8. ReSA: An Ontology-based Requirement Specification Language Tailored to Automotive Systems

Nesredin Mahmud, Cristina Seceleanu and Oscar Ljungkrantz.*In the 10th IEEE International Symposium on Industrial Embedded Systems (SIES)(pp. 1-10). IEEE, 2015.*

Abstract: Automotive systems are developed using multi-leveled architectural abstractions in an attempt to manage the increasing complexity and criticality of automotive functions. Consequently, well-structured and unambiguously specified requirements are needed on all levels of abstraction, in order to enable early detection of possible design errors. However, automotive industry often relies on requirements specified in ambiguous natural language, sometimes in large and incomprehensible documents. Semi-formal requirements specification approaches (e.g., requirement boilerplates, pattern-based specifications, etc.) aim to reduce requirements ambiguity, without altering their readability and expressiveness. Nevertheless, such approaches do not offer support for specifying requirements in terms of multi-leveled architectural concepts, nor do they provide means for early-stage rigorous analysis of the specified requirements.

In this paper, we propose a language, called ReSA, which allows requirements specification at various levels of abstraction, modeled in the architectural language of EAST-ADL. ReSA uses an automotive systems' ontology that offers typing and syntactic axioms for the specification. Besides enforcing structure and more rigor in specifying requirements, our approach enables checking refinement as well as consistency of requirements, by proving ordinary boolean implications. To illustrate ReSA's applicability, we show how to specify some requirements of the Adjustable Speed Limiter, which is a complex, safety-critical Volvo Trucks user function.

8.1 Introduction

Modern automotive systems rely on software that is rapidly growing in complexity and criticality. The requirements specification phase of developing large automotive systems represents a decisive factor in achieving a dependable system within demanding time-to-market constraints. Automotive systems development typically uses natural language documents as requirements artifacts that are exchanged between various stakeholders [10]. The inherent ambiguity of natural language, as well as the lack of rigor in using logical connectives between textual terms, can give rise to imprecise requirements. Moreover, complex requirements might suffer from lack of structure, rendering them incomprehensible. These symptoms imply poor requirements quality, which might negatively impact the system's verification and validation process.

Various researchers have proposed ways of structuring requirements expressed textually, via software requirement patterns [20][32], or the so-called *requirements boilerplates* [8][27]. Requirements boilerplates offer a more restricted grammar to write more precise textual requirements, and yet should be flexible enough to adapt new requirements structures. According to Hull et al. [18] a requirement boilerplate is a sequence of attributes connected by fixed syntax elements. For instance, `If <event> then <system> shall be <act>` is a boilerplate in which `event`, `system` and `act` are attributes, whereas `If`, `then`, `shall be` are fixed syntactic elements. By using such templates, the generated requirements comply to a predefined set of syntactic rules that guarantee that the requirements are well-formed.

Software architectures (SAs) show how the system is structured and how its components work together, representing the earliest stage at which design decisions can be analyzed. SAs are described in dedicated languages called *architecture description* languages, out of which EAST-ADL, with its four levels of abstraction, is one example already used in the automotive industry [26]. In order for the analysis of architectural models to be comprehensible, it is crucial that the system and subsystem requirements support architectural elements, such that the requirements are traceable across different layers of abstraction. Specifying particular requirements in terms of specific concepts valid for certain abstraction levels of the architectural model enables proving facts like refinement and consistency of requirements intra- as well as inter-abstraction levels. Although much research has been devoted to formal and semi-formal requirements specification (see section 8.2), none of the works addresses this issue.

Motivated by the above, in this paper we propose a semi-formal requirements specification language, called ReSA (**Re**quirement **S**pecification language for **A**utomotive systems), which provides support for defining requirements boilerplates tailored to automotive systems. The language is based on automotive systems ontology (introduced in section

8.4) that specifies the needed concepts at the system level, as well as the abstracted levels of the system's software architecture, thus providing a way of describing requirements in terms of syntactic elements specific to each of these levels. In section 8.5, we apply ReSA on a subset of three representative textual requirements of an automotive system called *Adjustable Speed Limiter*, which we describe in section 8.3. We show that refinement and consistency analysis of the set of ReSA requirements reduces to proving simple theorems over boolean formulas, as presented in section 8.6. The result of our work is a framework for specifying non-ambiguous, structured, typed and architecture-aware requirements, via boilerplates. Our ontology-based requirements specification language paves the way towards analyzable and traceable sets of requirements, across various levels of architectural descriptions of automotive systems. Finally, we present related work in section 8.7, before concluding the paper in section 8.8.

8.2 Requirements Specification Methods

A requirement can be defined as "A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document" [2]. In this paper, we consider a *requirement specification* as a way of describing requirements in textual form. Concretely, a (countable) requirement specification is a statement of the requirement in textual form. Several proposed methods aim to improve requirements specification, while maintaining some attributes of natural language, such as readability, and expressive power [7]. In the following subsections, we briefly discuss some relevant semi-formal, and also formal requirements specification methods.

8.2.1 Semi-formal Specification Methods

A semi-formal specification has defined syntax, but usually lacks semantics. *Requirements boilerplates*, as illustrated below (8.1), are semi-formal specifications originating from CESAR boilerplates [29]. The specification method uses restricted English grammar, but usually lacks formal semantics. A requirement boilerplate is constructed from attributes (shown in angle brackets), and fixed syntax elements, which are the rest of the boilerplate syntax. A requirement engineer selects boilerplates that suit the type of requirement to be specified, and fills the attributes in order to complete the specification.

$$\textit{If } \langle \textit{condition} \rangle, \ldots, \langle \textit{System} \rangle \textit{ shall be able to } \langle \textit{capability} \rangle \tag{8.1}$$

As opposed to usual natural language, a requirement boilerplate constrains the structure and choice of words that an engineer should use to construct a requirement. With this approach, specification errors can be reduced when authoring requirements. In order to support automated analysis of requirements, e.g., consistency checks, semi-formal specifications are often transformed into formal specifications, sometimes via patterns, e.g., the BOSCH Specification Pattern System (SPS) [20][32] is used to express natural language requirements in temporal logics.

8.2.2 Formal Specification Methods

Formal specifications are mathematical expressions of properties that a system should satisfy [13]. They have formal syntax that describes well-formed specifications, and also formal semantics that interprets the specification [22], e.g., as Z [30], LARCH [11], or four-variable models [31]. According to a study carried out by Woodcock et al. [37], formal methods can support a high-quality software development process in practice, nowadays being mostly applied in the safety-critical systems niche. Furthermore, the study reveals that the specification phase is time consuming and difficult. In this respect, semi-formal specifications could facilitate requirements formalization by first using restricted English grammar, and then transforming such specifications into formal ones, hence, alleviating the burden of complying to the mathematical rigor required when using formal specification methods.

