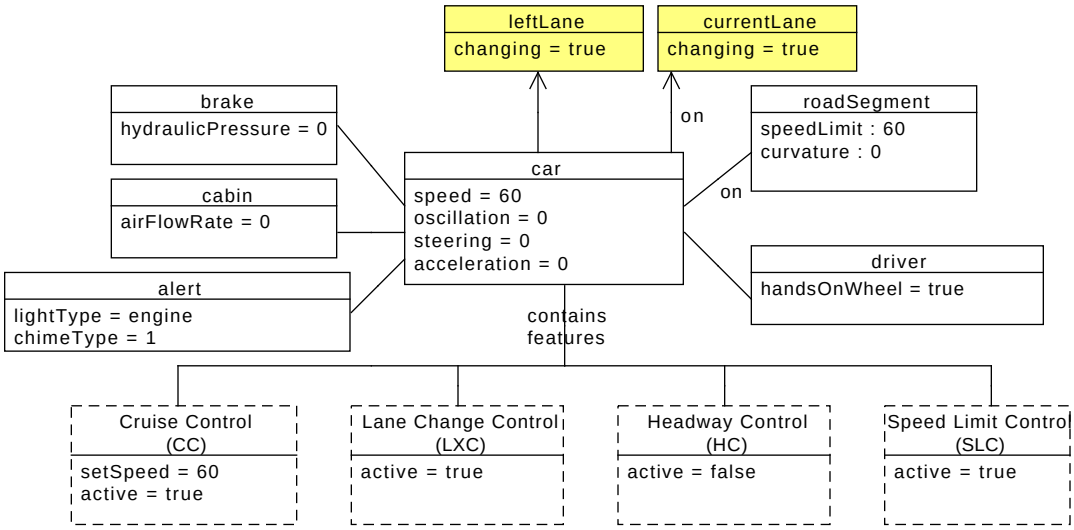


(a) World State ws_0

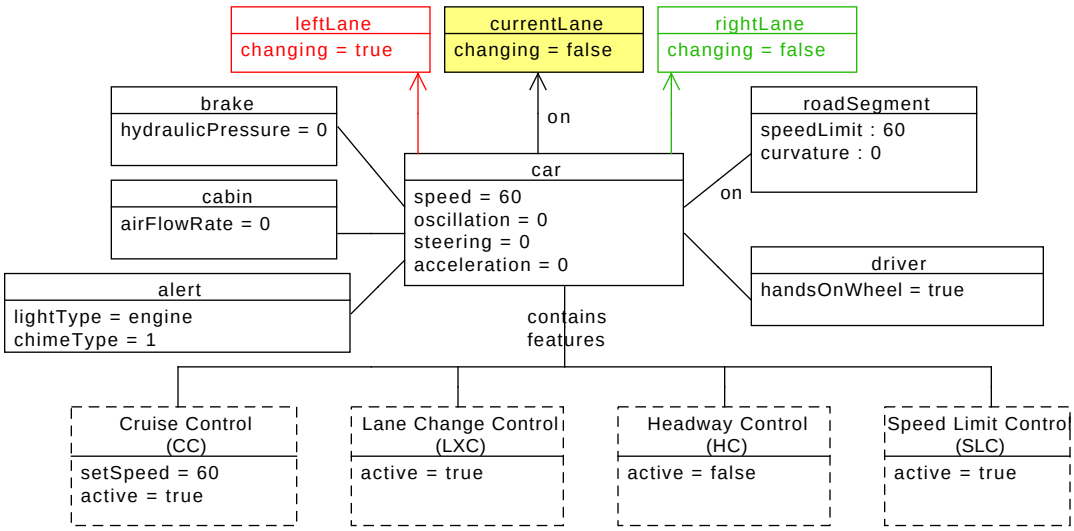


(b) World State ws_1

Figure 2.3: Progression of the world state as a result of executing WCAs



(a) World State ws_2



(b) World State ws_3

Figure 2.4: Progression of the world state as a result of executing WCAs

Finally, the world state ws_3 shows the effects of completing the lane change into the left lane. The value of the attribute *currentLane.changing* is set by the world-change action $car.currentLane.changing := false$ (shown in yellow), the former current lane is now a new lane on the right, and there is no longer a lane on the left. These changes are realized by the following world-change actions in the lane detection feature:

$$- car.leftLane$$
$$+ car.rightLane(changing = false)$$

2.2 Feature Interactions

Composing independently developed features naturally leads to feature interactions. A **feature interaction** encompasses any scenario in which a feature behaves differently in the presence of one or more other features in the system than it behaves in isolation.

Some interactions are intended. For example, the Headway Control (HC) feature extends the functionality of Cruise Control (CC) by reducing the acceleration of the vehicle if there is an upcoming obstacle in the road. In isolation, Cruise Control accelerates the vehicle to maintain a driver-set preference. This behaviour is inhibited in the presence of Headway Control, which prevents the vehicle from accelerating to the driver-set speed when there is an obstacle in the road. This feature interaction is a deliberate part of the design of Headway Control.

Other interactions cause unintended or unexpected behaviour. Consider the combination of Headway Control and Speed-Limit Control, the latter of which ensures that vehicle acceleration remains within the speed limit of the road. If the vehicle travels faster than the speed limit and at the same time approaches an obstacle, both features will simultaneously send messages to the actuators responsible for controlling vehicle acceleration. If their messages are different, the behaviour of the vehicle is undefined. Acceleration may be set to an unpredictable value.

The interactions we have described occur between two features. However, some interactions may manifest themselves only in particular combinations of three or more features [25]. Consider telephony a scenario in which user A calls user B. User A subscribes to Calling Number Delivery Blocking, a feature that hides her identity (in the form of her phone number) from the person she is calling. User B subscribes to Call Forwarding on Busy and forwards all of her calls to user C. User C subscribes to Ring Back When Free,

a feature that stores a calling number in the event that the subscriber is busy; Ring Back When Free tries to establish a connection to the deferred call as soon as the subscriber is free. When user A places a call to user B, which is then forwarded to user C, if user C is currently busy, there is ambiguity in the specifications as to whether user A's number or user B's number will be called back when user A is free. In the case of the latter, this might interfere with the appropriate behaviour of the Calling Number Delivery Blocking feature. When composing a system, software developers must look for interactions in all possible combinations of features.

The literature [8] lists multiple types of feature interactions, but for the purposes of this thesis we are concerned with only one type of interaction.

2.2.1 Feature Interactions due to Conflicts

A **conflict interaction** occurs when the set of actions in an execution step are impossible to execute simultaneously because they modify the same controlled variable. Alma et al. [24] proposed a general definition for these kinds of feature interactions in the automotive domain. Two features are in conflict if they attempt to modify the same controlled variable in the environment by assigning different values to the same actuator or to two distinct actuators that are responsible for that controlled variable.

For example, if two features try to set the same controlled variable to different values, there is no way for both of these requests to be satisfied in the same execution step. The result is an undefined or unpredictable world state. In the semantics of FORML, the resulting next world state is a special **conflict state**, which terminates the execution trace [32].

In Figure 2.2, we see the potential for conflict when both SLC and CC are Active and $car.CC.setSpeed > car.roadSegment.speedLimit$: SLC will try to decrease acceleration at the same time that CC tries to increase it:

$$CC.t2.a1 : car.acceleration := ccAcceleration()$$
$$SLC.t2.a1 : car.acceleration := slcAcceleration()$$

The next world state cannot be determined as the result of executing both of these actions, because the controlled variable $car.acceleration$ can have only one value assigned to it.

Chapter 3

Resolution

Our aim is to resolve feature interactions in a way that addresses key aspects of the feature interaction problem. We developed a strategy with the following high-level goals.

1. Maintain the advantages of feature-oriented software development. This includes preserving the ability to express feature behaviour as an independent feature machine that is oblivious to the presence of other features.

2. Enable conflict-free feature composition. Feature composition should resolve feature interactions due to conflict if they are present and should preserve feature behaviour in the absence of interactions.

3. Allow resolutions to be based on all conflicting actions rather than on the features that perform them. This limits the impact that adding or removing features has on the specification of resolutions.

4. Resolutions should be agnostic to the number of features in the system and the number of features in an interaction. In addition, the number of resolutions specified by developers should be small with respect to the number of interactions and should not grow super-linearly with the number of features.

5. The resolutions we devise should be deterministic and total. Determinism guarantees that, given a current world state and a set of changes to monitored variables, there should be only one possible value for each controlled variable in the next world state. As a result, system behaviour is predictable. Totality guarantees that there will always be a valid next world state.

We first give an overview of our approach to resolving feature interactions due to conflicting actions and describe how it fits into the execution model of a system composed of feature machines. We then present the details of our implementation and provide examples of resolutions in the automotive domain.

3.1 Overview

We draw inspiration for our approach from the Software Cost Reduction (SCR) [19] requirements method. SCR specifications follow a dataflow execution model [23], in which requirements are represented as a directed graph. Each node in the graph is a function that calculates the current value of a variable. Edges indicate the flow of data values between nodes. A node executes its function as soon as all of its required input data are available, and it outputs the result along directed edge(s) to the next node(s).

An SCR specification defines a function for each local and controlled variable. This function calculates what the value of the variable will be at the end of an execution step. Each function takes as input the current values of all monitored variables, the most recently computed values of local variables, and the current modes of all mode classes (analogous to states in a state-machine) and deterministically calculates the next value of the variable for which it is defined. There are no conflicts in an SCR specification because they are resolved during specification: each function that computes the value of a controlled variable encodes all features' contributions to that variable's next value.

Although SCR specifications are always conflict free, they do not allow for the parallel or incremental development of features. When a new unit of functionality is added to the system, all affected functions must be updated to reflect this behaviour. In SCR, there is no way to treat features as separate concerns. We want to allow for the independent and separate development of features, while also utilizing the ideas presented in SCR of controlled variable functions to eliminate the possibility of conflicts.

In our approach, we define a resolution module, similar to the controlled variable functions in SCR, for each controlled variable that defines the next value of that variable. A controlled variable resolver takes as input the features' actions on that variable and employs a feature-independent resolution strategy to assign a conflict-free value to the variable. In this way, our approach is unique in the fact that features do not need to be known in advance.

Figure 3.1 depicts the architectural structure of our approach. We define a feature machine for each feature. At the start of an execution step, the values of monitored vari-

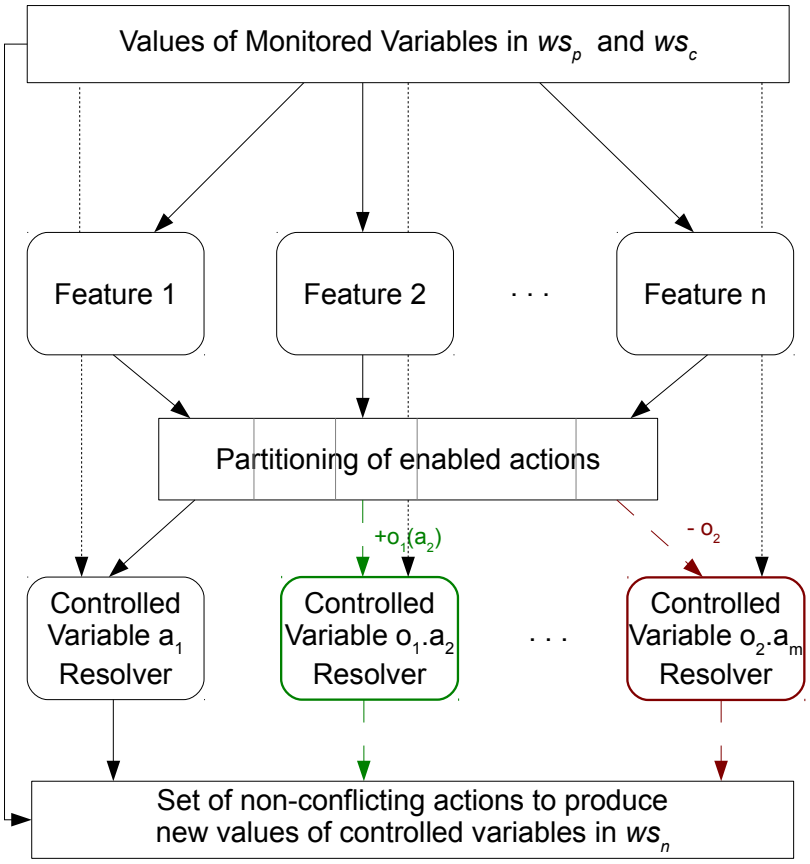


Figure 3.1: Architectural model

ables in the current and previous world states, ws_c and ws_p, respectively, prompt changes in the execution states of feature machines. The feature machines execute in parallel, each reacting to changes in the values of monitored variables. In each execution step, the outputs of each feature machine are the set of actions on the transitions that execute in that step. The features' actions are partitioned according to the controlled variable they modify. For each controlled variable in the current world state, we define a resolver module that calculates, given all of the features' actions on the controlled variable, a sequence of actions to be executed atomically on the variable. Note that our resolvers differ from the controlled-variable functions in SCR. In SCR, all information about how features affect the values of a controlled variable is encoded in the controlled variable function. The functions in SCR take as input only the current values of monitored variables. In our approach, each feature's contribution to the value of a controlled variable is encoded in its feature machine. The outputs of these machines, and inputs to the resolution modules, are *actions* on the controlled variables as well as the values of monitored variables. In this way, we use the notion of a controlled variable function to calculate conflict-free values of controlled variables in the next world state, but also support independent and modular modelling of features.

The next world state, ws_n is the result of executing in parallel the output actions of each resolver, together with the next states of each feature machine and the changes to non-controlled variables. Thus, a complete execution step of the requirements model proceeds as follows:

1. Changes occur to the values of one or more monitored variables.
2. Feature machines react in parallel by executing transitions and outputting transition actions.
3. Our resolution introduces a third phase in the execution step, in which the feature-machine actions pass through resolver modules, one resolver per controlled variable. The output of each resolver is a sequence of actions on the controlled variable to be performed atomically.
4. The next world state, ws_n, is determined by the result of performing the state transitions in step 2 the world-change actions produced in step 3, and environmental updates to the values of non-controlled variables.

The set of resolver modules in a system is dynamic. The number and type of resolvers changes as the number and type of controlled variables in the world state changes. Recall

that a world-change action can add or remove an object from the world state, where each object consists of one or more controlled variables. When a controlled variable is added to the world state, a corresponding resolver module is added to the architectural model of the system. Figure 3.1 shows the result of executing the action $+o_1(a_2)$ that adds the object o_1 with attribute a_2 to the world. A new resolver module for $o_1.a_2$ is added to the system. In future execution steps, the resolver module will take as input all actions that assign values to this controlled variable and will produce a resolution in the form of a conflict-free sequence of actions. When a controlled variable is removed from the world state, the corresponding resolver module is removed from the architectural model of the system. In Figure 2, object o_2 has attribute a_m. Therefore, after executing the action $-o_2$, the resolver for $o.a_m$ is removed and all subsequent requests to modify $o.a_m$ are ignored.

Note that not all controlled variables will be involved in feature interaction conflicts. If a variable is only ever modified by one feature, there is no need for the developer to specify a resolution module for it. Static analysis of the system may be performed to determine which variables require resolution modules [24]. This will further reduce the work of the developer.

The developer is responsible for specifying a resolution strategy for each type of controlled variable. We note that a new feature can introduce new concepts and attributes to the world model through world change actions, which in turn introduces new controlled variables for which resolution modules need to be written; but the number of variables for which resolver modules are written is often less than the number of features and therefore much less than the number of possible interactions. Moreover, as will be seen, the resolution does not depend on the specific features in the system, or the specific feature interactions to be resolved. As such, this approach to resolving interactions addresses the feature-interaction problem by drastically reducing the amount of work done by the developer.

3.2 Details

In this section, we focus on the details of the resolver modules. We describe a language that is suitable for expressing the inputs and outputs of a resolver as well as for specifying a resolver's resolution strategy. We then provide an implementation that uses situation calculus and a subset of the GOLOG programming language.

