Automatically Extracting Requirements Specifications from Natural Language

Shalini Ghosh1, Daniel Elenius1, Wenchao Li1, Patrick Lincoln1, Natarajan Shankar1, Wilfried Steiner2

1CSL, SRI International, Menlo Park. {shalini,elenius,li,lincoln,shankar}@csl.sri.com
2TTTech C. AG, Chip IP Design, A-1040 Vienna, Austria. wilfried.steiner@tttech.com

Abstract. Natural language (supplemented with diagrams and some mathematical notations) is convenient for succinct communication of technical descriptions between the various stakeholders (e.g., customers, designers, implementers) involved in the design of software systems. However, natural language descriptions can be informal, incomplete, imprecise and ambiguous, and cannot be processed easily by design and analysis tools. Formal languages, on the other hand, formulate design requirements in a precise and unambiguous mathematical notation, but are more difficult to master and use. We propose a methodology for connecting semi-formal requirements with formal descriptions through an intermediate representation. We have implemented this methodology in a research prototype called ARSENAL with the goal of constructing a robust, scalable, and trainable framework for bridging the gap between natural language requirements and formal tools. The main novelty of ARSENAL lies in its automated generation of a fully-specified formal model from natural language requirements. ARSENAL has a modular and flexible architecture that facilitates porting it from one domain to another. ARSENAL has been tested on complex requirements from dependable systems in multiple domains (e.g., requirements from the FAA-Isolette and TTEthernet systems), and evaluated its degree of automation and robustness to requirements perturbation. The results provide concrete empirical evidence that it is possible to bridge the gap between stylized natural language requirements and formal specifications with ARSENAL, achieving a promising level of performance and domain independence.

1 Introduction

Software systems operate in the real world, and often work in conjunction with complex physical systems. Many different stakeholders participate in the design and operation of these systems. In this setting, natural language descriptions and formal modeling languages each offer distinct advantages to the system designer. The informality of natural language can kick-start discussion among stakeholders in early design, but can lead to confusion, lack of automation, and errors. The
Table 1. Key innovations in ARSENAL.

| Challenges                                                                 | Key Insights                                                                 |
|---------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Bridge the gap between semi-formal natural language requirements and precise formal models. | Create a rich/expressive intermediate representation (IR), useful for generating outputs in multiple formalisms. |
| Create a general-purpose architecture that can be ported to different domains. | Encapsulate domain-specific components in modules (e.g., NL preprocessor, output generators), keeping rest of system domain-independent and reusable. |
| Incorporate semantics into formal model generation.                       | Add semantics to the formal model via rewrite rules and type inference algorithms in the model generator stage. |

Rigor of formal languages can eliminate broad classes of ambiguity, enable consistency checking, and facilitate automatic test case generation. However, mastery of formal notations requires a significant amount of training and mathematical sophistication.

Most of the costly errors often enter at the requirements stage as a result of confusion among stakeholders [8]: “If a defect is found in the requirements phase, it may cost $1 to fix. It is proffered that the same defect will cost $10 if found in design, $100 during coding, $1000 during testing [3].” In order to catch as many errors as possible during the requirements phase, iteration between the stakeholders through clear communication in natural language must be supported. Formal models and descriptions that can detect errors, incompleteness, ambiguity, and inconsistency in the requirements should also be used. By bridging the gap between semi-formal requirements and formal specifications, we can dramatically reduce the number of costly uncaught errors in requirements and enable high levels of assurance for critical complex systems. Figure summarizes the trade-off between natural language and formal requirements specifications. We aim to leverage the best of both natural and formal languages to support the system designer in achieving high assurance for critical systems. This paper’s primary objective is answering this question:

*Can we build a requirements engineering framework that combines the strengths of semi-formal natural language and precise formal notations?*

To that effect, we present the “Automatic Requirements Specification Extraction from Natural Language” (ARSENAL) methodology. ARSENAL uses state-of-the-art advances in natural language processing (NLP) and formal methods (FM) to connect natural language descriptions with their precise formal representations.

ARSENAL is an exploratory and experimental open-loop framework for extracting formal meaning from semi-formal text. ARSENAL provides a method for extracting relevant information from *stylized* natural language requirements.
documents and creating formal models with that information. The stylized
fragment imposes a degree of precision and consistency in the way concepts
are expressed, but it is neither as restrictive as templates nor as imprecise
as free-form text. Let us consider the following sentence, which is part of the
requirements specification for a regulator that regulates the temperature in
an isolette (an incubator for an infant that provides controlled temperature,
humidity, and oxygen):

**REQ1:** *If the Status attribute of the Lower Desired Temperature or the Upper
Desired Temperature equals Invalid, the Regulator Interface Failure shall be set
to True.*

This requirements sentence is written with terminology that is particular to
the domain, in a stylized language that facilitates precise and comprehensible
communication between the relevant stakeholders involved in different stages
of the isolette design. ARSENAL aims to provide a natural language front-end
to formal analysis that is robust and flexible across different forms of natural
language expressions in different domains, and customized to stylized usages
within each domain.

![Trade-off between natural language and formal specifications](image)

**Fig. 1.** Trade-off between natural language and formal specifications [1], inset showing
the design-iteration cycle of the ARSENAL methodology.
Some of the benefits of ARSENAL include: (1) resolution of semantic ambiguities and co-references in requirements, (2) consistency/redundancy checking and validation of requirements, (3) example generation for test cases, (4) putative theorem exploration, (5) traceability to connect implementations to requirements they implement, (6) feedback on requirements quality and hints for improvement to the end user, facilitating iterative requirements refinement. Overall, ARSENAL facilitates communication between stakeholders (e.g., formal methods modelers, requirements engineers), which is important for critical systems like avionics, and helps to resolve imprecise requirements.

