Extendable Toolchain for Automatic Compatibility Checks

Vincent Bertram[1], Alexander Roth[1], Bernhard Rumpe[1,2], and Michael von Wenckstern[1]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
[2] Fraunhofer FIT, Aachen, Germany

Abstract. Embedded software systems are highly configurable and consist of many software components in different variants and versions. However, component updates or upgrades often result in unpredictable incompatibilities with its environment. Existing research addresses this challenge by employing formal methods with a fixed set of encoded static compatibility checks, making it nearly impossible for engineers to add new or modify existing ones. This paper presents a highly adaptable infrastructure to define constraints for compatibility checks. The underlying approach transforms software components into instances of a C&C meta-model, enriched with OCL compatibility constraints at runtime, then evaluated by a solver. The result is transformed back into a C&C model showing compatibility or incompatibility. The easy to integrate infrastructure is based on industrial requirements and allows to add, modify or delete constraints without restarting the tool infrastructure.

1 Introduction

Software systems for embedded devices are highly configurable consisting of a variety of components in different variants and versions [10]. Because these components are tightly integrated into their environment, which consists of multiple other software components, updates/upgrades often result in unpredictable incompatibilities between its interacting components. In many cases updates/upgrades are possible by adapting the component. However, to determine compatibility, as defined in [18], static compatibility checks with a fixed set of constraints are proposed [16], which hampers adding new or modifying existing constraints. This paper presents an adaptable infrastructure based on industrial requirements to define and adapt compatibility constraints for checks between components using OCL at runtime. It is based on the meta-model as proposed in [3], which allows, in contrast to other verification frameworks [8,16,19], besides intra- also inter-model verification of constraints between Component & Connector (C&C) models of even different modeling languages such as *Simulink* [12] or *Modelica* [13]. The underlying approach transforms models of software components to instances of the C&C meta-model, which is enriched with OCL compatibility constraints. The instances and constraints are then evaluated by a solver and transformed into a user friendly C&C model showing (in)compatibility based

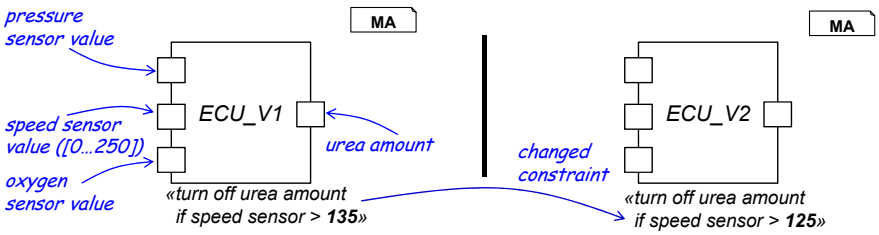

Fig. 1: Two versions of a simplified emission control ECU having different constraints

on a counter-example [15]. Our infrastructure allows to modify, delete, and add constraints and supports the addition of new transformations to check arbitrary (heterogeneous) C&C architectures without changing the infrastructure itself.

Next is an example in Sec. 2 which is followed by a set of industrial requirements in Sec. 3. Sec. 4 introduces the tools and languages used by the infrastructure proposed in Sec. 5. Sec. 6 describes the technical realization. Finally, Sec. 7 and Sec. 8 compare our approach to existing work and conclude this paper.

2 Motivating Example

As a motivating example, we use a simplified emission control system as used in automotive industry. The Electronic Control Unit (ECU) has an input for the pressure value of the exhaust system, a speed sensor for the current velocity, and an oxygen sensor. Using these values, the urea amount is triggered to clean emissions. Now a team of developers is responsible to develop a new version of the controller to maximize gas mileage. In this context, `ECU_V2` has been developed. It has the same ranges for all input ports but the constraint when to turn off the emission control has been changed from 135 km/h to 125 km/h (see Fig. 1).

Due to varying regulations in different countries (in Germany the emission control can be turned off if the speed is greater 120 km/h, whereas in the US it can be turned off if speed is greater 128 km/h), a developer team member is unsure if the new version can be used in the US and in Germany. Her doubts are justified: on the component's type level the different versions seem to be compatible. However, due to the additional constraint they are not. In particular, for the German market, the newer version is compatible (requires a turn off if speed > 120 km/h), whereas in the US the new version is *not* compatible because the emission control is turned off too early. Our proposed solution does not only allow to check such compatibilities between different component versions, but it also allows to dynamically en-/disable OCL constraints, e.g. local market regulations such as emission control, during tool runtime.

