Formal Verification of A Domain Specific Language for Run-time Adaptation

Shahid Khan*, Faiq Khalid†, Osman Hasan* and João M.P. Cardoso‡
*School of Electrical Engineering & Computer Science, National University of Sciences & Technology, Islamabad, Pakistan
†Department of Computer Engineering, Vienna University of Technology, Vienna, Austria
‡Faculty of Engineering, University of Porto, Porto, Portugal
Email: {shahid.khan1, osman.hasan}@seecs.nust.edu.pk, faiq.khalid@tuwien.ac.at, jmpc@acm.org

***Abstract*—Compared to general purpose programming languages, domain specific languages (DSLs) targeting adaptive embedded software development provide a very promising alternative for developing clean and error free run-time adaptations. However, the ability to use several rules in a single adaptation strategy, as allowed by some DSLs, may lead to conflicts and reachability issues, which can eventually lead to functional bugs. Traditionally, such conflict analysis is done using software testing or manual manipulation of automata based models of rules. However, both of these techniques are error-prone and thus can lead to unwanted situations. As an accurate alternative, we propose to use model checking for rule conflict and reachability analysis in DSL adaptation. In particular, this paper provides an approach to formally model DSL adaptation specifications, along with their rules, and identifies a set of generic temporal properties to check for reachability and other rule conflicts, using the finite and infinite state-space based model checking capabilities of nuXmv model checker. For illustration, formal analyses of an energy aware CPU scheduling algorithm, i.e, PAST, for adaptivity rules for a stereo navigation system and for a context aware application are presented.**

I. Introduction

Most of the recent embedded systems have very high performance requirements, which also vary with changing environments [1], [2]. Coping up with these varying requirements under the tight constraints of embedded systems is a big challenge. Run-time software adaptations tend to solve these emerging issues by allowing the embedded systems to adapt according to their requirements in run-time and thus meet their stringent goals. Apart from core functionality, the run-time software adaptations also tend to achieve nonfunctional requirements, e.g., reduce energy consumption if lower system battery level is detected.

Traditionally, these embedded systems are developed using mainstream general-purpose programming languages (GPLs), like C/C++/Java, which do not conveniently support programming of dynamic adaptive behaviors. Therefore, adaptive behaviors are usually defined using hard coded conditional expressions, parameters and exceptions or by using programming techniques, like context oriented programming, feature oriented programming and change-oriented software engineering [3]. All these approaches have known limitations e.g., introducing parameters and conditional expressions in GPL results in anti patterns, e.g., blob and spaghetti code effects [3]. Context oriented programming approaches, on the other hand, allow an organized inclusion of adaptations in the core application software. Although, this gives comprehensible code compared to GPL, the problem of comprehension persists especially for larger systems. Similarly, feature oriented programming approaches offer no support for situational and operational variability of the core application [3].

Domain specific languages (DSLs) can mitigate the above-mentioned problems of GPLs [4]. Many DSLs [5] are successfully being used to serve specialized domains, including databases [6], web applications [7], railway systems [8] and industry automation [9]. Recently, a DSL was proposed [3] to cater for the aforementioned problems in the context of adaptive software development for embedded systems. This approach partitions software development into two components, i.e., *core application* and *adaptive application*. The *core application* allows specifying the static (fixed) parts of the program behavior, which usually consists of the application source code written in any target language, e.g., C or Java. Whereas, the *adaptive application* comprises of DSL specification of adaptations. This separation of functional and adaptive behaviors usually results in a clean and more comprehensible code by allowing the programmer to focus on adaptations separately. The tool, supporting DSL based software development, involve both core and adaptive applications at compile time. The adaptation application of DSL specifications can be further decomposed into several adaptation strategies, with only one strategy active at a given time. *Rules* form the most important part of a DSL strategy as they are responsible to define the adaptation behavior of the whole strategy. There are many priority functions through which one can ensure conflict-free execution of rules [3]. However, the fact that several rules can be defined in a single adaptation strategy may lead to reachability issues and conflicts between concurrent or sequential rules. These conflicts may lead to functional bugs in the target embedded system, which is an extremely undesirable characteristic, given the extensive application of embedded systems in safety-critical domains. Therefore, the rule conflict and reachability analysis of DSL specifications has become an integral step in their design. For example, a DSL specification for an adaptation scenario is presented in Code Listing 1, that imports one function (`infAct`) in which one rule is defined with two strategies, `EVENT` and `EVERY RUN`, to adapt the inference when the battery of the device

```
infAct IS fftKnnInf(FftNrSamples=2048)
  { IDLE, WALK, JOG };
  RUN[period=1sec]
    {
     //default implementation
      activityContext = infAct();
    }
  RULES
    {
       EVENT: ENERGY.LEVEL.LOW || CPU.LOAD.HIGH
        { //Rule A
         infAct IS fftKnnInf(FftNrSamples=512);
        }
       EVERY RUN: RUN.ELAPSEDTIME > RUN.period
        { // Rule B
        }
    }
```

Code Listing 1: ACI DSL Code [10]

reaches a low level, the CPU reaches high load or the elapsed time takes longer than the defined period.