3.2.1 Input Action Language

Our resolution language needs to be rich enough to encode the inputs to a resolver module. This includes values of monitored variables and world-change actions. Recall the three main types of world-change actions that manipulate controlled variables: addition of an object, removal of an object, and variable assignment. Object addition causes a new resolver module for each of the object's attributes (if deemed necessary by static analysis) to be added to the execution model, whereas object removal causes the resolvers associated with the object's attributes to be removed from the execution model. Therefore, our resolution language needs to encode only variable assignments.

An assignment assigns a variable to the value of an expression. An expression may be a simple value, or it may be a computation on other variables or uninterpreted functions[1]. The resolution language must be able to encode any relevant information about assignment expressions, as the inputs to a resolver.

3.2.2 Requirements for a Resolution Language

How conflicting assignments to a controlled variable are resolved depends on multiple factors, including the variable's range of values, the system domain, and the effect the variable has on the behaviour of the system or on the system's environment. As such, the developer or domain expert are in the best position to determine the most appropriate resolution strategy for conflicting actions on a controlled variable. Our goal is to provide a language that is suitable for them to program appropriate resolutions.

Consider two features, A and B, that affect vehicle steering. Feature A monitors lane markings and detects that the vehicle has veered too far to the left and compensates by turning the vehicle to the right. Simultaneously, feature B monitors skidding during turns and detects that the vehicle is over-steering to the right, and attempts to correct this by turning the vehicle to the left. In order to achieve maximum stability, an appropriate resolution needs to consider the actions from both features. A reasonable resolution might be to set *car.steering* to the average of the values assigned by features A and B.

In general, the resolution language needs to be expressive enough to reason about a collection of actions and compute a conflict-free resolution. Examples of resolution strategies include computing the minimum, average, or sum of the assignment expressions that are output by the feature modules. These resolution strategies consider all assignments

[1]The details of uninterpreted functions may be contained in another document or are to be specified later in development.

Symbol	Type	Description
S_c	constant	starting state
$\text{do}(a, s)$	function	result of performing action a in situation s
$\text{Poss}(a, s)$	predicate	a can be performed in s

Table 3.1: Domain independent situation calculus symbols

to a controlled variable and they focus on the values of the expressions rather than the sources of the assignments. Such strategies offer an alternative to a win/lose resolution in which only one feature's actions (e.g., those of the highest-priority feature) are allowed to execute.

Even when it is possible to specify variable-specific resolutions, feature-based priorities sometimes still play a role and our resolution language should support them. For example, in automotive systems, it is common to give higher priority to actions that preserve driver safety, such as Speed-Limit Control (SLC), compared to actions that maintain driver-set preferences, such as Cruise Control (CC). Additionally, we may wish resolutions to prioritize driver actions over feature actions. As such, we categorize actions as being driver-controlled, safety, or non-safety and devise a resolution language that supports reasoning about action categories as well as the values of assignment expressions themselves.

3.2.3 Situation Calculus

Situation calculus [30] is a first-order language that is well-suited to expressing actions, domain-knowledge, and the effects that actions have on the current domain state. We chose to use situation calculus because it naturally supports the expression of both the input actions to our resolver modules and their resolutions.

Situation calculus constructs are grouped into three basic categories: situations, fluents, and actions. A **situation** is a first-order term that represents a world state, or a valuation of all variables. A situation is the result of performing a sequence of actions on a defined starting state, S_c[2].

Actions in the situation calculus are first-order logic terms that reflect a prescribed change to a situation, or world state. These actions may take one or more arguments as inputs. For our purposes, situation-calculus actions are analogous to world-change actions

[2]We deviate from the traditional situation-calculus terminology of "initial-state" to avoid confusion with the concept of the initial state for a state-machine model.

in a feature machine. Performing an action a on a situation S_c is expressed using the special function do, and results in a new situation s_n:

$$s_n = do(a, S_c)$$

Fluents are functions and predicates that take a situation as one of their arguments; they are referred to as fluents because their valuations depend on the situation to which they are applied. Fluents can be used to refer to the values of variables in a particular world state. For example, the functional fluent $carSpeed(s)$ returns the value of the car's speed in the world state represented by situation s.

The developer uses a combination of situations, actions, and fluents to specify allowable steps in the execution of a system. Additional specifications are characterized with axioms that constitute a domain theory $\mathcal{D}$. Starting-state axioms are assertions on the values of fluents in a starting state S_c. Successor-state axioms define the effects of performing an action a in a situation s. The developer uses fluents to specify assertions on variable values in the successor situation that results from performing an action. Precondition axioms specify whether an action a may be performed in a given situation s; they are expressed with the special predicate $Poss(a, s)$. Table 3.1 contains a summary of special situation-calculus constructs.

3.2.4 Encoding Inputs to a Resolver Module

The inputs to a resolver module for a controlled variable of type $o.a$ are (1) the values of monitored variables at the start of the execution step (i.e., the values of the monitored variables in the world state ws_c), and (2) the set of world-change actions that are output by all the feature modules and that assign values to $o.a$.

The developer encodes the inputs to a resolver as starting-state axioms, which assert constraints on the valuations of fluents in the starting situation S_c. The starting situation of each resolver module will reflect the current world state, ws_c. Given a monitored variable m in ws_c, the developer asserts the relational fluent $m(v, S_c)$ to be true if the value of m in ws_c is equal to v ($ws_c :: m = v$). To express the set of input actions, she asserts another relational fluent $assignRqst(L, S_c)$ where L represents the set of assignment expressions output by the feature modules. That is, if the features perform the actions $o.a := e_1, \ldots, o.a := e_n$, then $L = [e_1, \ldots, e_n]$. Note that the number n of actions output by the feature modules depends on the transitions that execute in the feature machines and varies between execution steps.

To distinguish between safety, driver, and non-safety actions, the developer may define *assignRqst* fluents for each category. The fluents $assignRqstSafety(L, S_c)$, $assignRqstDriver(L, S_c)$, and $assignRqstNonSafety(L, S_c)$ correspond to assignment expressions partitioned according to the above categories. In general, the developer may define a fluent for every level of prioritization she wishes to express. The purpose of this is to allow for resolutions that distinguish between broad categories of actions. Note that, throughout the development of each resolution module, the developer gains a sense of the appropriate categories features may belong to. During composition, the features must be grouped into these categories so that their actions can be appropriately partitioned before they are sent to the resolution modules.

Additionally, we can reason about the structure of an assignment expression e. For example, if it is important for the resolution to distinguish between assignments that depend on a monitored variable m, we can encode this information by asserting a fluent $dependsOnm(e, S_c)$. The expression e will then appear both in the list of input actions, and in this additional fluent. Such information may be useful when reasoning about a feature's motives for assigning a controlled variable. The level of detail these fluents express is left to the developer (e.g., whether an expression is dependent on a variable, increases a variable, increases a variable by a precise amount, and so on).

Example 1. Suppose a developer is responsible for programming the resolver module for the controlled variable *car.acceleration*. Relevant monitored variables include *car.speed* and *car.acceleration*. The developer distinguishes between three levels of prioritization: safety, non-safety, and driver features.

In a particular world-state ws_c, the feature modules for Cruise Control, Speed-Limit Control, and Basic Driving Service output the following respective actions:

$$car.acceleration := ccAcceleration()$$
$$car.acceleration := slcAcceleration()$$
$$car.acceleration := Acceleration() + 5$$

Thus, the starting-state axioms that encode these inputs are:

$$car.speed(ws_c :: car.speed, S_c)$$
$$car.acceleration(ws_c :: car.acceleration, S_c)$$
$$assignRqstSafety([slcAcceleration()], S_c)$$
$$assignRqstDriver([Accleration() + 5], S_c)$$
$$assignRqstNonSafety([ccAcceleration], S_c)$$

Symbol	Type	Description
empty(L)	Predicate	list L is empty
member(L, e)	Predicate	element e is in list L
first(L)	Function	returns the first element of the list L
rest(L)	Function	removes the first element from the list L
append(L, e)	Function	append element e to list L
order(L)	Function	orders list L from maximum to minimum
remove(L, e)	Function	remove element e from list L
average(L, v)	Predicate	average value in L is v
minimum(L, v)	Predicate	minimum value in L is v
maximum(L, v)	Predicate	maximum value in L is v
sum(L, v)	Predicate	sum of values in L is v
$<$, $>$, $=$, $\neq$	Predicate	equality and inequality
$+$, $-$, $*$, $/$	Function	arithmetic operations

Table 3.2: Domain independent resolution symbols

3.2.5 Encoding the Resolutions of a Resolver

Each resolver module is responsible for assigning a value to one controlled variable. Therefore, the developer defines only one situation-calculus action $assign(v)$ per resolver; the action assigns the resolved value v to the controlled variable in question.

The developer uses precondition axioms to specify how the value v is computed. A precondition axiom $Poss(a, s)$, dictates the conditions under which the situation-calculus action a may be performed in the situation s. Thus, the developer defines a precondition $Poss(assign(v), s)$ to specify the conditions under which the output of the resolver module in situation s is $assign(v)$. We express precondition axioms using the fluents described above. Some helper predicates and functions that we deem useful for specifying resolutions are listed in Table 3.2. This list is by no means exhaustive — the developer may define any first-order logic predicate and function needed to provide appropriate resolutions for her domain.

Example 2. Recall the acceleration example discussed in Example 1. There are several safety and convenience features that modify the controlled variable *car.acceleration*. These include Cruise Control (CC) and Speed Limit Control (SLC), which aim to keep the monitored variable *car.speed* at a driver-set preference and below the speed limit of the road, respectively. Additionally, the headway control (HC) feature

changes the vehicle's acceleration in response to upcoming obstructions or other cars on the road; and the driver can affect vehicle acceleration by pressing her foot on the accelerator pedal.

Based on our understanding of how these features ought to interoperate with each other, we devised the following resolution[3]. Our resolution considers three different levels of priority: driver, safety, and non-safety actions. Driver actions to modify the car's acceleration have the highest priority, followed by actions from safety features, followed by actions from non-safety features. If there are multiple driver-related input actions, the resolver module will assign the minimum value. If there are no driver-related actions, then safety actions will be considered. If there is more than one safety action, then the minimum value is selected to be the output action. For example, if there are no driver-related actions and the two safety features, Speed-Limit control and Headway Control, both contribute input actions $car.acceleration := e_1$ and $car.acceleration := e_2$, then the output action will be the minimum of these two values.

Our resolution is expressed in situation calculus as follows:

$$
\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstDriver(l, s) \wedge minimum(l, v)) \vee \\
&\quad (\forall l.(assignRqstDriver(l, s) \rightarrow empty(l)) \wedge \\
&\qquad \exists l_2.assignRqstSafety(l_2, s) \wedge minimum(l_2, v)) \vee \\
&\quad (\forall l.(assignRqstDriver(l, s) \vee assignRqstSafety(l, s) \rightarrow \\
&\qquad empty(l))) \wedge (\exists l_3.assignRqstNonSafety(l_3, s) \wedge \\
&\qquad minimum(l_3, v))
\end{aligned}
$$

The interpreter will first see if there are any elements in the list of input driver actions and take the minimum value of this list. If the list of driver actions is empty, it will attempt to find the minimum value of the list of input safety actions. Finally, if there are no driver or safety actions, the interpreter will output the minimum value of non-safety actions.

As long as there is at least one enabled action, there will be a value v that satisfies the above formula. If there are no input actions, the resolver will not output any actions and the value of the controlled variable will not change. In Section 4, we discuss the necessary and sufficient conditions to ensure that resolutions are deterministic and total.

[3]For the purposes of this thesis, it does not matter whether or not we have a correct understanding of how feature interactions ought to be resolved. What matters is whether our proposed resolution language is expressive enough to specify interesting, non-trivial resolution strategies.

```
/* Starting-State Axioms */
assignRqstDriver([SlcAcceleration],sc).
assignRqstSafety([Acceleration+5],sc).
assignRqstNonSafety([CcAcceleration],sc).

/* Precondition Axiom */
poss(assign(N),S) :- assignRqstDriver(R,S),minimum(N,R);
    assignRqstDriver([],S), assignRqstSafety(R,S), minimum(N,R);
    assignRqstDriver([],S), assignRqstSafety([],S), assignRqstNonSafety(R,S),
        minimum(N,R).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveAcceleration, pi(n,assign(n))).

/*Command to run resolver*/
do(pi(v,assign(v)),sc,S).
```

Figure 3.2: GOLOG implementation of *car.acceleration* resolution module

3.2.6 Implementation in GOLOG

We use the logic programming language, GOLOG [28], to implement our resolution modules. GOLOG is a language for writing complex control procedures for situation calculus actions that operate within the specifications of situation-calculus precondition and starting-state axioms. We use a GOLOG interpreter [28] written in SWI-Prolog [34]. The resolver modules themselves are likewise written and compiled with the SWI-Prolog compiler. Once compiled, these modules may be run from the command line.

We now explain how a single resolution module is implemented in a combination of situation calculus and GOLOG. As we walk through the explanation, we will refer to the resolution of the variable *car.acceleration* given in Examples 1 and 2. The full encoding of the resolution module in GOLOG is given in Figure 3.2.

Every resolver has its own domain theory, $\mathcal{D}$, which comprises the situation-calculus facts, fluents, and axioms that describe the module's inputs and resolution strategy. In general, the inputs to the resolver are encoded in situation calculus as starting-state axioms: one axiom for each priority level of input actions and one axiom for each monitored variable.

For example, in the *car.acceleration* example, the inputs to the resolution module are encoded as the situation-calculus starting-state axioms:

$$assignRqstSafety([slcAcceleration()], S_c)$$
$$assignRqstDriver([Accleration() + 5], S_c)$$
$$assignRqstNonSafety([ccAcceleration], S_c)$$

These axioms represent the world-change actions output by the feature machines in the world state ws_c. The actions are partitioned according to whether they represent a safety function, the driver, or a non-safety function, respectively. These starting-state axioms are shown at the top of Figure 3.2.

The resolution strategy for the resolver's controlled variable is encoded as a precondition axiom in situation calculus. The precondition axiom $Poss(assign(v), s)$ dictates which values v may be assigned to the resolver's controlled variable in a situation s (considering the input actions and monitored-variable values that pertain to the situation s). The resolution of conflicting actions for *car.acceleration* is encoded as the precondition axiom:

$$\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstDriver(l, s) \wedge minimum(l, v)) \vee \\
&\quad (\forall l.(assignRqstDriver(l, s) \rightarrow empty(l)) \wedge \\
&\qquad \exists l_2.assignRqstSafety(l_2, s) \wedge minimum(l_2, v)) \vee \\
&\quad (\forall l.(assignRqstDriver(l, s) \vee assignRqstSafety(l, s) \rightarrow \\
&\qquad empty(l))) \wedge (\exists l_3.assignRqstNonSafety(l_3, s) \wedge \\
&\qquad minimum(l_3, v))
\end{aligned}$$

This axiom encodes when it is possible to execute a world-change action of the form

$$car.acceleration := v$$

The GOLOG code for this axiom is shown in the second section of Figure 3.2.

The GOLOG program itself specifies how the resolution strategy encoded in $Poss(assign(v), s)$ is carried out to determine the resolution module's output. In simple cases, as in the acceleration example, the GOLOG program invokes the resolution strategy once to determine a single output action. In more complicated cases, GOLOG allows the developer to specify complex procedures by forming sequences or iterations of actions for the resolution module to output. We now describe the GOLOG programming language. Situation-calculus actions form the basis for GOLOG programs. Because our resolutions need to be deterministic, we restrict ourselves to the following GOLOG constructs, where δ, δ_1, and δ_2 range over all possible GOLOG programs.

a	situation calculus action
$\delta_1 \; ; \; \delta_2$	sequence
$\pi v.\delta$	nondeterministic choice of arguments
if ϕ **then** δ_1 **else** δ_2	conditional
while ϕ **do** δ	while loop

The GOLOG program to execute the acceleration resolution is:

$$\pi v.assign(v)$$

and will execute according to the assertions of the domain theory defined in Examples 1 and 2. The program $\pi v.assign(v)$ nondeterministically chooses arguments v to apply to find one v that satisfies the precondition axiom $Poss(assign(v), S_c)$, contained in the situation calculus domain theory $\mathcal{D}$. The output of this program will then be this single action $assign(v)$, which corresponds to the world-change action $car.acceleration := v$. This program is shown in Figure 3.2 as the control procedures for the resolution[4].

