Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1737–1746
DOI: 10.15439/2016F404
ACSIS, Vol. 8. ISSN 2300-5963

ReSA Tool: Structured Requirements Specification and SAT-based Consistency-checking

Nesredin Mahmud*, Cristina Seceleanu*, Oscar Ljungkrantz†
*Mälardalen University, Sweden, {nesredin.mahmud, cristina.seceleanu}@mdh.se
†Volvo Group Trucks Technology, Sweden, oscar.ljungkrantz@volvo.com

***Abstract*—Most industrial embedded systems requirements are specified in natural language, hence they can sometimes be ambiguous and error-prone. Moreover, employing an early-stage model-based incremental system development using multiple levels of abstraction, for instance via architectural languages such as EAST-ADL, calls for different granularity requirements specifications described with abstraction-specific concepts that reflect the respective abstraction level effectively.**

In this paper, we propose a toolchain for structured requirements specification in the ReSA language, which scales to multiple EAST-ADL levels of abstraction. Furthermore, we introduce a consistency function that is seamlessly integrated into the specification toolchain, for the automatic analysis of requirements logical consistency prior to their temporal logic formalization for full formal verification. The consistency check subsumes two parts: (i) transforming ReSA requirements specification into boolean expressions, and (ii) checking the consistency of the resulting boolean expressions by solving the satisfiability of their conjunction with the Z3 SMT solver. For validation, we apply the ReSA toolchain on an industrial vehicle speed control system, namely the Adjustable Speed Limiter.

I. Introduction

Most often, the development of dependable automotive systems that are nowadays increasingly complex [1] relies on intricate requirements, given the nature of the system that has to interact with the environment. Therefore, the importance of establishing non-ambiguous and consistent requirements is even higher than for closed systems. Despite this acknowledged situation, current specification methods and tools [2][3][4] lack adequate support to formally analyze the logical consistency of high-level natural language requirements, in order to improve the quality of their specification.

Moreover, to be able to manage the complexity of automotive embedded systems during development, incremental model-based design approaches that assume multiple levels of abstraction are becoming appealing to industry. Among others, dedicated architectural languages, such as Electrical and Software Technology - Architectural Description Language (EAST-ADL) [5] are good candidates for such approaches. In EAST-ADL, an automotive system's structure and function are modeled at multiple levels of abstraction, that is, vehicle, analysis, design, implementation levels, and each abstraction level employs distinct concepts worth considering during requirements specification. For instance, the vehicle level of EAST-ADL abstraction describes the high level function of the system. Therefore, it would be inappropriate to use concepts from the design level, such as ports, signals, hardware elements, to describe requirements at the vehicle level, since such details usually hinder communication with non-technical stakeholders. Consequently, the requirements specifications need to be adapted to the appropriate levels of abstraction.

In this paper, we propose an Eclipse-based tool chain for structured requirements specification in ReSA [6], which scales to multiple EAST-ADL levels of abstraction. ReSA is an ontology-based requirements specification language tailored to automotive embedded systems development, which uses requirements boilerplates to structure the specification in natural language. Furthermore, we propose a consistency-check function that seamlessly integrates into the tool chain, for the automated consistency check of requirements using Z3 SMT solver [7]. The consistency checking is a preliminary task during elicitation and specification of requirements that paves the way for formal verification at later stages of software development. Our approach for consistency checking does not require a behavioral, or architectural model of the system, which might increase its attractiveness to industry as there is often the case that no system models exist for industrial systems. Checking for requirements consistency has been widely used in the field of requirements engineering, e.g., to describe consistent use of terms (words, phrases), logical consistency of requirements statements, or consistency between requirements and subsequent refinements [8][9]. The term can also refer to checking against type errors, or circular definitions [10]. In this paper, the consistency checking refers to checking the logical consistency of ReSA requirements specifications, in Z3.

Consistency checking of requirements specification helps detect possible logical errors at early stages of software development, and reduce the communication cost between manufacturers and suppliers [11]. However, checking for logical consistency of requirements expressed in natural language is not an easy task, mainly because: (i) unconstrained natural language is inherently ambiguous when it comes to reasoning, (ii) substantial assumptions used during requirements specification are hidden, and (iii) the size and complexity of requirements specifications are considerable.

In this work, we reduce the problem of checking the logical consistency of ReSA requirements to a boolean satisfiability problem, hence we propose algorithms for transforming the ReSA specification into boolean expressions, encode the latter into Z3 assertions, and perform consistency check using the Z3 SMT solver. The remainder of the paper is organized as follows. In section II, we recall the main features of ReSA, the EAST-ADL levels of abstraction, xText grammar, and the boolean satisfiability problem. We introduce the ReSA toolchain in section III, after which we describe our consistency checking steps in section IV. The applicability of the tool is shown in section V, where we specify and check the consistency of sample requirements from an industrial use case, called the Adjustable Speed Limiter (ASL). We compare to related work in section VI, before concluding the paper in section VII.

II. Preliminaries

In this section, we overview the ReSA language, and its adaptation to EAST-ADL levels of abstraction, as well as the xText grammar, and the basic boolean satisfiability problem.

A. Overview of ReSA

ReSA [6] is an ontology-based requirements specification language tailored to automotive embedded systems development. The language (i) renders natural language terms (words, phrases), and syntax, (ii) uses an ontology that defines concepts and syntactic rules of the specification, and (iii) uses requirements boilerplates to structure specification.

1) Requirements Specification Ontology: A snippet of the ontology specification is shown below.

$$[System \ast x1][ActOnPara \ast x2][Para \ast x3] \quad (1)$$

$$(Is\text{-}fb\ ?x1\ ?x2)(Is\text{-}fb\ ?x2\ ?x3) \quad (2)$$

This ontology snippet defines requirements specification concepts (1), and syntactic rules between instances of concepts (2). The specification states that an instance of *System* precedes both an instance of *ActOnPara*, and an instance of *Para* in a requirement specification, e.g., `ASL:system shall control:ActOnPara vehicle speed:para`, is a valid example that conforms to the ontology specification.