3 Industrial Requirements

The proposed approach is developed with respect to a set of requirements, which are derived from the SPES_XT project[1]. They indicate that a differentiation be-

[1] `http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html`

tween engineers (who are using the toolchain) and developers (who are extending the toolchain), is required. Subsequently, a set of the main requirements is introduced:

(R1) *Compatibility constraints have to be defined in comprehensive and concise notation:* Most engineers are only familiar with Java- or C-like syntax. Therefore, specifying constraints must hide all model-checking aspects.

(R2) *The tool should support heterogeneous C&C architecture models:* Since in industry various C&C architectures including third party plugins or in-house developments are used, the tool should be easily extendable.

(R3) *Developers should be able to modify structural compatibility constraints:* Compatibility rules must be adaptable instead of hardwire them to support different local markets and to be flexible for changing laws.

(R4) *Meaningful and model related error messages:* Reported violations must be easy to comprehend for engineers, having no deep knowledge of formal methods. Therefore, expressive descriptions relating to source C&C models should point to components being affected with incompatibility.

(R5) *C&C model files should not be modified:* The tool should also derive compatibility statements (probably weaker ones, because of missing information) between older (without touching them) and newer *Simulink* models.

(R6) *Compatibility checking should be easy for engineers:* A seamless integration into existing C&C modeling tools such as *Simulink* should be supported.

4 Modeling C&C Architectures and Constraints

The realization of the proposed approach is based on the MontiCore (MC) framework [11], which is a light-weight and highly customizable language workbench supporting language development and facilitates all elements of language definition, language processing, and code generation.

A language family, realized with the MC framework, is the UML/P language family [17], which is rooted on the Unified Modeling Language (UML) family. It provides a set of modeling languages, supporting reduced modeling language concepts with clear semantics, addressing different aspects of software development. For example, the UML/P Class Diagram (CD) modeling language can be used to model structural aspects. Instances, of this structure can be described using the UML/P Object Diagram (OD) language, and constraints for the structural model can be defined by the UML/P object constraint modeling language (OCL/P). To model C&C architectures, *Simulink*, an industrial modeling tool is used. The MC framework has been extended to process *Simulink* models and to generate queries for the Microsoft Z3 Satisfiability Modulo Theories (SMT) solver [6], which finds solutions for first-order decisions problems.

5 Extendable Infrastructure for OCL Evaluation

An approach has been developed to check for structural compatibility of two components, which satisfies the requirements described in Sec. 3 and is based

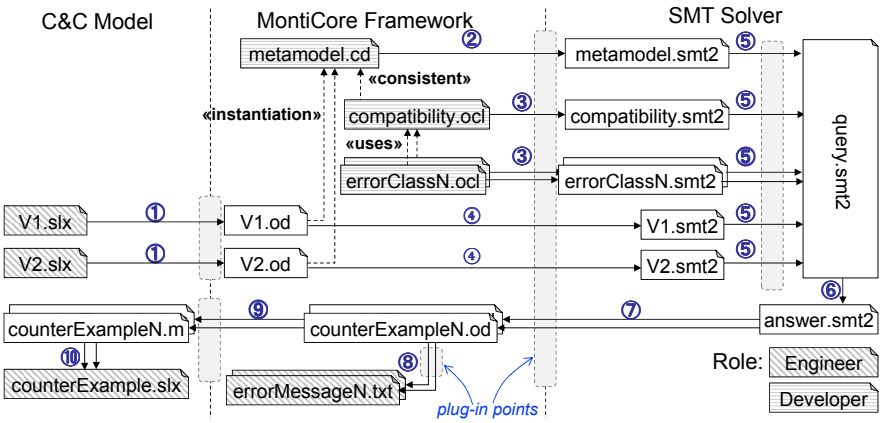

Fig. 2: Compatibility checking workflow on the example of two *Simulink* models

on C&C architectures and user-friendly (R1) constraint modeling language. The proposed approach, shown in Fig. 2, is composed of five *plug-in points*, which enable dynamic support for different modeling languages such as *Simulink*, *TargetLink* and *Modelica* and solvers such as Alloy [2], Yices [6], and Z3 [14] (R2). Since all transformations are dynamically executed during the checking process, redefinitions and extensions of compatibility definitions and compatibility variations (e.g. for local markets) are supported (R3).

