Verification of Feature Coordination using the Fluent Calculus

Ralph Hoch and Hermann Kaindl
Institute of Computer Technology, TU Wien, Austria

Keywords: Verification, Fluent Calculus, Situation Calculus, Model Checking.

Abstract: Previously, an approach based on the Situation Calculus was published for specifying feature coordination of a software system, but without a physical model or any additional autonomous agent in the environment. Hence, no verification of the feature coordination was possible in spite of its formal specification. Verification of *safety-critical* feature coordination is important, however, and requires additional models. This paper shows that a specification of a software coordinator can be formally verified using the *Fluent Calculus* (a derivative of the Situation Calculus), when combined with additional models. The overall qualitative model is a re-implementation of a recently published one based on synchronized finite-state machines, which was used for *model checking*. In fact, we show how the model in Fluent Calculus can be systematically derived from the finite-state machines. The results of verification using the Fluent Calculus correspond to those using model checking. We also contrast our approach using the Fluent Calculus with model checking. In summary, we present verification of (safety-critical) feature coordination using the Fluent Calculus.

1 INTRODUCTION

Feature interaction occurs, when the interplay of two or more features gives rise to an overall system behavior that is not easily deducible from the individual behaviors of the features involved, and often unexpected (Apel et al., 2013). Undesired feature interaction can be safety-critical, e.g., in automotive systems. Hence, the verification of such cyber-physical systems including (embedded) software is important.

An approach for detecting feature interactions automatically through *model checking* can be found in (Juarez-Dominguez et al., 2008). A feature interaction may result directly from conflicting requests to a single variable in the software (as studied for the speed of a vehicle in (Juarez-Dominguez et al., 2008)). This is sufficient for detecting that there is a feature interaction, but not for investigating its influence on the environment. This would require an additional physical model and a model of another vehicle, which is an independent autonomous agent.

Bocovich and Atlee (Bocovich and Atlee, 2014) also addressed feature interaction resulting from a conflict of features accessing the same software variable(s) at the same point in time. Since the granularity of features may be relevant, resolution for each software variable under conflict was proposed for feature *coordination* (with examples in the automotive domain). They showed how such a resolution can be specified using the *Situation Calculus*. However, without a physical model or any additional embodied entity in the environment, no verification is possible of whether such a resolution as specified for the software actually achieves its purpose in the overall cyber-physical system.

According to (Rathmair et al., 2016), an additional physical model is needed for model checking feature *coordination*. This work modeled the influence of speed also on the distance to another vehicle in the context of *Adaptive Cruise Control* (ACC), which is one of the more advanced features penetrating the automotive market (Winner and Schopper, 2015). It includes both *Cruise Control* (CC), as widely used in cars, and *Distance Control* (DC) of a vehicle A following another vehicle B. Overall, this work modeled a cyber-physical system qualitatively and used the resulting model for model checking the coordination of features.

Artificial Intelligence techniques like the Situation Calculus may be a good fit for modeling such a system qualitatively, since they have been proposed and used early on in the context of robot planning. Hence, they offer also planning capabilities, so that not only the specification of a feature coordinator is possible, but also some means for verifying whether it actually works should be feasible. In this sense, our motivation was to check this feasibility using the ACC example.

Hoch, R. and Kaindl, H.
Verification of Feature Coordination using the Fluent Calculus.
DOI: 10.5220/0006771401690179
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 169-179
ISBN: 978-989-758-300-1
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

We systematically transfer the model of (Rathmair et al., 2016) to the *Fluent Calculus*, a derivative of the Situation Calculus as used in (Bocovich and Atlee, 2014) for specifying feature coordination within software, and use the resulting model for a new approach to the verification of cyber-physical feature coordination. This approach involves a tool supporting the Fluent Calculus and a simple planner that we implemented on top of it. When the planner can find a sequence of actions for achieving a goal that formulates a property to be avoided, then it actually presents an example of what is to be avoided. This corresponds to a counterexample in model checking. Hence, we also contrast our new verification approach based on the Fluent Calculus with the widely studied verification approach based on model checking.

The remainder of this paper is organized in the following manner. First, we provide some background material for making this paper self-contained, and relate it to other work. Then we present a systematic transformation from synchronized finite-state machines (FSMs) to a representation in the Fluent Calculus. Based on it, we explain our verification approach and show that it can deliver the same results as model checking. Finally, we briefly contrast it with model checking.

2 BACKGROUND AND RELATED WORK

We cover here some background on and related work to the Fluent Calculus, feature coordination, model checking, and the given model of ACC that we transform to the Fluent Calculus for verification with our new approach.

2.1 Fluent Calculus

The original idea was introduced by McCarthy and Hayes (McCarthy and Hayes, 1969) long time ago. Their Situation Calculus consists of three elements:

1. Situations
   represent the evolving states of the domain, where certain conditions hold in each state.
2. Actions
   represent the changes between situations. A special predicate *poss* determines whether a specific *action* can be performed or not.
3. Fluents
   represent the elements of the domain that can change over time. Typically, predicates are used for this representation, which take a situation as an argument. An example is the fluent *carrying(o,s)*, which states if an object o is carried, e.g., by a robot, in situation s.