2) Requirements Boilerplate: The language uses *requirements boilerplates (or boilerplates)* [12] in order to structure a requirement. A boilerplate is a reusable specification template, which is constructed from variable, and fixed syntactic elements, e.g., `if <button> is <pressed> then <system> shall be <state> within <10><ms>`, where syntactic elements within pairs of angle brackets are variable syntactic elements, and the rest are fixed syntactic element. Table I displays the boilerplate elements of the language.

B. EAST-ADL Levels of Abstraction

The ReSA language can be tailored to express requirements at multiple levels of abstraction in the development of automotive systems. This helps achieving a consistent specification style across several abstraction levels. We show this for automotive embedded systems development based on EAST-ADL. EAST-ADL [14] is a model-driven approach to the development of complex automotive embedded systems. It covers a wide range of development aspects, such as analysis, design, implementation, verification&validation. The language uses various levels of abstraction to conceptualize a system with different degrees of detail, that is, vehicle, analysis, design, and implementation levels. We briefly describe the levels of abstraction in light of requirements modeling.

- Vehicle level: a vehicle is modeled using interconnected vehicle features, that satisfy high level requirements.
- Analysis level: the vehicle feature is refined using analysis level functions, that are design, and hardware independent. These functions satisfy the refined version the high level requirements specified at the vehicle level.
- Design level: the analysis level functions are refined using design level functions, that are enriched with periodic triggering, and execution time constraints. These functions satisfy the refined version of requirements specified at the analysis level.
- Implementation: the design level requirements are refined, and are satisfied by AUTOSAR [15] implementation, which we don't discuss it in this paper.

The specialization of the ReSA language to express requirements for EAST-ADL's levels of abstraction is done by specializing the ReSA concepts to appropriate concepts found in EAST-ADL. Table II shows an example of the specialization of the *System* concept at vehicle, analysis, and design levels of EAST-ADL levels of abstraction.

TABLE I: The ReSA Language Boilerplates

Boilerplate	Description
Simple	Instantiates a simple statement, and contains a modal verb, such as, *shall*, e,g., `system shall be activated.`
Proposition	Similar to *Simple*, except it is a proposition (or an assertive statement) [13, p.435], e.g., `button is pressed.`
Complex	Instantiates a complex statement, and is constructed from a Simple, a Proposition boilerplate, and an adverbial conjunctive (such as while, when, until). For example, `the error shall be reported while the fault is present`
Compound	Instantiates a compound statement, and is composed of two or more *Simple* or *Proposition* boilerplates and the logical operators, AND/OR, e.g., `system shall be activated` *and* `driver shall be notified.`
Conditional	Instantiates a conditional statement. The boilerplate can be instantiated to a different variant of conditional statements, i.e., *if*, *if-else*, *if-elseif*, or *if-elseif-else*, and conditional nesting.
Prepositional Phrase	Instantiates a prepositional phrase, and can be used to describe timing properties, occurrence of events, other complements to the subject of a main phrase. e.g., `within 5ms, by the driver`

C. XText Grammar

ReSA is implemented in xText Eclipse framework, a powerful, and popular Integrated Development Environment (IDE) for the development of Domain Specific Languages (DSL), and programming languages. The main component of the framework is the xText grammar language [16]. Among other constructs, the xText grammar contains the declaration of an xText file header (1-3), and parser rules (4-7). Line (1) states

TABLE II: Concept specialization for *System* concept

Vehicle-level	Analysis-level	Design-level
VehicleFeature (VF)	AnalysisFunctionType (AFT) FunctionalDevice (FD)	DesignFunctionType (DFT) BasicSoftwareFunction (BSF) LocalDeviceManager (LDM) HardwareFunctionType (HFT)

the grammar's name to be a valid java extension; (2) states the reuse of common terminal rules, e.g. rules for string, whitespace; (3) generates EPackage for the implementation of the grammar with the name *resaDSL* located at the stated Uniform Resource Identifier (URI). Rules (4-7) state the different parser rules in Extended Backus-Naur Form (EBNF) notation [17][18].

```
(1) grammar org.volvo.resadsl.ResaDsl
(2) with org.eclipse.xtext.common.Terminals
(3) generate resaDsl
    "http://www.volvo.org/resadsl/ResaDsl"
...
(4) UnAssignedPRule: AssignedRule;
(5) AssignedPRule: feature = STRING;
(6) DataTypePRule: 'dType ' name = ID;
(7) CrossRefPRule: feature = [DataTypePRule];
```

D. The Boolean Satisfiability Problem (SAT)

The consistency of ReSA specifications can be reduced to a satisfiability problem of boolean expressions (propositional formulas) [6]. A requirement specification in ReSA is constructed from one or more propositions connected by logical operators (*and*, *or*, *implies*, *not*), and parentheses. SAT techniques can be used to determine if the conjunction of ReSA requirements are satisfiable.

The *satisfiability problem* (SAT) [19] is defined as follows: given a propositional formula ϕ = f($x_1, \ldots, x_n$), over a set of boolean variables $x_1, \ldots, x_n$, decide whether or not there exists a truth assignment to the variables such that ϕ evaluates to true. SAT problem instances are usually expressed in a standard form called *conjunctive normal form* (CNF). A propositional logic formula is said to be in CNF if it is a conjunction (*and*) of disjunctions (or_s) of literals. A literal is either x, or its negation $\neg x$, for a boolean variable x. The disjunctions are called clauses.

Theorem 1 (Inconsistency of requirements specifications): Let $\Psi = \psi_1, ..., \psi_n$ denote the system requirements specification, where each of the formulas $(\psi_1, \ \ldots, \ \psi_n)$ encodes requirements. We say that the set is inconsistent if the following implication is satisfied: $\psi_1 \wedge \psi_2 \wedge \ldots \psi_n \Rightarrow False$.

In order to check the consistency of requirements specification, one has to disprove Theorem 1 by showing its negation *true*, that is, find a counterexample that satisfies the CNF of the requirements specification, Ψ. In this paper, we check the consistency of ReSA requirements via the Z3 tool [7]. Z3 is an efficient Satisfiability Modulo Theories (SMT) solver developed at Microsoft Research, which integrates several decision procedures for verification.

In the following consecutive sections, we describe the main contribution of the paper regarding the tool implementation, including its architecture, and consistency checking.