Traditionally, DSLs are tested using GPL software verification techniques, i.e., test cases are written in GPL and the verification process ensures program correctness with respect to written test cases [11]. Another way to perform DSL testing is through writing test cases in DSL itself by extending DSLs in a way to express test cases [11]. However, these kind of analysis cannot be done exhaustively because of the large design-space of the given problem, which in turn is due to the concurrent nature of DSL rule executions and the many possible orders of execution, and the limited computational resources. Thus the problem is usually analyzed for a small set of possibilities. This choice compromises the completeness of the analysis and thus the accuracy of the analysis results. Moreover, choosing the sample set is another major issue while conducting rule conflict analysis using software testing as there are no general rules to select the testing cases to find all possible corner cases. As a deductive alternative, which is more effective than testing, a nondeterministic finite automata (NFA) based approach for rule conflict and reachability analysis is proposed in [3]. According to this approach, NFAs for every DSL rule, used in the given application, are first defined. The Cartesian product of these rules is formed, which is then traversed to identify rule conflicts and reachability issues. The identified conflicts are removed by revising the rules iteratively.

To mitigate the above-mentioned limitations, in this paper we propose to use model checking, which is a widely employed formal method [12]. The idea of using model checking for DSL verification has been previously used in other domains including web [13] and railway interlocking [14]. Although model checking based rule conflict analysis [15] and reachability analysis [16] are also available, to the best of our knowledge, model checking has not been used for verifying DSL code specifying adaptivity rules. To bridge this gap, in this paper we provide a formal verification methodology for embedded software developers, who wish to ascertain the conflict free compliance of their adaptation specifications to customer requirements using formal methods.

We use the nuXmv model checker [17] to conduct the rule conflict and reachability analysis. The main motivation behind this choice is to benefit from the scalable Satisfiability Modulo Theory (SMT) [18] based Bounded Model Checking (BMC) [19] capabilities and efficient verification algorithms of nuXmv. The first step in the proposed methodology is to develop models of DSL rules, used in the given adaptive strategy, in the Symbolic Model verification (SMV) language [20]. We have also identified a set of temporal properties that allow us to identify conflict situations. Once the model and specifications are written, the nuXmv model checker can be used to automatically check whether the model satisfies the specifications or not. However, the main concern in this approach is to translate the strategies for adaptivity to the respective formal model. Therefore, we propose a translator which translates the DSL rules into its respective SMV model. For illustration, we use it to formally analyze an energy aware CPU scheduling algorithm, i.e. PAST [21], adaptivity rules for a stereo navigation application [22], and for a context aware application [10].

The main contributions of this paper are as follows:

1) To the best of our knowledge, this is the first methodology which formally analyzes the rules, operation and strategies for the DSL to identify the contradictions that are very difficult to detect using typical methods.
2) A comprehensive translation scheme that allows to convert the DSL strategies, along with their rules and operations, into the corresponding SMV model.

II. Model Checking and nuXmv

Model checking [12] is used as a verification technique for reactive systems, i.e., the systems whose behavior is dependent on time and their environment. The inputs to a model checker include a finite-state model of the system that needs to be analyzed along with the intended system properties, which are expressed in *temporal* logic, a logic that allows expressing time-dependent behaviors. The model checker automatically and exhaustively verifies if the properties hold for the given system while providing an error trace in case of a failing property. The state-space of a system grows exponentially with the increase in the size of system variables and their possible values. Thus, it becomes computationally impossible to explore the entire state-space with limited resources of time and memory for larger models. This problem, termed as *state-space explosion*, is usually resolved by using efficient algorithms and techniques, like symbolic and BMC [12]. The main idea behind BMC is to allow the model checker to check the given property for a partial model, based on the user provided depth. The model checker detects the failing property if it fails in this reduced model. Otherwise, the depth of BMC is incrementally increased in search of a failing property.

The nuXmv is an extended version of NuSMV [17] and inherits all its capabilities, including Propositional Satisfiability (SAT) algorithms for finite state systems. For infinite state systems, it introduces new data types of `Integers` and `Reals` and also provides the support of Satisfiability Modulo Theories (SMT) [18] for verification. To cater for

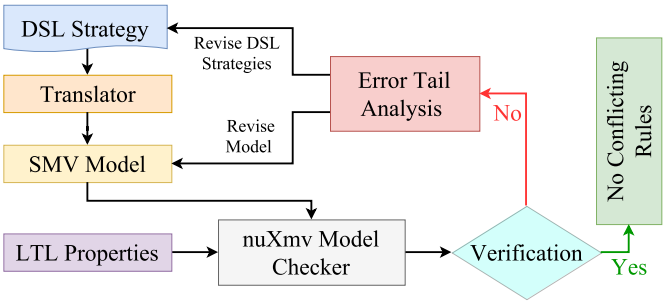

Fig. 1: Proposed Methodology

the state-space-explosion problem, nuXmv supports BMC. Although the approaches used by nuXMV are in general incomplete, a lasso-shaped counter example is always found if it is guaranteed to exists. The system to be modeled is expressed in the SMV language, which supports a modular programming approach. The properties to be verified can be specified in nuXmv using the Linear Temporal Logic (LTL), using temporal operators like Globally (G) and Finally (F) or the Computation Tree Logic (CTL).