The input of ARSENAL consists of requirements in stylized natural language with specific technical content. ARSENAL uses domain-specific semantic parsing to extract formulas in first-order logic and linear-temporal logic (LTL) from requirements text in the NLP stage. These are then converted to specifications in the FM stage, which can be used by formal verification tools like theorem provers (e.g., PVS [31]) and model checking tools (e.g., SAL [5]) as well as LTL synthesis tools (e.g. RATSY [6]) for automated analysis of the formal specifications.

The main challenges and key insights of ARSENAL are outlined in Table 1. The organization of the rest of the paper is as follows: Section 2 gives an overview of ARSENAL, while Sections 3 and 4 describe the NLP and FM stages in more detail. Section 5 discusses the results of our experiments with ARSENAL on the FAA-Isolette and TTEthernet requirements documents, followed by Section 6 which discusses the novelty of ARSENAL as compared to related research. Finally, Section 7 summarizes the contributions of this work and outlines possible future directions of research.

2 The ARSENAL Methodology

Fig. 2. ARSENAL pipeline.
In this section, we give an overview of the flow (shown in Figure 2) using the example requirement sentence REQ1 from the FAA-Isolette corpus, introduced in Section II.

Given requirements written in stylized natural language, ARSENAL first processes them using a natural language processing (NLP) stage. The NLP stage has a preprocessor that does some domain-independent processing (e.g., identifying arithmetic formulas) as well as domain-specific processing (e.g., identifying domain-specific nominal phrases corresponding to an entity). In REQ1, the preprocessor identifies terms like Lower Desired Temperature, Upper Desired Temperature and Regulator Interface Failure as phrases with a special meaning in the FAA-Isolette domain, and converts each of these phrases to a single term (corresponding to an entity in this domain), resulting in the following preprocessed requirements sentence:

\textit{If the Status\_attribute of the Lower\_Desired\_Temperature or the Upper\_Desired\_Temperature equals Invalid, the Regulator\_Interface\_Failure shall be set to True.}

The output of the preprocessor is analyzed by a semantic processor that first does shallow semantic parsing of the preprocessed requirements text using the Stanford Typed Dependency Parser (STDP) \cite{14}. STDP outputs typed dependencies such as:

\texttt{nsubj(equals, Status\_attribute)}

Each typed dependency indicates a semantic relation between parsed terms, e.g., the typed dependency above indicates that \texttt{Status\_attribute} is a subject of the verb \texttt{equals}. The next stage of the semantic processor converts these typed dependencies to entries in a symbol table in an intermediate representation (IR) of the form:

\texttt{Upper\_Desired\_Temperature: Upper\_Desired\_Temperature |entity|unique|or:\{Lower\_Desired\_Temperature\}}

The IR table maps each symbol to its metadata and to its relationships with other symbols. In the example above, the IR table entry for \texttt{Upper\_Desired\_Temperature} shows that it is an entity, it is unique, and is connected to another entity \texttt{Lower\_Desired\_Temperature} via the relation \texttt{or}. A detailed description of the IR table is given in Section \text{III}.

The next part of the ARSENAL pipeline is the Formal Methods (FM) stage, which converts the IR table to a formal model (in the current ARSENAL prototype, we generate a SAL \cite{5} model). ARSENAL effectively converts multiple NL requirements sentences, which describe a system module, into a unified SAL model. Using this model, we can potentially generate a proof or counter-example for certain system properties of interest.
Note that ARSENAL is a general purpose methodology. We can plug in different modules to various parts of the workflow, e.g., any state-of-the-art typed dependency parser in the NLP stage or formal analysis tool in the FM stage. In this instance of the ARSENAL pipeline, we use STDP and SAL in the NLP stage and FM stage respectively (as described in the following sections), but other tools can also be plugged into these stages.

3 Natural Language Processing

The NLP stage takes requirements in natural language as input and generates the IR table as output. The different components of the NLP stage (shown in Figure 3) are described in detail in this section.

**Preprocessor:** The first part of the NLP stage is a preprocessor. It seeks to extract better (more meaningful) parses to aid the Stanford parser using both domain-specific and domain-independent transformations on the requirements sentence. An example domain-specific preprocessing task is identifying entity phrases like “Lower Desired Temperature” and converting them to the term `Lower_Desired_Temperature`. Domain-independent preprocessing tasks include identifying and transforming arithmetic expressions, so that NLP parsers like the Stanford parser can handle them better. For example, the preprocessor replaces the arithmetic expression “[x + 5]” by `ARITH_x_PLUS_5`. The parser then treats this as a single term, instead of trying to parse the five symbols in the arithmetic expression. The preprocessor also encodes complex phrases like “is greater than or equal to” into simpler terms like `dominates`. In later processing (e.g., in the Formal Methods stage), ARSENAL decodes the original arithmetic expressions from the corresponding encoded strings.

**Stanford Typed Dependency Parser:** The next part of the NLP stage is the application of the Stanford Typed Dependency Parser (STDP) to the preprocessed sentence. The syntactic parser in STDP parses the requirements text to get unique entities called mentions, while the dependency parser generates grammatical relations between the mentions. The final output is a set of typed dependency (TD) triples between extracted terms, which encode the grammatical relationship between mentions extracted from a sentence. For the example requirement `REQ1`, the full set of TDs generated by STDP are shown in Figure 4. The Stanford typed dependencies representation provides a simple description of the grammatical relationships in a sentence, which can be understood easily and used effectively to extract textual relations without requiring deep linguistic expertise.

Note that the suffix of each mention is a number indicating the word position of the mention in the sentence. The position index helps to uniquely identify the mention in the case of multiple occurrences in the sentence. The TDs output by STDP are triples of the form:
Fig. 3. NLP Stage of ARSENAL pipeline.
Fig. 4. STDP output for REQ1.

⟨relation name⟩ (⟨governor term⟩, ⟨dependent term⟩).

Each triple indicates a relation of type “relation name” between the governor and dependent terms. For example, let us consider the TD:

prep_of(Status_attribute-3, Lower_Desired_Temperature-6)

It indicates that the mention Status_attribute is related to Lower_Desired_Temperature via a prepositional connective of type “of”.