8.3 Use-case: Adjustable Speed Limiter (ASL)

In this paper, we use an automotive function, that is, the Adjustable Speed Limiter (ASL), as proof of concept for our contribution. ASL is a vehicle speed control system that helps the driver to not exceed a predefined speed (ASLSetSpeed). This function is helpful especially in long highway drives, when a driver enters a low speed area. The predefined speed is configurable, according to the driver's request, through buttons located at the steering wheel level. ASL is a safety-critical user function, which is operational in Volvo trucks, as an alternative to other vehicle speed control functions, such as Cruise Control (CC), and EcoCruise (an extension to CC).

8.3.1 ASL System Description

ASL consists of a Human Machine Interface (HMI) for driver interaction, a main control subsystem that limits vehicle speed, and a display subsystem that notifies the driver. Figure 8.1 shows the main flow for ASL activation at current vehicle speed, that is, ASLSetSpeed is set to current vehicle speed.

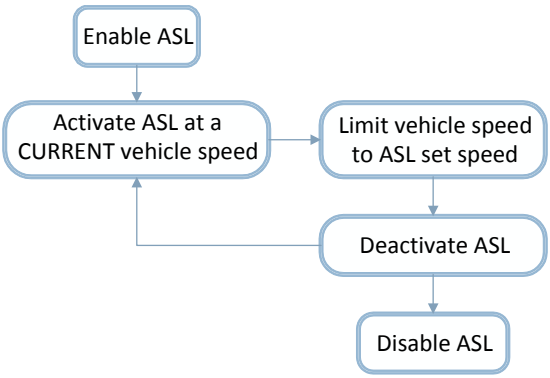

Figure 8.1: ASL activation at current vehicle speed, main flow

ASL is enabled by selecting the ASL mode using a freewheel located on the vehicle dashboard. If activation conditions are met, e.g., the vehicle is in running mode, ASL is activated. Next, the main control subsystem calculates the ultimate vehicle speed, which is the minimum of ASLSetSpeed and legal vehicle speed configured by the authority. The main control subsystem limits the engine torque not to exceed the ultimate vehicle speed. After activation, ASL can be deactivated, or disabled at any moment during operation.

8.3.2 EAST-ADL Model of ASL

EAST-ADL [5] is an architectural description language for automotive systems development. It is founded on levels of abstraction at which an automotive system is perceived with a certain detail. The levels of abstraction that are relevant to our work in this paper are briefly described as follows:

- **Vehicle-level** is the uppermost abstraction level. At this level an automotive system is realized through a set of vehicle features, which are functional or extra-functional traits of a vehicle. The vehicle features are structured into Technical Feature Model (TFM) that shows dependency between features, and realization of a high-level requirement.
- **Analysis-level**: here, a vehicle feature is realized through Functional Analysis Architecture (FAA), without regard to implementation details. The FAA is composed of abstract functional blocks (a.k.a. Analysis Functions).
- **Design-level**: analysis functions are further realized through Functional Design Architecture (FDA) with consideration of implementation issues, such as design constraints, and hardware resources. The FDA is composed of abstract design functions that implement application software, middleware, sensor/actuator handlers.

Figure 8.2 depicts the approach behind ASL function decomposition, and refinement of requirements. ASL, which is a Vehicle Feature (VF) in

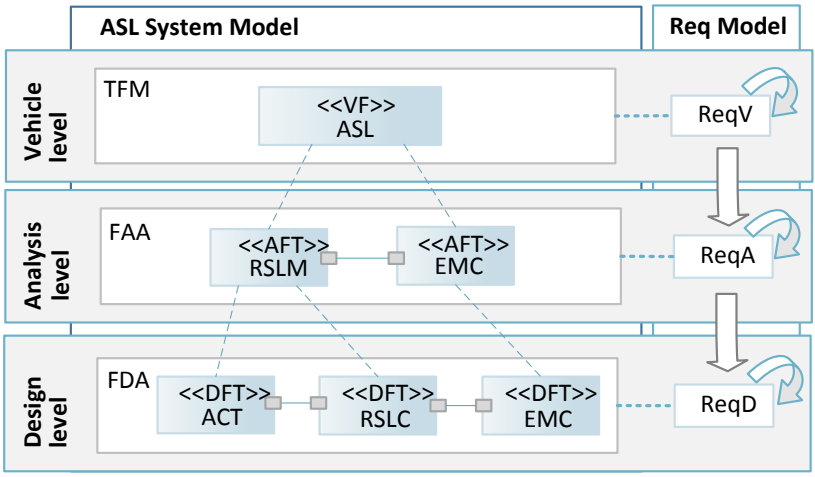

Figure 8.2: Adjustable Speed Limiter (ASL) system in EAST-ADL model

EAST-ADL, is decomposed into Road Speed Limit Manager (RSLM), and Engine Manager Control (EMC). RSLM is responsible for calculating the ultimate target speed. EMC is responsible for generating proportional power supply (a.k.a. mechanical torque) that matches the required longitudinal motion by RSLM. Furthermore, the diagram shows a specific case for RSLM decomposition into Activation (ACT), and Road Speed Limit Controller (RSLC) function blocks, which activate ASL, and calculate the ultimate target speed (`ulTargetSpd`), respectively.

Lower-level requirements are refinements of upper-level requirements, that is, $ReqV \Leftarrow ReqA \Leftarrow ReqD$ (a.k.a. vertical refinement). Requirements at the same levels of abstraction can also be refined, or decomposed, which is indicated through the curved arrows. In order to guarantee integrity of requirements amid development process, it is crucial to check the refinement of requirements within, as well as across levels of abstraction. In this regard, we represent requirements as boolean expressions that allow us to check refinement, and consistency of requirements. This is discussed in section 8.6.

8.3.3 Describing ASL Requirements in Natural Language: Issues

According to the current development status of ASL, there are 17 use cases and 32 scenarios that are formulated to capture functional and extra-functional requirements of ASL. There are also 244 functional and extra-functional requirements associated to logical components, which basically are refinements of the use cases and scenarios. The requirements are expressed in unconstrained natural language, in English, using a company

specific software engineering tool. In this subsection, we share some issues encountered in some relevant ASL requirements described in natural language.

Req# 1. *If ASL is active and the driver presses PAUSE button or vehicle exits vehicle mode running, the ASL function stops to limit the vehicle speed. The indication of the ASL active status is replaced with the ASL enabled status. The ASL switches to the enable state.*

Req#1 describes the actions to take, notably deactivation of ASL, if some conditions are met. Since no parentheses or commas are used, and given the precedence of 'and/or', the conditional expression is ambiguous.