III. The ReSA Toolchain

The ReSA toolchain is an Eclipse-based implementation of our requirements specification language [6]. The toolchain supports contextual content completion, and text validation features. Furthermore, it seamlessly integrates a function for checking the logical consistency of requirements using the Z3 SMT solver [7]. The toolchain also supports specifying requirements at different levels of software development, using appropriate concepts valid at a specific level of abstraction. We specialize this approach for EAST-ADL, with respect to the vehicle, analysis, and design level of abstraction. The Graphical User Interface of the toolchain is shown in Figure 1, displaying demo projects for ASL, both EAST-ADL generic specification, as well as EAST-ADL abstraction level aware specification. The toolchain is available for download from the web link: `https://github.com/nasmdh/ReSA-Tool-0.0.git`

A. The Toolchain Architecture

Figure 2 shows the architecture of the ReSA toolchain. It consists of requirements specification and consistency checking of requirements. The specification part is basically the ReSA specification editor (a.k.a. *Resa App*), and a domain model. During writing requirements specifications, domain elements can be accessed from the domain model, but also model elements can be populated during specification. Such approach allows the consistent use of terms among different requirements engineers, reduces typographic errors, and maintains a knowledge base for later system refinements. The consistency checking part consists of a consistency checking plugin that calls the Z3 SMT solver. The result of the consistency checking is returned to the editor perspective.

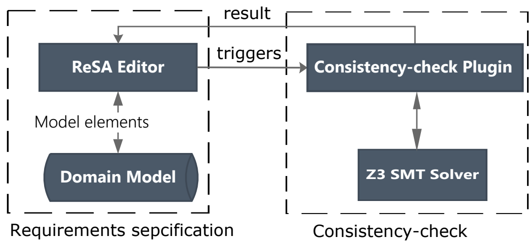

Fig. 2: The ReSA toolchain architecture

1) ReSA Specification Framework: Figure 3 shows the framework for specifying requirements with our ReSA tool. The framework consists of the Hierarchical Grammar, the ReSA Application, and the System Model. The Hierarchical Grammar is composed of a generic grammar, Gs, and grammar definitions for each EAST-ADL abstraction level, indicated by Gv, Ga, Gd, for vehicle, analysis, and design

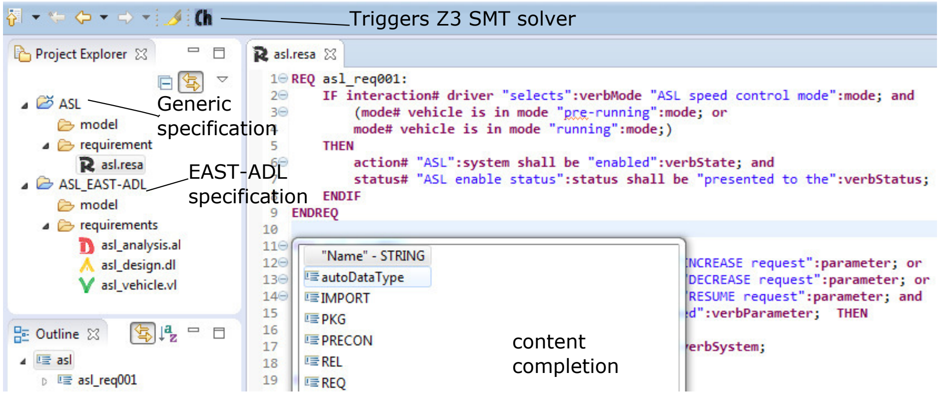

Fig. 1: The ReSA toolchain user interface

levels of abstraction, respectively. The grammar definitions for the EAST-ADL levels of abstraction are specializations of the generic grammar (indicated by the relation `<specialize>`), that is, concepts and syntactic rules are adapted to suit the specification at each levels. Through the `<import>` relation, concepts, and rules from the top level grammar are imported to the low level grammar, which enables referring to higher level concepts from lower level abstractions.

The ReSA Application is an implementation of the Hierarchical Grammar, and an editor for ReSA, indicated by the relation `<implements>`. The file extension **.resa* implements the application for the generic grammar, whereas file extensions **.vl*, **.al*, **.dl* represent the applications for vehicle, analysis, and design levels, respectively, and implement their corresponding grammar definition. The System Model provides access to the model elements of the application, during the specification at the respective abstraction level.

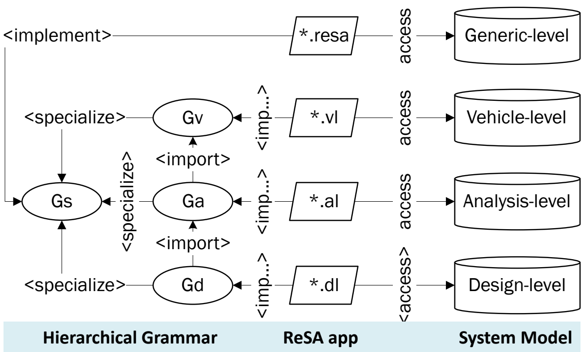

Fig. 3: Grammar architecture, and support for EAST-ADL

B. Implementation

We have implemented the toolchain in the xText Eclipse Framework. The framework provides an xText editor for grammar specification using the xText grammar language, and generates a start-up IDE based on Eclipse, which includes Parser, Compiler, Linker, and textual editor [16]. In this subsection, we go through the implementation of the ReSA grammar, and its adaptation to EAST-ADL.