Semantic Processor: The semantic processor starts with the STDP output and creates a mention table, by selecting each mention in STDP output and creating a hash from each mention to all TDs it is involved in. Subsequently, it uses the mention table to create the Intermediate Representation (IR) table, using Metadata tags and TypeRules.

Metadata tags: Different types of metadata tags annotate the entries in the IR table, using the algorithmic flow outlined in Figure 5:

1. TermType: Whether term is of the type entity, event, numeric, or predicate.
2. NegatedOrNot: Whether term is logically negated.
3. QuantifierType: Unique, all, exists.
4. Relations/Attributes: Temporal or normal.
5. Lists: Corresponds to the connectives and, or, implied-by.

These tags are used to associate semantics with the entries in the IR table. The metatag annotations are similar to role annotations in automatic semantic...
role labeling. In this stage, ARSENAL uses WordNet to identify word stems, and to find out whether a term is a known noun (if not, the term is marked as a unique entity). ARSENAL can also use special domain-specific ontologies or glossaries in this stage to annotate the IR entries with a richer set of metadata tags that can be used in downstream processing.

**TypeRules**: TypeRules are domain-independent semantic rules used by the semantic processor to create the IR table. For example, `nsubjpass(V, N)` in the STDP output indicates that the noun phrase N is the syntactic subject of a passive clause with the root verb V. The TypeRule corresponding to the TD `nsubjpass(V, N)` indicates that N is a who/what argument of relation V in the output formula. TypeRules have the form:

```
TD(ARG1, ARG2) : ACTION(ARG3, ARG4)
```

For example:

```
prep_upon(?g, ?d) : implies(?d, ?g)
```

**Matching of Typed Dependencies with TypeRules**: Matching this rule with the TD: `prep_upon(entering-17, set-4)` produces a match with `?g = entering-17`, `?d = set-4`. The action to execute is then: `implies(set-4, entering-17)`.

There are a number of different types of actions, each with its own semantics. The `implies(?x, ?y)` action adds an entry `impliedBy:?x` to the IR entry for `?y`.

There are different kinds of TypeRules, e.g., for handling implications, conjunctions/disjunctions, universal/existential quantifiers, temporal attributes/relations, relation arguments, and events.

**Rules with complex patterns**: Some TypeRules are complex and have multiple TDS that match on the left-hand side. For example:

```
advcl(?g, ?d) & mark(?d, if) : implies(?d, ?g)
```

Here, the current TD being matched must match the first part of the rule, `advcl(?g, ?d)`, but any other TD from the parsed sentence can be used to match the rest of the rule, i.e., `mark(?d, if)`. Note that “if” here is not a variable: it denotes that a mention of the word if must appear in that position in a TD to produce a match.

TypeRules can also contain an additional condition on the left-hand side. For example:

```
nsubj(?g, ?d) & event(?g) : rel(agent, ?g, ?d)
```

Here, we have an additional check that whatever mention matches `?g` is marked as an event in the IR (in step 4 of the algorithm in Figure 5) — `rel(agent, ?g, ?d)` adds a relation `agent=?d` to the IR entry for `?g`.

---

1 A detailed list of TypeRules is available at: [http://www.csl.sri.com/~shalini/arsenal/](http://www.csl.sri.com/~shalini/arsenal/).
| **Input:** | Requirements text, WordNet, Domain-specific ontology, TypeRules, Pre-processing rules. |
|-----------|----------------------------------------------------------------------------------|
| **Output:** | Intermediate Representation (IR) table. |

1. Run the requirement text through Stanford Dependency Parser. This produces a set of Typed Dependencies (TDs) and part of speech (POS) tags for all the words in the sentence.
2. Create a Mention Table by selecting each MentionId in Stanford Parser output and creating a hash from MentionId to all typed dependencies (TDs) it is involved in.
3. Initialize IR (empty at beginning).
4. Populate the IR. Iterate over MentionIds in the Mention Table in sequence. For each MentionId:
   (a) Get the POS tag for the MentionId.
   (b) Set word to the stem of the word, using the WordNet stemmer, and add an IR entry for word.
   (c) If word is a math expression encoded by the math preprocessor, set its IR type to arithmetic.
   (d) Else if word is marked as a unique entity in the ontology, set its IR type to entity and its quantifier to unique.
   (e) Else if word is marked as predicate in the ontology, set its IR type to pred.
   (f) Else if word is a number, set its IR type to num.
   (g) Else if word has a noun POS tag, set its IR type to entity. In addition, if the word is not found in WordNet, set its quantifier to unique (as it is presumably a proper name).
   (h) Else if word has a verb POS tag, set its type to event.
5. Execute the type rules. For each MentionId in the Mention Table, for each TD associated with MentionId in the Mention Table, for each type rule TR (from top to bottom):
   (a) Match the type rule with the TD, producing TD.
   (b) If step 5(a) was successful, i.e., the the left-hand-side of TR matches TD, execute the right-hand-side of TD.

**Fig. 5.** Detailed algorithmic flow of IR generation.
A plug-and-play architecture like ARSENAL has certain flexibilities that enables it to give better performance by increasing accuracy of requirements processing. For example, if the shallow semantic NL parser generates multiple candidate parses of the requirements, ARSENAL can use semantic rules to select the best parse. ARSENAL can also correct inconsistencies between different NLP modules, if they give conflicting results.

Figure 6 shows the full IR table corresponding to REQ1.

| Status_attribute-3 | : Status_attribute | entity | unique |
| Lower_Desired_Temperature-6 | : Lower_Desired_Temperature | entity | unique |
| Upper_Desired_Temperature-9 | : Upper_Desired_Temperature | entity | unique |
| equals-10 | : equal | predicate | arg2=Invalid-11, arg1=Status_attribute-3 |
| Invalid-11 | : Invalid | entity |
| Regulator_Interface_Failure-14 | : Regulator_Interface_Failure | entity | unique |
| be-16 | : be | event |
| set-17 | : set | event | to=True-19, object=Regulator_Interface_Failure-14 |
| True-19 | : True | bool |

Fig. 6. IR table for REQ1.