III. Proposed Methodology

The proposed methodology for the formal rule conflict and reachability analysis of DSL adaptation strategies is depicted in Fig. 1 and can be divided into four main steps. The first step is to extract the SMV model from the operations, declaration, rules and codes of DSL adaptation strategies using the proposed translator. In this paper, we have identified a set of temporal properties that can be used to check all kinds of rule conflicts. These temporal properties can then be incrementally customized to specify temporal properties for the given required adaptation behavior. Subsequently, the model and temporal properties are provided to the nuXmv model checker to automatically check if the two correspond to one another. In case a failing property is found then its error trail is generated in nuXmv to investigate its cause. A property can fail due to a modeling error or an actual rule conflict. In case, it is a modeling error then the model is updated to rectify it. Whereas in case of an actual rule conflict both the model and the DSL adaptation strategy have to be revised to fix the issue. In both cases, the process is repeated with the revised model until all the properties are satisfied. The proposed translator translates the given DSL code to the corresponding formal model accepted by nuXmv and the verification in inherently automatic in model checking.

Now, we provide details associated with the two main steps of methodology, i.e., formal modeling of DSL strategies in the SMV and the generic set of properties that we have identified to develop temporal properties for the given DSL strategy.

A. Formal Modeling of DSL Strategies

In order to automate the formal analysis, verification and validation of the given DSL [3], we developed a comprehensive translator that extracts the SMV model from the given DSLs model [3] by using the following rules.

1) ***Declaration:*** In DSL [3], the declaration section allows specification of variables, functions, and other components, which are used in all other sections of the code. Thus, the first step of the proposed translator is to extract all the declared and temporary variables from the given DSL strategy. Then, these extracted variables are translated into corresponding, VAR (state variables), IVAR (Input variables) and DEFINE, sections of SMV, based on the following set of rules:

a) Independent variable should be declared as IVAR.
b) Variables having fixed value or no dependence on conditional statements should be declared in DEFINE section.
c) Variable having dependence on conditional statements or recursive definitions should be declared as VAR and all its dependants must be defined in ASSIGN section.

2) ***Operations:*** The operations section of the DSL describes important operational blocks in the computational process. The operations can be translated and included in the `DEFINE` section of the corresponding SMV model.

3) ***Rules:*** The rules of a DSL specify multiple adaptation actions, which are responsible for performing the necessary changes that control the behavior of the target application. These rules are translated and defined in the `ASSIGN` section of the SMV model to describe the state transitions.

4) ***Codes:*** This section allows the developer to extend the DSL by adding functionality that is not present (e.g., platform-specific code), or to extend the application without making changes directly to it. However, being specific to the targeting language, it reduces the DSL's interoperability as language, and thus cannot be translated into SMV automatically.

The translator nondeterministically assigns values to the DSL variables with unassigned values. Such nondeterminism is among the major advantages of the proposed approach as it allows implicit checking of corner cases during the verification process. In this paper, we propose to investigate both finite and infinite state based reactive systems. The `real` variable of nuXmv is used for infinite state space. Nevertheless, from a programmer's perspective, the procedure of writing SMV models for both types of reactive systems is the same.

B. LTL Properties for DSL Strategies

The main objective of an adaptation is to optimize the system execution for every predefined changing environment. This objective is primarily met by the application of a set of rules. We propose to check the following five properties for every DSL strategy. We believe that the successful verification of these properties would ensure that there are no functional bugs, including conflicts, in the given strategies.

1) ***Reachability:*** In this property, we verify that every reachable state of the model (states reachable by program) has at least one incoming and one outgoing transition. Thus, this property ensures that every state of the model is reachable and there is no deadlock present in the model. The deadlocks are also automatically reported by nuXmv during the property verification step. To check reachability of a particular state, LTL properties can be defined. For example, considering an adaptive strategy which adjusts the number of samples

Algorithm 1 Pseudocode of the SMV Model

```
MODULE main
Require: 0 ≥ EnergyLevel ≤ 100 and Memory ≥ 0
Instantiate Modules
re : energy(EnergyLevel)
rm : memory(Memory)
MODULE energy(EnergyLevel)
Ensure: energyBudget
if EnergyLevel < 10 then
    energyBudget = 50
else
    energyBudget = energyBudget
end if
MODULE memory(Memory)
Ensure: energyBudget
if Memory < 1000 then
    energyBudget = 75
else
    energyBudget = energyBudget
end if
```

```
rules
{
 r_energy : every (1 sec)
  {
   if( energy <10)
    { energyBudget = 50
      //...
    }
  }
 r_memory : every (1 sec)
  {
   if( memory <1000)
    { energyBudget = 75
      //...
    }
}}
```