1) Generic Grammar: This grammar defines the generic rules of constructing requirements specification in automotive systems. It uses automotive concepts to typeset domain elements, and action verbs associated to instances of concepts. The grammar defines the syntax of the boilerplates, and the requirements specification that is built from the boilerplates.

a) Boilerplate Rules: The following grammar rules define how a requirement specification is structured using boilerplates. Line (1) defines an unassigned rule that delegates rules to the compound boilerplate (2), and the conditional boilerplate (4). Lines (2) and (4) define a left-refracturing to handle the left-recursive nature of compound, and conditional boilerplates. Line (4) defines a rule for the different cases of conditional boilerplates, i.e., if, if-else, if-elseif, if-elseif-else, and nested-if.

```
(1) Boilerplate : Compound | Conditional;
(2) Compound:
    cx=Simple ({cmOp.left=current} biOp=LgOp
                 rt=Compound)?;
(3) Condition:
    pr=Proposition ({cnOp.left=current}
                 biOp=LgOp rt=Condition)?;
(4) Conditional:
    'IF'cnl=Condition 'THEN'({cnlOp.left =
                 current} rt=Conditional)?
    then=Compound?
    ('ELSE' else=Compound | elseif=Elseif)?
    'ENDIF';
```

b) Syntactic Element Rules: The following grammar snippet states rules for constructing boilerplates elements. Rule (1) creates *datatypes*, that is, *system* and *state*; rule (2), (3) create syntactic elements. The syntactic elements can be typed inline, e.g., `"ASL":system`, or referred from a model; rule (4) creates the fixed syntax element *shall be*, and finally rules (5) and (6) create Simple, and Proposition boilerplates using the above parser rules, respectively. For example, Simple boilerplate, such as, <term:system> shall be <term:state>, and Proposition boilerplate, such as, <term:system> is <term:state>.

```
(1) System: name=STRING; State: name=STRING;
(2) System_Rule: System | system=[System];
(3) State_Rule: State | state=[State];
(4) PModal: "shall be" | "shall be able to";
(5) Simple:
    (not)? (sub=System_Rule modal=PModal
            obj=State_Rule";" |...);
(6) Proposition:
    (not)? (sub=System_Rule "is"
            obj=State_Rule";" |...);
```

2) Specialization of Generic Grammar to EAST-ADL: The requirements specification is adapted to EAST-ADL using specialization of types, and syntactic rules defined in the generic grammar. In the xText framework, such functionality can be supported using *grammar mixin*, a feature that allows the reuse of previously defined grammars. Rules (1), (2), (3) show how the datatype *System* is specialized at vehicle, analysis, and design levels, respectively. Furthermore, a rule in the generic grammar can be extended to cover more requirements specification scenarios, using the keyword *super*, e.g., (4) shows the extension of the *Main* rule with *MainDesign*, which includes a signal related specification, e.g., `<term:signal> shall be "received on":verb <term:port>`.

```
(1) System: VF;
(2) System: AF | FD | FP
(3) System: DF | BSFT | HFT
(4) Main returns reSADSL::Main:
    super | MainDesign
    MainDesign:
        sub=Signal_Rule modal=PModal
        verb=Verb_Rule  obj=InPort_Rule?';'
```

If we consider the previous example, that is, "ASL":system shall be "activated" :state, the counterpart specification of this requirement at Vehicle-level is, "ASL":VF shall be "activated":state.

IV. Automated Consistency checking

In this paper, consistency checking refers to checking the logical consistency of ReSA requirements specifications without the use of a formal architectural model. This is in contrast to the use of the term to describe the consistent use of terms (i.e., words and phrases) in a specification [8][9], or type checking, or identification of circular definitions [10]. Since manual inspection of requirements specification is cumbersome, and sometimes impractical for finding inconsistencies, computer-assisted (automated) methods and tools, such as model checking, and theorem proving, have gained popularity in embedded systems. However, most of these methods, and tools require a formal specification language, such as LTL, CTL, which is expensive to use in industry due to the associated cost employing formal methods. We address this challenge by using the ReSA language, a relatively readable, and close to natural language, and a seamless integration of the consistency checking (with z3) in the requirements specification toolchain.

A. Consistency Checking Approach

Our consistency checking approach does not require a behavioral,or architectural model, instead the input is simply requirements specification document written in ReSA. Since, such models are not readily available in practice, our approach is appealing and useful to industry. The problem of consistency checking is reduced to a satisfiability problem as follows. The requirements, Req_i, are expressed using propositional formulas, and the conjunction of these requirements, $\bigwedge_i Req_i$, is checked for satisfiability, $M \models \bigwedge_i Req_i$, where M is an interpretation (assignment of the proportional variables that satisfies) $\bigwedge_i Req_i$. The propositional formulas are mostly expressed by conditional statements, $(P \Rightarrow Q)$ that hold globally in the system, where P, Q are propositional formula, which contains $\wedge$, $\vee$, $\rightarrow$, and $\neg$ logical operators [6].

The consistency-check is briefly described as follows:

- input: ReSA requirement specification.
- step1: ReSA requirement specification is transformed into a boolean expression of propositions (or propositional formula); check Section IV-B.
- step2: Boolean expressions are encoded into the SMT-LIB2 format [20], with each of the expressions as an assertion.
- step3: Z3 SMT solver is triggered to check the satisfiability of the expressions; check Section IV-C.
- output:The user is notified of the consistency check result.

B. ReSA-to-boolean Transformation

Algorithm 1 shows a function that transforms a ReSA requirement specification into a propositional formula. A ReSA specification can be treated as a composition of propositions, logical operators, *(and, or)*, and fixed syntactic elements, like *if...else*. The propositions are instantiations of the `Simple`, or `Proposition` boilerplates. In the ReSA requirement of Example 1, `<Btn1: inDevice> is <pressed: actOnInDev>` is an instantiation of the *Proposition* boilerplate, and `<ASL: system> shall be <activated: state>` is an instantiation of the *Simple* boilerplate:

Example 1:

```
if <Btn1: inDevice> is <pressed: actOnInDev>;
then
   <ASL: system> shall be <activated: state>;
endif
```

Line 1 of Algorithm 1 reads the requirements specification (*.resa) file, and buffers the content into *reqsBuffer*. For each requirement specification, the *Simple* and *Proposition* boilerplates are respectively replaced with temporary variables for later use (3). Next, propositions *props* are extracted from the requirement specification $reqSpec$ (4), after which, for each proposition, propositional variables pvs are generated (5). Finally, a boolean expression is generated by substituting the temporary variables with pvs in the preserved requirement structure (6). Applying this algorithm to Example 1, we get $(p1 \Rightarrow p2)$, where $p1$ represents `<Btn1:inDevice> is <pressed:actOnInDev>;`, and $p2$ represents `<ASL:system> shall be <activated:state>;`.