The Prolog command to run the acceleration resolver is:

```
do(pi(v,assign(v)),sc,S)
```

where pi(v,assign(v)) is the Prolog equivalent of the GOLOG program above.

The GOLOG interpreter functions as a solver that fulfills the entailment [5]

$$\mathcal{D} \models Do(\delta, S_c, S_n)$$

where $\mathcal{D}$ is the set of precondition and starting-state axioms in situation calculus that make up the resolution domain theory, δ is the GOLOG program that defines the resolution module's procedure for executing situation-calculus actions, S_c is the starting situation, and S_n is a valid terminating situation. Given $\mathcal{D}$, δ, and S_c, the GOLOG interpreter attempts to find a terminating situation S_n that satisfies the above entailment. This terminating situation is represented as a sequence of situation-calculus actions performed from the starting state S_c using the function $do(a, s)$ as follows:

$$S_n = do(a_n, \ldots do(a_2, do(a_1, S_c)) \cdots)$$

[4]Note the additional control procedure primitive_action(assign(N)). This tells the GOLOG interpreter that assign(n) is to be treated as a situation-calculus action.

[5]Note that the GOLOG construct differs from the first-order function $do(a, s)$, which denotes the situation that results from performing the action a in s.

where $a_1, \ldots, a_n$ are situation calculus actions that can be mapped to the world-change actions to be performed at the end of the current execution step.

Revisiting the GOLOG programming language constructs, we explain their semantics in terms of what it means to fulfill the entailment:

$$\mathcal{D} \models Do(\delta, S_c, S_n)$$

1. a is an atomic GOLOG program consisting of a single primitive situation-calculus action with either no arguments or with only constants, defined values for all of its arguments[6]. Such a program may be executed from the starting state S_c only if the resolution domain theory allows that action to take place in S_c. Therefore, the predicate $Do(a, S_c, S_n)$ is characterized as:

$$Do(a, S_c, S_n) \stackrel{def}{=} Poss(a, S_c) \wedge S_n = do(a, S_c)$$

2. $\delta_1; \delta_2$ is the sequence of two GOLOG programs δ_1, δ_2. A sequence of GOLOG programs may be executed from the starting state S_c only if the resolution domain theory allows δ_1 to be executed in S_c and δ_2 to be executed from the terminating state of δ_1. Therefore, the predicate $Do(\delta_1; \delta_2, S_c, S_n)$ is characterized as:

$$Do(\delta_1; \delta_2, S_c, S_n) \stackrel{def}{=} \exists S_i. Do(\delta_1, S_c, S_i) \wedge Do(\delta_2, S_i, S_n)$$

   Sequences are used when the developer wants the output of a resolution module to be a sequence of actions.

3. $\pi x.\delta$ signifies the nondeterministic choice of the value for x, and applies that value as an argument to situation-calculus actions and fluents in the GOLOG program δ. For example, $\pi x.assign(x)$ is a GOLOG program that assigns a nondeterministically chosen value x to the resolver's controlled variable. Recall that, because GOLOG programs operate within the restrictions of our situation-calculus domain theory, a particular argument value v will be chosen only if $Poss(assign(v), S_c)$ is satisfied. Therefore, the predicate $Do(\pi x.\delta, S_c, S_n)$ is characterized as:

$$Do(\pi x.\delta, S_c, S_n) \stackrel{def}{=} \exists x. Do(\delta(x), S_c, S_n)$$

   where $\delta(x)$ is the application of the argument x to a situation calculus action or fluent in the program δ.

[6]The π construct is used for actions with variable arguments.

The developer uses this nondeterminism operate to choose nondeterministically from a set of possible solutions. This operator is also necessary when the task of the GOLOG program is to *find* an argument v that satisfies the precondition axiom $Poss(assign(v), S_c)$.

4. **if** ϕ **then** δ_1 **else** δ_2 Conditionals are expressed in GOLOG with pseudo-formulas, whose situation arguments have been suppressed. These are formulas with situation-calculus fluents where all situation arguments are suppressed. For example, the formula $car.speed(v) \wedge v > 30$ uses the fluent $car.speed(ws_c :: car.speed, S_c)$, but with the situational argument S_c suppressed. The predicate $Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, S_c, S_n)$ is characterized as follows:

   $$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, S_c, S_n) \stackrel{def}{=} (\phi(S_c) \wedge Do(\delta_1, S_c, S_n)) \vee (\neg\phi(S_c) \wedge Do(\delta_2, S_c, S_n))$$

   where the formula $\phi(S_c)$ is the situation-calculus formula $\phi(s)$ applied to the starting state S_c.

   Conditionals are used when the output of a resolution module depends on the valuation of a situation-calculus fluent in that starting state.

5. **while** ϕ **do** δ Iteration is expressed in GOLOG with a while loop that depends on a situation-calculus pseudo-formula. A situation S_n can result from executing a while loop from state S_c only if S_n is the result of applying δ to S_c n times and $\neg\phi$ does not hold in S_n. Additionally, for all $i < n$, applying δ to S_c i times results in the intermediate state S_i where ϕ holds in S_i. The predicate $Do(\textbf{while } \phi \textbf{ do } \delta, S_c, S_n)$ is characterized as:

   $$\begin{aligned} Do(\textbf{while } \phi \textbf{ do } \delta, S_c, S_n) \stackrel{def}{=} (\forall P.(\forall s_c.P(s_c, s_c)) \wedge (\forall s_c, s_i, s_n.P(s_c, s_i) \wedge \\ \phi(s_i) \wedge Do(\delta, s_i, s_n) \supset P(s_c, s_n)) \supset P(S_c, S_n)) \wedge \neg\phi(S_n) \end{aligned}$$

   This definition uses second order logic to express the nondeterministic iteration of the while loop. By quantifying over all binary relations, we are finding the smallest set such that (s_c, s_n) is in the set if and only if we can get from s_c to s_n by performing δ some number of times. In other words, we are determining whether the situation S_n is reachable from the starting situation S_c. Executing the program δ from the starting situation S_c will result in the terminating situation S_n if and only if

   (a) Base Case: $\forall s_c.P(s_c, s_c)$, we can always get from situation S_c to the situation S_c by performing zero iterations of the program δ.

(b) Inductive Case: $\forall s_c, s_i, s_n.P(s_c, s_i) \wedge \phi(s_i) \wedge Do(\delta, s_i, s_n)$, if we can execute some number of iterations of δ to get from a starting state s_c to an intermediate state s_i, then we can execute one more iteration of δ to get to the state s_n.

In other words, we can get from S_c to S_n by performing some number of iterations of the GOLOG program δ. When we have reached S_n, the condition $\neg\phi(S_n)$ must hold.

The developer uses while loops to iteratively perform the procedure δ, where the valuation of the situation-calculus fluent ϕ in each successive state determines whether the program δ is applied again.

Referring to the GOLOG program for the acceleration module in Figure 3.2, the interpreter attempts to find a situation S_n that satisfies the entailment $\mathcal{D} \models Do(\delta, S_c, S_n)$. The program $\pi v.assign(v)$ nondeterministically chooses arguments v, searching for some value v that satisfies the module's precondition axiom. Assuming the precondition axiom is deterministic and total, then there will be exactly one such value v. The output will be a situation S_n, expressed as a sequence of actions performed from the starting state S_c

$$S_n = do(assign(v), S_c)$$

This output corresponds to the world change action $car.acceleration := v$. The next world state ws_n is calculated as the result of performing this action, along with the outputs of all other resolution modules and the update values of non-controlled variables.

The resolution examples discussed up to this point output a single action to be performed on a controlled variable. For some variables, it may be advantageous to output a sequence of actions to be performed atomically on a controlled variable. In this case, we use successor-state axioms to express the changes that each situation-calculus action makes to the current situation (i.e., variable valuations). For example, if the value of $car.acceleration$ is v_c in the world state ws_c, then performing the action $assign(v)$ will result in a new situation $do(assign(v), S_c)$ in which the value of $car.acceleration$ is now v. The next action is performed on the new situation, and so on, through the atomic list of actions until a final situation is reached. To encode the effects of each situation-calculus action, we specify the successor-state axiom

$$car.acceleration(v, do(a, s)) \equiv a = assign(v)$$

This states that the value of the variable $car.acceleration$ will be v after performing the single action a in the previous situation s, where a is the situation calculus action $assign(v)$.

The developer does not need to create a successor-state axiom for every combination of fluents and actions; only for those that are relevant to the resolution in question. These axioms form part of the domain theory $\mathcal{D}$ described above. The following example illustrates the use of successor-state axioms.

Example 3. The variable *alert.lightType* is controlled by alert features that try to get the driver's attention. If multiple features want to set a particular light to different values, one possible resolution is to satisfy all requests sequentially. For example, if one feature turns a light off and another turns the light on, the resolution is to have the light blink on and off, to alert the driver to a possible conflict among the features associated with the warning light. The inputs to the resolution are

$$alert.lightType(ws_c :: alert.lightType, S_c)$$
$$assignRqst(L, S_c)$$

The resolution will cycle through the values in $assignRqst(L, S_c)$, setting the variable *alert.lightType* to each value in sequence for a fixed number of iterations. In order to do this, we first order the list L, to produce a deterministic output, and then append it to itself 3 times. This is done with an intermediary situation calculus action, *setup*, which is encoded with the following successor state axiom:

$$assignRqst(l, do(a, s)) \equiv a = setup \wedge l = append(append(order(l), order(l)), order(l))$$

The resolution will then iterate through the list, setting *alert.lightType* to the first element of the list, removing this element, and repeating with the remainder of the list. This behaviour requires the following successor state axioms:

$$alert.lightType(v, do(a, s)) \equiv a = assign(v)$$
$$assignRqst(l, do(a, s)) \equiv a = assign(v) \wedge l = rest(k) \wedge assignRqst(k, s)$$

The resolution is characterized in the precondition axiom

$$Poss(assign(v), s) \equiv assignRqst(l, s) \wedge first(l, v)$$

The GOLOG program is

$$\begin{array}{l} setup; \\ \quad \textbf{while } (assignRqsts(l, now) \wedge \neg empty(l)) \\ \quad\quad \textbf{do } \pi n.assign(n) \end{array}$$

This program first sets up the list by ordering it (to ensure determinism) and copying it three times to produce three cycles through the original values. It then uses a while loop to iterate through the entire list. The condition on the while loop checks whether the list is empty. If it is not empty, the first value in the list will be assigned and removed from the list.

The result from this program is a sequence of actions

$$S_n = do(assign(v_n), \ldots do(assign(v_1), S_c)) \cdots)$$

In general, we translate the sequence of situation calculus actions output by the resolution module back into the terms of our requirements modelling language as the sequence of variable assignment actions to be performed on the variable in ws_c to compute the value of the variable in the next world state ws_n.

$$alert.lightType := v_1, \ldots, alert.lightType := v_n$$

Example 4. The controlled variable *car.steering* is controlled by features that affect the direction in which the car is travelling. These include Lane-Centring Control (LCC) and several stability features such as Traction Control (TC) and Stability Control (SC). The driver can also affect *car.steering* by rotating the steering wheel.

For this resolution, we distinguish between driver, safety, and non-safety actions. The purpose of stability features is to correct the bad effects of driver steering on slippery surfaces. Thus, we give these safety-related actions the highest priority. Driver-related actions have the next highest priority, followed by non-safety actions. If there are multiple assignments at the same priority level, the variable *car.steering* is set to the average of the assignment expressions. Note that the resolution is not win/lose in that the actions from all features affect the outcome of the resolution.

Inputs to the resolution module for *car.steering* are world-change actions of the form

$$car.steering := \langle expr \rangle$$

output by the feature modules of LCC, TC, SC, BDS, and any other features that modify vehicle steering. We partition these actions further, according to whether they represent driver, safety, or non-safety-related actions. These actions, along with the values of relevant monitored variables, such as the current value of *car.steering*, are input into the resolution module. In this implementation, inputs are expressed as starting-state axioms for the situation S_c, as they represent the current world state ws_c.

The inputs to the resolver are:

$$car.steering(ws_c :: car.steering, S_c)$$
$$assignRqstDriver(L, S_c)$$
$$assignRqstSafety(L, S_c)$$
$$assignRqstNonSafety(L, S_c)$$

The resolution is encoded as a precondition axiom that specifies the conditions under which the resolver module outputs $assign(v)$ as the conflict-free assignment.

$$\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstSafety(l, s) \wedge average(l, v)) \vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \rightarrow empty(l)) \wedge \\
&\qquad \exists l_2.assignRqstDriver(l_2, s) \wedge average(l_2, v)) \vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \vee assignRqstDriver(l, s) \rightarrow empty(l)) \wedge \\
&\qquad \exists l_3.assignRqstNonSafety(l_3, s) \wedge average(l_3, v))
\end{aligned}$$

This resolution gives priority to safety actions and will take the average of multiple actions at the same priority level, if they exist.

The GOLOG program is

$$\pi v.\ \text{assign}(v)$$

A successful resolved assignment is one that satisfies the precondition axiom specified above: $Poss(assign(v), S_c)$. In this case, as in the acceleration example, the precondition axiom is deterministic and the GOLOG program outputs a sequence of only one assignment action, expressed as the terminating situation $S_n = do(assign(v), S_c)$ where S_n satisfies the entailment:

$$Do(\ \pi v.\ \text{assign}(v), S_c, S_n)$$

This corresponds to the single world-change action $car.steering := v$, which is the output of the resolution module.

Chapter 4

Analysis

In this chapter, we demonstrate that our resolutions to feature interactions have the desired properties that we listed in the beginning of Chapter 3: that the resolutions are conflict-free, are deterministic and total, and preserve the features' actions in the absence of an interaction.

We also show that our resolution addresses the Feature Interaction Problem by focusing on the outputs of a system. This reduces the work of the developer to be linear in the number of types of output variables. The focus on outputs also allows the developer to specify win/win resolutions, maintains feature obliviousness, and allows for agnosticism with respect to the number of features in the system.

The most important goal of our work is to enable conflict-free feature composition. Recall that a feature interaction due to conflict occurs when two or more features attempt to simultaneously execute a set of incompatible world-change actions. The resolution approach that we have devised eliminates conflicts by computing a conflict-free sequence of actions. Such a computation is performed for the actions on each controlled variable in each execution step of the system.

Theorem 1. *The set of action sequences output by the resolver modules are conflict-free.*

Proof. A feature may execute one or more world change actions in each step. These actions each belong to one of three categories: adding an object to the world, removing an object from the world, and modifying the value of a controlled variable. Possible conflicts occur when:

1. Two or more features attempt to set the value of the same controlled variable to different values, or

2. One feature attempts to remove an object from the world while another feature attempts to modify it.

Case 1: All of the features' assignments to the same controlled variable are forwarded to the same resolver module. The resolver outputs one sequence of actions to be executed sequentially, thus the resolvers' actions do not interact. Furthermore, each resolver produces a resolved sequence of actions for a distinct controlled variable; thus the output actions of the resolver modules do not conflict[1].

Case 2: An action to remove an object from the world state results in the removal of all the resolver modules associated with the object's controlled variables. All attempts to assign a value to any of these variables are ignored. In this way, object removal has priority over assignment.

In both cases, the set of output actions is conflict-free. □

A second goal of our work is that resolutions should be deterministic: for each set of features' actions, a resolver produces a unique sequence of assignments to the corresponding controlled variable. Determinism is not guaranteed by the approach itself. The developer is responsible for ensuring that the resolutions she programs are deterministic by demonstrating that, in every resolver module, the axiom $Poss(assign(v), s)$ is satisfied by only one possible value v in any situation s:

$$\textbf{Obligation 1}.\ \forall v_1.\forall v_2.\forall s.Poss(\text{assign}(v_1), s) \\ \wedge Poss(\text{assign}(v_2), s) \rightarrow v_1 = v_2$$

Then there is only one possible sequence of output actions from the resolver and only one possible value for the variable in next world state ws_n calculated as the result of performing these actions.