(`FftNrSamples`) considered by a Fast Fourier Transform (FFT) algorithm from 2,048 to 512 whenever `CPU_load` increases from 80%, the following property checks the reachability regarding default state, i.e., 2,048:

```
G (CPU_LOAD < 80 & FFT.FftNrSamples = 512 ->F
(FFT.FftNrSamples = 2048))
```

This property ensures that there is always a path, starting from a state in which `CPU_LOAD` is less than 80 and the `FFT.FftNrSamples` is equal to 512, which leads the system eventually to a state where `FftNrSamples` of the FFT is set to 2,048 again.

2) ***Trigger Similarity:*** In this property, we check that multiple triggering conditions, occurring simultaneously, do not lead to contradictory behavior. The trigger similarity condition must be specified such that it fails whenever there are contradictory behaviors of adaptation in the given model. The case when a system can be in state of `adaptation1` and `adaptation2` simultaneously is a classical example of trigger similarity. The corresponding property can be expressed in LTL as follows:

```
G !(adaptation1 & adaptation2)
```

which states that always for all states it is not true that both adaptations, `adaptation1` and `adaptation2` are true simultaneously.

3) ***Action Similarity:*** This property checks if different concurrent or sequential rules access the same resources or parameters [3]. If concurrent rules access the same resources then there is a possibility to set different values simultaneously, which can raise conflicts. Similarly, another kind of conflict occurs if one rule tries to increase the value of a parameter value whilst the other one tries to decrease it. For example, the case when the two concurrent rules `memory` and `energy`, presented in Algorithm 1, try to set a single parameter `energyBudget`, with different values can be checked by the following property.

```
G (r1.energyBudget = r2.energyBudget)
```

which states that the value of `energyBudget` of the `memory` rule is always going to be equal to value of `energyBudget` of the `energy` rule.

4) ***Overriding Rules:*** Based on triggering conditions, some rules may become redundant. For instance, consider the case when three energy rules that set energy state of the system to `A`, `B`, `C` based upon whether the energy level is less than 10, 20 or 30, respectively. The following properties can be used to check if such conflict occurs in the given model.

```
G ( EnergyLevel < 10 -> X state = A)
G ( EnergyLevel < 20 -> X state = B)
G ( EnergyLevel < 30 -> X state = C)
```

The first LTL specification states that, whenever the value of the variable `EnergyLevel` is less than 10 then the next transition should lead the system to the energy state `A`. Alternatively, the second and third LTL specification states that, if energy level is less that 20 or 30, in next state, the system should be in the energy state `B` or `C`, respectively.

5) ***Incompatible Requirements:*** Adaptation requirements are provided by the customer but it is the responsibility of the developer to ensure that different adaptation requirements do not contradict each other. For example, consider the case when in a single adaptation strategy, one rule `r1` moves the system to a low energy state and another rule `r2` tries to increase its computation rate. The physical interpretation of these rules is of contradictory nature and thus this behavior can be checked using the following LTL property

```
G !(r1.state=LowEnergy & r2.state=HighRate)
```

which states that the system can not be in `LowEnergy` and `HighRate` states simultaneously.

The proposed methodology and modeling strategy, described in this section, can be used to model the DSL adaptation strategies, associated with any embedded system. The generic properties mentioned here, can be incrementally refined to describe any particular scenario and thus facilitate automatic rule conflict and reachability analysis.

IV. Case Studies

In this section, we conduct the formal rule conflict and reachability analysis using the proposed methodology, for the classical energy aware CPU speed algorithm PAST [21], adaptivity rules for a Stereo Navigation application [22] and adaptivity rules for a context inference application [10]. The proposed approach allowed us to discover some conflicting scenarios, which were undetected by the traditional analysis methods for the same DSL specifications. Our experiments have been conducted using Version 1.1.1 of nuXmv running on Windows 10 and 3GHz CPU with 4GB RAM.