Definition 2: A proposition p_2 is the negation of proposition p_1 ($p_2 = not\ p_1$), if there exists a word at position i of p_2

Algorithm 1: ReSA to boolean transformation

```
reqsBuffer[]← ReadReqs(*.resa)
Function ReSAToBoolean(reqsBuffer)
    foreach req in reqsBuffer do
        (id, reqSpec...) ← ParseReq(req)
        reqSpecStruct ←
          PreserveReqSpecStruct(reqSpec)
        props[]← ParseProps(reqSpec)
        foreach p in props do
            pvs[]←GetPropVars(p)
        end
        booleanExps[]←
          GenerateBooleanExp(reqSpecStruct,pvs)
    end
    return : booleanExps
end
```

(word$_i^{p_2}$) that is the antonym (opposite) of a word at position i of p_1 (word$_i^{p_1}$), while the rest of p_2 syntactic structure matches p_1 (valid also for the reverse case, that is, $p_1 = not\ p_2$).■

The antonyms dictionary is a two dimensional list of antonyms (or words with opposites). The first word in the list represents a root word, and the rest represent opposite words to the root word. An opposite word is replaced with its root word. For example, the antonyms dictionary contains the word *activated* as a root word, and its opposite word *deactivated*. For the example below, we say that p_2 is a negation of p_1:

$$p_1 = ASL : system\ \ shall\ \ be\ \ \textbf{activated} : state$$

$$p_2 = ASL : system\ \ shall\ \ be\ \ \textbf{deactivated} : state$$

Algorithm 2: Generates a proposition variable

```
1 antonymsBuffer[]← ReadAntonyms(antonyms.txt)
2 propositionsBuffer[]← Null
  Function GetPropVars(proposition)
3     pn ←NormalizeProp(proposition, antonymsBuffer)
4     foreach p in propBuffer do
          if pn = p then
              return : p.pv
          end
      end
5     newPv ←GeneratePropVars()
      AddProp(pn, newPv)
      return : newPv
  end
```

Algorithm 2 implements Definition 2, hence, replaces an antonym with its root word (3). Further, Line (4-5) checks the *propositionsBuffer* for match of the proposition pn, and returns its propositional variable if found. Otherwise, Line 5 generates a new propositional variable for the proposition pn, and Line (5) adds the new proposition, and its propositional variable to the *propositionsBuffer*.

C. Consistency Checking

In this section, we introduce Algorithm 3 that illustrates the function for invoking the Z3 SMT Solver. Line (1) transforms boolean expressions into an SMT-LIB2 format, which is the input format of Z3; line (2) creates a logical context that enables interaction with the solver; line (3) parses SMT-LIB2 into the context, and finally, lines (4) and (5) create an instance of the solver, and invoke the solver, respectively.

Algorithm 3: consistency-check using Z3 SMT Solver

```
  Function CheckConsistency(booleanExp)
1     smtLibStr ← GenerateSMTLIBStr(booleanExp)
2     ctx ← new Context()
3     ctx.parseSMTLIBString(smtLibStr, null, null, null,
        null)
4     z3Solver ← ctx.mkSolver()
5     return : z3Solver.check(ctx)
  end
```

V. Industrial Use Case: Adjustable Speed Limiter

We have conducted an initial validation of our approach on requirements from the Adjustable Speed Limiter (ASL) [6]. ASL is an automotive safety-critical function, which is found along other vehicle limitation and control functions, such as Cruise Control (CC), in modern Volvo trucks. It limits the truck speed not to exceed a predefined and configurable vehicle speed. ASL provides an HMI interface for interaction with the driver, and has access to the powertrain engine in order to limit the engine positive torque. Therefore, it is a complex and safety-critical function.

ASL realizes 304 functional and extra-functional requirements, such as timing, safety, vehicle configurability, and variability. The requirements of ASL are found at multiple levels of abstraction according to EAST-ADL requirements modeling approach, that is, requirements defined at the lower level of abstraction are refinements of the upper level abstraction. In our validation process, we rewrite the requirements of ASL, which have been previously written in natural language (English), in ReSA. Furthermore, we evaluate the language and the tool with practitioners at Volvo Group Trucks Technology (VGTT). In this section, we show the validation result, and explain the consistency check function of the ReSA toolchain.

A. ASL Requirements Expressed in ReSA

Requirements of ASL describe a wide range of ASL functional and extra-functional properties, including:

- Interaction of the function with the driver (Human Machine Interface, HMI Requirements)
- High level ASL functions, which are less technical, and independent of implementation (High level FR).
- Functional-block Responsibility Requirement (Functional-block RR) briefly describe the responsibility of a functional block in precise and short statement.
- Low level functional requirements are more technical and implementation dependent (Low level FR).
- Performance Requirements express, such as timing, and concurrency, related requirements.
- Safety Requirements, such as response during faulty operation of ASL function.