Req# 2. *If RSL controller is determined to be faulty except if ultTargetSpd is in the error or spare range, value Error shall be used while the fault is present, Else if an external Road Speed Limit request is active in signal ultTargetSpd and the RSL controller is active, value active shall be used, Else value inactive shall be used.*

The above requirement is about determining RSL controller status. Due to the intricate conditional's structure, the requirement specification becomes hard to grasp. The requirement is to return status (i.e., error, active, inactive) under different conditions (e.g., RSL controller is faulty).

Req# 3. *If the IncreaseRequest is active during more than a configurable time (i.e., Speed control button long press time), ASLSetSpeed shall be* ***increased every 500 ms to the next multiple of 5 km/h****.*

With Req#3, we intend to show a timing requirement specification, and assignment of values from function expression in a generic form.

Our empirical study of ASL requirements suggests: 1) there is a tendency to use very technical automotive concepts, 2) the specifications tend to express requirements at different levels of abstraction, e.g., Req 1 at high-level abstraction, and Req 2 at design-level abstraction, and 3) intricate specifications give rise to ambiguous and complicated statements. In light of these findings, we can observe that current specification methods could be improved towards integrating features that address the above issues. Hence, we rewrite the above requirements using a specification language that we propose in the next section, in an attempt to improve their quality.

8.4 ReSA: A Requirements Specification Language

Here, we propose ReSA as a prototype of requirements specification language, for the automotive domain. The language uses human readable terms (words, phrases) to construct functional and extra-functional

requirements that resemble closely the requirements specified using plain English language. ReSA consists of a *grammar*, and a requirement specification *ontology*. The grammar provides syntactic rules that enable the specification of well-formed requirements; the ontology provides the language with *types*, and *axiomatic expressions* for constructing typed terms into the requirement specification.

8.4.1 Requirements Specification Ontology

The vernacular sense of an *ontology* is found in philosophy. According to Stanford Encyclopedia of Philosophy dictionary, it is a field that studies the existence of entities (or things) [17]. In addition, it also studies entity features, and the relations that exist between entities. In 1990s, Guber introduced the concept of ontology to Artificial Intelligence in order to facilitate knowledge sharing, and assist software development process [15]. Since then, the application of ontology has been gaining momentum in software engineering [16], and knowledge-based engineering [34]. For terminological definitions and clarification of ontology, we refer the reader to the work of Giaretta et. al [14].

Definition 1. *An ontology is a formal specification of a shared conceptualization* [4].

In the computer science community, there is no unique definition of ontology [14]. However, the above definition by Borst is sufficient for our needs. A conceptualization is a way of viewing a system at some level of abstraction in order to hide details and consequently, tame the system's complexity. Therefore, an ontology can be considered as means of formally expressing a part of conceptualization. The form of conceptualization that we are interested in this research is *requirement specification conceptualization*, that is, an abstraction that allows engineers to describe a requirement at a certain level of abstraction. An ontology can be used to capture part of this conceptualization, that is, the syntactic and semantic constructs of a requirement specification, which support engineers to express clear, unambiguous requirements.

Structurally, an ontology contains concepts (a.k.a. classes in object oriented vocabulary), conceptual relations (a.k.a. properties), and instances of concepts (a.k.a. individuals). In this paper, a *concept* refers to a class, or collection of entities in an ontology; a *conceptual relation* relates concepts in an ontology to describe some property of the ontology. Concepts can be abstract or concrete. If a concept is concrete, e.g., Driver, it does not require instantiation [21]. In our quest for developing ReSA, we identify the *system-level ontology* (short for system-level requirement specification ontology), and *abstraction-level ontologies* (short for abstract-level

requirement specification ontologies). This classification is on meta-level, at which our specification language is constructed.

System-level Ontology

This ontology expresses a part of the requirement specification that is generic to automotive system development. The ontology describes specifications related to driver interaction with a system (e.g., a driver pressing a button to activate an automotive function), specifications related to main functionality of an automotive function (e.g., controlling a vehicle speed), and specifications related to driver notification regarding a status of an automotive function.

In total, the system-level ontology is composed of 13 concepts (*S*), 24 conceptual relations, and 10 axiomatic expressions. The concepts act as *types* for terms in a requirement specification. The conceptual relations describe the precedence order of typed terms as they appear in a requirement specification. The axiomatic expressions describe statements, which are constraints on how requirements need to be specified.

Figure 8.3 shows graphically the system-level ontology for describing the main functionality of an automotive function, by a node-link diagram [19]. The entities having oval shape are *concepts* of the ontology, except `Driver`,

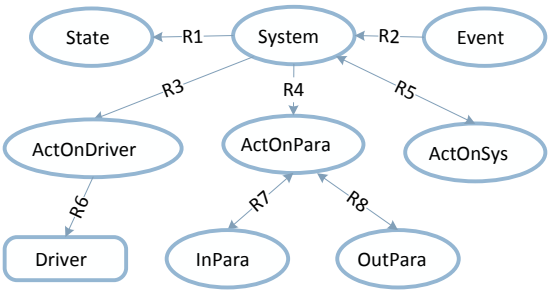

Figure 8.3: Part of system-level ontology chart

which is a concrete concept. The concepts are described as follows:

- `System` - refers to any physical or logical entity that can process an input and return an output, e.g., ASL (automotive function), RoadSpeedLimitManager (a logical component of ASL).
- `InPara` - refers to input parameters, or properties of `System`. Usually, changing the value of `InPara` affects the output value of `System`, e.g., `ASLSetSpeed` is an input parameter of ASL.
- `OutPara` - refers to output parameters of `System`, e.g., `ultTargetSpd` is an output parameter of ASL.
- `State` - refers to operational modes of `System`, e.g., ASL is `enabled` or `disabled`, `activated` or `deactivated`.

- `Event` - refers to internal or external occurrence that needs to be handled by `System`, e.g., the occurrence of internal fault in ASL (internal), driver request to activate ASL (external).
- `ActOnSys`, `ActOnPara`, `ActOnDriver` - refer to (action) verbs that precede `System`, `InPara/Outpara`, `Driver`, respectively, in a requirement specification. The main goal of classifying the action verbs is to identify the right semantics that matches a specific concept.

The edges shown in Figure 8.3 refer to conceptual relations. The relations have type label, `Is-fb` (a short for *Is followed by*). The type label has a signature (or stereotype) $< S, S >$, and it describes the precedence order of two adjacent typed terms, that is, a term with type S is followed by a term with another type S.