This property also ensures the resolution of all feature interactions due to nondeterminism. Since the enabled actions of all features are considered at every execution step, and a resolution module always output a deterministic set of actions, the possibility for nondeterminism does not exist.

Theorem 2. *Given the set of situation-calculus facts, fluents, and axioms $\mathcal{D}$ for a resolver module,* **Obligation 1** *on the precondition axiom in $\mathcal{D}$, and the corresponding resolution*

[1]It is possible that actions on different controlled variables may interfere to produce a non-conflict type of interactions. At present, these interactions would need to be explicitly resolved. In the future, we will investigate more generic methods for resolving such interactions.

procedure δ, then the following entailment holds:

$$\mathcal{D} \models \forall s_c, s_n, s'_n. Do(\delta, s_c, s_n) \supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n)$$

Proof. This is a sketch of the proof — full details are given in the appendix.

This is proven by structural induction on the GOLOG resolution program, δ. The base case proves the entailment for a program $\delta = a$ where a is a primitive situation calculus action:

$$\mathcal{D} \models \forall s_c, s_n, s'_n. Do(a, s_c, s_n) \supset (Do(a, s_c, s'_n) \supset s_n = s'_n)$$

By the soundness of first-order logic, we can prove this entailment holds by using natural deduction to formally deduce the following:

$$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(a, s_c, s_n) \supset (Do(a, s_c, s'_n) \supset s_n = s'_n)$$

The inductive step assumes two GOLOG programs δ_1 and δ_2 that both satisfy the above entailment, and we prove that a GOLOG program of each of the following forms satisfy the above entailment.

- Sequence $\delta = \delta_1; \delta_2$

  $$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(\delta_1; \delta_2, s_c, s_n) \supset (Do(\delta_1; \delta_2, s_c, s'_n) \supset s_n = s'_n)$$

- Nondeterministic choice of arguments $\delta = \pi v. \delta_1$

  $$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(\pi v. \delta_1, s_c, s_n) \supset (Do(\pi v. \delta_1, s_c, s'_n) \supset s_n = s'_n)$$

- Conditional $\delta = \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2$

  $$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s_n) \supset (Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s'_n) \supset s_n = s'_n)$$

- While loop $\delta = \textbf{while } \phi \textbf{ do } \delta_1$

  $$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(\textbf{while } \phi \textbf{ do } \delta_1, s_c, s_n) \supset (Do(\textbf{while } \phi \textbf{ do } \delta_1, s_c, s'_n) \supset s_n = s'_n)$$

□

A third goal is that the resolutions be total: for any non-empty set of features' actions on a controlled variable, there exists a non-empty sequence of conflict-free actions output by the variable's resolver module. To ensure that the resolutions are total, the developer must demonstrate that each resolver's precondition axiom can be satisfied.

The case in which there are no input actions to the resolver also needs to be considered. If there are no features attempting to modify a controlled variable, then no world-change actions should be executed on this variable in the current execution step. Thus, if there are no input actions to a resolver module, the resolver should have no outputs. We only want to output actions if there is at least one action being performed on the variable in question.

Obligation 2. $\forall s.\,(\exists v.Poss(assign(v), s) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$

where $assignRqst(L, s) \wedge \neg empty(L)$ is true if and only if there is at least one input action, at any priority level, in the situation s.

Note that particular attention must be paid to computations that have while loops. For every such loop, the developer must demonstrate that the loop terminates:

$$\forall s.(\exists s'.(\forall P.(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge \phi[s_2] \wedge Do(\delta_1, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s')) \wedge \neg\phi[s'] \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

Theorem 3. *Given the set of situation-calculus facts, fluents, and axioms $\mathcal{D}$, for a resolver module;* **Obligation 2** *on the resolver's precondition axiom; the corresponding GOLOG program δ; and obligations for each while loop in δ, then the following entailment holds:*

$$\mathcal{D} \models \forall s_c.\,(\exists s_n.Do(\delta, s_c, s_n) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$$

Proof. As above, this is proven by structural induction on the GOLOG program δ. For each step of the proof, we consider two cases: one in which there exist input actions, and one in which there are no input actions. See Appendix for details. □

As a fourth goal, a resolution should preserve the functionalities of the features being composed. If the set of features' actions on a controlled variable in an execution step are non-conflicting (there is only one input assignment), then the resolution should include all actions on that variable. This requires an obligation on both the precondition axiom and the GOLOG program δ:

Obligation 3. $\mathcal{D}, \mathbf{Ob3}, assignRqst([v], S_c) \vdash \exists s_n.Do(\delta, S_c, S_n) \wedge S_n \approx do(assign(v), S_c)$

where $assignRqst([v], S_c)$ states that there is only one input assignment action to the resolution module. Additionally, we use the notation $S_n \approx do(assign(v), S_c)$ to account for sequences of situation calculus actions that, when translated back into the modelling language, are equivalent to performing the assignment action $o.a := v$. See Example 3 in Section 3.2.6 for more details.

Theorem 4. *Given the set of situation-calculus facts, fluents, and axioms $\mathcal{D}$, for a resolver module;* **Obligation 3***; and the corresponding GOLOG program δ, then feature functionality will be preserved in the absence of feature-interaction conflicts.*

Proof. The obligation

$$\mathcal{D}, \mathbf{Ob3}, assignRqst([v], S_c) \vdash \exists s_n. Do(\delta, S_c, S_n) \wedge S_n \approx do(assign(v), S_c)$$

along with the determinism property proved in Theorem 1, ensures that if there is only one input action the resolution module for the controlled variable $o.a := v$, encoded as the starting-state axiom $assignRqst([v], S_c)$, then the output of the resolver module will be a sequence of actions that translate back into the modelling language action $o.a := v$. Note that since we require the translation to be equivalent to this action, we also except outputs of the form:

$$o.a := v, \ldots, o.a := v$$

where the variable $o.a$ is repeatedly assigned the same value v. This must be proven for every resolver module.

□

4.1 Case Study

We conducted a case study to analyze whether the expressive power of the situation calculus is suitable for expressing feature interaction resolutions. We examined the requirements specifications of 24 automotive features that were provided to us by our industrial partners, although the feature names we give are based on common features found on the internet. In the models we investigated, we encountered a total of 8 controlled variables. Two of these variables were modified by only one feature. The remaining 6 were modified by several features. This supports our claim that new controlled variables are introduced to a system at a much slower rate than new features. Table 4.1 shows the variables, along with the number of features that modify them.

Variable name	Number of features that modify it
car.acceleration	4
car.steering	6
alert.lightType	8
brake.hydralicPressure	9
alert.chimeType	5
cabin.airFlowRate	6

Table 4.1: Case Study Variables

Three of these variables, *car.acceleration*, *car.steering*, and *alert.lightType* were used during the course of developing our approach and the implementation discussed in Section 3. We examined the requirements specification of the features, their FORML models, and any proposed resolutions mentioned in the specifications to determine the necessary inputs and functionality for the resolution modules.

The remaining three variables, *brake.hydraulicPressure*, *alert.chimeType*, and *cabin.airFlowRate*, were used to evaluate whether or not the resolution language is strong enough to express appropriate resolutions for a variety of controlled variables. We now provide the details of the variables in our case study and their appropriate resolutions.

4.1.1 Brake Pressure

There are three categories of features that modify the controlled variable

brake.hydraulicPressure

. The Automatic Braking (AB) feature enhances driver actions by detecting when the driver applies a large amount of pressure to the brake pedal. It optimizes *hydraulicPressure* to minimize stopping distance. This feature falls into the first category of driver-related behaviour. The second category includes safety or stability features such as Trailer Stability (TS) and Stability Control (SC). These features apply different levels of brake pressure to each of the four wheels to keep the vehicle on a straight path and counteract vehicle oscillation, respectively. The third category of features maintains driver-set acceleration preferences or provides feedback to the driver. These are grouped into a non-safety feature category.

We give the highest priority to safety and stability features. There are two ways in which safety features apply brake pressure. Some safety features reduce vehicle speed by

applying an even application of brake pressure to all four wheels. Other safety features apply brake pressure unevenly with the goal of controlling vehicle oscillation or stability. Our resolution differentiates between these cases by observing the monitored variable *car.oscillation*. If *car.oscillation* is less than a low threshold value, the resolution sets *brake.hydralicPressure* to the maximum of all assignment values. If it is greater than the threshold, we assume that multiple stability features are working to correct the oscillation of the vehicle and our resolution sets *brake.hydralicPressure* to the average of all assignment values.

The precondition axiom for the resolution is

$$
\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstSafety(l, s) \wedge ((maximum(l, v) \wedge \\
&\qquad car.oscillation(s) < threshold) \vee (car.oscillation(s) \geq \\
&\qquad threshold \wedge average(l, v))))\vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \rightarrow empty(l)) \wedge \\
&\qquad (\exists l_2.assignRqstDriver(l_2, s) \wedge maximum(l_2, v)))\vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \vee assignRqstDriver(l, s) \\
&\qquad \rightarrow empty(l)) \wedge (\exists l_3.assignRqstNonSafety(l_3, s) \\
&\qquad \wedge maximum(l_3, v)))
\end{aligned}
$$

The GOLOG program for the resolution module is

proc resolve $\pi v.$ assign(v)

This outputs a sequence of one assignment, which adheres to the restrictions stated in the precondition axiom.

4.1.2 Warning Chime

The controlled variable *alert.chimeType* is used to alert the driver by features such as Cruise Control (CC), Basic Braking (BB), Parking Brake (PB), Manual Park Brake (MPB), and Road Change Alert (RCA). As the primary purpose of this variable is to capture the driver's attention, we operate under the assumption that values for this variable can be ranked from less to more urgent. We define the function $rankList(L)$ that takes a list of values and returns a corresponding list of rankings, and we define the function $getType(x)$

that returns the chime type that corresponds to the ranking x. The inputs to the resolution are encoded in the predicate $assignRqst([\ldots], s_c)$.

The resolution will set $alert.chimeType$ to the most urgent chime value. The precondition axiom is

$$Poss(assign(v), s) \equiv \exists l.assignRqst(l, s) \wedge r = rankList(l) \\ \wedge maximum(r, x) \wedge v = getType(x)$$

Similar to the example above, the GOLOG program outputs one assignment:

proc resolve $\pi v.$ assign(v)

4.1.3 Air Flow Rate

There are several features that control cabin temperature and air quality. Each of these features modifies the variable $cabin.airFlowRate$. The Air Quality System (AQS) circulates air to reduce pollution levels, Air Conditioning (AC) and Heater Control (HC) use air flow to circulate cooler or warmer temperatures, and Air Recirculation (AR) recirculates air at the driver's request.

We prioritize features that circulate air to improve air quality over features that circulate air to improve the air temperature. The input actions from air-quality features are encoded in the predicate $assignRqstQuality([\ldots], s_c)$. Inputs from air-temperature features are represented as
$assignRqstTemp([\ldots], s_c)$. In both cases, we set the air flow rate to be the maximum assigned value. The precondition axiom for this variable is

$$Poss(assign(v), s) \equiv \\ (\exists l.assignRqstQuality(l, s) \wedge maximum(l, v)) \vee \\ (\forall l.(assignRqstQuality(l, s) \rightarrow empty(l)) \wedge \\ \exists l_2.assignRqstTemp(l_2, s) \wedge maximum(l_2, v))$$

We also make use of the simple GOLOG program

proc resolve $\pi v.$ assign(v)

4.2 Discussion

One goal of our resolution approach is to provide the modeller with a language that is powerful enough to express resolution functions that are tailored to fit the domain and the behaviour of each controlled variable of a system. In general, situation calculus can be used to refer to the current values of all monitored variables and to features' assignment expressions, and can be used to compute relatively sophisticated resolutions. If needed, the developer can define additional first-order-logic functions and predicates that are helpful to reason about input actions or specify output actions. The purpose of our case study was to gauge the expressiveness of our approach by specifying variable-specific resolutions for a diverse set of controlled variables. We found that the controlled variables in our case study called for unique variable-specific resolutions, and we were able to express all desired resolutions.

Situation calculus is capable of expressing win/win resolutions, where input is taken from all features and not just those that have the highest priority in the system. In the case of the steering example, all features contribute to the value of the controlled variable *car.steering*. Thus, all features "win" in that they all contribute to the value of this variable.

One of the major advantages of our resolution is the absence of a required priority scheme among features. The developer still has the option to define priorities between groups of features, as we did in our examples by grouping of features into safety and non-safety categories. However, this classification of features into groups is easier than identifying a total or partial ordering on all features. When adding new features, or removing existing features from the system, the developer does not need to reassess the prioritization. She need only determine to which category the new feature belongs. Classification decisions do not need to be revised as new features are added to the system.

Additionally, the developer does not need to know how many features modify each controlled variable. Our resolution approach is agnostic to the number of features in the system as well as the number of feature interactions that arise from their composition. The inputs to the resolution modules are lists of arbitrary size. Thus, features can be developed independently and can be added to the system incrementally. If a feature introduces a new type of controlled variable, the developer does need to introduce a new resolution module. The work involved in specifying a new resolver includes determining the appropriate inputs, writing the situation-calculus precondition axiom and any necessary successor state axioms, and also satisfying the three proof obligations for determinism, totality, and preservation of original behaviour that are given in the beginning of this chapter. However, our case

study suggests that the introduction of new controlled variables are rare; we discovered a total of only 8 controlled variables, 6 of which are modified by more than one feature, in a group of 24 automotive features. In this way, our approach preserves the advantages of feature-oriented software development. Features can still be developed independently and added and evolved incrementally.

The focus on controlled variables allows all feature interactions due to conflicts to be resolved with a linear amount of work in the number of types of controlled variables that are modified by multiple features. At composition time, a simple static analysis can be performed on each controlled variable to determine whether the behavioural specification of more than one feature includes a world-change action to modify this variable. If it is modified by more than one variable, the possibility of a conflict interaction exists. Then, the developer need specify only one resolution module per type of susceptible controlled variable, which will be instantiated every time an instance of that controlled variable is added to the world through a world-change action. As we saw in the case study, new types of controlled variables appear to be introduced at a much slower rate than features in the system; however, further studies should be conducted to provide a more comprehensive evaluation. This addresses the scalability aspect of the Feature Interaction Problem.

Some of the resolutions we considered suggest that there are dependencies among controlled variables. For example, *brake.hydralicPressure* affects *car.acceleration*. Our resolution does not consider dependencies between controlled variables and therefore does not address feature interactions that arise due to these dependencies. We conjecture that such interactions could be addressed by clustering related variables and resolving their conflicts or by imposing a partial ordering on controlled variables and using the resolutions of some variables as inputs to the resolver modules of others. We leave these investigations to future work.

4.2.1 Threats to Validity

The requirements documents on which we base our resolutions provide information on a subset of automotive features. We specified what we considered to be appropriate resolutions to conflicting assignments made by these features. It is possible that other features could modify the same variables in a way that would warrant a different resolution strategy. This would weaken our claim that the addition of features does not impact the resolution strategies. These claims should be validated with future case studies.

This case study was conducted solely on an automotive domain. We can conclude that the behaviour of each controlled variable considered required a unique resolution.

It is possible that resolutions in another domain would be more difficult to express in our resolution language. The conclusions we draw about the variation in appropriate resolutions across controlled variables, as well as the claim that controlled variables are introduced at a much slower rate than the rate of new features should be backed up with future case studies on more comprehensive collections of features and across different domains.

Additionally, the resolutions presented in the previous section were interpreted and written by the author of this thesis. To ensure that no bias was introduced to conform the behaviour of these controlled variables to something expressible in our resolution language, more case studies should be conducted across different domains by different developers.

Chapter 5

Related Work

5.1 Priority-Based Resolutions

The majority of related work on resolving feature interactions relies on a priority ranking among features [11, 14, 18, 20, 22]. Priority-based approaches need a total or partial ordering on features to support the resolution strategy. When a new feature is developed, its place in the priority ordering must be determined, making it difficult to add new features. In the case of a conflict, only the actions of the highest-priority feature are executed, blocking the behaviour of all other features. Our resolution considers all enabled actions, regardless of feature priority.