A. PAST Algorithm

PAST is an energy aware CPU clock speed setting algorithm that predicts the behavior in the next window of time based on the observation acquired during a fixed window of time in the past [21]. These time windows are designated as intervals. PAST, as given in Algorithm 2, consist of four parts. The percentage of time for which the CPU runs during the interval is computed in first part. The run_cycles originate from two sources, i.e., the run-time in the trace data for the interval, and the $excess\_cycles$. The $excess\_cycles$ is used to represent the carry over from the previous interval. The carry over is generated if the CPU speed is set so slow

Algorithm 2 Pseudo-code of the PAST Algorithm [21]

Initialize:
run_cycles: Number of non-idle CPU Cycles in last interval
$idle\_cycles$: Number of Idle CPU Cycles
$excess\_cycles$: Number of Cycles left over from previous interval
while (1) **do**
  $idle\_cycles = hard\_idle + soft\_idle$
  $run\_cycles+ = excess\_cycles$
  $run\_percent = run\_cycles/(idle\_cycles + run\_cycles)$
  $next\_excess = run\_cyclesspeed * (run\_cycles + soft\_idle)$
  if $excess\_cycles < 0$ **then**
    $excess\_cycles = 0$
  end if
  $energy = (run\_cycles - excess\_cycles) * speed * speed$
  if $excess\_cycles > idle\_cycles$ **then**
    $newspeed = 1.0$
  else if $run\_percent > 0.7$ **then**
    $newspeed = speed + 0.2$
  else if $run\_percent < 0.5$ **then**
    $newspeed = speed - (0.6 - run\_percent)$
  end if
  if $newspeed > 1.0$ **then**
    $newspeed = 1.0$
  end if
  if $newspeed < min\_speed$ **then**
    $newspeed = min\_speed$
  end if
  $speed = newspeed$
  $excess\_cycles = next\_excess$
end while

that it cannot accommodate the load provided during the interval. This initial value is reduced by the number of cycles actually performed at the current speed and the soft idle time. The idle time is squeezed by protracting the run-times in the interval and this ability is represented by $next\_excess$ ($next\_excess = run\_cyclesspeed * (run\_cycles + soft\_idle)$) calculation. The only candidate available for elimination of idle is the "soft" idle , for instance waiting for keyboard events. The excess cycles and soft idle time during an interval are mutually related, the former approaches $run\_cycles * (1 - oldspeed)$ with the later approaching zero. The cycles for which the CPU is unable to serve during the interval are subtracted out first. They are subsequently accounted for in the next interval. This accommodation of unserved cycles is probably done at an increased CPU speed. The speed setting policy is defined in the fourth step. The clock rate adjustment is a simple heuristic that tries to smooth the fast to slow processing transitions. If the idle time of the system is lower than its busy time, then the clock speed is increased. Whereas, it is slowed down if the system is mostly idle. The reduction in clock speed is attained by assigning the new_speed to min_speed. The targeted, DSL strategy and rules are: The clock *"Speed Bound"* should remain within min_speed and 1. For the *"Energy Optimization"*, the clock speed should be slow, if the CPU is more idle.

The first step of the proposed methodology is to translate the PAST algorithm into the corresponding SMV model. This algorithm is represented using the *Declarations*, *Rules* and *Operations* sections as described in Section III-A. In the translated model, the run_cycles, $idle\_cycles$, $run\_percent$, $excess\_cycles$ and $energy$ is declared in the `DEFINE` section. Similarly, rule dependent parameters, such as $newspeed$, are declared in `VAR` section and all rules are translated in the form of assignment and the transition conditions of the state variables. The initialization and default values of `newspeed` and `excess_cycles` are set as `min_speed` and `next_excess`, respectively. Moreover, the minimum speed of the clock is taken as 0.2 as mentioned in [21]. The translated model of PAST algorithm is given below:

```
MODULE main
 VAR
  excess_cycles : real;
  next_excess : real;
  run_cycles : real;
  newspeed : real;
 DEFINE
  hard_idle := 10;
  soft_idle := 0;
  idle_cycles := hard_idle + soft_idle;
  run_percent := run_cycles/idle_cycles * ( 1 - run_cycles/idle_cycles +
                 ((run_cycles/idle_cycles)^2 - (run_cycles/idle_cycles)^3);
  speed := newspeed;
  energy := (run_cycles - excess_cycles) *
                speed^2;
  min_speed := 0.2;
  ASSIGN
   init(run_cycles):= 0;
   init(run_cycles):= run_cycles +
                        next(excess_cycles);
   init(newspeed):= min_speed;
   next(newspeed):=
    case
     excess_cycles > idle_cycles : 1;
     run_percent > 0.7 : speed + 0.2;
     run_percent < 0.5 : speed - (0.6 -
                                  run_percent);
     newspeed > 1 : 1;
     newspeed < min_speed : min_speed;
     TRUE : min_speed;
    esac;
   init(next_excess):= run_cycles - speed *
                    (run_cycles + soft_idle);
   next(next_excess):= run_cycles - speed *
                    (run_cycles + soft_idle);
   init(excess_cycles):= next_excess;
   next(excess_cycles):=
    case
      excess_cycles < 0 : 0;
      TRUE : next_excess;
    esac;
```

Code Listing 2: SMV model of PAST algorithm

The PAST algorithm contains two strategies, which are analyzed using five LTL properties presented in Section III-B.

1) ***Speed Bound*** strategy selects the clock speed in such a way that it remains within the given bound. This strategy consists of a single rule and thus have no conflicts.