4 Formal Analysis

Formal methods have proven effective at providing assurance and finding bugs in a multitude of domains such as electronic designs, software development, and protocol standardization. The *lingua franca* of formal methods is *logic*, which provides an unambiguous semantics for the (formal) language describing a design, and the means to reason with it. However, most people who experience or interact with computers today are “end-users” — they are not expert logicians, and their way of describing their usage to others is through natural language. In many cases, even domain experts, such as circuit designers, resort to natural language as the main medium for communicating their model of a design to consumers of the model (e.g., other designers, implementers and verification engineers), as evidenced by the large proportion of design documents still written in (stylized) natural language today. Hence, formal methods encapsulated in NLP layers can bring greater accessibility to the engineering discipline at the requirements stage, by liberating end-users from the burden of learning formal logic. In addition, ARSENAL helps even formal method experts with the ability
to create a first formal model quickly and automatically from NL descriptions. We next discuss the different parts of the FM stage (shown in Figure 7).

Fig. 7. FM Stage of ARSENAL pipeline.

4.1 Formula Generation

There are multiple output adapters in ARSENAL, which convert the IR table (with the semantic metadata annotations) to different output forms. The current ARSENAL implementation includes FOL and LTL adapters, which convert the
IR table to first-order logic (FOL) and linear temporal logic (LTL) formulas respectively. In this paper, we discuss the SAL model adapter, which converts the IR table to a SAL model. The SAL model represents a transition system whose semantics is given by a Kripke structure, which is a kind of nondeterministic automaton widely used in model checking [13].

ARSENAL uses translation rules to generate the SAL formulas from the IR table. The translation rules of the SAL adapter are shown in Figures 9 and 8. The translation from IR to a formula is given by the function $T_f$ which is defined inductively starting from the root entry, e.g., $T_f(e(\text{set-17}))$. Note that $e$ is a function that expands a mention to the IR table entry for that mention. The formula rules in $T_f$ invoke translation rules for terms, $T_{tl}$ and $T_{tr}$, for terms on the left-hand side (LHS) and right-hand side (RHS) of the formula respectively. The translation rules for terms are shown in Figure 8.

| Rule  | IR Entry | Translated Terms (Ttl/Ttr) |
|-------|----------|-----------------------------|
| VALUE | value | of=X | X (a variable) |
| DOT   | X | entity | of=Y | LHS: $T_{tl}(e(Y)).X$, RHS: $T_{tr}(e(Y)).X$ |
| NUM   | X | num | X (a number) |
| BOOL  | X | boolean | X (a boolean) |
| ARITH | X | arithmetic | X (an arithmetic expression) |
| NMOD  | X | entity | $[M_1, \ldots, M_n]$ | LHS: $M_1, \ldots, M_n, X$ (a variable), RHS: $M_1, \ldots, M_n, X$ (a constant) |

Fig. 8. Translation rules for terms.

Once the output formula is created from the IR table, we check if the formula includes all the mentions and relations in the IR table. ARSENAL first creates a graph from the IR table, in which each node is a mention entry in the IR table and each (directed) edge indicates if a mention is related to another via relations. It then runs Depth First Search (DFS) on this graph, starting at the root node of the IR table, “coloring” each node as visited as soon as DFS visits that node. When DFS terminates, it checks to see which nodes have not been covered. These are the nodes that are disconnected from the root node, and will hence not be processed by the translation algorithm (and hence not be part of the output formula). ARSENAL shows the uncolored nodes (i.e., unconverted mentions) to the end-user. This approach is very useful for debugging, since it helps to keep track of provenance and coverage of IR mentions in the output formula.

2 The output trace of applying the translation rules on the IR table is shown in [http://www.csl.sri.com/~shalini/arsenal/](http://www.csl.sri.com/~shalini/arsenal/)
Fig. 9. Translation rules for formulas.

4.2 SAL Model Generation

We continue to use REQ1 to illustrate how ARSENAL produces a SAL model from the generated formulas in the previous step. At its core, SAL is a language for specifying transition systems in a compositional way. A transition system is composed (synchronously or asynchronously) of modules, where each module consists of a state type, an invariant definition on this state type, an initialization condition on the state type, and a binary transition relation on the state type. The state type is defined by four pairwise disjoint types of variables: input, output, local and global — input and global variables are observed variables of a module, while output, global and local variables are controlled variables of a module. Note that the SAL model-checkers use Linear Temporal Logic (LTL), a modal temporal logic with modalities referring to time, as their underlying assertion language. This is an appropriate language for formally expressing requirements, since many requirements have temporal operators (e.g., eventually, always).

In order to unambiguously define a transition system, we need to additionally distinguish controlled variables that are state variables from wires (variables that do not directly define the state space). We need to know the type of any variable. Since we consider only self-contained modules in SAL, a variable can then belong to one of the following five categories: input, state only, state and output, output only, and wire. By differentiating state variables from wires, we can unambiguously map them to the corresponding section, namely DEFINITION, or TRANSITION (INITIALIZATION is distinguished by the use of the verb “initialize”).
We use the first to describe constraints over wires, and the others to describe evolutions of state variables. For example, if the `Regulator_Interface_Failure` variable in REQ1 was a state variable, then the SAL model generator would have produced the following transition instead.

**TRANSITION**

Upper_Desired_Temperature.Status_attribute = Invalid
OR Lower_Desired_Temperature.Status_attribute =
Invalid --> Regulator_Interface_Failure' = TRUE

The SAL model would hence be different, even though generated from the same sentence. Currently, ARSENAL requires the user to provide this additional information only after the NLP stage, thus keeping it separate from the model-independent part of the pipeline.

| Expression                  | Inference                       |
|-----------------------------|---------------------------------|
| $X \times Y, \times \in \{<,>,\leq,\geq\}$ | $X$ and $Y$ are numbers         |
| $X = \text{a number}$       | $X$ is a number                  |
| $X = \text{a named value } C$ | $X$ has enum type containing $C$|
| $X = Y$                     | $X$ and $Y$ have same type       |

Table 2. Rules for Gathering Type Evidence

During the model generation stage, ARSENAL gathers type evidences for each variable across all sentences and performs type inference by organizing them into equivalence classes. Further, in case of a type conflict, a warning is produced to indicate inconsistency in the NL sentences, thus helping the user to refine their requirements documentation at an early stage. Table 2 summarizes the rules ARSENAL currently implements for gathering type evidence.