Based on previous work on the Situation Calculus such as (Reiter, 1991), Thielscher (Thielscher, 1998) developed the Fluent Calculus. It differs from the Situation Calculus in how situations are treated and how fluents are used. The Fluent Calculus defines that a new state after the execution of an action is equal to the previous state with exceptions to the effects of the action. In addition, fluents are treated as functional terms. The fluents from the Situation Calculus are stripped off the situation parameter, and special predicates, e.g., *holds*, are introduced. These special predicates take a functional term and a state as an argument. They are used to check whether specific conditions hold in a specific state or not. For example, the fluent *carrying(o,s)* from the Situation Calculus translates to a functional term *carrying(o)* in the Fluent Calculus. Hence, this term is not depended on the current state anymore. To check whether this term holds in a specific state, the *holds* predicate is used, e.g., *holds(carrying(o), state)*.

Hence, the Fluent Calculus provides a formalism to model specific *actions* that lead from one situation to another. This is specified using the *poss* and *state_update* predicates. These predefined predicates model the preconditions (*poss* statement) and effects (*state_update*) of an action. Together, they provide a formal specification of an action.

An implementation of the Fluent Calculus called FLUX (Thielscher, 2005) is available for the constraint logic programming system ECLiPSe.[1] To illustrate how such an action in Fluent Calculus is applied in FLUX, we use a simple example. Let us assume that a vehicle B is currently driving with *medium speed* and wants to change to *high speed*. This simplistic example already utilizes the main parts of the Fluent Calculus. The driving speed of the vehicle changes over *time* and thus is a *fluent*. The change of the driving speed is performed by an *action*. For this example, we assume that the name of this action is *switchMediumSpeedToHighSpeedVehicleB*. What is missing, is the situation of the domain. The current situation of vehicle B is medium speed. This information is only part of a more complex situation specification, but is sufficient for this example. In essence, the action *switchMediumSpeedToHighSpeedVehicleB* can only be executed if the current speed of vehicle B is medium speed. This is a precondition for the execution of the action. The

[1] ECLiPSe Constraint Programming System: http://www.eclipseclp.org

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

holds predicate checks if a specific value *vehicleBIsSetTo(B, mediumSpeed)* holds in state *Z*. Listing 1 shows this precondition in FLUX.

```
poss(switchMediumSpeedToHighSpeedVehicleB(B), Z)
    :-
  % Input has to be of type VehicleB %
  knows_val([B], vehicleB(B), Z),
  % Precondition that the speed of B is set to
       mediumSpeed in State Z %
  holds(vehicleBIsSetTo(B, mediumSpeed), Z).
```

Listing 1: Preconditions of the medium-to-high-speed action in FLUX.

When the action is executed, it has effects on the situation of the domain, e.g., the speed of vehicle B changes. In our example, vehicle B is set to *high speed*. However, this is not sufficient, since we also have to specify that vehicle B is not driving with medium speed anymore. This fact is removed from the state. Listing 2 shows the effects of the action in FLUX. Basically *vehicleBIsSetTo(B, highSpeed)* is added and *vehicleBIsSetTo(B, mediumSpeed)* removed from Z1 and results in Z2.

```
state_update(Z1,
     switchMediumSpeedToHighSpeedB(B), Z2, []) :-
  update(Z1, % original state %
    % Statements added to the State %
    [vehicleBIsSetTo(B, highSpeed)],
    % Statements removed from the State %
    [vehicleBIsSetTo(B, mediumSpeed)],
    Z2  % new State is stored in Z2 %
  ).
```

Listing 2: Effects of the medium-to high-speed-action in FLUX.

In fact, we previously proposed a verification approach based on the Fluent Calculus already in (Hoch et al., 2015), which verifies sequences of semantically specified services against the specifications of its atomic services. In this paper, we propose a completely different verification approach, where all possible sequences are exhaustively tried out by a planner and checked against a goal condition that is actually to be avoided.

2.2 Related Work on Feature Coordination

Automatically detecting and analyzing feature interactions is outside the scope of this paper, which focuses on coordinating features with known interactions. Even if these are known, coordinating the features involved poses additional challenges (Jackson and Zave, 1998).

The formulation of resolutions as proposed in (Bocovich and Atlee, 2014) uses the Situation Calculus implemented in GOLOG (Levesque et al., 1997). This work did not investigate, however, the physical effects from a feature interaction of a software variable, and it did not include independent physical variables outside the control of the software. Hence, no verification was possible of whether such a resolution as specified actually achieves its purpose.

In fact, we can also use this approach for specifying a coordination of CC and DC as given in Formula 1.

$$\begin{aligned} &Poss(setSpeedA(SPEED), Z) \equiv \\ &\exists ValueCC.ccVehicleAIsSetTo(cc, ValueCC) \wedge \\ &\exists ValueDC.dcVehicleAIsSetTo(dc, ValueDC) \wedge \\ &\quad minimum(ValueCC, ValueDC, SPEED). \end{aligned} \quad (1)$$

The actual implementation in FLUX differs slightly from this equation as we utilize several *Poss* statements to express the minimum. Each *Poss* statement has a set of non-overlapping conditions and the minimum is built by choosing the fitting *Poss* statement. Since we use a qualitative model we chose this approach.