Definition 2. *According to the notion of subtyping (or subtype polymorphism), if B is a subtype of A, relations (or operations) that hold on instances of A also hold on instances of B. The subtyping is expressed mathematically as* $B \leq A$. *Furthermore, subtyping is closed under transitivity, that is, if* $B \leq A$ *and* $C \leq B$, *then* $C \leq A$ *is true* [24].

Following the above definition, if `S` is a type associated to the top level concept in system-level ontology, it is true that the following subtyping relations hold for the trace `R4->R7` (figure 8.3): `(System` $\leq$ `S)`, `(ActOnPara` $\leq$ `S)`, `(InPara` $\leq$ `S)`. As a result, the conceptual relation, `Is-fb`, also holds on the subtypes, e.g., `<System, ActOnPara>` is valid. The following extended CGIF (Conceptual Graph Interchange Format) encoding 1 shows a representation of the ontology for the trace `R4->R7`. CGIF is a knowledge representation language, based on first-order logic [33]. Therefore, CGIF notations can be transformed into semantically equivalent notations, such as first-order predicate logic [36]. The trace represents an axiomatic expression for constructing a requirement boilerplate. We use a similar approach to encode the rest of the ontology.

CGIF encoding: 1 For trace R4 $\rightarrow$ R7 of fig 8.3

```
1: [System *x1][ActOnPara *x2][InPara *x3]
2: (Is-fb ?x1 ?x2)(Is-fb ?x2 ?x3)
```

According to the semantics of CGIF language [25], the concepts are represented in pairs of square bracket, as indicated in line #1. Conceptual relations are represented in pairs of parentheses, as indicated in line #2. The co-reference labels (or variables, x1, x2, x3), which connect concepts and conceptual relations, are defined by *, and referenced by ?.

Abstraction-level Ontologies

Abstraction-level ontologies refer to Vehicle-, Analysis-, and Design-level ontologies. Each of these ontologies is a specialization of system-level

ontology for EAST-ADL levels of abstractions, via *subtyping* relations. These abstraction-level ontologies consist of concepts, and conceptual relations that are pertinent to their specific abstraction levels. This approach allows consistent and proper use of words and phrases at the right level of abstraction, to specify requirements. For instance, logical components at Design-levels are characterized by ports, signals, so it is not proper to use such terms to define requirements at Vehicle-, or Analysis-levels, as such a specification reduces comprehensibility of the requirement for non-technical stakeholders. In this way, type-checking can be enforced on requirements specifications, thereby, minimizing typing errors. For that matter, system-level ontology can be specialized to levels of abstraction found in automotive development models other than EAST-ADL [6].

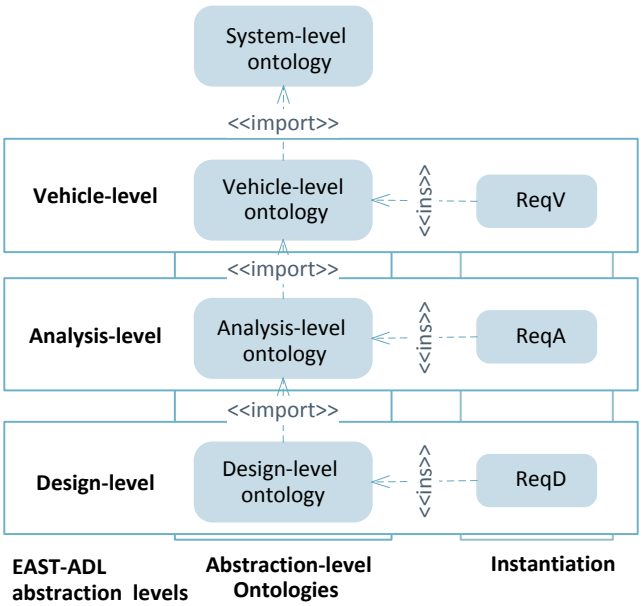

Figure 8.4: ReSA language meta-level

EAST-ADL levels of abstraction not only provide separation of concerns, but also top-down refinement of an automotive function. In this respect, an upper-level abstraction is a basis for further refinement at a lower-level abstraction. Therefore, upper-level ontologies should be accessible from lower-level ontologies. This is indicated in Figure 8.4 by relation «`import`». The import allows lower-level ontologies to access concepts and conceptual relations from upper-level ontologies (referred to as *ontology projection*) [4]. ReqV, ReqA, and ReqD designate instantiations of requirement boilerplates from Vehicle-, Analysis-, and Design-level ontologies, respectively.

System-level concepts	EAST-ADL abstraction-level concepts		
	Vehicle-level	**Analysis-level**	**Design-level**
System	VehicleFeature (VF)	AnalysisFunctionType (AFT) FunctionalDevice (FD)	DesignFunctionType (DFT) BasicSoftwareFunction (BSF) LocalDeviceManager (LDM) HardwareFunctionType (HFT)
InPara/OutPara	Stimuli/Response	InSig/OutSig; Signal is a tuple <request, value>, and the fields are accessed through the infix notation '.', as <signal name>.request, <signal name>.value	

Table 8.1: Example: System-level, and abstraction-level concepts subtyping in an EAST-ADL specification [1]

Table 8.1 shows system-level concepts, that is, `System`, `InPara`, `OutPara`, and their subtypes from EAST-ADL concepts. For instance, at Vehicle level, `(VF ≤ System)`, `(Stimuli ≤ InPara)`; at Analysis level, `(AFT ≤ System)(InSig ≤ InPara)`; at Design level, `(DFT ≤ System)(InSig ≤ InPara)`. In this way, one can construct a taxonomy of types that is later used to typeset terms in a requirement specification. Figure 2 shows the CGIF encoding for the trace `R4->R7`, specialized to the EAST-ADL levels of abstraction at Vehicle-, Analysis-, and Design-level ontologies.

CGIF encoding: 2 For trace R4 → R7

```
1: [VF *x1][ActOnPara *x2][Stimuli *x3]
   Infer: (Is-fb ?x1 ?x2)(Is-fb ?x2 ?x3)
                                  ▷ Vehicle-level ontology
2: [AFT *x1][ActOnPara *x2][InSig *x3]
   Infer: (Is-fb ?x1 ?x2)(Is-fb ?x2 ?x3)
                                ▷ Analysis-level ontology
3: [DFT *x1][ActOnPara *x2][InSig *x3]
   Infer: (Is-fb ?x1 ?x2)(Is-fb ?x2 ?x3)
                                  ▷ Design-level ontology
```

The encoding shows how precedence operations at abstraction-level ontologies, e.g., `Is-fb`, can be inferred from the system-level ontology. For instance, line#1 defines concepts from Vehicle-level ontology, and the relation `Is-fb` can be inferred from the system-level ontology according to the transitivity property of subtyping: `(System ≤ S)`, and `(VF ≤ System)`, then `(VF ≤ S)`. Line#2, line#3 use a similar approach to line#1.