4.3 Verification and Synthesis with LTL

In addition to specifying a transition system, sentences in NL may describe high-level requirements that the system must satisfy. Often times, this requirement can be precisely captured using temporal logic. In this paper, we use Linear Temporal Logic (LTL) \[27\], whose semantics can be interpreted over Kripke structures. We consider two problems that leverage LTL to reason about potential inconsistencies in a NL documentation, namely verification and synthesis. For verification, we use model checking to analyze whether a SAL model satisfies its LTL specification. In general, the application of model-checking tools involves a nontrivial step of creating a mathematical model of the system and translating the desired properties into a formal specification. ARSENAL automates this process with minimal user guidance. Given a model $M$ as a (labeled) transition system and a specification $\psi$ in LTL, both produced in the NLP stage of ARSENAL, we check if $M \models \psi$. When the model does not satisfy the specification, a negative answer (often in the form of a counterexample) is presented to the user.
as a certificate of how the system fails the specification. In this paper, we use SAL's bounded model checking [12] capability as the main workhorse for finding such inconsistencies.

ARSENAL also provides the flexibility to generate only specifications from the natural language requirements. Consistency means the specification is satisfiable, that is, whether a model exists for the specification. This problem is known as LTL satisfiability checking and it can be reduced to model checking [33]. Given an LTL formula $\psi$, it is satisfiable precisely when a universal model $M$ does not satisfy $\neg \psi$. A counterexample that points to the inconsistency is produced when $\psi$ is not satisfiable.

Given an LTL specification, it may also be possible to directly synthesize an implementation that satisfies the specification. Realizability, the decision problem of determining whether such an implementation exists, can be used to further inspect the requirements document for inconsistencies. If the specification is realizable, then a Moore machine can be extracted as an implementation that satisfies the specification. Thus, the benefit of LTL synthesis is a correct-by-construction process that can automatically generate an implementation from its specification. In general, LTL synthesis has high computational complexity [23]. However, it has been shown that a subclass of LTL, known as Generalized Reactivity (1) [GR(1)], is more amenable to synthesis [32] and is also expressive enough for specifying complex industrial designs [7].

Formal specification can precisely capture the desired properties of a design. However, it is common for formal specifications to be incomplete. Assumptions or constraints about the environment are particularly hard to capture. In this section, we describe a technique to generate candidate environment assumptions as suggestive solutions to make an unrealizable specification realizable. This is motivated by the fact that, in many scenarios, simply producing an unrealizable answer is not very useful for an user. Playing a two-player game according to the counterstrategy can be useful [6], but it requires considerable effort and time, not to mention the expertise in formal method that an user is assumed to have. To overcome this problem, we propose finding potentially missing assumptions about the environment, and then recommending them to the user as NL sentences in an interactive way. Throughout the process, the user remains oblivious to the underlying formal analysis performed, and can just reason with the NL feedback directly.

Given a LTL specification $\psi$ that is satisfiable but not realizable, the assumption mining problem is to find $\psi_a$ such that $\psi_a \rightarrow \psi$ is realizable. Our algorithm for computing $\psi_a$ follows the counterstrategy-guided approach in [6], which has shown to be able to generate useful and intuitive environment assumptions for digital circuits and robotic controllers. The algorithm is based on [25], and is summarized below.

**Counterstrategy-guided synthesis of environment assumptions.** Given the unrealizable specification $\psi$, the method first computes a counterstrategy.  

\footnote{Universal means $M$ contains all possible traces over the set of atomic propositions.}
The counterstrategy summarizes the next moves of the environment in response to the current output of the system, which will force a violation of the specification. The method then uses a template-based mining approach to find a specification \( \phi \) that is satisfied by the counterstrategy. \( \neg \phi \) is added as a new conjunct to \( \psi_a \) and \( \psi_a \land \psi_e \rightarrow \psi_s \) is checked for realizability again. By asserting the negation of \( \phi \) as an assumption to the original specification, the method effectively eliminates the moves by the environment that adhere to the counterstrategy. The process iterates until the resulting specification becomes realizable. At any step of the iteration, the user is asked to verify the correctness of the mined assumption. We present them as NL sentences, which we generate by mapping the boolean and temporal operators to English connectives.

In the next section, we demonstrate the usefulness of these techniques when they are incorporated into ARSENAL and applied to different corpora.

5 Evaluation

In this section, we present results on analyzing the FAA-Isolette corpus [21] and a portion of the TTEThernet requirements document [39] to demonstrate ARSENAL’s ability in handling complex NL sentences and different corpora. To better understand the degree of automation and robustness ARSENAL can achieve, we separately evaluate different stages of the ARSENAL pipeline.

5.1 NLP Stage: Evaluation

**Degree of Automation Metric** In this section, we report automation results of ARSENAL on both the FAA-Isolette (FAA) and the TTEThernet (TTE) corpora. Specifically, we evaluate the accuracy of ARSENAL’s NLP pipeline on translating each NL sentence into the corresponding logical formula automatically, without any manual correction. This metric measures the degree to which ARSENAL runs in an automated mode.