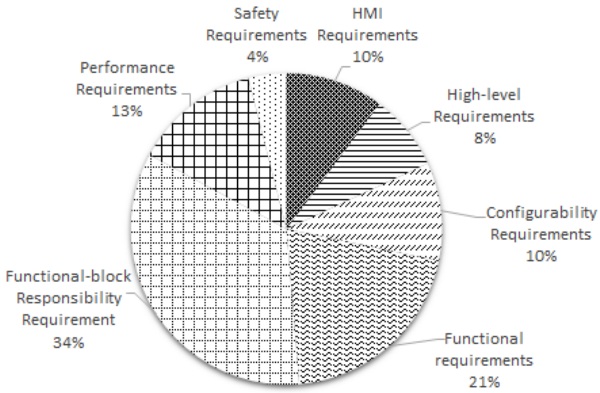

Fig. 4: The ASL Requirements Distribution

Req# 1 (ASL activation display): ...HMI Requirement
```
if <ASL:vf> is <selected:actOnSys> then
  <"the ASL indication light":hft> shall be
  <"lit":actOnDev>; on <the free wheel>;
endif
```
Req# 2 (ASL activation): ...High-level FR
```
if <increaseBtn:inDev> is <pressed:actOnDev>
then
  <ASL:vf> shall be <activated:state>;
  within <0.25><s>;
endif
```
Req# 3 (ASL activation): ...Configurability Requirement
```
<ASL_min:ffp> and <ASL_max:ffp> shall be
<configurable>;
```
Req# 4 (RSLM - ASL activation): ...Functional-block RR
```
<RSLM:fd> shall be <responsible>; for
<"activating ASL">;
```
Req# 5 (ASL activation request): ...Design-level FR
```
if <"ASL activation request":ffp> is
  <received:actOnPara>; while <ASL:vf> is
  <overriden:state>;
then
  <"ASL target speed":ffp> shall be set to
  <"ASL set speed":ffp>; and
  <"ASL":state> shall be <activated:state>;
endif
```
Req# 6 (ASL activation request): ...Performance Requirement
```
<The engine torque":ffp> shall <"release
control on":actOnPara> <"engine":hct>;
within <0.25><s>;
```
Req# 7 (ASL activation request): ...Safety Requirement
```
if <"fault affecting ASL function":event>
  occurs; while <ASL:vf> is <active:state>;
then
  <ASL:vf> shall be <deactivated:state>;
  in <a safeway>;
endif
```
In order to observe how much of information is encoded in the different requirements categories mentioned above, we analyze the boilerplates that are used to express requirements of ASL. Figure 5 shows that *Simple* and *Proposition* boilerplates are the most widely used boilerplates, followed by *Compound* boilerplates. Even though the number of Functional-block Requirements are more than the Low-level Functional

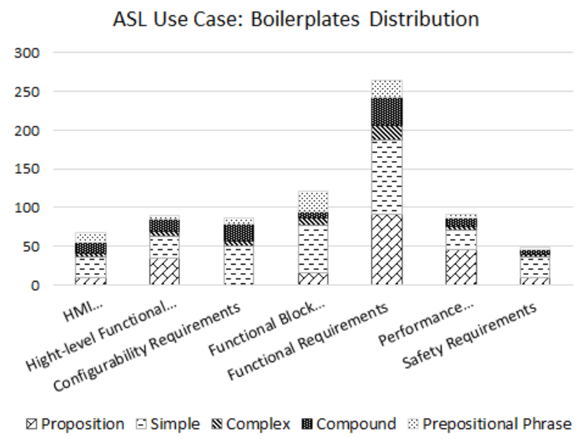

Fig. 5: The ASL Boilerplates Distribution

Requirements, as shown in Figure 4, the amount of information encoded in the requirements is higher in the Low-level Functional Requirements, as indicated in Figure 5. This is witnessed by the fact that far more boilerplates are used to express the Low-level Functional Requirements than to express Functional-block Requirements.

B. Evaluation of the ReSA Toolchain with Practitioners

We have carried out an initial evaluation of the ReSA tool with 8 practitioners from VGTT. The practitioners include requirements engineers, software engineers and architects, test engineers, and researchers. The main goal of the evaluation is to get an initial result of using the tool. The evaluation criteria can be accessed using the web link, https://goo.gl/HwQ1vO. The response from the practitioners is Table III.

TABLE III: The ReSA toolchain evaluation

Role	Summary Feedback
Software engineers	found structuring of requirements appealing; they suggested more expressiveness in the language.
Verification engineers	found the tool usable, especially in test case development.
Requirement engineers	found that reusability and extensibility of the specification method was appealing, and suggested adding alternative graphical specification.
Researchers	found the specification method, and specialization to EAST-ADL abstractions useful.

In the following subsection, we show a sample of the ASL specification in ReSA. Further, we apply our consistency checking approach on ASL requirements.

C. Consistency Checking on the Use Case

Using the ReSA toolchain, we express 37 functional requirements of the ASL system, that are related to the activation and deactivation of the system. Next, we check consistency of the requirements specifications using the consistency-checking feature of the toolchain. In this subsection, we show the point of inconsistency reported by toolchain, and also demonstrate how the consistency-checking function works.

The following requirement describes enabling ASL.

```
req req001_ENABLING_ASL:
(p1) if interaction#driver "selects":verb
              "ASL speed control":mode; and
(p2) (mode# vehicle is in mode
              "pre-running"; or
(p3) mode# vehicle is in mode "running";)
     then
(p4) action# "ASL":system shall be
              "enabled":verbState; and
(p5) status# "ASL enabled":status shall be
              "presented to" driver
     endif
endreq
```

The boolean expression for the above requirement becomes `(p1 ∧ (p2 ∨ p3)) ⇒ (p4 ∧ p5)`.

To show how the consistency function catches inconsistencies in requirements specifications, we now introduce a bogus requirement for disabling ASL, as follows:

```
req req002_bogus_DISABLING_ASL:
(p6) if interaction#driver "selects":verb
              "ASL speed control":mode
     then
(p7)   action# "ASL":system shall be
                "disabled":verbState;
    endif
endreq
/* "disabled" is antonymic to "enabled" */
```

Since the word *disabled* is antonimic to the word *enabled*, `p7` becomes the negation of `p4` according Definition 6; and `p6` is equivalent to `p1`. Therefore, the boolean expression for the above requirement becomes `p1 ⇒ ¬p4`. To demonstrate how the consistency check function works, let us assume, and assert in the specification that ASL is enabled, and the vehicle is in pre-running mode. The SMT-LIB2 equivalent format of the above two requirements including the assertions, as obtained from our transformation, appears as follows:

```
(set-option :produce-unsat-cores true)
; declare boolean constant
(declare-const p1 Bool)
...
;req001_ENABLING_ASL
(assert (! (=>(and p1 (or p2 p3))
(and p4 p5)) :named req001))
;req002_bogus_DISABLING_ASL
(assert (!(=> p1 (not p4)) :named req002))

;Assert that driver selects ASL control
(assert (! (= p1 true) :named assumption1))
;Assert vehicle is in pre-running mode
(assert (! (= p2 true) :named assumption2))
```

If the Z3 solver is triggered to check the satisfiability of the 37 requirements specifications, it returns *unsat*, as there exists an inconsistency within the requirements specification. Obviously, the ASL cannot be activated and deactivated at the same time, given the assumptions, and this inconsistency is identified using the toolchain. A feature of Z3's unsat-core tries to localize the region of inconsistency by listing the requirements associated with the inconsistency problem using the labels of requirements defined during the requirements specification. For example, the following result from the solver indicates that the region of inconsistency is related to the two requirements, and the assertions we made.

```
unsat
(req001 req002 assumption1 assumption2)
```

The engineer is supposed to use this feedback from the solver, and make necessary changes to the specification, and repeat the consistency checking process until no more inconsistency is found.