GOLOG differs from FLUX in the way it handles variable resolution. GOLOG uses (backward) regression as opposed to the forward progression of FLUX. More information on this and how GOLOG programs can be interpreted in FLUX is given in (Schiffel and Thielscher, 2005).

2.3 Background on Model Checking

Model checking (or property checking) is a formal verification technique based on models of system behavior and properties, specified unambiguously in formal languages (see, e.g., (Baier and Katoen, 2008)). The behavioral model of the system under verification is often specified using an FSM, in our case using synchronized FSMs. Their expressiveness is sufficient for our case, but Petri nets, e.g., could be used as well, if needed (depending on the tool used). The properties to be checked on the behavioral model are formulated in a specific property specification language, usually based on a temporal language. Several tools such as NuSMV (NuSMV, 2014) exist for performing these checks by systematically exploring the state-space of the system. When such a tool finds a property violation, it reports it in the form of a counterexample.

Many model checking approaches use *Computational Tree Logic* (CTL) for property specification, with the following CTL operators:

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

- *AG* (Always Globally): an expression p is true in the initial state s_0 and in each state of all transitions $s_0 \to s_1 \to s_2 \to \cdots \to s_n$.
- *EF* (Eventually in the Future): for an expression p and an initial state s_0, there exists a state sequence $s_0 \to s_1 \to s_2 \to \cdots \to s_n$ such that p is true in s_n.

2.4 A Given Minimalist Model of ACC

Our example, as adopted from (Rathmair et al., 2016), is based on *Adaptive Cruise Control* (ACC). The high-level structure of this model is illustrated in Figure 1.

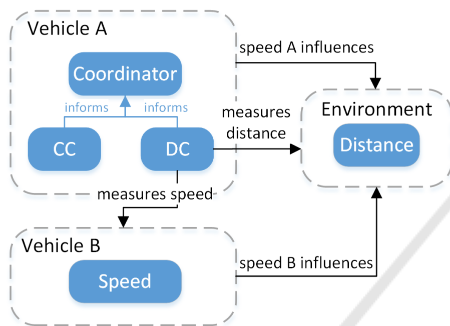

Figure 1: High-level structure of ACC model used.

It contains a model of vehicle A, which involves a Coordinator of CC and DC. It also contains vehicle B, whose behavior is, of course, not under control of vehicle A. And it contains a simple physical model of the environment, where the distance between vehicles A and B is important with respect to a potential accident.

The property that such a model has been checked against in (Rathmair et al., 2016) formalizes such an accident, more precisely a rear-end collision accident that DC has to avoid, and hence also ACC. Formula 2 defines this property in CTL in the sense that it is always (globally) true that the physical distance is different from a collision:

$$AG(state_phy_dist \neq COLLISION) \quad (2)$$

3 MODELING CYBER-PHYSICAL FEATURE COORDINATION IN THE FLUENT CALCULUS

Rather than creating a new model from scratch, we built upon the qualitative model of ACC as defined in (Rathmair et al., 2016). In contrast to the original model, we used the Fluent Calculus for modeling the cyber-physical system including the feature coordination. Hence, we transform the given model in synchronous FSMs to a model represented in the Fluent Calculus, where both models are supposed to capture the same overall behavior of ACC. In fact, we specify a systematic transformation approach that could, in principle, also be automated.

Instead of replicating exactly the same FSMs from (Rathmair et al., 2016) here, we present isomorphic FSMs below, which have already action names as labels, i.e. names of the actions to be formally defined in the Fluent Calculus. This is primarily for didactic reasons, so that it will be easier to see the correspondence of the FSMs with the actions as defined in the Fluent Calculus.

First, we have to specify how the following terms in the Fluent Calculus are derived:

- terms identifying the state machines,
- terms for each state in each state machine,
- terms for each state transition.

Based on these terms, we define the preconditions and effects of all actions as derived from the conditions of state transitions, the state from where a transition is going out (its root), and the state where the transition leads to (its target).

Before we actually explain the details of the derivation of these terms, let us show the FSM on the behavior of vehicle B as an example. This FSM, as given in Figure 2, is isomorphic to the FSM labeled *Vehicle B* in (Rathmair et al., 2016, Figure 6). The action names on transitions are abbreviated in Figure 2. They are automatically inferred from the original model according to the rules defined in the text. For example *MtL* translates to *switchMediumSpeedToLowSpeedVehicleB*.

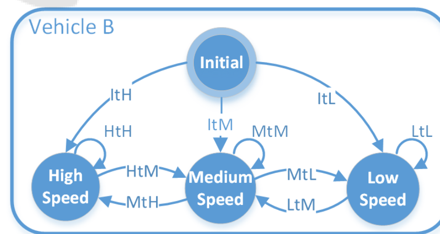

Figure 2: FSM of vehicle B.