The results are summarized in Table 3. When evaluating accuracy, the correct outputs were given a score of 1.0, wrong outputs were given a score of 0.0, while partially correct results were given partial credit of 0.5. A translation was deemed partially correct if there was one error and incorrect if there was more than one error.

| Corpus | Total | Correct | Partial | Wrong | Degree of Automation |
|--------|-------|---------|---------|-------|----------------------|
| TTE    | 36    | 24      | 8       | 4     | 78%                  |
| FAA    | 42    | 39      | 2       | 1     | 95%                  |

Note that when ARSENAL fails to give the correct output automatically from the NLP stage while processing requirements, we correct the error...
Table 4. Results of perturbation test on ARSENAL.

| Perturbation Type     | TTEthernet domain (TTE) | FAA-Isolette domain (FAA) |
|-----------------------|-------------------------|---------------------------|
|                       | Total sentences | Perturbed sentences | Accuracy  | Total sentences | Perturbed sentences | Accuracy  |
| First (And → Or)     | 36            | 16                  | 81%       | 42            | N/A                 | N/A       |
| All (And → Or)       | 36            | 16                  | 87%       | 42            | 13                  | 92%       |
| All (Is → Is not)    | 36            | 17                  | 100%      | 42            | 13                  | 92%       |
| If A then B → B if A  | 36            | N/A                 | N/A       | 42            | 40                  | 65%       |

manually so that the input to the FM stage is correct. The following sentence is one of the sentences in FAA for which ARSENAL partially captures the logical semantics.

**REQ2:** If the Regulator Mode equals NORMAL, the Temp attribute of the Display Temperature shall be set to Temp attribute of the Current Temperature rounded to the nearest integer.

The logical formula output by ARSENAL is:

\[(\text{Regulator\_Mode} = \text{NORMAL} \implies \text{Display\_Temperature.Temp\_attribute} = \text{Current\_Temperature.Temp\_attribute})\]

The reason ARSENAL only handles the first half of the sentence correctly is that the phrase “rounded to the nearest integer” implies there is a function that can take a real/ floating-point number as input and produce its nearest integer as output. Currently, ARSENAL does not have support for arbitrary functions — in the future, we plan to incorporate more domain-specific knowledge and have built-in support for frequently occurring functions.

**Degree of Perturbation Metric** We define an evaluation criteria for measuring the robustness of ARSENAL, i.e., if perturbations/modifications are made to a requirements sentence using certain rewrite rules, whether ARSENAL can still generate the right output formula.

For the given dataset (e.g., FAA or TTE), we do perturbations to the requirements in that dataset using a transformational grammar, having operators that transform the text. The transformations in this grammar are based on allowable terminals in SAL, e.g., we can replace “always” by “eventually”, “or” by “and”, “is” by “is not”, etc. By applying these transformation operators to the FAA dataset, we can generate a “perturbed” dataset. This is similar in principle to generating test cases by fuzz testing [19].

Table 4 shows the results of our experiments on the FAA and TTE datasets. Note that total number of requirements was 42 in FAA and 36 in TTE. Out of the 36 requirements in TTE, the “And → Or” rewrite rule affected 16 requirements. We ran two types of “And → Or” transformations — in the first case, we modified only the first occurrence of “And” in the requirements sentences, while
in the second case we modified all occurrences of “And” in the sentences. When ARSENAL was run on these transformed requirements, thirteen of them gave output formulas that were correct w.r.t. the modified requirements sentence for the “First (And → Or)” rewrite rule, while fourteen of them gave output formulas that were correct for the “All (And → Or)” rewrite rule, giving an accuracy of 13/16 ≈ 81% and 14/16 ≈ 87% respectively. Similar numbers were calculated for other rules on FAA and TTE.

For FAA, only 2 sentences had more than one AND in them — so we did not run the “First (And→Or)” transformation on FAA, since the results for that would have been quite close to the “All (And→Or)” rule. For TTE, none of the 36 sentences had the “If A then B” structure. ARSENAL’s lower accuracy of 65% on the FAA domain for the “If A then B if A” rule was mainly caused by incorrect parse output from STDP on the perturbed sentences.

5.2 FM Stage: Evaluation

Verification Both the SAL model and theorems are automatically generated by ARSENAL from their NL descriptions. In this section, we demonstrate the usefulness of incorporating verification technologies in the ARSENAL pipeline to identify problems in NL documents.

TTEthernet. In the TTEthernet corpus, we consider the NL requirements that describe the synchronization state machine in TTEthernet. Figure 10 shows the diagram of this state machine (conditions for transitions are not shown). The machine starts at the ES_INTEGRATE state, and the ES_SYNC_ABS state indicates that the end-system has synchronized with other systems in the cluster.

This corpus contains 36 sentences.

ARSENAL can handle complex requirements sentences, generating the correct formula automatically. An example requirement describing part of the behavior in the ES_UNSYNC state is shown below.

REQ3: When an end system is in ES_UNSYNC state and receives a cold-start frame, it shall (a) transit to ES_FLOOD state, (b) set local_timer to es_cs_offset, (c) set local_clock to 0, (d) set local_integration_cycle to 0, and (e) set local_membership_comp to 0.

Notice that this sentence has a more complicated structure than REQ1 and includes five itemized actions. The part of the overall SAL model generated from REQ3 is shown in Figure 11. Observe that ARSENAL was able to infer that the end-system has an enumerated type (Type0) which contains named values ES_UNSYNC_state and ES_FLOOD_state. It was also able to set correctly

4 The requirements corpora for the FAA-Isolette and TTEthernet domains, and the corresponding SAL models generated by ARSENAL, are available at: http://www.csl.sri.com/~shalini/arsenal/
the type of local_integration_cycle and local_membership_comp to INTEGRER. In this example, the user asserted that all the five LOCAL variables are state variables. Hence, the actions over these variables were considered as state updates and mapped to the TRANSITION section.

A formal method expert was asked to review the model and found it was compatible with (and in fact, included more information than) a similar model that he handcrafted in [39]. We then asked one of the original creators of the TTEthernet documentation to provide a high-level specification that should be verified for this model. The sentence in English is given below, followed by the corresponding LTL theorem in SAL syntax generated by ARSENAL.

REQ4: If the end system is in ES_FLOOD state, it shall eventually not be in ES_FLOOD state.