Some priority-based approaches offer finer-grained resolutions. Laney et al. [26, 27] propose resolutions in which priorities are considered at the granularity of individual feature requirements. During feature composition, the developer specifies which aspects of a feature's behaviour may be relaxed in the event of a conflict. Interactions are resolved on a case-by-case basis. Thus, this approach does not address the feature interaction problem: the number of interactions to consider, resolve, and verify is potentially exponential in the number of features. In addition, the resolutions are win/lose in that only the highest-priority requirements are satisfied in case of a conflict.

5.2 Precedence-Based Resolutions

Precedence-based resolution strategies [3, 9, 21], in which features are executed in a specified order, display similar problems to priority-based approaches. Features are given a

total or partial precedence ordering, and the task of determining a precedence order for n features requires that the developer consider up to $n!$ orderings.

Some work has been done to mitigate the task of specifying priorities or precedences among large collections of features by categorizing features [40] and using automated detection of feature interactions to find acceptable orderings [40]. However, these approaches still suffer from course-grained resolutions based on feature priorities and offer only win/lose resolutions to feature interactions.

5.3 Negotiation-Based Resolutions

Griffeth and Velthuijsen reduce a developer's work by resolving conflicts through automated negotiation [16]. The general idea behind negotiation-based resolution is to offer alternative feature behaviours in the event of a conflict, to maintain the essential intent of the feature developer. This approach has been applied to multi-agent systems [33] using situation calculus as the action language. Negotiation requires multiple rounds of communication between negotiating agents that act on the behalf of features. Many safety-critical systems have strict timing requirements and cannot afford of multiple rounds of communication. Our approach resolves interactions in a single multi-phase execution step, by calculating variable-specific resolutions to conflicting assignments. These calculations are fast and each resolver is independent, so all resolvers may execute in parallel. Furthermore, features themselves do not need to interact with each other. This promotes feature modularity and obliviousness — key attributes of feature-oriented software development.

Finkelstein et al. propose a strategy for negotiating inconsistencies in software by deferring judgement to the user [13]. The reasoning behind this action is that a user has their own perspective and expectations as to how a product should behave, and can use their judgement as a run-time resolution strategy. This approach is infeasible in the automotive domain, as feature interactions occur and require resolutions too quickly to involve user input.

5.4 Undoing Conflicts

A run-time technique to resolve feature interactions was introduced by Marples et al. [29], to rollback, or undo, previous actions when the system detects that multiple features are responding to the same stimulus. When this occurs, a Feature Manager explores all

possible resolutions by sending and receiving messages to and from all features to reach a stable, conflict-free state. However, the exploration of all possible resolutions and the message passing between features can be a lengthy process; this approach is not suitable for systems with strict timing requirements. Additionally, if features fail to participate in the negotiation, they are terminated from the system.

Some approaches prevent the activation of low-priority features, or terminate the lowest priority feature involved in a conflict [20]. This removes undesired feature interactions by undoing any behaviour that feature contributed to the system. Our approach allows all features to remain in the system, and considers the actions they output, regardless of their priority-level.

Alma Juarez-Dominguez mentions in her work another resolution strategy related to relaxed requirements that only considers feature interactions due to conflicts only if the values of the conflicting variable differ by a certain threshold [12]. This strategy reduces the number of interactions that need to be dealt with by ignoring those that pose very little trouble to the behaviour of the system. Appropriate threshold values depend on the variable being modified and the actuators in the system. This specification is left to the domain expert and is, similar to our approach, linear in the number of actuators.

Chapter 6

Conclusions

In this thesis, we have presented an approach for resolving feature interactions that addresses key aspects of the Feature Interaction Problem by providing means for developers to specify an appropriate resolution strategy for each controlled variable rather than for each possible feature interaction.

We described a language necessary for expressing the inputs and outputs of each resolution module and provided an implementation in situation calculus and GOLOG. Additionally, we defined proof obligations for the developer, to ensure that each resolution is deterministic, total, and preserves the behaviour of feature modules in the absence of feature interactions.

We showed that the desired resolution for a controlled variable depends on the roles that the variable plays in overall system behaviour. Our approach allows for resolution strategies that are tailored to the specifics of each controlled variable. We provided evidence, in the form of a case study that different controlled variables warrant different resolution strategies.

Existing approaches to resolving feature interactions lie at two extremes. Resolving every feature interaction individually provides detailed and appropriate resolutions, but poses a scalability problem as more features are introduced into the system. Devising one resolution strategy to handle all feature interactions addresses the scalability issue, but results in sub-optimal resolutions. Our approach provides an alternative to these extremes by addressing the scalability problem and enabling the developer to specify detailed resolutions. The work involved in resolving feature interactions is linear in the number of types of controlled variables.

This approach makes possible resolutions that are not win/lose. The actions from all features are considered when determining the resolution for feature interactions. Thus, the developer has more information to work with and more flexibility when deciding the course of action in the event many features are trying to modify the same controlled variable.

Finally, this approach preserves the advantages of feature-oriented software development by allowing for conflict-free feature composition. The developer does not need to be aware of the number of features in the system. This eases the task of adding and removing features as well as supports feature modularity and obliviousness.

Appendix A

Proofs

This appendix contains the details for the proofs outlined in Chapter 4. We use the soundness property of first and second-order logic to provide natural deduction proofs for the entailments that follow. Our natural deduction proofs refer to the following inference rules, taken from [39].

(Ref) $\Sigma, A \vdash A$

($\neg_E$) If $\Sigma, \neg A \vdash B$ and $\Sigma, \neg A \vdash \neg B$, then $\Sigma \vdash A$.

($\neg_I$) If $\Sigma, A \vdash B$ and $\Sigma, A \vdash \neg B$, then $\Sigma \vdash \neg A$.

($\supset_E$) If $\Sigma \vdash A \supset B$ and $\Sigma \vdash A$, then $\Sigma \vdash B$.

($\supset_I$) If $\Sigma, A \vdash B$, then $\Sigma \vdash A \supset B$.

($\wedge_E$) If $\Sigma \vdash A \wedge B$, then $\Sigma \vdash A$ and $\Sigma \vdash B$.

($\wedge_I$) If $\Sigma \vdash A$ and $\Sigma \vdash B$, then $\Sigma \vdash A \wedge B$.

($\vee_E$) If $\Sigma, A \vdash C$ and $\Sigma, B \vdash C$, then $\Sigma, A \vee B \vdash C$.

($\vee_I$) If $\Sigma \vdash A$, then $\Sigma \vdash A \vee B$ and $\Sigma \vdash B \vee A$.

($\equiv_E$) If $\Sigma \vdash A \equiv B$, then $\Sigma \vdash A \supset B$ and $\Sigma \vdash B \supset A$.

($\forall_E$) If $\Sigma \vdash \forall x.A(x)$, then $\Sigma \vdash A(t)$.

($\forall_I$) If $\Sigma \vdash A(u)$ where u does not occur in Σ, then $\Sigma \vdash \forall x.A(x)$.

($\exists_E$) If $\Sigma, A(u) \vdash B$ where u does not occur in Σ or in B, then $\Sigma, \exists x.A(x) \vdash B$.

($\exists_I$) If $\Sigma \vdash A(t)$, then $\Sigma \vdash \exists x.A(x)$ where $A(x)$ is the result of replacing some occurences of t in $A(t)$ with x.

($=_E$) If $\Sigma \vdash A(t_1)$ and $\Sigma \vdash t_1 = t_2$, then $\Sigma \vdash A(t_2)$ where $A(t_2)$ is the result of replacing some occurences of t_1 in $A(t_1)$ with t_2

($=_I$) $\Sigma \vdash u = u$

We use the following domain-independent axioms in $\mathcal{D}$:

$Ax.1.\ \forall a, s_0.\exists s.s = do(a, s_0)$
$Ax.2.\ \forall a, v, v'.\neg(v = v') \supset \neg(a(v) = a(v'))$
$Ax.3.\ \forall s, a, s'.\neg(a = a') \supset \neg(do(a, s) = do(a', s))$

Theorem 2. *Given a resolution domain theory $\mathcal{D}$ in the situation calculus and a Golog resolution procedure δ, the following restriction on the precondition axiom in $\mathcal{D}$:*

$$\textbf{Ob 1.}\ \forall v_1.\forall v_2.\forall s.Poss(assign(v_1), s) \wedge Poss(assign(v_2), s) \supset v_1 = v_2$$

is a necessary and sufficient condition for the following entailment:

$$\mathcal{D} \models \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n) \supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n)$$

Proof. We first show this is a sufficient condition with an inductive proof on the structure of the Golog resolution procedure δ. The inductive definition of $Do(\delta, s_c, s_n)$ is given in Section 3.2.6.

Base Case

- Primitive action $\delta = a$

$$Do(a, s_c, s_n) \stackrel{def}{=} Poss(a, s_c) \wedge s_n = do(a, s_c)$$
$$Do(a, s_c, s'_n) \stackrel{def}{=} Poss(a, s_c) \wedge s'_n = do(a, s_c)$$

To prove

$$\mathcal{D} \models \forall s_c, s_n, s'_n.Do(a, s_c, s_n) \supset (Do(a, s_c, s'_n) \supset s_n = s'_n)$$

we give a natural deduction proof of the following:

$$\mathcal{D} \vdash \forall s_c, s_n, s'_n.(Poss(a, s_c) \wedge s_n = do(a, s_c)) \supset (Poss(a, s_c) \wedge s'_n = do(a, s_c) \supset s_n = s'_n)$$

1. $\mathcal{D}, (Poss(a, s_c) \wedge s_n = do(a, s_c)), (Poss(a, s_c) \wedge s'_n = do(a, s_c)) \vdash s_n = do(a, s_c)$ $\wedge_E$
2. $\mathcal{D}, (Poss(a, s_c) \wedge s_n = do(a, s_c)), (Poss(a, s_c) \wedge s'_n = do(a, s_c)) \vdash s'_n = do(a, s_c)$ $\wedge_E$
3. $\mathcal{D}, (Poss(a, s_c) \wedge s_n = do(a, s_c)), (Poss(a, s_c) \wedge s'_n = do(a, s_c)) \vdash s'_n = s_n$ $=_I$
4. $\mathcal{D}, (Poss(a, s_c) \wedge s_n = do(a, s_c)) \vdash (Poss(a, s_c) \wedge s'_n = do(a, s_c)) \supset s'_n = s_n$ $\supset_I$
5. $\mathcal{D} \vdash (Poss(a, s_c) \wedge s_n = do(a, s_c)) \supset ((Poss(a, s_c) \wedge s'_n = do(a, s_c)) \supset s'_n = s_n)$ $\supset_I$
6. $\mathcal{D} \vdash \forall s_c, s_n, s'_n.(Poss(a, s_c) \wedge s_n = do(a, s_c)) \supset ((Poss(a, s_c) \wedge s'_n = do(a, s_c)) \supset s'_n = s_n)$ $\forall_I$

Inductive Step: Assume the programs δ_1 and δ_2 satisfy the properties

$$P1.\ \mathcal{D} \models \forall s_c, s_n, s'_n.Do(\delta_1, s_c, s_n) \supset (Do(\delta_1, s_c, s'_n) \supset s_n = s'_n)$$
$$P2.\ \mathcal{D} \models \forall s_c, s_n, s'_n.Do(\delta_2, s_c, s_n) \supset (Do(\delta_2, s_c, s'_n) \supset s_n = s'_n)$$

- Sequence $\delta = \delta_1; \delta_2$

$$Do(\delta_1; \delta_2, s_c, s_n) \stackrel{def}{=} \exists s_i.Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)$$
$$Do(\delta_1; \delta_2, s_c, s'_n) \stackrel{def}{=} \exists s'_i.Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)$$

To prove

$$\mathcal{D} \models \forall s_c, s_n, s'_n.Do(\delta_1; \delta_2, s_c, s_n) \supset (Do(\delta_1; \delta_2, s_c, s'_n) \supset s_n = s'_n)$$

we give a natural deduction proof of the following:

$\mathcal{D} \vdash \forall s_c, s_n, s'_n.(\exists s_i.Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)) \supset ((\exists s'_i.Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \supset s_n = s'_n)$

1. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_1, s_c, s_i)$ $\wedge_E$
2. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_1, s_c, s'_i)$ $\wedge_E$
3. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_1, s_c, s_i) \supset (Do(\delta_1, s_c, s'_i) \supset s_i = s'_i)$ $P1$
4. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash s_i = s'_i$ $\supset_E$
5. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_2, s_i, s_n)$ $\wedge_E$
6. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_2, s'_i, s'_n)$ $\wedge_E$
7. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_2, s_i, s'_n)$ $=_E$
8. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash Do(\delta_2, s_i, s_n) \supset (Do(\delta_2, s_i, s'_n) \supset s_n = s'_n)$ $P2$
9. $\mathcal{D}, (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)), (Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \vdash s_n = s'_n$ $\supset_E$
10. $\mathcal{D} \vdash (Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)) \supset ((Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \supset s_n = s'_n)$ $\supset_I$
11. $\mathcal{D} \vdash \forall s_c, s_n, s'_n.(Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n)) \supset ((Do(\delta_1, s_c, s'_i) \wedge Do(\delta_2, s'_i, s'_n)) \supset s_n = s'_n)$ $\forall_I$

- Nondeterministic choice of action arguments $\delta = \pi v.\delta$

$$Do(\pi v.\delta, s_c, s_n) \stackrel{def}{=} \exists v.Do(\delta(v), s_c, s_n)$$
$$Do(\pi v.\delta, s_c, s'_n) \stackrel{def}{=} \exists v.Do(\delta(v), s_c, s'_n)$$

Here, the notation $\delta(v)$ stands for the application of the argument v to one or more primitive actions or conditions in δ. Therefore, to prove

$$\mathcal{D} \models \forall s_c, s_n, s'_n.Do(\pi v.\delta(v), s_c, s_n) \supset (Do(\pi v.\delta(v), s_c, s'_n) \supset s_n = s'_n)$$

it suffices to prove the following two formulas for all actions a and all conditions ϕ,

$$\mathcal{D} \models \forall s_c, s_n, s'_n.(\exists v.Do(a(v), s_c, s_n)) \supset ((\exists v.Do(a(v), s_c, s'_n)) \supset s_n = s'_n) \quad \text{(A.1)}$$
$$\mathcal{D} \models \forall s_c, s_n, s'_n.(\exists v.Do(\phi(v)?, s_c, s_n)) \supset ((\exists v.Do(\phi(v)?, s_c, s'_n)) \supset s_n = s'_n) \quad \text{(A.2)}$$

First, note that the only action we worry about in our domain $\mathcal{D}$ is $assign(v)$. So, to prove formula (6.1), we provide a natural deduction proof of the following:

$$\begin{aligned}\mathcal{D} \vdash &\forall s_c, s_n, s'_n.(\exists v.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)) \\ &\supset ((\exists v.Poss(assign(v), s_c) \wedge s'_n = do(assign(v), s_c)) \supset s_n = s'_n)\end{aligned}$$

1. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c)$ *Ref*
2. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash Poss(assign(v_1), s_c)$ $\wedge_E$
3. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash Poss(assign(v_2), s_c) \wedge s'_n = do(assign(v_2), s_c)$ *Ref*
4. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash Poss(assign(v_2), s_c)$ $\wedge_E$
5. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash \forall v_1.\forall v_2.\forall s.Poss(assign(v_1), s) \wedge$
   $Poss(assign(v_2), s) \supset v_1 = v_2$ **Ob 1**
6. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash Poss(assign(v_1), s_c) \wedge Poss(assign(v_2), s_c) \supset v_1 = v_2$ $\forall_E$
7. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
   $\wedge\, s'_n = do(assign(v_2), s_c) \vdash v_1 = v_2$ $\supset_E$

8. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
$\wedge\, s'_n = do(assign(v_2), s_c) \vdash s_n = do(assign(v_1), s_c)$ $\wedge_E$
9. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
$\wedge\, s'_n = do(assign(v_2), s_c) \vdash s'_n = do(assign(v_2), s_c)$ $\wedge_E$
10. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
$\wedge\, s'_n = do(assign(v_2), s_c) \vdash s'_n = do(assign(v_1), s_c)$ $=_E$
11. $\mathcal{D}, Poss(assign(v_1), s_c) \wedge s_n = do(assign(v_1), s_c), Poss(assign(v_2), s_c)$
$\wedge\, s'_n = do(assign(v_2), s_c) \vdash s_n = s'_n$ $=_I$
12. $\mathcal{D}, \exists v.(Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)), (\exists v.Poss(assign(v), s_c)$
$\wedge\, s'_n = do(assign(v), s_c)) \vdash s_n = s'_n$ $\exists_E$
13. $\mathcal{D} \vdash (\exists v.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)) \supset$
$(\exists v.(Poss(assign(v), s_c) \wedge s'_n = do(assign(v), s_c)) \supset s_n = s'_n)$ $\supset_I$
14. $\mathcal{D} \vdash \forall s_c, s_n, s'_n.(\exists v.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)) \supset$
$(\exists v.(Poss(assign(v), s_c) \wedge s'_n = do(assign(v), s_c)) \supset s_n = s'_n)$ $\forall_I$

Now, we prove formula (6.2) by showing a natural deduction proof of

$$\mathcal{D} \vdash \forall s_c, s_n, s'_n.(\exists v.\phi(v)[s_n] \wedge s_c = s_n) \supset ((\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \supset s_n = s'_n)$$

1. $\mathcal{D}, (\exists v.\phi(v)[s_n] \wedge s_c = s_n), (\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \vdash \phi(v_1)[s_n] \wedge s_c = s_n$ $\exists_E$
2. $\mathcal{D}, (\exists v.\phi(v)[s_n] \wedge s_c = s_n), (\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \vdash s_c = s_n$ $\wedge_E$
3. $\mathcal{D}, (\exists v.\phi(v)[s_n] \wedge s_c = s_n), (\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \vdash \phi(v_2)[s'] \wedge s_c = s'_n$ $\exists_E$
4. $\mathcal{D}, (\exists v.\phi(v)[s_n] \wedge s_c = s_n), (\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \vdash s_c = s'_n$ $\wedge_E$
5. $\mathcal{D}, (\exists v.\phi(v)[s_n] \wedge s_c = s_n), (\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \vdash s_n = s'_n$ $=_E$
6. $\mathcal{D} \vdash (\exists v.\phi(v)[s_n] \wedge s_c = s_n) \supset ((\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \supset s_n = s'_n)$ $\supset_I$
7. $\mathcal{D} \vdash \forall s_c, s_n, s'_n(\exists v.\phi(v)[s_n] \wedge s_c = s_n) \supset ((\exists v.Poss(\phi(v)[s'_n] \wedge s_c = s'_n)) \supset s_n = s'_n)$ $\forall_I$

- Conditional $\delta =$ **if** ϕ **then** δ_1 **else** δ_2

$$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s_n) \stackrel{def}{=} Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)$$
$$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s'_n) \stackrel{def}{=} Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n)$$

To prove

$\mathcal{D} \models \forall s_c, s_n, s'_n. Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s_n) \supset (Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s_c, s'_n) \supset s_n = s'_n)$

we give a natural deduction proof of the following.

$$\mathcal{D} \vdash \forall s_c, s_n, s'_n. Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n) \supset (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n) \supset s_n = s'_n)$$

1. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash \neg\neg\phi[s_c]$ $\neg_I$
2. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash \neg(\neg\phi[s_c] \wedge s_i = s_c)$ $\neg_I$
3. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash \neg(Do(\neg\phi?; \delta_2, s_c, s_n))$ $\neg_I$
4. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash Do(\phi?; \delta_1, s_c, s_n)$ $\vee_E$
5. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash \neg(Do(\neg\phi?; \delta_2, s_c, s'_n))$ $\neg_I$
6. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash Do(\phi?; \delta_1, s_c, s'_n)$ $\vee_E$
7. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash Do(\phi?; \delta_1, s_c, s_n) \supset (Do(\phi?; \delta_1, s_c, s_n) \supset s_n = s'_n)$ Seq
8. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $, \phi[s_c] \vdash s_n = s'_n$ $\supset_E$
9. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
   $\vdash \phi[s_c] \supset s_n = s'_n$ $\supset_E$
10. $\mathcal{D}, (Do(\phi?; \delta_1, s_c, s_n) \vee Do(\neg\phi?; \delta_2, s_c, s_n)), (Do(\phi?; \delta_1, s_c, s'_n) \vee Do(\neg\phi?; \delta_2, s_c, s'_n))$
    $, \neg\phi[s_c] \vdash \neg\phi[s_c]$ $\neg_I$

11. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash \neg(\phi[s_c] \wedge s_i = s_c)$ $\neg_I$

12. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash \neg(Do(\phi?;\delta_1,s_c,s_n))$ $\neg_I$

13. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash Do(\neg\phi?;\delta_2,s_c,s_n)$ $\vee_E$

14. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash \neg(Do(\phi?;\delta_1,s_c,s'_n))$ $\neg_I$

15. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash Do(\neg\phi?;\delta_2,s_c,s'_n)$ $\vee_E$

16. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash Do(\neg\phi?;\delta_2,s_c,s_n) \supset (Do(\neg\phi?;\delta_2,s_c,s_n) \supset s_n = s'_n)$ Seq

17. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$, \neg\phi[s_c] \vdash s_n = s'_n$ $\supset_E$

18. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$\vdash \neg\phi[s_c] \supset s_n = s'_n$ $\supset_E$

19. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$\vdash \neg\phi[s_c] \vee \phi[s_c] \supset s_n = s'_n$ $\supset_E$

20. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$\vdash \neg\phi[s_c] \vee \phi[s_c]$ $\vee_I$

21. $\mathcal{D}, (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)), (Do(\phi?;\delta_1,s_c,s'_n) \vee Do(\neg\phi?;\delta_2,s_c,s'_n))$
$\vdash s_n = s'_n$ $\vee_I$

22. $\mathcal{D} \vdash (Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)) \supset ((Do(\phi?;\delta_1,s_c,s'_n) \vee$
$Do(\neg\phi?;\delta_2,s_c,s'_n)) \supset s_n = s'_n)$ $\supset_I$

23. $\mathcal{D} \vdash \forall s_c,s_n,s'_n.(Do(\phi?;\delta_1,s_c,s_n) \vee Do(\neg\phi?;\delta_2,s_c,s_n)) \supset ((Do(\phi?;\delta_1,s_c,s'_n) \vee$
$Do(\neg\phi?;\delta_2,s_c,s'_n)) \supset s_n = s'_n)$ $\forall_I$

- While loop $\delta =$ **while** ϕ **do** δ_1

$$Do(\delta = \textbf{while } \phi \textbf{ do } \delta_1, s_c, s_n) \stackrel{def}{=} (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n])$$

$$Do(\delta = \textbf{while } \phi \textbf{ do } \delta_1, s_c, s'_n) \stackrel{def}{=} (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s'_n]$$

To prove

$$\mathcal{D} \models \forall s_c, s_n, s'_n.Do(\textbf{while } \phi \textbf{ do } \delta_1, s_c, s_n) \supset (Do(\textbf{while } \phi \textbf{ do } \delta_1, s_c, s'_n) \supset s_n = s'_n)$$

we give a natural deduction proof of the following:

$$\begin{aligned}\mathcal{D} \vdash \forall s_c, s_n, s'_n.(&\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \\ &\supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n] \supset ((\forall P.(\forall s_1.P(s_1,s_1)) \\ &\wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \\ &\wedge \neg\phi[s'_n])) \supset s_n = s'_n)\end{aligned}$$

In this proof, we make use of the binary relation R, which we characterize as follows:

$$R(s_1,s_3) \equiv (\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3)) \qquad (Def_R)$$

1. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash R(s_1,s_3) \equiv (\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3))$ $\quad Def_R$
2. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash (\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3)) \supset R(s_1,s_3)$ $\quad \equiv_E$
3. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n))$ $\quad \wedge_E$

4. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash (\forall s_1.R(s_1,s_1))$
$\wedge (\forall s_2.R(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset R(s_1,s_3)) \supset R(s_c,s_n))$ $\forall_E$
5. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash R(s_c,s_n)$ $\supset_E$
6. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash (\forall P.(\forall s_1.P(s_1,s_1))$
$\wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n'))$ $\wedge_E$
7. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash (\forall s_1.R(s_1,s_1))$
$\wedge (\forall s_2.R(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset R(s_1,s_3)) \supset R(s_c,s_n'))$ $\forall_E$
8. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash R(s_c,s_n')$ $\supset_E$
9. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash R(s_c,s_n) \equiv$
$(\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_n))$ Def_R
10. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash R(s_c,s_n) \supset$
$(\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_n))$ $\equiv_E$
11. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3))$
$\supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2)$
$\wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash (\forall s_1.$
$R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_n))$ $\supset_E$

12. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash (\forall s_2. R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_n))$ $\wedge_E$

13. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash R(s_c,s_i) \wedge \phi[s_i] \wedge Do(\delta_1,s_i,s_n))$ $\forall_E$

14. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash Do(\delta_1,s_i,s_n))$ $\wedge_E$

15. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash R(s_c,s'_n) \equiv (\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s'_n))$ Def_R

16. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash R(s_c,s'_n) \supset (\forall s_1.R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s'_n))$ $\equiv_E$

17. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash (\forall s_1. R(s_1,s_1)) \wedge (\forall s_2.R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s'_n))$ $\supset_E$

18. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash (\forall s_2. R(s_c,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s'_n))$ $\wedge_E$

19. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s'_n)) \wedge \phi[s'_n] \vdash R(s_c,s_i) \wedge \phi[s_i] \wedge Do(\delta_1,s_i,s'_n))$ $\forall_E$

20. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash Do(\delta_1,s_i,s_n'))$ $\wedge_E$

21. $\mathcal{D}, (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n], (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \vdash s_n = s_n'$ P_1

22. $\mathcal{D} \vdash (\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n)) \wedge \neg\phi[s_n] \supset ((\forall P.(\forall s_1.P(s_1,s_1)) \wedge (\forall s_1,s_2,s_3.P(s_1,s_2) \wedge \phi[s_2] \wedge Do(\delta_1,s_2,s_3) \supset P(s_1,s_3)) \supset P(s_c,s_n')) \wedge \phi[s_n'] \supset s_n = s_n')$ $\supset_I$

Thus, for any resolution procedure δ, the restriction **Ob 1** on the precondition axiom is sufficient to ensure the resolution module is deterministic.

We now show that this is a necessary condition. Consider the resolution procedure $\delta = \pi v.assign(v)$. Assume the obligation

$$\forall v_1.\forall v_2.\forall s.Poss(assign(v_1),s) \wedge Poss(assign(v_2),s) \supset v_1 = v_2$$

does not hold in $\mathcal{D}$. Note that by de Morgan's law,

$$\neg(\forall v_1.\forall v_2.\forall s.Poss(assign(v_1),s) \wedge Poss(assign(v_2),s) \supset v_1 = v_2) \equiv \exists v_1.\exists v_2.\exists s.(Poss(assign(v_1),s) \wedge Poss(assign(v_s),s)) \wedge \neg(v_1 = v_2)$$

We show the entailment

$$\mathcal{D} \models \forall s_c,s_n,s_n'.Do(\delta,s_c,s_n) \supset (Do(\delta,s_c,s_n') \supset s_n = s_n')$$

will not hold by providing a natural deduction proof of the following:

$$\mathcal{D}, \exists v_1.\exists v_2.\exists s_n.(Poss(assign(v_1),s_n) \wedge Poss(assign(v_2),s_n)) \wedge \neg(v_1 = v_2) \vdash \neg(\forall s_c,s_n,s_n'.Do(\delta,s_c,s_n) \supset (Do(\delta,s_c,s_n') \supset s_n = s_n')$$

1. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Poss(assign(v_1), s) \wedge s_1 = do(assign(v_1), s)$ $\wedge_I$
2. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Do(assign(v_1), s, s_1)$ Def
3. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Poss(assign(v_2), s) \wedge s_2 = do(assign(v_2), s)$ $\wedge_I$
4. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Do(assign(v_2), s, s_2)$ Def
5. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Do(\delta, s, s_1) \supset (Do(\delta, s, s_2) \supset s_1 = s_2$ $\forall_E$
6. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash Do(\delta, s, s_2) \supset s_1 = s_2$ $\supset_E$
7. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash s_1 = s_2$ $\supset_E$
8. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash \neg(v_1 = v_2) \supset \neg(assign(v_1)) = assign(v_2))$ $Ax.2$
9. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash \neg(assign(v_1)) = assign(v_2))$ $\supset_E$
10. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_1 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash \neg(assign(v_1)) = assign(v_2))$
$\supset \neg(do(assign(v_1), s) = do(assign(v_2), s))$ $Ax.3$

11. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_2 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash \neg(do(assign(v_1), s) = do(assign(v_2), s))$ $\supset_E$

11. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_2 = do(assign(v_1), s), s_2 = do(assign(v_2), s), \forall s_c, s_n, s'_n.Do(\delta, s_c, s_n)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n) \vdash \neg(s_1 = s_2)$ $=_E$

12. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2),$
$s_2 = do(assign(v_1), s), s_2 = do(assign(v_2), s) \vdash \neg(\forall s_c, s_n, s'_n.Do(\delta, s_c, s)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n))$ $\neg_I$

13. $\mathcal{D}, (Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2) \vdash$
$s_2 = do(assign(v_1), s) \wedge s_2 = do(assign(v_2), s) \supset \neg(\forall s_c, s_n, s'_n.Do(\delta, s_c, s)$
$\supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n))$ $\supset_I$

14. $\mathcal{D}, \exists v_1.\exists v_2.\exists s.(Poss(assign(v_1), s) \wedge Poss(assign(v_2), s)) \wedge \neg(v_1 = v_2) \vdash$
$\neg(\forall s_c, s_n, s'_n.Do(\delta, s_c, s_n) \supset (Do(\delta, s_c, s'_n) \supset s_n = s'_n))$ $\exists_E$

Thus, we know that the restriction is a necessary condition to ensure that all resolution procedures δ satisfy the proof obligation **Ob1**. □

Theorem 3. *Given a resolution domain theory $\mathcal{D}$ in the situation calculus and a Golog resolution procedure δ, if in $\mathcal{D}$, the following restriction is on the precondition axiom:*

$$\textbf{Ob 2.}\ \forall s.\,(\exists v.Poss(assign(v), s) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$$

is a necessary and sufficient condition for the following entailment:

$$\mathcal{D} \models \forall s_c.\,(\exists s_n.Do(\delta, s_c, s_n) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$$

unless δ contains one or more loops. In this case, for every loop of the form **while** ϕ **do** δ_1*, the following additional property must hold:*

$$\begin{aligned}\forall s.(\exists s'.(\forall P.(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge \phi[s_2] \\ \wedge Do(\delta_1, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s')) \wedge \neg\phi[s'] \\ \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))\end{aligned}$$

Proof. We first show this is a sufficient condition by inducting on the Golog resolution procedure δ.