The systematic transformation approach to this derivation is as follows:

- Each FSM is a term $\{FSM.name\}\{FSM.context\}$ with an argument. The argument $\{FSM.name\}$ is a constant used during the verification process to ensure that actions are only performed when appropriate. The suffix $\{FSM.context\}$ is not necessary, but makes the term more human readable.

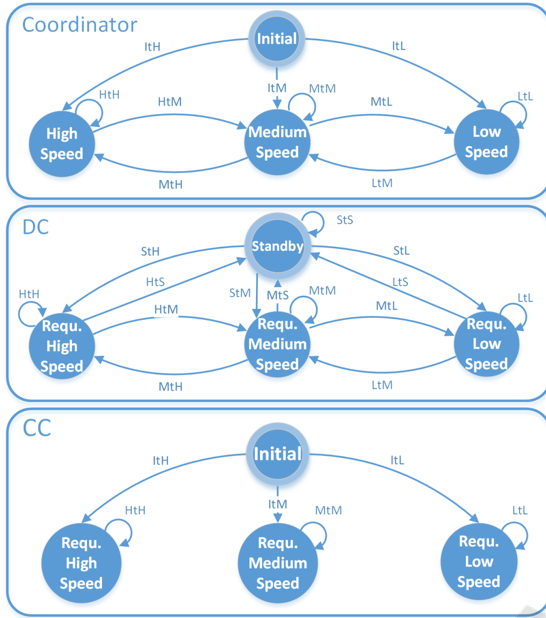

Figure 3: FSM of vehicle A.

Example: FSM of CC is *ccVehicleA(cc)*. *FSM.context* is *VehicleA* since CC is executed for Vehicle A.

- Each FSM is also a compound term {*FSM.name*} {*FSM. context*} *IsSetTo(object, value)* with two arguments. This term may change over time and thus is a *fluent*. The first argument is the object on which the term operates and the second argument is the value.

  Example: FSM of CC results in *ccVehicleAIsSetTo(object, value)*, where object and value are to be reset for concrete values, e.g., *ccVehicleAIsSetTo(cc, requLowSpeed)*.

- Each state of an FSM is a value that the term of this FSM can hold. The value is represented as a constant and is the name of the state {*FSM.state.name*}{*FSM.name*}.

  Example: The state *Requ Low Speed* of the CC FSM in Vehicle A results in the value *requLowSpeedCC*. This value is used with the corresponding term *ccVehicleAIsSetTo* of the FSM. The result is *ccVehicleAIsSetTo(cc, requLowSpeedCC)*.

- Each transition in an FSM is transformed to an *action* in the Fluent Calculus. The name of the action is defined through the root and target state of the transition and is *switch* {*FSM. transition.root.name*} *To* {*FSM.transition.target.name*} {*FSM. name*}.

  Example: The transition in the CC FSM from *Requ Medium Speed* to *Requ Low Speed* results in the action *switchRequMediumSpeedToRequLowSpeed*.

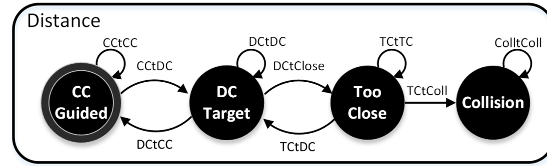

Figure 4: FSM of the distance classification.

Much as Figure 2 shows the action names for *vehicle B*, Figure 3 shows the action names for *vehicle A* and Figure 4 the distance classification in the environment.

The FSM in Figure 2 contains the action *switchMediumSpeedToHighSpeedVehicleB* as a label of a transition. In fact, it corresponds to the example of an action specified in the Fluent Calculus above (in the Background section). Still, we have to explain how the action specifications can be systematically transformed from the FSMs (based on the terms derived above).

The systematic transformation of transitions in FSMs to actions in the Fluent Calculus is as follows:

- Transitions between states are transformed to actions. Actions consist of *state_update* and *poss* statements.
- Conditions of the actions are given as propositional logic formulas in disjunctive normal form.
- The arguments of an action are *all* distinct elements of the propositional logic formula.
- The root of the state transition in the FSM is a precondition in the corresponding *poss* statement of the action in the Fluent Calculus.
- For each disjunctive part of the condition of a transition, one *poss* statement is created.
- The effects of an action are modeled via a *state_update* predicate.
- The positive effect of an action is the target node.
- The negative effect of an action is the root node.

To demonstrate this systematic transformation, let us consider the transition from low speed to medium speed in the Coordinator FSM. It consists of two separate *or*-connected conditions condCo_M and condCo_H given as propositional formulas in (Rathmair et al., 2016). These conditions consist of several logical expressions related to states in the FSMs. The disjunctive normal form of this condition is shown in Listing 3.

```
(
  (requMediumSpeedCC ∧ ¬requLowSpeedDC) ∨
  (¬requLowSpeedDC ∧ requMediumSpeedDC)
) ∨
(
  (
    requHighSpeedCC ∧
    ¬requLowSpeedDC ∧
    ¬requMediumSpeedDC
  )  ∨
  (
    ¬requLowSpeedDC ∧
    ¬requMediumSpeedDC ∧
    requHighSpeedDC
  )
)
```

Listing 3: Example condition on a transition.