THEOREM main |- G((end_system = ES_FLOOD_state =>
F(NOT(end_system = ES_FLOOD_state))));

We applied bounded model checking, a model checking technique that checks if the model satisfies the requirement within a bounded number of transitions, and found a counterexample. This counterexample reveals that if the environment keeps sending a coldstart_frame to this module, then local_timer, which maintains a count to timeout in the ES_FLOOD state, will keep resetting to 0 and thus preventing any transition out of the ES_FLOOD_state to occur.



tte_example : CONTEXT =
BEGIN
  Type1 : TYPE = {coldstart_frame};
  Type0 : TYPE = {ES_UNSSYNC_state, ES_FLOOD_state};
  Type2 : TYPE = {es_cs_offset};
main : MODULE =
BEGIN
  LOCAL local_integration_cycle : INTEGER
  LOCAL local_membership_comp : INTEGER
  LOCAL local_clock : INTEGER
  LOCAL end_system : Type0
  LOCAL local_timer : Type2
  INPUT in_channel : Type1
TRANSITION
  [ (end_system = ES_UNSSYNC_state AND in_channel = coldstart_frame) -->
    end_system' = ES_FLOOD_state;
    local_timer' = es_cs_offset;
    local_clock' = 0;
    local_integration_cycle' = 0;
    local_membership_comp' = 0 ]
END;
END

Fig. 11. SAL Model for REQ3.
This helped us identify the missing assumption (absent in the original documentation) that was needed for system verification. In fact, modular verification is one of the most difficult tasks in verification since it requires the precise specifications of the constraints on the environment. These constraints are often implicit and undocumented. In this case, the interaction of multiple end-systems should ensure that any end-system will not receive a \texttt{coldstart\_frame} infinitely often before it can exit the \texttt{ES\_FLOOD\_state}.

**FAA-Isolette.** Figure 12 (a) shows one of the finite state machines corresponding to the regulator function in the FAA document. In addition to NL requirements specified in this document, NL sentences were manually written for each transition in the FSMs (including the one shown here). An example sentence is shown below.

**REQ5:** \textit{If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.}

![Fig. 12. Original FSM (a) and Modified FSM (b) for Regulator.](image)

This experiment seeks to evaluate if ARSENAL can faithfully generate the transition system corresponding to the description (including the FSM) in the design document. Similar to the analysis performed for the TTEthernet example, verification was used to validate the generated FAA model. The corresponding SAL theorem generated by ARSENAL is shown below.

\begin{verbatim}
THEOREM main |- G((Regulator_Mode = FAILED => NOT (F(Regulator_Mode = NORMAL))));
\end{verbatim}
This theorem states that if the FSM is in the FAILED state, then it cannot go back to the NORMAL state (F in the theorem means eventually). Applying model checking, we verified that the generated SAL model satisfied the theorem. In general, for systems with a large state space like the TTEthernet example, it would be difficult to prove such theorems by manual inspection alone.

To demonstrate the applicability of ARSENAL in identifying inconsistencies in NL requirements, we added a sentence corresponding to the transition from the FAILED state to the INIT state, as shown in Figure 12 (b). For the modified model, ARSENAL quickly produced a counterexample that showed a path from the FAILED state to the NORMAL state, thus violating the aforementioned theorem. This shows that by integrating an NLP pipeline with formal analysis engines, ARSENAL can bring significant benefits to the requirements engineering community by detecting problems in NL requirements.

**Synthesis** We further applied the LTL realizability and synthesis analysis to the FAA-Isolette corpus. In this scenario, each sentence in the corpus is interpreted by ARSENAL as an end-to-end requirement on the target implementation. Hence, any non-input variable is considered as an output. Additionally, in order to work with the GR(1) synthesis tool RATSY, all the variables are converted to the bit-level.

Application of LTL [GR(1)] synthesis to these formulas produced an unrealizable result; no Moore machine existed to satisfy the formulas. At this point, the user can either interact with the tool (RATSY) to debug the specification or directly examine candidate assumptions generated by ARSENAL. The latter is more user-friendly since it assumes no knowledge of formal methods and other tools on the part of the user, and enables a user to directly refine the existing NL requirements. For the FAA-Isolette example, ARSENAL produces the following candidate assumption to make the specification realizable.

\[G ! (\text{Regulator\_Status}=1 \land \text{Regulator\_Init\_Timeout}=1)\]

To better understand why this assumption is necessary for ARSENAL to generate a SAL model from the set of sentences, observe that in Figure 12 the INIT state has two outgoing transitions, one to the NORMAL state and the other to the FAILED state. When both \text{Regulator\_Status} and \text{Regulator\_Init\_Timeout} are true, the state machine can nondeterministically choose to go to either of the states. Such behavior is not desirable in an actual implementation, which is supposed to be deterministic. Hence, the original specification is not realizable.

In this example, if the NL sentences describing the transitions were written differently, in a way that \text{Regulator\_Status}=1 and \text{Regulator\_Init\_Timeout}=1 were mutually exclusive, then the specification would be realizable and an implementation could also be generated automatically in Verilog. In general, the notion of unrealizability captures a much wider class of bugs than nondeterminism, and assumption mining helps to generate candidate fixes and facilitates interaction, especially with end-users.
6 Related Work

There is a rich and diverse body of research related to requirements engineering. The main advantages of ARSENAL over prior work are a less restrictive NL front-end, a more powerful FM analysis framework, and a stronger interaction between the NL and FM stages.

Kress-Gazit et al. [22], Smith et al. [37] and Shimizu et al. [36] propose grammars for representing requirements in controlled natural language. The natural language interfaces suggested in these papers are restrictive — ARSENAL can extract information automatically from a wider range of natural language styles.

