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

Nesredin Mahmud∗, Cristina Seceleanu∗, Oscar Ljungkrantz†

∗Malardalen 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.

\[
\begin{align*}
\text{System} \times x_1 & \times \text{ActOnPara} \times x_2 \times \text{Para} \times x_3 \\
\text{If}-\text{fb} \times x_1 \times x_2 & \times \text{If}-\text{fb} \times x_2 \times x_3 
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Boilerplate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Instantiates a simple statement, and contains a modal verb, such as, shall, e.g., system shall be activated.</td>
</tr>
<tr>
<td>Proposition</td>
<td>Similar to Simple, except it is a proposition (or an assertive statement) [13, p.435], e.g., button is pressed.</td>
</tr>
<tr>
<td>Complex</td>
<td>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.</td>
</tr>
<tr>
<td>Compound</td>
<td>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.</td>
</tr>
<tr>
<td>Conditional</td>
<td>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.</td>
</tr>
<tr>
<td>Prepositional Phrase</td>
<td>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 &lt;ms&gt;, by the driver.</td>
</tr>
</tbody>
</table>

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 specifying 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.

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

<table>
<thead>
<tr>
<th>Vehicle-level</th>
<th>Analysis-level</th>
<th>Design-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>VehicleFeature (VF)</td>
<td>AnalysisFunctionType (AFT)</td>
<td>DesignFunctionType (DFT)</td>
</tr>
<tr>
<td>FunctionalDevice (FD)</td>
<td>BasicSoftwareFunction (BSF)</td>
<td>SoftwareFunction (SFT)</td>
</tr>
<tr>
<td></td>
<td>LocalDeviceManager (LDM)</td>
<td>HardwareFunctionType (HFT)</td>
</tr>
</tbody>
</table>

devolved 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 RE SA 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](image)

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
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.

**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>.
Our consistency checking approach does not require a behavioral, or architectural model, instead the input is simply the 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, Reqᵢ, are expressed using propositional formulas, and the conjunction of these requirements, \( \bigwedge \text{Reqᵢ} \), is checked for satisfiability, \( M \models \bigwedge \text{Reqᵢ} \), where \( M \) is an interpretation (assignment of the propositional variables that satisfies) \( \bigwedge \text{Reqᵢ} \). 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 \( \land \), \( \lor \), \( \rightarrow \), and \( \neg \) logical operators [6].

The consistency-check is briefly described as follows:

**input:** ReSA requirement specification.

1. **step1:** ReSA requirement specification is transformed into a boolean expression of propositions (or propositional formula); check Section IV-B.
2. **step2:** Boolean expressions are encoded into the SMT-LIB2 format [20], with each of the expressions as an assertion.
3. **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, \( \text{(and, or)} \), and fixed syntactic elements, like \text{if...else}. The propositions are instantiations of the Simple, or Proposition boilerplates. In the ReSA requirement of Example 1, \(<\text{Btn1: inDevice}> \) is \!<\text{pressed: actOnInDev}>\) is an instantiation of the Proposition boilerplate, and \(<\text{ASL: system}> \) shall be \!<\text{activated: state}>\) is an instantiation of the Simple boilerplate:

**Example 1:**

\[
\begin{align*}
\text{if} \ <\text{Btn1: inDevice}> \ <\text{pressed: actOnInDev}>; \\
\text{then} \ <\text{ASL: system}> \ &\text{shall be} \ <\text{activated: state}>; \\
\text{endif}
\end{align*}
\]

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

**Definition 2:** A proposition \( p₂ \) is the negation of proposition \( p₁ \) (\( p₂ = \neg p₁ \)), if there exists a word at position \( i \) of \( p₂ \)
In this section, we introduce Algorithm 3 that illustrates the consistency check function of the ReSA toolchain. In the algorithm, we define a function `CheckConsistency` that takes a boolean expression as input and returns `true` if the expression is satisfiable, i.e., it is consistent. The function works by generating an SMT-LIB2 format string `smtLibStr` and creating a new context `ctx` with an instance of the solver. Finally, it invokes the solver to check the consistency of the expression.

Algorithm 3: consistency-check using Z3 SMT Solver

```plaintext
Function CheckConsistency(booleanExp)
1  smtLibStr ← GenerateSMTLIBStr(booleanExp)
2  ctx ← new Context()
3  ctx.parseSMTLIBString(smtLibStr, 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 (VGT). 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 <overridden: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 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>
<thead>
<tr>
<th>Role</th>
<th>Summary Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software engineers</td>
<td>found structuring of requirements appealing; they suggested more expressiveness in the language.</td>
</tr>
<tr>
<td>Verification engineers</td>
<td>found the tool usable, especially in test case development.</td>
</tr>
<tr>
<td>Requirement engineers</td>
<td>found that reusability and extensibility of the specification method was appealing, and suggested adding alternative graphical specification.</td>
</tr>
<tr>
<td>Researchers</td>
<td>found the specification method, and specialization to EAST-ADL abstractions useful.</td>
</tr>
</tbody>
</table>

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:

```lisp
(req req001_ENABLELING_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

\[(p1 \land (p2 \lor p3)) \Rightarrow (p4 \land p5).\]

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

```lisp
(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
```

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 \(p1 \Rightarrow \neg p4\). 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:

```lisp
(set-option :produce-unsat-cores true)
; declare boolean constant
(declare-const p1 Bool)
....
;req001_ENABLELING_ASL
(assert (! (= p1 true) :named assumption1))
;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.

```lisp
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 isSAT 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 synthesizes 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 VGGT 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 VGGT, such as from Scania. In the near future, the toolchain will be integrated into Synligare Eclipse\(^1\) for the EAST-ADL language.

REFERENCES


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


\(^1\) https://github.com/Arccore/synligare