2) ***Energy Optimization*** selects the clock speed, such that the the CPU operates at the minimum clock speed during the idle time . This strategy fails the following reachability property.

```
G(run_percent <= 0.7 & run_percent => 0.5 ->
X(newspeed < min_speed | newspeed > 1))
```
(I)

This property states that if the $run\_percent$ is between 0.5 and 0.7 then the clock speed should be within the

allowed range ($min\_speed <= newspeed => 1$). However, Algorithm 2 does not select the clock when the $run\_percent$ lies between 0.5 and 0.7, which means that the CPU does not operate if it remains 50% to 70% active. This problem can be fixed in two ways: Firstly, this condition can be merged with greater than 70% or less than 50% rules for $run\_percent$ and the second alternative is to assign it the min_speed.

B. Context Inference (CI)

Context aware applications are capable of inferring, e.g., user's context and then adjusting their behavior accordingly. This key characteristic makes them quite useful in many environments and they find wide applications in tour guides, healthcare system support and social networking. Our scope, in this paper, is limited to a context inference based mobile application for daily activity monitoring [10]. In this application, mobile phone infers the user activities like running, walking, standing and sitting. with a set of sensors and publishes this activity on social networking sites, like twitter and Hi5. Six DSL adaptation strategies have been proposed for this application [3][10]:

1) ***Algorithm Parameter (AP)*** strategy selects one of available five configurations to be active at a time, based on the CPU deadline.
2) ***Inference Levels (AL)*** strategy is composed of two rules that are responsible for adjusting the context inference computation rate based on the present state of energy level. The low inference rate is selected if the battery power is found to be below 70%. A further degradation in the battery level, i.e., less than 40%, reduces the inference rate further.
3) ***Adaptable Inference (AI)*** strategy consists of four rules. Rule A is used to adjust the inference computation rate based on the context history. Whereas, Rule B reduces the running frequency of different functions if the energy level is between 40% to 60%. Rules C and D further switch off some functionalities when the energy levels fall below 40% and 25%, respectively.
4) ***Adaptable Context Inference (ACI)*** strategy consist of two rules [10]. Rule A changes the FftNrSamples for the Fast Fourier Transport (FFT) algorithm from 2,048 to 512 when the CPU load increases or the energy levels become low. Rule B is responsible for increasing the running period of the strategy if the elapsed time is greater than the running period.
5) ***Sensor Availability (SA)*** This strategy consists of three rules, which monitor energy levels and sensor availability, and based on these conditions, one sensor from the GSM, WiFi and GPS is selected.
6) ***Adaptable Level Inference (ALI)*** strategy consists of four rules: Rule A checks the context history and if it is the same, then the inference computation rate is reduced. Rules B and C are energy related and Rule D monitors the CPU load and as soon as the CPU load increases from 70%, this rule sets off the level 2 inference computation algorithms.

```
1: MODULE main
2:  IVAR
3:   EnergyLevel : Low,High;
4:   CPULoad : Low,High;
5:   elapsed_time : integer;
6:  VAR
7:   fftNrSamples : 2048,512;
8:   run_period : 2..6;
9:  ASSIGN
10:   init(fftNrSamples) := 2048 ;
11:   init(run_period) := 5 ;
12:   next(fftNrSamples) :=
13:    case
14:     EnergyLevel = Low | CPULoad = High : 512;
15:     TRUE : fftNrSamples;
16:    esac;
17:   next(run_period):=
18:    case
19:      elapsed_time > run_period : 2;
20:      TRUE : run_period;
21:    esac;
```

Code Listing 3: ACI Translated SMV Model

Similar to the PAST algorithm, we have also translated the CI aware application, presented in [10], into the corresponding SMV model [23]. Code Listing 1 shows one of the context inference scenarios, where a strategy for sensor changes according to the battery level or sensor unavailability is enforced. For illustrating the translation process, the corresponding SMV model is shown in Code Listing 3. Line 1 of Code Listing 1 declares the inference function `infAct`, Lines 2-3 define how and when the inference process is computed. whereas, Lines 4-8 of Code Listing 1 define the rules defining strategies to adapt the inference when specific events or conditions occur, i.e., when the battery of the device reaches a low level, when CPU reaches high load or when the inference elapsed time takes longer than the defined period. The default implementation scenario of activity context of DSL code is translated into the respective default case statements and defined strategies for rules A and B are translated into the `ASSIGN` section of the corresponding SMV model as shown in Code 3.