The numbers (1) to (10) in Fig. 2 represent the processing steps: (1) transforms C&C input models to ODs, which are an instantiation of the meta-model for all well-established and commonly used C&C modeling languages. More information about this meta-model can be found in one of our previous work [3]. (2), (3) and (4) generate for all UML/P language family artifacts (CDs, OCL constraints, and ODs) solver specific ones. The parts are merged to one query document in (5), and evaluated by the solver in (6), which produces an artifact containing - in case of incompatibility - instantiations of error classes. Each error class is transferred back to one or several counterexample ODs in (7). These error class ODs are used to generate user-friendly text messages in (8) and to produce in (9) Matlab code generating visual counterexamples with minimal witnesses when executed in (10). Underspecification, which may occur in transformation steps (3) and (4), when checking compatibility constraints between older (not enriched model with less information) and newer (more information) C&C components (R5), is handled by dynamically not executing particular constraints.

The toolchain is divided into a C&C Engineer Frontend, a MC Developer Frontend, and a Solver Backend, each of which addressed by either an automotive engineer modeling Advanced Driver Assisted System (ADAS) functionality (gray-slashed artifacts) or a quality control/assurance (QA/QC) developer specifying compatibility constraints and different error classes (gray-lined artifacts).

Engineer View The automotive engineer initiates the compatibility check in *Simulink* (R6), therefore the engineer must not learn a new tool. The result annotated with error descriptions is presented as a *Simulink* model (R4), since

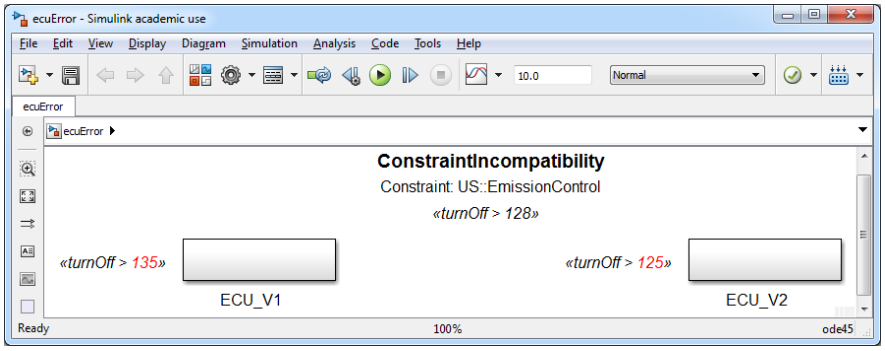

Fig. 3: Structual compatibility errors are illustrated as *Simulink* model witnesses

```
def boolean infix (Number v) in (Range r) is:                    OCL/P
   result = v >= r.min && v <= r.max &&
      (~r.res || (v - range.min) % range.res == 0)
```

(Range *r* has no optional association *res* to Resolution)

```
(define-fun In_Number_Range((v Number) (r Range)) Bool                Z3
   (and (GreaterThan_Number_Number v (minimum r))
        (LessThan_Number_Number v (maximum r))
        (or (not (resDefined r))
            (Equals_Number_Number
                (Mod_Number_Number (Minus_Number_Number v (minimum r))
                                   (resolution r))
                (mk-number 0)  ))))
```

Fig. 4: Comparison between OCL code and Z3 code generated by ③ in Fig. 2

the engineer already knows *Simulink* with its syntax and semantics, thus, he is able to understand the error messages. The witness generation algorithm is based on previous work done by Ringert [15] and is showing the constraint violating part of the model (see Fig. 3). The example points directly to the violated constraint `US::EmissionControl` when checking backwards compatibility of `ECU_V2` to `ECU_V1`. As the algorithm does not find a witness violating `EU::EmissionControl`, backwards compatibility is still given for the German market.