Hence, there are *or*-connected blocks of *and*-connected atoms. Each *or*-block, e.g., *(requMediumSpeedCC ∧ ¬requLowSpeedDC)*, is treated as one set of preconditions for an action, in this case the *switchLowSpeedToMediumSpeedCoordinator* action. That is, the action can only be executed if (at least) one set of preconditions is fulfilled. E.g., the precondition set *(requMediumSpeedCC ∧ ¬requLowSpeedDC)* states, that the corresponding action can only be executed if the cruise control requests medium speed and, at the same time, the distance control does not request low speed.

In the Coordinator FSM, the condition on the transition states that the CC FSM has to be in state *requMediumSpeed* and the DC FSM cannot be in state *requLowSpeed* to trigger the transition. In the Fluent Calculus, these states are terms ({*FSM.state.name*}{*FSM.name*}) related to the terms of the FSM ({*FSM.name*} {*FSM.context*}*IsSetTo(object, value)*) and, thus, we can systematically derive a representation. For example, the atomic element *requMediumSpeedCC* in the condition is a value of *ccVehicleAIsSetTo* and only holds if *ccVehicleAIsSetTo(cc, requMediumSpeedCC)* is true. That is, this part of the condition only evaluates to true if cruise control requests medium speed. This is one part of the precondition set. The second part is *¬requLowSpeedDC* and is transformed to *dcVehicleAIsSetTo(DC, requLowSpeedDC)*. Together, they are one precondition set for the corresponding *switchLowSpeedToMediumSpeedCoordinator* action.

Preconditions are checked via the *poss* predicate in the Fluent Calculus. In fact, there can be many *poss* statements with different conditions for one action. We utilize this and generate one *poss* statement per precondition set. Each *poss* statement also contains the root state of the transition in the FSM as an additional precondition. Hence, the switch from low speed to medium speed has the additional precondition that the coordinator has currently set the speed to low speed. Corresponding *poss* statements in FLUX for the first two *o*r-connected sets in Listing 3 are shown in Listing 4.

```
% requMediumSpeedCC and not requLowSpeedDC %
poss(switchLowSpeedToMediumSpeedCoordinator(A,
    DC, CC), Z) :-
  % object that the action operates on %
  knows_val([A], vehicleA(A), Z),
  % root state in FSM as a precondition %
  holds(coordinatorVehilceAIsSetTo(A, lowSpeed),
      Z),
  % requMediumSpeedCC precondition %
  holds(ccVehicleAIsSetTo(CC, requMediumSpeedCC),
      Z),
  % not requLowSpeedDC precondition %
  knows_not(dcVehicleAIsSetTo(DC,
      requLowSpeedDC), Z).

% requMediumSpeedDC and not RequLowSpeedCC %
poss(switchLowSpeedToMediumSpeedCoordinator(A,
    DC, CC), Z) :-
  % object that the action operates on %
  knows_val([A], vehicleA(A), Z),
  % root state in FSM as a precondition %
  holds(coordinatorVehilceAIsSetTo(A, lowSpeed),
      Z),
  % requMediumSpeedDC precondition %
  holds(dcVehicleAIsSetTo(DC, requMediumSpeedDC),
      Z),
  % not requLowSpeedCC precondition %
  knows_not(ccVehicleAIsSetTo(CC,
      requLowSpeedCC), Z).
```

Listing 4: Example poss statements for actions.

The arguments of the action are given through the elements it depends on. In Listing 4, the arguments are A, DC and CC, or the vehicle, the distance control and the cruise control, respectively. It has to be noted that this is not necessarily required, the arguments could also be omitted, but we include them for better readability.

Still missing are the effects of an action. The effects are the fluents that are added and removed when an action is performed. In our example, the effect of switching from low speed to medium speed is that the vehicle is driving with medium speed after execution of the action. Additionally, the fact that the vehicle is driving low speed has to be removed. Hence, the positive effect is the addition of the target state as a value, and the negative effect is the removal of the root state as a value. The effects are specified via the *state_update* predicate and shown in Listing 5.

```
state_update(Z1,
    switchLowSpeedToMediumSpeedCoordinator(A, _,
    _), Z2, []) :-
  update(Z1,
    % positive effect: adding medium speed %
    [coordinatorVehilceAIsSetTo(A, mediumSpeed)],
    % negative effect: removing low speed %
    [coordinatorVehilceAIsSetTo(A, lowSpeed)],
    Z2
  ).
```

Listing 5: Example state_update statement of an action.

Using the systematic transformation defined above, a very verbose model results. Often, many state transitions in the FSM have the same conditions but different root states, or they have repeating conditions on different state transitions. For example, the condCo_M condition shown above is used in a transition from low speed to medium speed, medium speed to medium speed and high speed to medium speed.