8.4.2 ReSA Context-Free Grammar

The ReSA grammar, *G*, is not only a set of syntactic rules used to guarantee that a requirement specification is well-formed. The grammar also encodes *types*, and operations on *typed terms* inferred from the ontologies. This enables type-checking of a requirement specification for detecting and removing type errors during authoring requirements. The grammar is a 6-tuple, as follows:

$$G = (N, N_s, T_a, T, S, P)$$

- N - is a finite set of non-terminal strings, and not part of the ontology
- N_s - is a finite set of non-terminal strings that are concepts of the system-level ontology. The strings start with a capital letter
- T_a - is a finite set of terminal strings that are concepts of the abstraction-level ontologies. The strings start with a capital letter

- T - is a finite set of terminal strings that are fixed terms of a requirement specification
- S - is a finite set of non-terminal strings, and marks the start of a requirement specification. It is a subset of the non-terminal strings, N
- P - is a finite set of production rules used to derive strings of the language.

In Backus-Naur Form (BNF) [12], the production rules appear as follows:

$$\langle \textit{non-terminal} \rangle ::= \textit{expression},$$

where '::=' refers to a substitution of *non-terminal* string with *expression*. The grammar is context-free; therefore, the left-hand side of the production rules have a single non-terminal string; at the right-hand side of the production rule, *expression* could be terminal, or non-terminal string, in any order. While non-terminal strings should be enclosed within pairs of angle brackets, this is optional for terminal strings [23].

8.4.3 Sentential forms (Requirements Boilerplates)

The language, $ReSA(G)$, is an infinite set of strings recursively built from the production rules of the grammar, P. In fact, the strings of the language are sentential-forms (include variables, or place-holders), as opposed to strings that are constructed from terminal strings only (without variables). Hereafter, the forms will be referred to as *requirements boilerplates*. The constructs of a requirement boilerplate and their syntax are discussed in this section. A syntax of these constructs has arguments enclosed between square brackets: [argument]. If the argument is optional, it is enclosed between braces: {argument}. The argument can be a typed term, or untyped term. A typed term is enclosed in a pair of angle brackets, `<term:[type]>`, where `[type]` refers to concepts of the ontologies. The requirement boilerplate will be referred to as *boilerplate*. Every boilerplate is prefixed by [*ReqV* | *ReqA* | *ReqD*], to indicate that a boilerplate belongs to a Vehicle-, Analysis-, or Design-level of an EAST-ADL model, respectively.

Simple Boilerplates

These boilerplates are constructed from a single main clause, and a single prepositional phrase (abbreviated as *prep phrase*). The main clause is an independent statement, and a prepositional phrase complements the main clause with a constraint, or other information that elaborates the main clause, e.g., reference to a standard, look-up table, etc. A *simple boilerplate* is the smallest requirement boilerplate that can be generated from the language.

$\langle \textit{simple} \rangle ::= \langle \textit{main} \rangle \mid \langle \textit{main} \rangle \ \langle \textit{prepositionphrase} \rangle$

Syntax :

$$\underbrace{[\textit{Subject}]\ [\textit{Modal verb}]\ [\textit{Verb}]\ \{\textit{Object}\}}_{\text{main clause}}\{\textit{Prep phrase}\}$$

- *Subject* and *Object* refer to typed terms, `<term:T>`, where `T` $\leq$ `{System, InPara, OutPara}`
- *Modal verb* refers to fixed syntax terms, e.g., `{shall, may, shall be able to}`
- *Verb* refers to typed terms, `<term:T>`, where `T,` $\in$ `{ActOnSys, ActOnPara, State}`

An instantiation of the above syntax can be shown below:

Boilerplate 3 : Simple boilerplate

1: `ReqV: <term:VF> shall <term:ActOnPara> <term:Stimuli>` ▷ vehicle-level
2: `ReqA: <term:AFT> shall <term:ActOnPara> <term:InSig>` ▷ analysis-level
3: `ReqD: <term:DFT> shall <term:ActOnPara> <term:InSig>` ▷ design-level

Proposition Boilerplates

These boilerplates allow one to define declarative statements that return true or false truth-values. A declarative statement is used as a condition inside a conditional statement, or to construct a complex sentence, which we discuss later. Such boilerplates can capture internal (8.2) or external events (8.3), operational states of a system (8.4), vehicle modes (8.5), or comparison of quantitative values (8.6).

$$\langle proposition \rangle ::= \langle event \rangle \mid \langle operationalstate \rangle \mid \langle vehiclemode \rangle \mid \langle comparisonproposition \rangle$$

Syntax :

$$\langle term : Event \rangle \; occurs \; \{Prep\; phrase\} \ldots \tag{8.2}$$

$$\langle Driver \rangle \; [verb] \; [Object] \; \{Prep\; phrase\} \ldots \tag{8.3}$$

where *Verb* is `<term:ActOnInDev|ActOnPara>`, *Object* is `<term:InDevice|InPara>`

$$[Subject] \; is \; [Object] \; \{Prep\; phrase\} \ldots \tag{8.4}$$

where *Subject* is `<term:System>`; *Object* is `<term:State>`

$$Vehicle\; is\; in \; [Object] \; \{Prep\; phrase\} \ldots \tag{8.5}$$

where *Object* is `<term:mode>`

$$[Operand1] \; [Comp\; operator] \; [Operand2] \; \ldots \tag{8.6}$$

where $Operand1$, and $Operand1$ refer to a variable or a literal value. $Comp\; operator$ are operators, which are discussed later, 8.4.3

Boilerplate 4 : Proposition boilerplate

```
1: <term:Event> occurs...
2: <Driver> <term:ActInDev> <term:InDevice>...
3: <term:VF> is <term:State>...
4: Vehicle is in <term:Mode>...
5: <variable> <is greater than> <constant>...
```

Operators

The logical operators (AND, OR) conjugate two or more simple boilerplates, or proposition boilerplates in order to create a compound boilerplate. The logical unary operator (NOT) negates a simple or a proposition boilerplate. The comparison operators compare quantitative values. The operator AND has higher precedence than OR; however, parentheses could be used in order to enforce a different order of the boolean evaluation.

⟨*operator*⟩ ::= ⟨*logical*⟩ | ⟨*comparison*⟩

⟨*logical*⟩ ::= 'AND' | 'OR' | 'NOT'

⟨*comparison*⟩ ::= 'equals' | 'not equals'
| 'less than' | 'greater than'
| 'less than or equal to'
| 'greater than or equal to'

Prepositional phrase Boilerplates

These boilerplates allow one to build a prepositional phrase. A prepositional phrase is constructed from a preposition followed by an object. This type of prepositional phrase is mainly used to express a constraint that is suffixed to a simple, or a propositional boilerplate. The constraint could be timing, occurrence of events, or assignment of values.