VI. Related work

The related work focuses on toolchains that use template-based specification methods, computer-processable Controlled Natural Languages (CNLs), and perform automated consistency checking without the need for system models. These are in contrast to tools that use tabular specification techniques [21], graphical specification methods, or formal specification methods, e.g., Z notation, LARCH, Linear Temporal Logic (LTL) [22].

A. Template-based Specification Tools

In this paper, we define a template-based specification method that uses predefined specification templates extracted from experience in requirements engineering, in order to express requirements in a more structured way. The most popular methods of this type are requirements boilerplates [23][24][25], and Specification Pattern System (SPS) [26][27][28]. Specification templates are reusable artifacts, and consist of variable and fixed syntactic elements, where the variable part is filled by the engineer. The specification templates facilitate communication among engineers due to the fact that engineers use the same templates for similar requirements from a common repository of templates. The challenges of template-based approaches are: 1) the selection of an appropriate template out of seemingly similar templates; the Natural Language Processing (NLP) technique is found to ease this challenge in the case of DODT tool [2] while manual intervention is still necessary, and 2) the extension of the template repository with new templates for requirements that could not be expressed with the existing templates. The templates extension requires a careful approach, as templates could be ambiguous, or conflicting to each other. Therefore, such extension mechanism should subscribe to some syntactic, or semantic rules. By using the ReSA tool, the creation of new boilerplates is constrained by the syntactic and semantic rules of the ReSA language.

Boilerplate tools, such as DODT, and Requirements Authoring Tool (RAT), use requirement boilerplates to express requirements. Requirements boilerplate, e.g., the `if <button> is <pressed> then <system> shall be <activated>; within <0.25><sec>; endif,` is a typical boilerplate in ReSA language. The primary goal of using boilerplates is to provide structure to requirements, and make them readable, and more comprehensible than their temporal logic counterparts. However, some tools use knowledge management, e.g., an ontology, to analyze the quality of requirements (such as consistency, completeness, redundancy, vagueness), according to quality metrics defined in their knowledge-base.

DODT [2] is a research prototype tool, which is developed in the European CEASAR project. The tool supports boilerplates, and unconstrained natural language (English) to write requirements. The unconstrained natural specification is matched to existing boilerplates using Natural Language Processing (NLP) technique, and boilerplate mismatches are manually corrected. The tool can assess the quality of requirements specification based on the analysis of Ambiguity, Inconsistency, Completeness, Opacity, and Noise, by referring to the ontology that defines attributes, attribute relations, various axioms of the boilerplates, e.g., for contradiction, subclassing, equivalence [29]. RAT [30] is an industry level tool, which is being developed by the REUSE Company. It supports advanced features, such as guides writing of requirements using *IntelliSense* from Microsoft, quality analysis on-the-fly using metrics, such as Inconsistency, Ambiguity, Overlapped requirements, non-atomicity with the help of a separate knowledge-manager that stores vocabulary, patterns, syntax, and semantic representation. Due to its proprietary nature, it is not clear if the boilerplate extension mechanism relies on any syntactic or semantic rules like in the ReSA toolchain.

The Specification Pattern System (SPS) proposed by Dwyer et al. [27] is a set of property specification patterns, that can better be understood, and used by domain practitioners than, for instance, LTL specifications. Konard and Cheng extended the SPS with real-time support [26]. Using ReSA, temporal requirements can be expressed using the *Prepositional Phrase* boilerplate, e.g, `within <time>, after <time>...`; however, our transformation is limited to proportional formulas only. The toolchain by Post and Hoenicke [28] is an implementation of the real-time SPS grammar. The toolchain allows expression of requirements in restricted English grammar, e.g., `Globally, it is always the case that P holds after at most 10 seconds`, where `P` is a property to be checked, and the pattern translation into Duration Calculus [31]. Furthermore, their toolchain can check inconsistency, rt-inconsistency (checks timing boundaries), and vacuity (requirements that can never be enabled). However, we couldn't gain access to the toolchain to do hands-on experience. As compared to the boilerplate-based specification, the SPS mentioned above uses architectural elements in constructing the property, e.g., `vehicleSpeed > setSpeed`, where `vehicleSpeed`, and `setSpeed` are elements of our ASL architecture. Moreover, the SPS has representations in formal logic. Unlike boilerplate-based specification, the SPS targets behaviour description, therefore its coverage is limited, but more precise due to its formalized nature. ReSA, on the other hand can express a wide range of requirement types, including behavioural, and requirements that express performance, and safety. Further more, as compared to the SPS, ReSA is close to natural language. Elen et. al [32] propose an existential bounded consistency analysis using Bounded Model Checking (BMC), and implement their prototyping using iSAT model checker. The analysis does not require a system model, and checks if a run exists that satisfies the specification in BTC pattern [33].

B. *Computer-processable CNL Tools*

Computer-processable Constrained Natural Languages (CNLs), such as the Attempto Control English (ACL), and the Processable ENGlish (PENG), use limited words, phrases, syntax and semantics of natural language express texts in a simplified English language. Moreover, computer-processable CNLs have formal semantics, e.g., in first-order-logic (FOL), which makes them amenable to automated analysis, that is for checking logical consistency, redundancy, and ambiguity. The ReSA toolchain uses transformation of requirements to proportional formula to do the consistency checking, and supports features, such as specification guide, and provides tips for error correction during requirements specification.

ACE supports the construction of simple, and composite sentences (complex and compound), coordination of phrases using *and*, subordination, quantification, negation, and query-answer interfaces [34]. Texts in ACE can be translated into formal specifications, such as FOL [35]. The Attempto toolchain is a suite of tools. The tool has support for text completion, and inline checking for ambiguity, inconsistency via its predefined lexicon, and grammar rules. Attempto does not allow the use of passive sentences, verb phrases, modal verbs, which is natural to use in requirements specification, for example, `system shall be activated`. Inspired by ACE, PENG [36] is also a computer-processable language. The PENG system uses ECORE, which is a look-ahead editor, in order to predictively provide possible alternatives during writing. This feature lowers the burden of memorising the syntax rules of PENG. Yan et. al [37], in the tool SpecCC, transformed their own CNL into LTL, and synthesize the LTL specification using G4LTL in order to check for realizability.