With this in mind, the transformation can be optimized. All conditions that are repeated on different state transitions leading to the same state can be written as a single *poss* statement. In this case, the root nodes are *o*r-connected preconditions. Hence, condCo_M can also be written as shown in Listing 6. Note, that also the name of the action has changed, since it does not depend on the root state anymore.

```
% requMediumSpeedDC and not requLowSpeedDC %
poss(switchToMediumSpeedVehicleA(A, DC, CC), Z) :-
  % object that the action operates on %
  knows_val([A], vehicleA(A), Z),
  (
    % root state low speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        lowSpeed), Z);
    % root state medium speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        mediumSpeed), Z);
    % root state high speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        highSpeed), Z)
  ),
  % requMediumSpeedCC precondition %
  holds(ccVehicleAIsSetTo(CC, requMediumSpeedCC),
      Z),
  % not requLowSpeedDC precondition %
  knows_not(dcVehicleAIsSetTo(DC,
      requLowSpeedDC), Z).

% not requLowSpeedCC and requMediumSpeedDC %
poss(switchToMediumSpeedVehicleA(A, DC, CC), Z) :-
  % object that the action operates on %
  knows_val([A], vehicleA(A), Z),
  (
    % root state low speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        lowSpeed), Z);
    % root state medium speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        mediumSpeed), Z);
    % root state high speed as a precondition %
    holds(coordinatorVehilceAIsSetTo(A,
        highSpeed), Z)
  ),
  % requMediumSpeedDC precondition %
  holds(dcVehicleAIsSetTo(DC, requMediumSpeedDC),
      Z),
  % not requLowSpeedCC precondition %
  knows_not(ccVehicleAIsSetTo(CC,
      requLowSpeedCC), Z).
```

Listing 6: Example optimized poss statements for actions.

The effects of this action have to be adapted as well. The result is shown in Listing 7.

```
state_update(Z1, switchToMediumSpeedVehicleA(A,
    _, _), Z2, []) :-
  update(Z1,
    % positive effect: adding medium speed %
    [coordinatorVehilceAIsSetTo(A, mediumSpeed)],
    % negative effect: removing low speed AND
        high speed %
    [
      coordinatorVehilceAIsSetTo(A, lowSpeed),
      coordinatorVehilceAIsSetTo(A, highSpeed)],
    Z2
  ).
```

Listing 7: Example of optimized state_update statement of an action.

Certain self-transitions can be omitted as well. For example, it is not necessary to set the speed in cruise control to high speed if it is already at high speed. However, for the sake of verification, the approach described above is sufficient and further optimizations are omitted here.

With the approach defined above, the environment part is transformed accordingly. After the transformation, the Coordinator shown in Figure 3 already contains the means to resolve conflicts of features. This approach is similar to the one introduced by Bocovich et al. (Bocovich and Atlee, 2014) (and sketched above), since actions may influence the same variable. In contrast, in our case the variable is not a single statement, but a value of different terms. The conflict between the values of these terms are resolved in the coordinator according to the definition in (Rathmair et al., 2016). Although we do not use the same variable, the approach of Bocovich et al. could be implemented as well. Currently, our model uses *requMediumSpeedDC* and *requMediumSpeedCC* to define values of terms related to DC and CC. However, there could also be one general term *requestSpeedIsSetTo(speed, Value)* that states a request on the speed variable. In this case, the variable resolution of Bocovich et al. can be used directly.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4 VERIFICATION BASED ON THE FLUENT CALCULUS USING A PLANNER

Much as verification through (unbounded) model checking provides a proof (or disproof) of given properties (such as strict avoidance of a collision situation), we strive for such a verification based on the Fluent Calculus. A key question is how to handle such a property in this context, and how to check it. The Situation Calculus and the Fluent Calculus have been used traditionally, e.g., for robot planning, where a formula specifies a *goal* condition. In such a setting, the planner tries to find a sequence of actions (called a *plan*) that leads to a situation where this condition is fulfilled. Our key idea for *verification* using a planner is to formulate a property to be avoided (e.g., a collision) as a goal condition. If the planner finds a related plan for achieving such a goal condition, then it actually presents an example of what is to be avoided. This corresponds to a counterexample in model checking. Otherwise, however, the planner needs to perform an exhaustive search that does not find any such goal for proving that no property violation can occur. Note, that this approach uses a planner for verification, while the approach in (Bensalem et al., 2014) uses a model checker for planning.

Under such an exhaustive search regime, each and every combination of actions is visited, so that the search space is fully explored. Any search approach could be used, in principle, for implementing such an exhaustive planner. We chose *depth-first search* because of its linear-space requirements, and since it is already provided in the PROLOG-based tool environment. In essence, all possible sequences of actions are tried recursively until a situation with the given goal condition is found, or all possibilities are exhausted. Starting from the given root node, the first child node is generated. From this newly generated node, again the first child node is generated. In this way, the depth-first search follows one path in the tree until it cannot be extended any further (having reached a leaf node), from where it backtracks to the previous node and continues with the next node, etc. This is illustrated in Figure 5(a). This search continues until a situation satisfying the goal condition is found, or there is no further action to be tried.