Syntax :

$$\ldots \underbrace{[timing \mid occurence \mid assignment]}_{\text{preposition}} [Object] \tag{8.7}$$

The preposition is represented by the non-terminal string ⟨*prepositionphrase*⟩. The objects of the prepositional phrase, *Object*, are terminal strings following the prepositions indicated by ⟨*timing*⟩, ⟨*occurrence*⟩, and ⟨*assignment*⟩.

⟨*prepositionphrase*⟩ ::= ⟨*timing*⟩ | ⟨*occurrence*⟩ | ⟨*assignment*⟩

⟨*timing*⟩ ::= 'at' ⟨*time*⟩ | 'after' ⟨*time*⟩ | 'until' ⟨*time*⟩
| 'within' ⟨*time*⟩ | 'for' ⟨*time*⟩
| 'between' ⟨*time*⟩ 'and' ⟨*time*⟩ | 'every' ⟨*time*⟩

⟨*occurrence*⟩ ::= ‘`after`’ ⟨*event*⟩ | ‘`before`’ ⟨*event*⟩
| ‘`between`’ ⟨*event*⟩ ‘`and`’ ⟨*event*⟩

⟨*assignment*⟩ ::= ‘`set to`’ ⟨*value*⟩
| ‘`set to less than`’ ⟨*value*⟩
| ‘`set to greater than`’ ⟨*value*⟩
| ‘`set to less than or equal to`’ ⟨*value*⟩
| ‘`set to greater than or equal to`’ ⟨*value*⟩
| ‘`set to between`’ ⟨*value*⟩ ‘`and`’ ⟨*value*⟩,

⟨*time*⟩, and ⟨*value*⟩ could be literal, or variable. Moreover, they could represent a single value, list, multi-values expressed with inequality, or a computed value, e.g., return value of a function.

An instantiation of the above syntax is shown below:

Boilerplate 5 : Prepositional phrase boilerplate

```
1: <within> <quantity> <unit>                    ▷ timing
2: <after> <term:Event>                           ▷ event
3: set to <greater than> <quantity> <unit>              ▷
   assignment
```

Complex Boilerplates

These boilerplates capture complex statements. A complex sentence is constructed from a single simple statement, and a subordinate clause. The subordinate clause is an independent clause, which cannot describe a requirement by itself. Complex boilerplates allow one to describe a requirement that is more complex than a requirement that can be described with a simple boilerplate, e.g., to limit requirement validity to a specific occasion using a conjunctive, e.g., `while`.

⟨*complex*⟩ ::= ⟨*simple*⟩ ⟨*subordinate*⟩

⟨*subordinate*⟩ ::= ⟨*conjunctive*⟩ ⟨*proposition*⟩

⟨*conjuctive*⟩ ::= ‘`while`’ | ‘`when`’

Boilerplate 6 : Complex boilerplate

1: `ReqV:` [Boilerplate 3] `<when>` [Boilerplate 4]
2: `ReqA:` [Boilerplate 3] `<when>` [Boilerplate 4]
3: `ReqD:` [Boilerplate 3] `<when>` [Boilerplate 4]

Conditional Boilerplates

These boilerplates allow one to create conditional statements. Each has *condition* and *consequent* parts. The condition part is specified by a boolean

expression of one or more propositions connected using binary operators (AND, OR). The consequent part is expressed by a simple or a compound boilerplate. The compound boilerplate is a composition of two or more simple boilerplates conjugated using binary operators (AND, OR).

Syntax :

$$PRECONDITION : [condition] \tag{8.8}$$

If conditional

$IF [condition]\ THEN$
$\quad [consequent]$
$ENDIF$

If-else conditional

$IF [condition]\ THEN$
$\quad [consequent]$
$ELSE$
$\quad [consequent]$
$ENDIF$

If-elseif conditional

$IF [condition]\ THEN$
$\quad [consequent]$
$ELSEIF [condition]\ THEN$
$\quad [consequent]$
$ENDIF$

If-elseif-else conditional

$IF [condition]\ THEN$
$\quad [consequent]$
$ELSEIF [condition]\ THEN$
$\quad [consequent]$
$ELSE$
$\quad [consequent]$
$ENDIF$

$\langle conditional \rangle ::= \langle if \rangle \mid \langle ifelse \rangle \mid \langle ifelseif \rangle \mid \langle ifelseifelse \rangle$

$\langle condition \rangle ::= \langle proposition \rangle$
$\mid \ (\langle proposition \rangle\ \langle binary \rangle\ \langle proposition \rangle)$

$\langle consequent \rangle ::= \langle simple \rangle \mid (\langle simple \rangle\ \langle binary \rangle\ \langle simple \rangle)$
$\mid \ \langle complex \rangle \mid (\langle complex \rangle\ \langle binary \rangle\ \langle complex \rangle)$

Boilerplate 7 : Conditional boilerplate

```
ReqV:                                                    ▷ vehicle-level
IF <Driver> <term:ActOnInDev> <term:InDevice>
THEN
  <term:VF> shall <term:ActOnPara> <term:Stimuli>
ENDIF
```

8.4.4 Implementation

The language is implemented in xText [3], which is a framework for developing domain-specific languages. xText has rich runtime components,

such as the parser, Type-Safe Abstract Syntax Tree (AST), the serializer, allowing an end-to-end development of a domain-specific language, that is, from encoding the grammar of the language to a specifically tailored IDE for the language.

8.5 Applying ReSA to ASL Requirements

To illustrate its usefulness and validate our approach, we have applied ReSA to specify the ASL's functional and timing requirements. The requirements range from end-to-end functional requirements at Vehicle-level, to component requirements at Design-level of EAST-ADL. Here, we show the ReSA description of the ASL requirements introduced in section 8.3 (see ReSA_Req 1 to ReSA_Req 3).