VII. CONCLUSION

In the automotive industry there is a stringent need for semi-formal requirements specification methods and tools that integrate seamlessly into industrial practice. In this paper, we propose an implementation of the previously proposed ReSA requirements specification language, and provide algorithms for the logical consistency checking of requirements formulated in ReSA for a particular system. Our consistency checking approach first automatically transforms ReSA requirements specifications into expressions in propositional logic first, and then uses Z3 SMT solver to check the satisfiability of the boolean specifications.

In order to handle the complexity of automotive embedded systems development, the use of multiple levels of abstraction is a known, and usually common practice for designing a complex electrical/electronic function in architectural languages such as EAST-ADL. In this paper, we specialize the ReSA toolchain to support specifications tailored to EAST-ADL levels of abstraction. We have conducted a validation of the toolchain on the Adjustable Speed Limiter use case. The language is expressive enough to express the 304 use case requirements. Furthermore, the toolchain has also undergone an initial evaluation by VGTT engineers, who answered questionnaires and specified certain requirements with our tool. In our future work, we plan to scale the consistency checking to support requirements with temporal, and quantifiers properties. We also plan to extend the validation process to various automotive use cases, including from other companies besides VGTT, such as from Scania. In the near future, the toolchain will be integrated into Synligare Eclipse[1] for the EAST-ADL language.

REFERENCES

[1] M. Garg and R. Lai. Measuring the Constraint Complexity of Automotive Embedded Software Systems. In *Data and Software Engineering (ICODSE), 2014 International Conference on*, pages 1–6, Nov 2014.

[2] S. Farfeleder, T. Moser, A. Krall, T. Stlhane, H. Zojer, and C. Panis. DODT: Increasing Requirements Formalism using Domain ontologies for Improved Embedded Systems Development. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 271–274, April 2011.

[3] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, chapter RATSY – A New Requirements Analysis Tool with Synthesis, pages 425–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28 – July 2, 1998 Proceedings*, chapter SCR: A Toolset for Specifying and Analyzing Software Requirements, pages 526–531. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[5] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, et al. The EAST-ADL Architecture Description Language for Automotive Embedded Software. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 297–307. Springer, 2010.

[6] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA: An Ontology-based Requirement Specification Language Tailored to Automotive Systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, jun 2015.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] Didar Zowghi and Vincenzo Gervasi. The Three Cs of Requirements: Consistency, Completeness, and Correctness. In *International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage*, pages 155–164, 2002.

[9] M.P.E. Heimdahl and N.G. Leveson. Completeness and Consistency in Hierarchical State-based Requirements. *Software Engineering, IEEE Transactions on*, 22(6):363–377, Jun 1996.

[10] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, July 1996.

[11] No Silver Bullet. Essence and Accidents of Software Engineering, FP Brooks. *IEEE Computer*, 20(4):10–19, 1987.

[12] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer Science & Business Media, 2010.

[13] Andrew Radford. *Minimalist Syntax: Exploring the Structure of English*. Cambridge University Press, 2004.

[14] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL-An Architecture Description Language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

[15] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, 2009.

[16] TypeFox items. XText 2.5 Documentation, 2013.

[17] Lars Marius Garshol. BNF and EBNF: What are They and How Do They Work?, 2008.

[18] Jianan Yue. Transition from EBNF to Xtext. *Alternation*, 1:1, 2014.

[19] Devlin David and Barry OSullivan. Satisfiability as a Classification Problem. In *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*. 2008.

[20] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard Version 2.0. 2010.

[21] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR: A Toolset for Specifying and Analyzing Software Requirements. In *Computer Aided Verification*, pages 526–531. Springer, 1998.

[22] Axel van Lamsweerde. Formal Specification: a Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000.

[23] Vegard Johannessen. CESAR-text vs. Boilerplates: What is More Effcient-requirements? Written as Free Text or Using Boilerplates (templates)? 2012.

[24] Alistair Mavin and Philip Wilkinson. Big Ears (The Return of "Easy Approach to Requirements Engineering"). In *2010 18th IEEE International Requirements Engineering Conference*, pages 277–282. IEEE, sep 2010.

[25] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy Approach to Requirements Syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322. IEEE, aug 2009.

[26] Sascha Konrad and Betty HC Cheng. Real-time Specification Patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[27] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in Property Specifications for Finite-state Verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999.

[28] Amalinda Post, Igor Menzel, and Andreas Podelski. Applying Restricted English Grammar on Automotive RequirementsDoes It Work? A Case Study. In *Requirements Engineering: Foundation for Software Quality*, pages 166–180. Springer, 2011.

[29] Andreas Mitschke. *EAST-ADL Domain Model SpecificationVersion*. EAST-ADL Association, 02 2012. Version 2.0.

[30] The REUSE Company. Requirements Authoring Tool, 2016.

[31] Zhou Chaochen, Charles Anthony Richard Hoare, and Anders P Ravn. A Calculus of Durations. *Information processing letters*, 40(5):269–276, 1991.

[32] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting Consistencies and Inconsistencies of Pattern-based Functional Requirements. In *Formal Methods for Industrial Critical Systems*, pages 155–169. Springer, 2014.

[33] BTC Embedded Systems. Attempto tools, 2016.

[34] Norbert E. Fuchs and Rolf Schwitter. Attempto Controlled Natural Language for Requirements Specifications. In *Proc. Seventh Intl. Logic Programming Symp. Workshop Logic Programming Environments*, pages 25–32, 1995.

[35] Attempto Project. Attempto Tools, 2013.

[36] Rolf Schwitter. English as a Formal Specification Language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 228–232. IEEE, 2002.

[37] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal Consistency Checking over Specifications in Natural Languages. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 1677–1682. IEEE, 2015.

[1] https://github.com/Arccore/synligare