Base Case

- Primitive action $\delta = a$

  To prove

  $$\mathcal{D} \models \forall s_c.\,(\exists s_n.Do(a, s_c, s_n) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$$

  we give a natural deduction proof of the following:

  $$\mathcal{D} \vdash \forall s_c.\,(\exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$$

  1. $\mathcal{D} \vdash \forall s.\,(\exists v.Poss(assign(v), s) \equiv \exists L.assignRqst(L, s) \wedge \neg empty(L))$ **Ob 2**
  2. $\mathcal{D} \vdash (\exists v.Poss(assign(v), s_c) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$ $\forall_E$
  3. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \vdash \exists v.Poss(assign(v), s_c)$ $\equiv_E$
  4. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c) \vdash \forall a, s_0.\exists s_n.s_n = do(a, s_0)$ $Ax.1$
  5. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c) \vdash \exists s_n.s_n = do(assign(v), s_c)$ $\forall_E$

6. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c), s_n = do(assign(v), s_c)$
$\vdash Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\wedge_I$
7. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c), s_n = do(assign(v), s_c)$
$\vdash \exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\exists_I$
8. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c), \exists s_n.s_n =$
$do(assign(v), s_c) \vdash \exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\exists_E$
9. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c) \vdash \exists s_n.s_n$
$= do(assign(v), s_c) \supset \exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\supset_I$
10. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), Poss(assign(v), s_c) \vdash$
$\exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\supset_E$
11. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), \exists v.Poss(assign(v), s_c) \vdash$
$\exists s_n.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\exists_E$
12. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \vdash \exists v.Poss(assign(v), s_c) \supset \exists s_n.$
$Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ $\supset_I$
13. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \vdash \exists s_n.Poss(assign(v), s_c) \wedge s_n$
$= do(assign(v), s_c)$ $\supset_E$
14. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \vdash \forall s_c.\exists s_n.Poss(assign(v), s_c) \wedge$
$s_n = do(assign(v), s_c)$ $\forall_I$
15. $\mathcal{D} \vdash \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \supset \forall s_c.\exists s_n.Poss(assign(v), s_c) \wedge$
$s_n = do(assign(v), s_c)$ $\supset_I$
16. $\mathcal{D}, \neg(\exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \neg(\exists v.Poss(assign(v), s_c))$ $\equiv_E$
17. $\mathcal{D}, \neg(\exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \neg(\exists s_n.s_n = Poss(assign(v), s_c) \wedge$
$s_n = do(assign(v), s_c))$ Def
18. $\mathcal{D} \vdash \exists L.assignRqst(L, s_c) \wedge \neg empty(L) \equiv \forall s_c.\exists s_n.Poss(assign(v), s_c) \wedge$
$s_n = do(assign(v), s_c)$ $\supset_I$

Inductive Step: Assume the programs δ_1 and δ_2 satisfy the properties

$P1.$ $\mathcal{D} \models \forall s_c.(\exists s_n.Do(\delta_1, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$
$P2.$ $\mathcal{D} \models \forall s_c.(\exists s_n.Do(\delta_2, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$

- Sequence $\delta = \delta_1; \delta_2$

To prove

$$\mathcal{D} \models \forall s_c.(\exists s_n.Do(\delta_1;\delta_2,s_c,s_n) \equiv \exists L.assignRqst(L,s_c) \wedge \neg empty(L))$$

We give a natural deduction proof of the following

$$\mathcal{D} \vdash \forall s_c.(\exists s_n.\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_2,s_i,s_n) \equiv \exists L.assignRqst(L,s_c) \wedge \neg empty(L))$$

1. $\mathcal{D} \vdash \forall s_c(\exists s_n.Do(\delta_1,s_c,s_n) \equiv \exists L.assignRqst(L,s_c) \wedge \neg empty(L))$ $P1$
2. $\mathcal{D} \vdash \exists s_n.Do(\delta_1,s_c,s_n) \equiv \exists L.assignRqst(L,s_c) \wedge \neg empty(L)$ $\forall_E$
3. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L) \vdash \exists s_n.Do(\delta_1,s_c,s_n)$ $\equiv_E$
4. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_c,s_i) \vdash \exists s_n.Do(\delta_1,s_i,s_n)$ $\forall_E$
5. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_c,s_i), Do(\delta_1,s_i,s) \vdash Do(\delta_1,s_c,s_i)$
   $\wedge Do(\delta_1,s_i,s_n)$ $\wedge_I$
6. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_c,s_i), Do(\delta_1,s_i,s_n) \vdash$
   $\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_1,s_i,s_n)$ $\exists_I$
7. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), \exists s_n.Do(\delta_1,s_c,s_n), Do(\delta_1,s_i,s_n) \vdash$
   $\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_1,s_i,s_n)$ $\exists_E$
8. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_i,s_n) \vdash \exists s.Do(\delta_1,s_c,s_n) \supset$
   $\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_1,s_i,s_n)$ $\supset_I$
9. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_i,s_n) \vdash \exists s_i.Do(\delta_1,s_c,s_i)$
   $\wedge Do(\delta_1,s_i,s_n)$ $\supset_E$
10. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), Do(\delta_1,s_i,s_n) \vdash \exists s_n.\exists s_i.Do(\delta_1,s_c,s_i)$
    $\wedge Do(\delta_1,s_i,s_n)$ $\exists_I$
11. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L), \exists s_n.Do(\delta_1,s_i,s_n) \vdash \exists s_n.\exists s_i.Do(\delta_1,s_c,s_i)$
    $\wedge Do(\delta_1,s_i,s_n)$ $\exists_E$
12. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L) \vdash \exists s_n.Do(\delta_1,s_i,s_n) \supset \exists s_n.\exists s_i.Do(\delta_1,s_c,s_i)$
    $\wedge Do(\delta_1,s_i,s_n)$ $\supset_I$
13. $\mathcal{D}, \exists L.assignRqst(L,s_c) \wedge \neg empty(L) \vdash \exists s_n.\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_1,s_i,s_n)$ $\supset_E$
14. $\mathcal{D} \vdash \exists L.assignRqst(L,s_c) \wedge \neg empty(L) \supset \exists s_n.\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_1,s_i,s_n)$ $\supset_I$
15. $\mathcal{D}, \neg(\exists L.assignRqst(L,s_c) \wedge \neg empty(L)) \vdash \neg(\exists s_n.Do(\delta_1,s_c,s_n))$ $\equiv_E$
16. $\mathcal{D}, \neg(\exists L.assignRqst(L,s_c) \wedge \neg empty(L)) \vdash \neg(\exists s_n.\exists s_i.Do(\delta_1,s_c,s_i) \wedge Do(\delta_2,s_i,s_n))$ Def

17. $\mathcal{D} \vdash \exists s_n.\exists s_i.Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L)$ $\equiv_I$
18. $\mathcal{D} \vdash \forall s_c.\exists s_n.\exists s_i.Do(\delta_1, s_c, s_i) \wedge Do(\delta_2, s_i, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L)$ $\forall_I$

- Nondeterministic choice of action arguments $\delta = \pi v.\delta_1$

  As above, since we only have one action that takes a single argument, to prove

  $$\mathcal{D} \models \forall s_c.(\exists s_n.Do(\pi v.\delta_1, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

  it suffices to prove

  $$\mathcal{D} \vdash \forall s_c.(\exists s_n.\exists v.Do(assign(v), s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

  This has the same proof as the base case step above.

- Conditional $\delta =$ **if** ϕ **then** δ_1 **else** δ_2

  To prove

  $\mathcal{D} \models \forall s_c.(\exists s_n.Do($**if** ϕ **then** δ_1 **else** $\delta_2, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$

  we give a natural deduction proof of:

  $\mathcal{D} \vdash \forall s_c.(\exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$
  $\equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$

  1. $\mathcal{D}, \phi[s_c], s_i = s_c \vdash \exists s_i.s_i = s_c$
  2. $\mathcal{D}, \phi[s_c], s_i = s_c \vdash \forall s_c.(\exists s_n.Do(\delta_1, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$ $P1$
  3. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c], s_i = s_c \vdash \exists s_n.Do(\delta_1, s_i, s_n)$ $\equiv_E$
  4. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c], s_i = s_c, Do(\delta_1, s_i, s_n) \vdash \phi[s_c] \wedge$
     $s_i = s_c \wedge Do(\delta_1, s_i, s_n)$ $\wedge_I$
  5. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c], s_i = s_c, Do(\delta_1, s_i, s_n) \vdash \exists s_i.\phi[s_c] \wedge$
     $s_i = s_c \wedge Do(\delta_1, s_i, s_n)$ $\exists_I$
  6. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c], s_i = s_c, Do(\delta_1, s_i, s_n) \vdash (\exists s_i.\phi[s_c] \wedge$
     $s_i = s_c \wedge Do(\delta_1, s_i, s_n))$
     $\vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\vee_I$
  7. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c], s_i = s_c, Do(\delta_1, s_i, s_n) \vdash \exists s_n.(\exists s_i.\phi[s_c]$
     $\wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n))$
     $\vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\exists_I$

8. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c] \vdash \exists s_i.s_i = s_c \supset \exists s_n.Do(\delta_1, s_i, s_n) \supset$
$\exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_I$
9. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c] \vdash \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge$
$Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_E$
10. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c \vdash \exists s_i.s_i = s_c$ $=_I$
11. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c \vdash \forall s_c.\exists s_n.Do(\delta_2, s_c, s_n)$ $P1$
12. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c \vdash \exists s_n.Do(\delta_2, s_i, s_n)$ $\forall_E$
13. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c, Do(\delta_2, s_i, s_n) \vdash \neg\phi[s_c] \wedge$
$s_i = s_c \wedge Do(\delta_2, s_i, s_n)$ $\wedge_I$
14. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c, Do(\delta_2, s_i, s_n) \vdash \exists s_i.\neg\phi[s_c] \wedge$
$s_i = s_c \wedge Do(\delta_2, s_i, s_n)$ $\exists_I$
15. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c, Do(\delta_2, s_i, s_n) \vdash (\exists s_i.\phi[s_c] \wedge$
$s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\vee_I$
16. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c], s_i = s_c, Do(\delta_2, s_i, s_n) \vdash \exists s_n.(\exists s_i.\phi[s_c]$
$\wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\exists_I$
17. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c] \vdash \exists s_i.s_i = s_c \supset \exists s_n.Do(\delta_2, s_i, s_n) \supset$
$\exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_I$
18. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \neg\phi[s_c] \vdash \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge$
$Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_E$
19. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)), \phi[s_c] \vee \neg\phi[s_c] \vdash \exists s_n.(\exists s_i.\phi[s_c] \wedge$
$s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\vee_E$
20. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \phi[s_c] \vee \neg\phi[s_c] \supset \exists s_n.(\exists s_i.\phi[s_c] \wedge$
$s_i = s_c \wedge Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_I$
21. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n))$
$\vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_E$
22. $\mathcal{D} \vdash \exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \supset \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n))$
$\vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n))$ $\supset_I$
23. $\mathcal{D}, \neg(\exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \neg(\exists s_n.Do(\delta_1, s_c, s_n))$ $\equiv_E$
24. $\mathcal{D}, \neg(\exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \vdash \neg(\exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge$
$Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n)))$ Def
25. $\mathcal{D} \vdash \exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \equiv \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_1, s_i, s_n))$
$\vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n)))$ $\equiv_I$
26. $\mathcal{D} \vdash \forall s_c.\exists L.assignRqst(L, s_c) \wedge \neg empty(L)) \equiv \exists s_n.(\exists s_i.\phi[s_c] \wedge s_i = s_c \wedge$
$Do(\delta_1, s_i, s_n)) \vee (\exists s_i.\neg\phi[s_c] \wedge s_i = s_c \wedge Do(\delta_2, s_i, s_n)))$ $\forall_I$

- While loop $\delta = \textbf{while } \phi \textbf{ do } \delta_1$

  To prove

  $$\mathcal{D} \models Do(\textbf{while } \phi \textbf{ do } \delta_1, S_c, S_n)$$

  we give a natural deduction proof of:

  $$\mathcal{D} \vdash \forall s_c.(\exists s_n.((\forall P.(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge Do(\phi?; \delta_1, s_2, s_3) \supset P(s_1, s_3)) \supset P(s_c, s_n)) \wedge \phi[s_n]) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

  Note that by assumption we have the additional condition

  $$\forall s.(\exists s'.(\forall P.(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge \phi[s_2] \wedge Do(\delta_1, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s')) \wedge \neg\phi[s'] \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

  for the while loop. This condition is equivalent to what we are trying to prove.

Thus, for any resolution procedure δ, the proof obligation **Ob 2** is sufficient to ensure that the following entailment holds:

$$\mathcal{D} \models \forall s_c.(\exists s_n.Do(\delta, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

We now show that this is a necessary condition. Consider the resolution procedure $\delta = \pi v.assign(v)$. Assume the condition

$$\forall s_c.(\exists v.Poss(assign(v), s_c) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

does not hold in $\mathcal{D}$. We first consider the case where

$$\neg(\forall s_c.(\exists v.Poss(assign(v), s_c) \subset \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

Note that by de Morgan's law,

$$\neg(\forall s_c.(\exists v.Poss(assign(v), s_c) \subset \exists L.assignRqst(L, s_c) \wedge \neg empty(L)))$$
$$\Leftrightarrow \exists s_c.\forall v.\neg Poss(assign(v), s_c) \wedge \exists L.assignRqst(L, s_c) \wedge \neg empty(L)$$

We show that the entialment for totality will not hold by providing a natural deduction proof of the following:

$$\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), \exists s_c.\forall v.\neg Poss(assign(v), s_c) \vdash \neg(\forall s_c.(\exists s_n.Do(\delta, s_c, s_n) \equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L)))$$

1. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s), \forall s_c.\exists s_n.Do(\delta, s_c, s_n), \forall v.\neg Poss(assign(v), s')$
   $\vdash \forall s_c.\exists s_n.\exists v.Poss(assign(v), s_c) \wedge s_n = do(assign(v), s_c)$ Def
2. $\mathcal{D}, \exists s_n.\forall v.\neg Poss(assign(v), s), \forall s_c.\exists s.Do(\delta, s_c, s_n), \forall v.\neg Poss(assign(v), s')$
   $\vdash \exists s_n.\exists v.Poss(assign(v), s') \wedge s_n = do(assign(v), s')$ $\forall_E$
3. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s), \forall s_c.\exists s_n.Do(\delta, s_c, s_n), \forall v.\neg Poss(assign(v), s'),$
   $Poss(assign(v), s') \wedge s_n = do(assign(v), s') \vdash Poss(assign(v), s')$ $\wedge_E$
4. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s), \forall s_c.\exists s_n.Do(\delta, s_c, s_n), \forall v.\neg Poss(assign(v), s'),$
   $Poss(assign(v), s') \wedge s_n = do(assign(v), s') \vdash \neg Poss(assign(v), s')$ $\forall_E$
5. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s), \forall v.\neg Poss(assign(v), s'),$
   $Poss(assign(v), s') \wedge s_n = do(assign(v), s') \vdash \neg(\forall s_c.\exists s_n.Do(\delta, s_c, s_n))$ $\neg_I$
6. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s), \exists s_n.\forall v.\neg Poss(assign(v), s'),$
   $\exists s_n.\exists v.Poss(assign(v), s') \wedge s_n = do(assign(v), s') \vdash \neg(\forall s_c.\exists s_n.Do(\delta, s_c, s_n))$ $\exists_E$
7. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s) \vdash (\exists s_n.\forall v.\neg Poss(assign(v), s')) \supset$
   $(\exists s_n.\exists v.Poss(assign(v), s') \wedge s_n = do(assign(v), s')) \supset \neg(\forall s_c.\exists s_n.Do(\delta, s_c, s))$ $\supset_I$
8. $\mathcal{D}, \exists s.\forall v.\neg Poss(assign(v), s) \vdash \neg(\forall s_c.\exists s_n.Do(\delta, s_c, s_n))$ $\supset_E$
9. $\mathcal{D}, \exists L.assignRqst(L, s_c) \wedge \neg empty(L), \exists s.\forall v.\neg Poss(assign(v), s) \vdash \neg(\forall s_c.\exists s_n.Do(\delta, s_c, s_n))$ $+$

Now consider the second case, where

$$\neg(\forall s_c.(\exists v.Poss(assign(v), s_c) \supset \exists L.assignRqst(L, s_c) \wedge \neg empty(L))$$

Note that by de Morgan's law,

$$\neg(\forall s_c.(\exists v.Poss(assign(v), s_c) \supset \exists L.assignRqst(L, s_c) \wedge \neg empty(L)))$$
$$\Leftrightarrow \exists s_c.\exists v.Poss(assign(v), s_c) \wedge \forall L.\neg(assignRqst(L, s_c) \wedge \neg empty(L))$$

We show that the entialment for totality will not hold by proving the following:

$\mathcal{D}, \forall L.\neg(assignRqst(L, s_c) \wedge \neg empty(L)), \exists v.Poss(assign(v), s_c) \vdash \neg(\forall s_c.(\exists s_n.Do(\delta, s_c, s_n)$
$\equiv \exists L.assignRqst(L, s_c) \wedge \neg empty(L)))$

Note that the natural deduction proofs shown above prove exactly this, that for all δ, there will exists some s_n such that $Do(\delta, s_c, s_n)$ if the precondition axiom allows assignment in s_c.