The CI aware application contains six strategies. We have analyzed the five LTL properties presented in Section III-B for all of them and the results are summarized below:

1) ***Algorithm Parameter (AP):*** This strategy consists of a single rule, and was evidently found to have no conflicts, as shown in Table I.
2) ***Inference Levels (AL):*** We found a problem in the rules specification, regarding the same parameter manipulation by two rules, i.e., high inference rate was selected when the battery level is below 40%, and on the contrary low inference rate was selected when battery level is above 40%. Swapping the values of inference rates eliminated this issue in this strategy. The strategy also failed the reachability property as the default state was found to be unreachable after mobile recharging activity, as shown in Table I. We resolved this issue by adding a condition that the default inference rate is reachable whenever the `battery_level` >70%.
3) ***Adaptable Inference (AI):*** We detected four reachability violations again. Rule A was found to not set the compu-

tation rate to its default value even if the context history changes. The other three issues are again related to the recharging condition, like the problem found in the last strategy, i.e., Rules B, C, and D adjust or switch off some functionalities based upon reducing energy levels but they do not switch on the functionalities, when the energy is recouped, as shown in Table I. These issues were resolved by introducing additional conditional checks associated with the unreachable states. We also found an issue of over riding rules between Rules B and C as once the energy goes below 40%, the Rule C switches off two algorithms. Recharging of device will activate Rule B again which will redundantly adjust parameters of these switched off algorithms.

4) ***Adaptable Context Inference (ACI):*** We detected three reachability failures in this strategy, as shown in Table I. Rule A was not found to be resetting the FFT parameters back to 2,048, when the CPU load becomes low or the energy levels are recouped. The third reachability failure was found to be related to the run period settings. We included additional checks in the SMV models to eliminate these issues. Moreover, two rules of this strategy violate the incompatible requirements property, because high CPU load and increased inference elapsed time are correlated events. This strategy fires two actions at this correlated event and reduces the window size for the FFT algorithm as well as increases the inference period. We used predicates to discern the reason of increased inference elapsed time and perform appropriate adaptation.
5) ***Sensor Availability (SA):*** Several reachability issues were spotted with this strategy. For example, the desired GPS sensor, once deselected, was found to be unavailable even if the system energy level goes back to 100% or GPS service is available again, as shown in Table I. Similarly, WiFi, the second priority sensor, could not be selected again if it becomes temporarily unavailable. This strategy also suffers from the action similarity conflict when the energy is <50% and the GSM and WiFi sensors are available but the GPS is unavailable. It was found that Rules 1 and 2 try to set the sensor value differently every 5 mins.
6) ***Adaptable Level Inference (ALI):*** We found two reachability issues with this strategy, as shown in Table I. Firstly, Rule A sets the inference rate to low but never sets it high despite variations in the context history. Secondly, Rule D does not set the inference computation level 2 ON when CPU load becomes less than 80%. Moreover, this strategy was found to have one violation of action similarity property, i.e., when the energy level is between 60 and 30 % and the context history is the same then Rules A and B try to set different values for the `run.period`. Another action similarity conflict arises between Rules A and C when the energy level is below 30%. Similarly, an overriding rule violation between Rules C and D was also found, i.e., when the CPU load is above 80% and the battery power is below 30%, then Rule C redundantly adjusts the window size of the `inference2` algorithm whilst Rule D has already switched off level 2 algorithms including `inference2`.

TABLE I: Identified Failures in Context Inference [3][10].

Strategy Name	Number of Rules	Failing Property	Failing Rules	Time
Strategy 1	1	Nil	Nil	Nil
Strategy 2	2	Reachability	2	0.06
		Action Similarity	2	0.03
Strategy 3	4	Reachability	4	0.05
		Reachability	2	0.02
		Reachability	2	0.02
		Reachability	2	0.11
		Overriding Rules	2	0.02
Strategy 4	2	Reachability	1	0.03
		Reachability	1	0.03
		Reachability	1	0.05
		Incompatible Requirement	2	0.05
Strategy 5	3	Reachability	1	0.03
		Reachability	1	0.02
		Incompatible Requirements	2	0.03
		Action Similarity	2	0.03
Strategy 6	4	Reachability	1	0.02
		Reachability	1	0.03
		Action Similarity	2	0.02
		Action Similarity	2	0.06
		Overriding Rules	2	0.05

C. Stereo Navigation

Guidance Navigation Satellite System (GNSS) is used to provide vehicle navigation related services. However, there is always a possibility that GNSS based service becomes temporarily unavailable. In such cases, a stereo navigation application can be used to perform the navigation related activities. This application uses cameras to acquire stereo vision capability. From processed images and saved maps, the stereo navigation application performs different navigation related activities. To maintain Quality of Service (QoS), stereo navigation application has to undergo many real-time adaptations. In this case study, six adaptations are used [22]. In all these available adaptation strategies, candidate parameters for adaptation are image resolution and number of iterations of a RANdom SAmple Consensus (RANSAC) algorithm.

The first five strategies consist of only one rule so they are not interesting for rule conflict and reachability analysis. We performed formal analysis of the sixth strategy, i.e., `Image Resolution, RANSAC Gradual Adapt`. This strategy consists of three rules and adjusts both candidate parameters, i.e., image resolution and RANSAC iterations based on the slack which is a function of CPU load. We identified an issue related to action similarity, according to which there are repeated traces where in one state image resolution is increased by Rule `r3` and in the next state it is decreased by other rule, i.e., `r2`. The action similarity violation with respect to RANSAC iterations was also found to be present in Rules `r3` and `r4`. The identified failures are summarized in Table II.