It is well known, however, that depth-first search can, in general, result in infinite cycles. This happens if and when an action leads from one situation to an already visited situation on the same path. Figure 5(b) illustrates an example of such a cycle. To prevent this, usually the already visited nodes are stored for guaranteeing that only new situations are explored

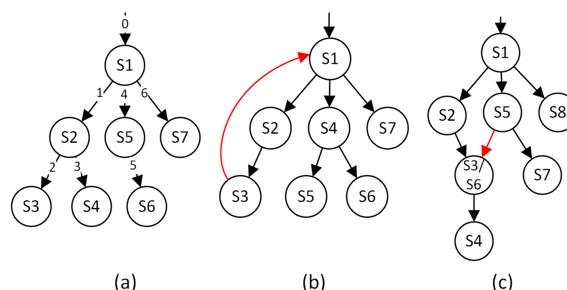

Figure 5: Search space of planner.

in each path. When the already visited situations are stored, the search can look them up and recognize that a newly generated situation, e.g., node *S3* in the figure, is the same as an already visited one, e.g., node *S1*. This path is terminated and the planner backtracks to continue with the next situation, e.g., node *S4*. With this cycle detection and avoidance, a termination of the search is guaranteed for a finite state space.

Storing only the visited nodes for the path from the root to the currently visited situation has linear-space requirements. However, this may lead to visiting situations more than once, if they can be reached from the root via several different paths. This is called a directed acyclic graph (DAG). An example of such a graph is illustrated in Figure 5(c). We chose to store all visited states for detecting DAGs as well and for terminating corresponding branches. For example, in Figure 5, node *S3/S6* can be reached through two different paths. However, the child nodes of *S3/S6* only have to be expanded once. When arriving again from node *S5*, the path can be terminated. In general, this approach has exponential space requirements, but it can still be used when enough storage is available (like in our example). For implementing such an approach to storing nodes, many PROLOG engines provide a *tabling* technique. The engine in ECLiPSe, however, has no built-in support for tabling, which made a custom implementation necessary.

So far, this planning approach looks straightforward, but there is an intricacy involved. As described in the previous section, each FSM has a predicate with corresponding actions. In the model-checking tool NuSMV (used in (Rathmair et al., 2016)), all state transitions of all the synchronous FSMs happen in one execution cycle. This is in contrast to our planner, where the actions are executed one after the other. To ensure that each part of the overall system has all information available for its execution, a linearization is required. Since some parts of the system depend on environment values, a specific order of execution can be derived. For example, the distance control depends on the value of the distance classification. The coordinator depends on

the distance and cruise control. In contrast, the cruise control and vehicle B do not depend on the other parts of the model. Hence, a possible execution sequence for the planner is:

1. Vehicle B
2. CC (of vehicle A)
3. DC (of vehicle A)
4. Coordinator (of vehicle A)
5. Distance classification

However, this execution sequence does not take into account that sensor values can only be read with a delay, which is essential in the model in (Rathmair et al., 2016). Since the distance control measures the speed of vehicle B and also the distance, there is such a delay involved. In the original model of (Rathmair et al., 2016), this is solved by using temporary variables that hold the values of the previous cycle. In our approach, this delay of sensors is implemented by rearranging the execution sequence of the actions. Since DC has to use the values of the previous cycle, the actions of vehicle B and the distance classification are the last steps in an execution cycle. By doing so, DC can only access the values of the previous cycle, since the new values are generated thereafter. Figure 6 illustrates this execution cycle actually used by our planner, whose implementation in FLUX is given in Listing 8.

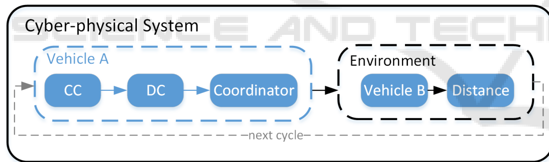

Figure 6: Execution cycle.

```
schedule_plan(Z,P,Zn):-
  (
    % first: execute CC %
    schedule_planCCA(Z,P,Z1),
    % second: execute DC, use result of CC as
        world state %
    schedule_planDCA(Z1,P,Z2),
    % third: execute Coordinator, use result of
        DC as world state %
    schedule_planCoodA(Z2,P,Z3),
    % fourth: execute Vehicle B, use result of
        Coordinator as world state %
    schedule_planVehB(Z3,P,Z4),
    % fifth: execute DistanceClassification, use
        result of Vehicle B as world state %
    schedule_planDistance(Z4,P,Z5)
  ),
  schedule_plan(Z5,P,Zn). % continue with next
      cycle %
```

Listing 8: Planner in FLUX.

Each *schedule_planXX* statement calls a function that determines, which specific action of the FSM is executed. The possible actions are all actions that are generated for a specific FSM. Only one of the actions, based on their preconditions, is chosen and executed. Listing 9 shows the possible actions for Vehicle B.

```
schedule_planSpeedVehB(Z,[VehB|P],Zn):-
  (VehB=switchToLowSpeedVehicleB(_),
      poss(VehB,Z), state_update(Z,VehB,Z1,[]));
  (VehB=switchToMediumSpeedVehicleB(_),
      poss(VehB,Z), state_update(Z,VehB,Z1,[]));
  (VehB=switchToHighSpeedVehicleB(_),
      poss(VehB,Z), state_update(Z,VehB,Z1,[]));
  (VehB="VehB - Nothing",Z1=Z,true).
```

Listing 9: Vehicle B planner.