Developer View The developer defines compatibility constraints and counterexamples as well as additional regional constraints as presented in Sec. 2 in OCL/P using the MC developer frontend. These counterexample specifications are used to generate useful feedback for automotive engineers in case of component incompatibility as shown in Fig. 3. Fig. 4 illustrates how MC transforms OCL/P code to SMT code for use with the Z3 solver. A more detailed textual description of the code and how a counterexample in OCL/P looks like can be found in [4]. Our student survey found out that OCL/P code is easier to learn than SMT code, because the syntax is Java like, especially with the infix operator notation, whereas most students are not very familiar with SMT's prefix notation making it more difficult to match what arguments belongs to what prefix-operator. A more complex example, comparing two ADAS, and showing how unoptimized generated SMT code actually looks like is available at:

`http://rise4fun.com/Z3/2AsLg`

6 Performance Considerations

The overall performance of this approach mainly depends on the generator. In particular, how it generates solver-specific code from the UML/P language family artifacts. Since our motivating example is too small, a bigger model comparing interface compatibility of two ADAS is taken from the SPES_XT project. Tab. 1[2] shows the impact of the generator where model A is compatible to model B, and model A has 126 components as well as model B has 97 components. Note that all artifacts are translated into SMT code using different generator strategies, and then evaluated by Microsoft's Z3 solver.

This table underlines that the verification execution time mainly depends on the optimizations implemented by the generator. The difference between the 2nd and 3rd column is that in the 2nd one no simplify constraints had been generated and the 3rd one added the `simplify solve-eqs smt` command to the generated SMT code. Due to the highly customizable toolchain infrastructure providing several plug-in points, it is possible to integrate different optimization steps. Tab. 1 also shows that it was a good idea to introduce CDs and OCL constraints as intermediate layer, because this layer is independent of all solver specific optimizations; translating *Simulink* models directly to SMT code and formulating the compatibility constraints directly in Z3 code would probably result in changing them all the time and polluting the model with solver strategy annotations just for performance reasons - which would make them very hard to read and nearly impossible to reuse at the end.

model	**time [s]**	**time* [s]**	**change in generated Z3 code**
m1	timeout	10.08	-
m2	126.68	10.44	remove custom datatypes
m3	93.55	12.86	change encoding of meta-model
m4	138.38	10.47	use `ite` (if-then-else) instead of `implies` after quantifier
m5	70.74	8.34	replace enumeration datatypes by integers
m6	19.05	4.33	replace id hash with an unique id starting at zero
m7	15.17	4.23	remove unnecessary `ite`s when translating OCL to Z3

Table 1: Impact of generated SMT code on Z3's execution time.
* added `simplify solve-eqs smt` as solver strategy to the generated artifact.

In the first version (m1) the meta-model, the ODs and the OCL constraints were nearly one by one translated to SMT code. The result was quite readable SMT code, as this code shows for the `Connector` element, which connects a source port, `(List Name)` represents its full qualified name, with a target port (see Fig. 5). Note that accessing some parts of a C&C model element could also be done very intuitively, e.g. `(source con)` where `con` is a `Connector` variable.

In the last generated model version (m7) all meta-model element structures were flattened to integers. This means there is no such `Connector` element in Z3 code; instead of every element is represented by an unique `id` and accessing a C&C element part is now done by accessing a function (see Fig. 5). Additionally, all names, e.g. port names, have been replaced by an unique integer.

[2] measured by Nicolai Strodthoff in his bachelor thesis

```
; meta-model definition                                                    Z3 ...
(declare-datatypes () ((Connector (mk-connector (source (List Name))
   (target (List Name)) (id ID)))))                          (code is an excerpt)
; instance creation
(mk-connector (insert n_switch1 (insert n_out1 nil))
  (insert n_mul (insert n_in2 nil)) id_1593458942)
```

```
(define-fun getConnectorSourceFromId ((id Int)) (List Int)                  Z3 ...
(declare-datatypes () ((Connector (mk-connector (source (List Name))
   (ite (= id 2) (insert 2 (insert 56 nil))
   (ite (= id 14) (insert 0 (insert 56 nil))
```

Fig. 5: Z3 code used in first version (top) and last version (bottom)

The entire post-processing step such as flattening data structures and replacing names by integer number is done by using the plug-in point (5) in Fig. 2 after all documents have been merged to one query document. The last optimization from m6 to m7 used the plug-in point (6) by removing `ite`s in Boolean OCL constraints. The code `(ite cond1 true (ite cond2 false (ite cond3 true ...)) true)` will be replaced by `(or cond1 cond3 ...)`.