The proposed formal analysis methodology allowed us to find many interesting issues that were unidentified by the manual NFA based analysis. The verification times for the formally verified properties are in fractions of a second, which further advocates the usage of the proposed approach for rule conflict and reachability analysis of sizable adaptation definitions. It is pertinent to reiterate that, we specified and checked

TABLE II: Identified Failures in Stereo Navigation [22]

Strategy Name	Number of Rules	Failing Property	Failing Rules	Time
Strategy 6	3	Action Similarity	2	0.52
		Action Similarity	2	0.38

each property one by one. The conflicts identified in DSL specifications were rectified using restriction predicates and priority functions. From the experience of conducting these case studies, we have learned that the definition of properties was also found to be quite tricky because the identification of properties that correctly investigate presence of conflicting situations can become very difficult if the customer specifications/requirements are incomplete, which is a common occurrence. However, the verification process provides very useful insights and thus the models and properties can be refined during the process.

V. Conclusion

This paper presented a symbolic model checking methodology for formal rule conflict and reachability analysis of a domain specific language (DSL) specially designed for specifying run-time adaptivity strategies. The proposed approach has the ability to investigate both finite and infinite state machines. Moreover, reachability analysis is also performed using simple temporal properties, which can be easily understood and implemented by any embedded system developer. For illustration purposes, a number of adaptivity strategies for energy aware CPU speed algorithm, adaptivity rules for a stereo navigation system and for a context inference application have been verified using the proposed approach and conflicts have been identified.

References

[1] P. Koopman, "Embedded system design issues (the rest of the story)," in *Computer Design: VLSI in Computers and Processors*, 1996, pp. 310–317.

[2] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime Software Adaptation: Framework, Approaches, and Styles," in *Companion of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 899–910.

[3] A. C. Santos, "A DSL for Specifying Run-time Adaptations for Embedded Systems," Ph.D. dissertation, Universidade De Lisboa, 2014.

[4] D. Ghosh, *DSLs in Action*. Manning Publications Co., 2010.

[5] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[6] S. S. Harkins and M. W. Reid, "Structured query language," in *SQL: Access to SQL Server*, 2002, pp. 1–5.

[7] D. M. Groenewegen, Z. Hemel, L. C. Kats, and E. Visser, "WebDSL: a Domain-specific Language for Dynamic Web Applications," in *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. ACM, 2008, pp. 779–780.

[8] A. E. Haxthausen and J. Peleska, "A Domain Specific Language for Railway Control Systems," in *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology,(IDPT2002), Pasadena, California*, 2002, pp. 23–28.

[9] C. Preschern, A. Leitner, and C. Kreiner, "Domain Specific Language Architecture for Automation Systems: An Industrial Case Study," *ECMFA12 Proceedings*, 2012.

[10] A. C. Santos, P. C. Diniz, J. M. Cardoso, and D. R. Ferreira, "A Domain-specific Language for the Specification of Adaptable Context Inference," in *Embedded and Ubiquitous Computing*, 2011, pp. 268–273.

[11] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[12] O. Hasan and S. Tahar, "Formal verification methods," *Encyclopedia of Information Science and Technology, IGI Global Pub*, pp. 7162–7170, 2015.

[13] A.-M. Törsel, "A testing tool for web applications using a domain-specific modelling language and the nusmv model checker," in *Software Testing, Verification and Validation*, 2013, pp. 383–390.

[14] Y. Cao, Q. Lu, T. Xu, T. Tang, H. Wang, and Y. Xu, "Integrating dsl-cbi and nusmv for modeling and verifiying interlocking systems," in *Secure Software Integration & Reliability Improvement Companion*, 2011, pp. 136–143.

[15] A. Awad, M. Weidlich, and M. Weske, "Consistency checking of compliance rules," in *Business Information Systems*, 2010, pp. 106–118.

[16] A. W. To and L. Libkin, "Recurrent reachability analysis in regular model checking," in *Logic for Programming Artificial Intelligence and Reasoning*, 2008, pp. 198–213.

[17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification*, 2014, pp. 334–342.

[18] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[20] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Computer Aided Verification*, 1996, pp. 419–422.

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Mobile Comp.*, 1994, pp. 449–471.

[22] A. C. Santos, J. M. Cardoso, P. C. Diniz, D. R. Ferreira, and Z. Petrov, "A DSL for Specifying Run-time Adaptations for Embedded Systems: An Application to Vehicle Stereo Navigation," *The Journal of Supercomputing*, vol. 70, no. 3, pp. 1218–1248, 2014.

[23] S. Khan, O. Hasan, F. Khalid, and J. M. Cardoso, "Formal Verification of DSL for Run-time Adaptation," http://save.seecs.nust.edu.pk/projects/fDSL/, 2017.