The sequence given in Listing 8 is executed until the goal state is reached or the exhaustive search is finished. In our case, the goal condition is a property to be avoided. It is fulfilled if and when the distance classification detects a collision. Listing 10 specifies this property in FLUX.

```
schedule_plan(Z,_,Z):-
  knows_val([Distance], distanceIsSetTo(Distance,
      collision), Z).
```

Listing 10: Property to be avoided.

5 CONTRASTING WITH MODEL CHECKING

Now let us contrast this new approach with model checking. We do this in terms of verification results, execution time for verification of our example, and conceptually.

First of all, it is important that our new approach using the Fluent Calculus delivers the same verification results as model checking (as reported in (Rathmair et al., 2016)). Hence, we ran the planner as specified above, and it did not find a plan (a sequence of actions) leading to a goal specified through the collision property. This verified the model, and this verification result is consistent with the verification result reported in (Rathmair et al., 2016) for the model that we transformed ours from systematically (as specified above). Rathmair et al. model-checked a slightly simpler model as well (one without the *Too Close* state in the distance classification). This led to a counterexample of a collision. We modified our derived model in the same way and, in fact, our planner found a sequence of actions leading to a goal state with the collision condition. Therefore, also our approach based on the Fluent Calculus did not verify this simpler model. Again, this result is consistent with the one reported in (Rathmair et al., 2016).

According to (Rathmair et al., 2016), a verification run against the collision property using the model checking tool NuSMV takes a fraction of a second on a usual PC. We measured the execution times needed for verification by our planner, and they are also within a fraction of a second.

Hence, for this minimalist model, the execution times for verification are negligible for both approaches. As our planner is not really optimized for speed, it is conceivable that model checking will run faster for larger search spaces than our planner. Still, such a comparison for scaling is left for future work. Anyway, in theory the complexity is the same for both approaches (without bounds).

We use the Fluent Calculus as a means for verification in this paper, but such a model may be used for other purposes as well. Since the model primarily specifies *actions*, these specifications may also be used for planning or plan execution. After all, related approaches have been proposed long time ago in the context of robot planning and execution, see, e.g., (Nilsson, 1982). This is in contrast to model checking, where only verification is possible.

6 CONCLUSION

In this paper, we propose a new approach to formal and automated verification based on the Fluent Calculus. While previously only the specification of a software feature coordinator was published in (Bocovich and Atlee, 2014), our approach can formally verify it. This requires additional models of the environment (an autonomous agent and some simple physics), which facilitate verifying safety of software more generally.

This approach is fully implemented for the model in (Rathmair et al., 2016), from which our representation in the Fluent Calculus is systematically transformed from. We also propose this transformation here, which could be more generally used for deriving representations in the Fluent Calculus. This calculus (much as the Situation Calculus) is usually employed for planning and plan execution, where re-planning is more flexible at run-time than using FSMs.

It is hard to judge, in general, which formal verification approach is easier to use. For someone having specified a feature coordinator according to (Bocovich and Atlee, 2014) through such a calculus, using it for verification purposes as well seems more appropriate than using FSMs and model checking.

ACKNOWLEDGMENT

The FeatureOpt project (No. 849928), is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program “ICT of the Future” between June 2015 and May 2018. More information can be found at https://iktderzukunft.at/en/.

REFERENCES

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press, Cambridge, MA, USA.

Bensalem, S., Havelund, K., and Orlandini, A. (2014). Verification and validation meet planning and scheduling. *International Journal on Software Tools for Technology Transfer*, 16(1):1–12.

Bocovich, C. and Atlee, J. M. (2014). Variable-specific resolutions for feature interactions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 553–563, New York, NY, USA. ACM.

Hoch, R., Kaindl, H., Popp, R., Ertl, D., and Horacek, H. (2015). Semantic Service Specification for V&V of Service Composition and Business Processes. In *Proceedings of the 48nd Annual Hawaii International Conference on System Sciences (HICSS-48)*, Piscataway, NJ, USA. IEEE Computer Society Press.

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

Juarez-Dominguez, A. L., Day, N. A., and Joyce, J. J. (2008). 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. ACM.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59 – 83.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh.

Nilsson, N. J. (1982). *Principles of Artificial Intelligence*. Springer, Berlin, Heidelberg, Germany.

NuSMV (2014). NuSMV: a new symbolic model checker.

Rathmair, M., Luckeneder, C., and Kaindl, H. (2016). Minimalist qualitative models for model checking cyber-physical feature coordination. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, USA. IEEE.

Reiter, R. (1991). The frame problem in situation the calculus: A simple solution (sometimes) and a complete-

ness result for goal regression. In Lifschitz, V., editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA.

Schiffel, S. and Thielscher, M. (2005). Interpreting golog programs in flux. In *In 7th International Symposium On Logical Formalizations of Commonsense Reasoning, The*, USA. AAAI Press.

Thielscher, M. (1998). Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, 2:179–192.

Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *TPLP*, 5(4-5):533–565.

Winner, H. and Schopper, M. (2015). Adaptive cruise control. In *Handbuch Fahrerassistenzsysteme*, pages 851–891. Springer.