ReSA_Req 1 : Rewriting Req1

ReqV: ▷ Vehicle-level

```
IF (<ASL:VF> is <activated:State> AND (<Driver>
<presses:ActOnInDev> <PAUSE button:InDevice> OR
   vehicle is not in <running mode:Mode))
THEN
  <ASL:VF> shall <stop to limit:ActOnPara> <vehicle
speed:Stimuli> AND
  <ASL status:Response> shall be <set to:ActOnPara>
<enabled:State> AND
  <ASL:VF> shall be <enabled:State>
ENDIF
```

ReSA_Req 2 : Rewriting Req2

ReqD: ▷ Design-level

```
IF <fault affecting RSL controller:Event> occurs AND
<ultTargetSpd.value> is not in {<error>, <spare>}
interval
THEN
  Status <Error:Status> shall be displayed while <the
fault:Event > is <active:State>
ELSEIF <ultTargetSpd.request:InPara> is <active:State>
AND <RSL controller:DFT> is <activated:State>
THEN
  Status <active:Status> shall be displayed.
ELSE
 Status <inactive:Status> shall be displayed.
ENDIF
```

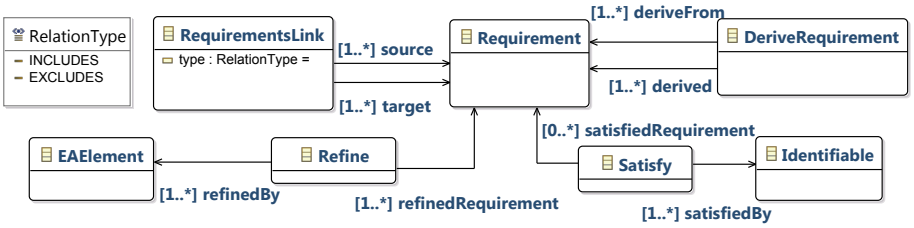

Figure 8.5: EAST-ADL Compatible Requirement Meta-model [1]

ReSA_Req 3 : Rewriting Req3

ReqV: ▷ Vehicle-level

```
IF <IncreaseRequest:Stimuli> is <active:State> <for>
<greater than> <SpeedCtrlBtnLongPressTime>
THEN
  <ASLSetSpeed:Stimuli> shall be set to <next multiple
of> <5> <kmph> every <500> <ms>
  <ASLSetSpeed:Stimuli> shall be <limited
to:ActOnPara> <Vmax>.
ENDIF
```

8.6 Requirements Analysis

Figure 8.5 is a requirement relationship meta-model based on the requirement relationship meta-model from EAST-ADL specification. Classes and attributes that are relevant to our discussion are indicated in the diagram. In this section, we show how ReSA requirement specification complements requirement modeling in EAST-ADL. Furthermore, we propose a consistency check theorem that helps to identify contradictions during requirement decomposition, or refinement process. The meta-model consists of a requirement class, modeling elements classes (EAElement, Identifiable), and requirement relations (Refine, Satisfy, Derive). Every relation has a client and a source modeling element.

- The Refine class indicates refinement of source (supplier) requirement with one or more refining clients, EAElement, which are either requirements, or modeling elements.
- The Satisfy class indicates satisfying a supplier requirement with one or more satisfying clients, Identifiable that are design/architectural elements.
- The DeriveRequirement class indicates derivation of new client requirements from a supplier requirement.
- Requirement dependency is shown through the RequirementsLink class. In order for a requirement to be satisfied, it might require another requirement, which is indicated through the type INCLUDES, or it might be unsatisfiable

in a presence of another requirement, which is indicated through the type EXCLUDES.

Let us assume a set of ReSA requirements in form of conditionals. In the following, we formalize such requirements as conjunctions of implications, where $A, B, ..., N$ are predicates that are either antecedents (left-hand side of each implication), or consequents (right-hand side of each implication), respectively:

If conditional	If-else conditional
$A \Rightarrow B$	$C \Rightarrow D$
	$\wedge$
	$\neg C \Rightarrow E$

If-elseif conditional	If-elseif-else conditional
$F \Rightarrow G$	$J \Rightarrow K$
$\wedge$	$\wedge$
$\neg F \wedge H \Rightarrow I$	$\neg J \wedge L \Rightarrow M$
	$\wedge$
	$\neg J \wedge \neg L \Rightarrow N$

In our framework, one should be able to prove the refinement relationships described above, both intra- as well as inter-abstraction levels of the software's architectural model, according to the abstraction-level ontology of ReSA, which ensures the fact that the defined boilerplates are instantiated with architectural elements valid for a specific level of abstraction. The definition below introduces formally the simple notion of refinement:

Definition 3 (Requirement refinement)**.** *Consider ReSA requirements* Req_1 *and* Req_1'. *The refinement of* Req_1 *by* Req_1' *is defined as follows:*

$$\text{Req\_1}' \text{ Refine Req\_1} \triangleq \text{Req\_1}' \Rightarrow \text{Req\_1}$$

Given a $A \Rightarrow B$ requirement, a refinement of such requirement according to Definition 3 holds if the antecedent A is weakened, or the consequent B is strengthened. That is $A \vee A' \Rightarrow B$ refines $A \Rightarrow B$, and so does $A \Rightarrow B \wedge B'$.

Finding inconsistencies in early-stage ReSA requirements reduces to finding contradictions in the set of boolean implications. We say that a set of ReSA requirements (of the form assumed above) is inconsistent if the following implication holds:

Theorem 1 (Inconsistency of requirements)**.** *Assuming* $Ai, Bi, ..., Nl$ *predicates of requirements Reqi, Reqj, Reqk,*

Reql $i, j, k, l \in [1..n]$, *we say that the set of requirements* $\bigwedge_i Reqi \wedge \bigwedge_j Req\,j \wedge \bigwedge_k Reqk \wedge \bigwedge_l Reql$ *is inconsistent if the following implication holds:*

$$(\bigwedge_i(\neg Ai \vee Bi) \wedge \bigwedge_j((\neg Cj \vee Dj) \wedge (Cj \vee Ej)) \wedge$$

$$\wedge \bigwedge_k((\neg Fk \vee Gk) \wedge (Fk \vee \neg Hk \vee Ik))$$

$$\wedge \bigwedge_l((\neg Jl \vee Kl) \wedge (Jl \vee \neg Ll \vee Ml) \wedge (Jl \vee Ll \vee Nl))$$

$$\Rightarrow False$$

We know that an interpretation will satisfy the formula above if and only if it satisfies the formula's universal closure, that is, for any i, and any variable value of $Ai, ..., Ni$ the negation of the whole left-hand side conjunct of Theorem 1 holds. In order to disprove the inconsistency theorem, one needs to prove that its negation is true, that is, find a counterexample that would satisfy the conjunction of all ReSA requirements. Such problem can be easily solved by a SAT solver or a theorem prover. Applying the consistency check to the set of three ASL ReSA specified requirements renders them consistent.

8.7 Related Work

There is a growing interest, and research in requirements formalization in the realm of requirement engineering due to compliance requirements in safety standards, and the benefits that could be gained from formalization likewise. This is predominantly common for safety-critical applications, e.g., in automotive, rail, and aerospace industries. This is justified by the fact that safety-critical functions need be tested, and verified at early stages of system development in order to detect and reduce software errors. In this section, we discuss the relevant state-of-the-art, and state-of-practice on requirement specification approaches.