We have now shown that the restriction

$$\forall s.\exists v.Poss(assign(v), s)$$

is a necessary and sufficient condition for totality. □

Appendix B

Implementations

This appendix contains source code for the GOLOG implementations of the resolution modules.

accelResolution.pl

```
#!/usr/bin/swipl -q -t main -f

:- use_module(library(optparse)).

opts_spec([ [opt(feature), longflags(['feature']), help('list of feature rqst')]
    , [opt(goal),longflags(['goal']),help('goal to be called')]
    , [opt(safety),longflags(['safety']),help('list of safety rqst')]
    , [opt(driver),longflags(['driver']), help('list driver rqst')]]).

main :-
    opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(feature(FeatureList), Opts),
    memberchk(driver(DriverList), Opts),
    memberchk(safety(SafetyList), Opts),
    (nonvar(DriverList) -> call_call(assert(driverRqsts(DriverList,sc)), Result)
        ; true),
    (nonvar(SafetyList) -> call_call(assert(safetyRqsts(SafetyList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(assert(accelRqsts(FeatureList,sc)), Result)
```

```
        ; true),
    (nonvar(FeatureList) -> call_call(do(resolveAcceleration,sc,S), Result),
        writeln(S) ; true),
    halt.

call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/* To compile:
 *    swipl -g main -o accelResolution.exe -c golog_swi.pl helper.pl accelResolution.pl
 *
 * Inputs:
 *    --driver [..]: driver-related WCAs
 *    --safety [..]: safety-related WCAs
 *    --feature [..]: nonsafety-related WCAs
 */

/*************************************************************/
/* Acceleration Resolution Module */
/*************************************************************/

/* Precondition Axioms */
poss(assign(N),S) :- driverRqsts(R,S),min(N,R);
    driverRqsts([],S), safetyRqsts(R,S), min(N,R);
    driverRqsts([],S), safetyRqsts([],S), accelRqsts(R,S), min(N,R).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveAcceleration, pi(n,assign(n))).
```

steeringResolution.pl

```
#!/usr/bin/swipl -q -t main -f

:- use_module(library(optparse)).
```

```
opts_spec([ [opt(feature), longflags(['feature']), help('list of feature rqst')]
, [opt(goal),longflags(['goal']),help('goal to be called')]
, [opt(safety),longflags(['safety']),help('list of safety rqst')]
, [opt(driver),longflags(['driver']), help('list driver rqst')]]).

main :-
opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(feature(FeatureList), Opts),
    memberchk(driver(DriverList), Opts),
    memberchk(safety(SafetyList), Opts),
    (nonvar(DriverList) -> call_call(assert(driverRqsts(DriverList,sc)), Result)
        ; true),
    (nonvar(SafetyList) -> call_call(assert(safetyRqsts(SafetyList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(assert(steerRqsts(FeatureList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(do(resolveSteering,sc,S), Result),
        writeln(S) ; true),
    halt.

call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/** To compile **/
/*swipl -g main -o steeringResolution.exe -c golog_swi.pl helper.pl steeringResolution.

/*************************************************************/
/* Steering Resolution Module */
/*************************************************************/

/* Precondition Axioms */
poss(assign(N),S) :- safetyRes(N,S),!.
poss(assign(N),S) :- driverRes(N,S),!.
```

```
poss(assign(N),S) :- nonsafetyRes(N,S),!.


/*Resolution Scenarios*/

/*Highest priority to safety rqst*/
safetyRes(N,S) :- safetyRqsts(L,S), avg(N,L).
/*Next highest to driver*/
driverRes(N,S) :- safetyRqsts([],S), driverRqsts(L,S), avg(N,L).
/*Next highest to nonsafety*/
nonsafetyRes(N,S) :- safetyRqsts([],S), driverRqsts([],S), nonsafetyRqsts(L,S),
        avg(N,L).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveSteering, pi(n,assign(n))).
```

warningLightResolution.pl

```
#!/usr/bin/swipl -q -t main -f

:- dynamic lightRqsts/2.

:- use_module(library(optparse)).

opts_spec([ [opt(feature), longflags(['feature']), help('list of feature rqst')]
    , [opt(goal),longflags(['goal']),help('goal to be called')]
    ]).

main :-
    opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(feature(FeatureList), Opts),
    (nonvar(FeatureList) -> call_call(assert(lightRqsts(FeatureList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(do(resolveWarningLights,sc,S), Result),
        writeln(S) ; true),
    halt.
```

```
call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/** To compile **/
/*swipl -g main -o warningLightResolution.exe -c golog_swi.pl
    helper.pl warningLightResolution.pl */




/*************************************************************/
/* Warning Light Resolution Module /
/*************************************************************/

/* Precondition Axioms */
poss(setLight(N),S) :- lightRqsts(L,S), member(N,L).

/*Successor Axioms */
lightRqst(L,do(A,S)) :- A=setup, lightRqsts(Lc,S), append(Lc,Lc,L1), append(L1,Lc,L).
lightRqsts(R,do(A,S)) :- A=setLight(N), lightRqsts(Rp, S), remove(N, Rp, R).

/* Defined Fluents */
noRqsts(S) :- lightRqsts([],S).

/*Control Procedures */
primitive_action(setLight(N)).
primitive_action(setup).

/*proc(resolveWarningLights, while(some(n,lightRqst(n)), pi(n,setLight(n)))).*/
proc(resolveWarningLights,  setup : (star(pi(n,setLight(n))) : ?(noRqsts(now)))).
```

brakePressure.pl

```
#!/usr/bin/swipl -q -t main -f
```

```
:- use_module(library(optparse)).

opts_spec([ [opt(feature), longflags(['feature']), help('list of feature rqst')]
    , [opt(goal),longflags(['goal']),help('goal to be called')]
    , [opt(safety),longflags(['safety']),help('list of safety rqst')]
    , [opt(oscil),longflags(['oscillation']),help('current value of car.oscillation')]
    , [opt(driver),longflags(['driver']), help('list driver rqst')]]).

main :-
    opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(feature(FeatureList), Opts),
    memberchk(driver(DriverList), Opts),
    memberchk(safety(SafetyList), Opts),
    memberchk(oscil(Osc), Opts),
    (nonvar(DriverList) -> call_call(assert(driverRqsts(DriverList,sc)), Result)
        ; true),
    (nonvar(SafetyList) -> call_call(assert(safetyRqsts(SafetyList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(assert(accelRqsts(FeatureList,sc)), Result)
        ; true),
    (nonvar(Osc) -> call_call(assert(oscillation(Osc,sc)), Result) ; true),
    (nonvar(FeatureList) -> call_call(do(resolveBrakes,sc,S), Result),
        writeln(S) ; true),
    halt.

call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/* To compile:
 *    swipl -g main -o brakePressure.exe -c golog_swi.pl helper.pl brakePressure.pl
 *
 * Inputs:
 *    --driver [..]: driver-related WCAs
```

```
 *    --safety [..]: safety-related WCAs
 *    --feature [..]: nonsafety-related WCAs
 */

/*************************************************************/
/* Brake Pressure Resolution Module */
/*************************************************************/

/* Precondition Axioms */
poss(assign(N),S) :- safetyRes1(N,S),!.
poss(assign(N),S) :- safetyRes2(N,S),!.
poss(assign(N),S) :- driverRes(N,S),!.
poss(assign(N),S) :- slowestRes(N,S).

/*Resolution Scenarios*/

/*Highest priority to safety rqst w/o oscillation*/
safetyRes1(N,S) :- safetyRqsts(R,S), oscillation(O,S), O<10,max(N,R).

/*Highest priority to safety rqst w/ oscillation*/
safetyRes2(N,S) :- safetyRqsts(R,S), oscillation(O,S), O>= 10, avg(N,R).

/*Next highest priority to driver actions */
safetyRes(N,S) :- safetyRqsts([],S), driverRqsts(R,S), max(N,R).

/*Otherwise, pick slowest speed*/
slowestRes(N,S) :- driverRqsts([],S), safetyRqsts([],S), accelRqsts(R,S), max(N,R).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveBrakes, pi(n,assign(n))).
```

warningChime.pl

```
#!/usr/bin/swipl -q -t main -f

:- use_module(library(optparse)).
```

```
opts_spec([ [opt(feature), longflags(['feature']), help('list of feature rqst')]
    , [opt(goal),longflags(['goal']),help('goal to be called')]]).

main :-
    opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(feature(FeatureList), Opts),
    (nonvar(FeatureList) -> call_call(assert(assignRqsts(FeatureList,sc)), Result)
        ; true),
    (nonvar(FeatureList) -> call_call(do(resolveChime,sc,S), Result),
        writeln(S) ; true),
    halt.

call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/* To compile:
 *    swipl -g main -o warningChime.exe -c golog_swi.pl helper.pl warningChime.pl
 *
 * Inputs:
 *    --feature [..]: all WCAs
 */

/*************************************************************/
/* Warning Chime Resolution Module */
/*************************************************************/

/* Precondition Axioms */
poss(assign(N),S) :- assignRqsts(L,S), max(N,L).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveChime, pi(n,assign(n))).
```

airFlowRate.pl

```
#!/usr/bin/swipl -q -t main -f

:- use_module(library(optparse)).

opts_spec([ [opt(quality), longflags(['quality']), help('list of feature rqst')]
    , [opt(goal),longflags(['goal']),help('goal to be called')]
    , [opt(temp),longflags(['temp']), help('list driver rqst')]]).

main :-
    opts_spec(OptsSpec),
    opt_arguments(OptsSpec, Opts, _PositionalArgs),
    memberchk(quality(QualList), Opts),
    memberchk(temp(TempList), Opts),
    (nonvar(QualList) -> call_call(assert(qualRqsts(QualList,sc)), Result) ; true),
    (nonvar(TempList) -> call_call(assert(tempRqsts(TempList,sc)), Result) ; true),
    (nonvar(TempList) -> call_call(do(resolveAirFlowRate,sc,S), Result),
        writeln(S) ; true),
    halt.

call_call(Goal, true) :-
    call(Goal), !.

call_call(_Goal, false).


/* To compile:
 *    swipl -g main -o airFlowRate.exe -c golog_swi.pl helper.pl airFlowRate.pl
 *
 * Inputs:
 *    --quality [..]: air quality-related WCAs
 *    --temp [..]: temperature-related WCAs
 */

/*************************************************************/
/* Air Flow Rate Resolution Module */
/*************************************************************/

/* Precondition Axioms */
```

```
poss(assign(N),S) :- qualityRes(N,S),!.
poss(assign(N),S) :- temperatureRes(N,S).

/*Resolution Scenarios*/

/*Highest priority to quality rqst*/
qualityRes(N,S) :- qualRqsts(R,S),max(N,R).

/*Next highest priority to temperature features */
temperatureRes(N,S) :- qualRqsts([],S), tempRqsts(R,S), max(N,R).

/*Control Procedures */
primitive_action(assign(N)).
proc(resolveAirFlowRate, pi(n,assign(n))).
```

References

[1] Pansy K. Au and Joanne M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 153–167, 1997.

[2] Ed Baroth and Chris Hartsough. Visual object-oriented programming. chapter Visual Programming in the Real World, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995.

[3] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[4] T.F. Bowen, Ching-Hua Chow, F.S. Dworak, Nancy Griffeth, and Yow-Juian Lin. Views on the feature interaction problem. Technical Report Technical Memorandum TM-ARH-012849, Bellcore, October 1988.

[5] T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62, 1989.

[6] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, January 2003.

[7] Muffy Calder and Evan H. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK*. IOS Press, 2000.

[8] E.J. Cameron, N. Griffeth, Y. Lin, and H. Velthuijsen. "Definitions of Services, Features, and Feature Interactions", December 1992. Bellcore Memorandum for Discussion, presented at the International Workshop on Feature Interactions in Telecommunications Software Systems.

[9] A. Chavan, L. Yang, K. Ramachandran, and W. H. Leung. Resolving feature interaction with precedence lists in the feature language extensions. In *Proceedings of the 9th International Conference on Feature Interactions (ICFI)*, pages 114–128, 2007.

[10] Yi-Liang Chen, S. Lafortune, and Feng Lin. Priority assignment algorithms for resolving blocking in modular control of discrete event systems. In *Proceedings of the 35th IEEE Conference on Decision and Control,*, volume 3, pages 2743–2748, Dec 1996.

[11] Yi-Liang Chen and Stephane Lafortune. Resolving feature interactions using modular supervisory control with priorities. In *Feature Interactions in Telecommunications Systems IV*, pages 108–121. IOS Press, 1997.

[12] Alma L. Juarez Dominguez. *Detection of Feature Interactions in Automotive Active Safety Features*. PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2012.

[13] A. C W Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, Aug 1994.

[14] Norbert Fritsche. Runtime resolution of feature interactions in architectures with separated call and feature control. In *Feature Interactions in Telecommunications Systems III*, pages 43–63. IOS Press, 1995.

[15] Anders Gammelgaard and Jens E. Kristensen. Interaction detection, a logical approach. In *Feature Interactions in Telecommunications Systems*, pages 178–196, 1994.

[16] N.D. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *Feature Interactions in Telecommunications Systems*, pages 217–235, 1994.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[18] J.D. Hay and J.M. Atlee. Composing features and resolving interactions. In *ACM SIGSOFT Foundations of Software Engineering (FSE)*, pages 110–119, 2000.

[19] Constance L. Heitmeyer. Software cost reduction. Technical report, Naval Research Laboratory, 2002.

[20] S. Homayoon and H. Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *IEEE Communications Magazine*, 26(12):42–46, Dec 1988.

[21] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[22] Y. Jia and J.M. Atlee. Run-time management of feature interactions. In *ICSE Workshop on Component-Based Software Engineering (CBSE)*, 2003.

[23] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[24] Alma L. Juarez-Dominguez, Nancy A. Day, and Jeffrey J. Joyce. Modelling feature interactions in the automotive domain. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, pages 45–50, New York, NY, USA, 2008. ACM.

[25] Mario Kolberg, Evan H. Magill, Dave Marples, and Stephan Reiff-Marganiec. Second feature interaction contest. In Calder and Magill [7], pages 293–310.

[26] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *12th IEEE International Proceedings of the Requirements Engineering Conference*, pages 122–131, Sept 2004.

[27] R.C. Laney, T.T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In *Proceedings of the 9th International Conference on Feature Interactions (ICFI)*, pages 129–144, 2007.

[28] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *J. Logic Programming*, pages 59–84, 1997.

[29] D. Marples and E.H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *Feature Interactions in Telecommunications Systems V*, pages 115–134, 1998.

[30] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[31] P. Shaker, J.M. Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 151–160, Sept 2012.

[32] Pourya Shaker. *A Feature-Oriented Modelling Language and a Feature-Interaction Taxonomy for Product-Line Requirements*. PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2013.

[33] Steven Shapiro and Yves Lespérance. Modeling Multiagent Systems with CASL - A Feature Interaction Resolution Application. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 244–259. Springer Berlin Heidelberg, 2001.

[34] SWI-Prolog. Swi-prolog [online].

[35] U.S. National Highway Traffic Safety Administration. Safercar.gov [online].

[36] D.M. Weiss and R.C.T. Lai. *Software Product Line Engineering, A Family Based Development Process*. Addison Wesley, 1999.

[37] Pamela Zave. Requirements for evolving systems: A telecommunications perspective. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE)*, pages 2–9, 2001.

[38] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Trans. Softw. Eng. Methodol.*, 2(4):379–411, October 1993.

[39] Lu Zhongwan. *Mathematical Logic for Computer Science*. World Scientific, New Jersey, second edition, 1989.

[40] P. Ann Zimmer and Joanne M. Atlee. Ordering features by category. *Journal of Systems and Software*, 85(8):1782–1800, August 2012.