This study shows that the modular and extendable architecture supports flexibility, i.e., replace the Z3 solver with another one like Alloy, and adaptations of generator algorithms to optimize performance.

7 Related Work

Besides our toolchain, there exist several frameworks for compatibility analysis. We want to mention here six representatives: (1) Dajsuren's architectural framework [7], which checks consistencies between multiple software views by lifting their detailed information up to more abstract functional views. (2) *ActiveTreaty* [20] validates component compatibility (interface as well as behavioral one) using contracts which can be described in Java or OCL using the Dresden OCL [9] Eclipse plugin. (3) *USE* [5] allows to specify and check OCL constraints on UML models. (4) *MATE* [19,1] defines constraints directly in *Simulink*; this results in modifying all existing *Simulink* models when adding new constraints (violating (R5)). (5) *Massif* [8] on the other hand, provides a specific *MATLAB* Ecore meta-model and a tool instantiating these meta-models (satisfying (R5)); but the *Simulink* specific meta-model hampers support for heterogeneous models (see (R2)). (6) *SimCheck* [16] can check (only) intra-model compatibilities and can generate counterexamples by annotating *Simulink* models; however, *SimCheck* hard-wires their data type, unit and dimension checks directly into *Simulink* models by annotating them with 7-tuple structures basing directly on their solver language; which makes it very hard to maintain these constraints for developers (see (R3)).

8 Conclusion

Update of software components are very unpredictable due to different versions, variants, and configuration options. Based on a set of industrial requirements and

on running example, a highly adaptable infrastructure to check compatibility constraints is presented. It is based on a generic meta-model and employs OCL at runtime, which allows high customizability and heterogeneous support for C&C architectures. The underlying approach supports different requirements of engineers and QA/QC developers to satisfy future infrastructure extensions and provide a user friendly witness-based error interface.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: Moflon:a standard-compliant metamodeling framework with graph transformations. In: ECMDA-FA (2006)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software & Systems Modeling (2008)
3. Bertram, V., Manhart, P., Plotnikov, D., Rumpe, B., Schulze, C., von Wenckstern, M.: Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In: Modellierung (2016)
4. Bertram, V., Roth, A., Rumpe, B., von Wenckstern, M.: Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL. In: OCL16 (2016)
5. Brüning, J., Gogolla, M., Hamann, L., Kuhlmann, M.: Evaluating and Debugging OCL Expressions in UML Models. In: TAP (2012)
6. Cok, D.R., Stump, A., Weber, T.: The 2013 Evaluation of SMT-COMP and SMT-LIB. Journal of Automated Reasoning (2015)
7. Dajsuren, Y., Gerpheide, C.M., Serebrenik, A., Wijs, A., Vasilescu, B., van den Brand, M.G.: Formalizing correspondence rules for automotive architecture views. In: QoSA (2014)
8. Hegedüs, A., Starr, R.R., Búr, M., Nascimento, L., Dóczi, R., Mirachi, S., Ráth, István und Horváth, A.: Massif: Matlab simulink integration framework for eclipse (2015), http://github.com/FTSRG/massif
9. Hussmann, H., Demuth, B., Finger, F.: Modular architecture for a toolset supporting OCL. Science of Computer Programming (2002)
10. Kasoju, A., Petersen, K., Mäntylä, M.V.: Analyzing an automotive testing process with evidence-based software engineering. IST (2013)
11. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a Framework for Compositional Development of Domain Specific Languages. STTT (2010)
12. Mathworks: Simulink User's Guide. Tech. rep. (2015)
13. Modelica Association: Modelica - A Unified Object-Oriented Language for Systems Modeling. Tech. rep. (2012)
14. Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS (2008)
15. Ringert, J.O.: Analysis and Synthesis of Interactive Component and Connector Systems. Shaker Verlag (2014)
16. Roy, P., Shankar, N.: Simcheck: A contract type system for simulink. Innov. Syst. Softw. Eng. (2011)
17. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer (2016)
18. Rumpe, B., Schulze, C., von Wenckstern, M., Ringert, J.O., Manhart, P.: Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In: SPLC (2015)
19. Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: The mate approach: Enhanced simulink and stateflow model transformation. In: MathWorks Automotive Conference (2007)
20. Wilke, B.C., Dietrich, J., Demuth, B.: Event-Driven Verification in Dynamic Component Models. In: WCOP (2010)