Requirement boilerplates

Requirement boilerplates have gained popularity in academic research as well as industry, e.g., CESAR RSL [29], ontology-based requirement boilerplates (DODT tool) [8], Requirement Authoring Tool (RAT) [35], Easy Approach to Requirement Syntax (EARS) [28]. Unlike these methods, ReSA is a specification language intended for automotive systems. Besides, current requirement boilerplates share the same set of attributes to express requirements at any level of abstraction. In order to improve precision and type-checking of requirements, we treat each level of abstraction as distinct, therefore, have distinct attributes (or concepts in this paper). DODT tool supports domain-specific ontology in order to guide and improve requirement

specification. In this paper, we propose instead an abstract ontology that acts as an abstract type-system for requirement terms, and guides engineers during specification.

Pattern-based requirement specifications

The Specification Pattern System (SPS) [20] is based on a restricted English grammar that has been recently extended to support requirements specification for automotive systems [32][9]. Concerning readability, the requirements boilerplates syntax appear close to natural language, and are more readable. To our findings, there is no clear criterion that differentiates the two specification methods. As compared to SPS, ReSA allows specifications that render natural language. However, SPS has semantically equivalent specifications in first-order-logic, CTL, etc., which ReSA has not implemented yet.

8.8 Conclusions and Future work

In this paper, we propose a structured requirement specification language for automotive systems. The specification language is human readable, and renders natural language sentence construction. It uses the notion of ontology to capture automotive concepts, and precedence order of typed terms and axiomatic expressions that allow the construction of unambiguous requirements specifications. The language is extended to express requirements designated to EAST-ADL levels of abstraction. However, the approach can also be applied to support abstraction levels of industrial development process models other than EAST-ADL (e.g., SysML). Moreover, we show how refinement and consistency checks can be carried out, to allow requirements analysis at early stages of system development, prior to full formal verification. Finally, we have applied the language to express a representative subset of ASL requirements at different levels of abstraction.

The ReSA requirement specification language is one step closer to formal specification. Future work includes complete implementation of the language followed by its validation on the ASL system that has to satisfy 244 functional and extra-functional requirements. In addition, we plan to work further on the automatic transformation of ReSA requirements to formal TCTL requirements that could be fed to ViTAL [26], our recently developed tool-chain for formal verification of EAST-ADL models, which uses the UPPAAL model-checking engine. Our ultimate goal is to be able to integrate formal verification of automotive architectural models into the industrial system development.

8.9 References

[1] East-adl v2.1.12 specification. `http://www.east-adl.info/Specification/V2.1.12/html/index.html`. Accessed: 2015-03-11.

[2] *IEEE Guide for developing system requirements specifications : IEEE Std 1233-1998(R2002).* Beuth, 1998.

[3] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, and Peter Friese. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, 2008.

[4] Willem Nico Borst. *Construction of engineering ontologies for knowledge sharing and reuse*. Universiteit Twente, 1997.

[5] ATESST2 Consortium. EAST-ADL Profile Specification, 2.1 RC3 (Release Candidate). pages 10–75. www.atesst.org, 2010.

[6] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. East-adl—an architecture description language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

[7] Jeremy Dick and Juan Llorens. *Using statement-level templates to improve the quality of requirements*. Integrate Systems eEngineering ltd, 2012.

[8] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, 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.

[9] Predrag Filipovikj, Mattias Nyberg, and Guillermo Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *22nd IEEE International Requirements Engineering Conference*, August 2014.

[10] D. Firesmith. Common requirements problems, their negative consequences, and the industry best practices to help solve them. *Journal of Object Technology*, 6(1):17–33, 2007.

[11] Stephen J Garland, Kevin D Jones, A Modet, and Jeannette M Wing. Larch: languages and tools for formal specification. *Springer-Verlag Texts and Monographs in Computer Science*, 1993.

[12] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work. *acedida pela última vez em*, 16, 2003.

[13] Marie-Claude Gaudel. Formal specification techniques. In *Proceedings of the 16th international conference on Software engineering*, pages 223–227. IEEE Computer Society Press, 1994.

[14] Pierdaniele Giaretta and N Guarino. Ontologies and knowledge bases towards a terminological clarification. *Towards very large knowledge bases: knowledge building & knowledge sharing*, 25, 1995.

[15] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, 43(5–6):907 – 928, 1995.

[16] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

[17] Thomas Hofweber. Logic and ontology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2014 edition, 2014.

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

[19] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia Giannopoulou. Ontology visualization methods—a survey. *ACM Comput. Surv.*, 39(4), November 2007.

[20] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[21] W. Kusnierczyk. Nontological engineering. In *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, volume 150 of *Formal ontology in information systems: proceedings of the fourth international conference (FOIS 2006)*, pages 39 – 50, 2006.

[22] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

[23] Peter Linz. *An introduction to formal languages and automata*. Jones & Bartlett Publishers, 2011.

[24] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[25] Common Logic. Information technology–common logic (cl): A framework for a family of logic based languages (final draft–reference number: Iso/iec fdis 24707: 2007 (e)). *Geneva: ISO Copyright Office*, 2007.

[26] R. Marinescu, H. Kaijser, M. Mikučionis, C. Seceleanu, H. Lönn, and A. David. Analyzing industrial architectural models by simulation and model-checking. In *Third International Workshop on Formal Techniques for Safety-Critical Systems (FSTCS2014)*. IEEE CS, Nov. 2014.

[27] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy approach to requirements syntax (ears). In *Requirements Engineering Conference, 2009. RE '09. 17th IEEE International*, pages 317–322, Aug 2009.

[28] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (ears). In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 317–322. IEEE, 2009.

[29] Andreas Mitschke. *Definition and exemplification of RSL and RMM D-SP2-R2.2-M2*. CESAR PROJECT, 2010-10-08.

[30] Gerard O'Regan. Z formal specification language. In *Mathematics in Computing*, pages 109–122. Springer London, 2013.

[31] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41 – 61, 1995.

[32] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. *Requirements Engineering*, 17(1):19–33, 2012.

[33] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[34] Rudi Studer, V Richard Benjamins, and Dieter Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1):161–197, 1998.

[35] Kunal Verma and Alex Kass. *Requirements analysis tool: A tool for automatically analyzing software requirements documents*. Springer, 2008.

[36] Michel Wermelinger. Conceptual graphs and first-order logic. In Gerard Ellis, Robert Levinson, William Rich, and JohnF. Sowa, editors, *Conceptual Structures: Applications, Implementation and Theory*, volume 954 of *Lecture Notes in Computer Science*, pages 323–337. Springer Berlin Heidelberg, 1995.

[37] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.