Zowghi et al. [42], Gervasi et al. [17], Scott et al. [35], Xiao et al. [40], and Ding et al. [41] process constrained natural language text using NLP tools (e.g., CFG, Cico) and perform different types of checks (e.g., consistency, access control) on the requirements. Compared to these methods, ARSENAL uses more state-of-the-art NLP techniques that can be made domain-specific using resources like the domain-specific ontology, customized regex-based template matching in preprocessing, etc. ARSENAL also uses more advanced model checking tools (e.g., SAL), which can represent theorems and invariants in a much more expressive logic (e.g., LTL).

Behavior-driven development (BDD) is a way to give natural language specifications for the software testing phase. Drechsler et al. [15], Soeken et al. [38], and Harris [21] show different ways of translating high-level natural language requirements to tests in the BDD framework, which can then be used by BDD tools like Cucumber [1] or RSpec [2]. ARSENAL is more general than these approaches — instead of considering requirements specifications at the test phase, it considers NL requirement specifications that can be specified and verified in the design phase.

Ormandjieva et al. [30], QUARS [16], and Goguen [20] focus on assessing the quality of requirements documents — they do not create formal models from the requirements for downstream formal analysis (e.g., consistency checks) like ARSENAL. Malin et al. [26], Boyd [11], and Nikora et al. [29] do linguistic modeling and information extraction from requirements documents, but do not handle consistency checks or downstream formal methods analysis (e.g., using SAL) like ARSENAL. Attempto Controlled English (ACE) [34], RECORD [10] and T-RED [9] are user-oriented tools for requirements collection, reuse, and documentation. These are interactive tools requiring inputs from domain experts, and are not as automated as ARSENAL.

7 Conclusion & Future Work

The key accomplishments of ARSENAL are outlined in Figure 13.

In the future, we would place primary emphasis on making the ARSENAL framework more robust. We want to test ARSENAL on multiple other domains and datasets, and design more evaluation metrics like the ones discussed in this paper (e.g., automation and perturbation metrics) to evaluate the performance
1. Creating an NLP workflow for generating the IR:
   a) ARSENAL does semantic parsing using the combination of a type dependency parser, metatags and type rules,
   b) Resolves co-references and ambiguities in complex requirements sentences,
   c) Handles both domain-independent (e.g., for arithmetic expressions) and domain-specific pre-processing.

2. Creating a FM workflow to generate a complete SAL model from the IR:
   a) ARSENAL has multiple output generators, to generate the appropriate output (e.g., FOL formula, SAL model) for a domain.
   b) For SAL model generation from IR, ARSENAL:
      (i) Has principles to determine which formula should go to which part of the SAL model automatically.
      (ii) Automatically determines the SAL types, when the user only provides the input type categories.
      (iii) Guides the user to come up with the right formulation of the FM theorem, in natural language.
      (iv) Provides a debugging environment to the FM expert, helping to discover missing assumptions in the text.
   c) ARSENAL generates counter-examples, constructs proofs of properties, and uses realizability to check inconsistency of requirements

3. Connecting the NLP and FM stages to create an end-to-end pipeline for both FAA-Isolette and TTEthernet domains:
   a) ARSENAL was developed on the FAA domain and later ported to the more complex TTEthernet domain,
   b) Has a modular design that helped isolate the parts that needed to be changed (e.g., pre-processor) without modifying the core parts,
   c) Has many algorithms (e.g., type rules) that are quite robust to porting to a new domain.

4. Designing novel evaluation metrics to assess the performance of ARSENAL (detailed numbers in Section 5.1):
   a) ARSENAL is automated to a large degree (as measured by the degree of automation metric).
   b) It is robust to requirements perturbation (as measured by the degree of perturbation metric).

5. Saving significant development cycles of the end-user:
   a) ARSENAL generates the first-cut formal model from the voluminous requirements that have become standard in modern
      CPS systems, for which manually creating a model requires significant effort from the end-user,
   b) Needs the user input to be provided only once per application domain,
   c) Allows user training efforts in formal modeling to be minimized.

Fig. 13. Key accomplishments in ARSENAL.

of the ARSENAL pipeline as we improve it. We would also like to create benchmark datasets for evaluating different aspects of ARSENAL. Apart from SAL models, we have also experimented with other logical model outputs, e.g., first-order logic. We plan to continue generating other logical models, which could be suitable for other types of formal analysis. We would also like to explore the creation of richer system models, by composing models generated from separate requirements corpora.

The current ARSENAL system also has a statistics generator, which generates statistics about the distribution of entities, typed dependencies, etc. in a requirements corpus. We use the generator to identify important type rules (e.g., from dominant TDs) and important preprocessing rules (e.g., from dominant entities) for ARSENAL. We would like to use these statistics and apply machine learning to automatically customize different parts of ARSENAL (e.g., type rules, translation rules) for a given domain and requirements corpus.

In this paper, we only consider requirements in natural language text. In the future, we would also like to parse flow-charts, diagrams and unstructured tables in requirements, as well as handle events, intervals, and other complex constructs in requirements. We would like to generalize the ARSENAL pipeline to domains other than state machines, e.g., probabilistic systems.

References

1. Cucumber. http://cukes.info.
2. Rspec. http://en.wikipedia.org/wiki/RSpec.
3. Software defects - do late bugs really cost more? Slashdot, March 2010.
4. J. Babcock. Good requirements are more than just accurate. Practical Analyst: Practical Insight for Business Analysts and Project Professionals, December 2007.
5. S. Bensalem, V. Ganesh, Y. Lakhmech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saüdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of
6. R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seebor. Ratsy: A new requirements analysis tool with synthesis. In T. Touili, B. Cook, and P. Jackson, editors, Computer Aided Verification, volume 6174 of Lecture Notes in Computer Science, pages 425–429. Springer Berlin Heidelberg, 2010.

7. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weighhofer. Automatic hardware synthesis from specifications: A case study. In Design, Automation Test in Europe Conference Exhibition (DATE), pages 1–6, 2007.

8. B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. IEEE Transactions on Software Engineering, 14(10):1462–1477, October 1998.

9. T. Boman and K. Sigerud. Requirements elicitation and documentation using T-rex. Master’s thesis, University of Umeå, 1996.

10. J. Borstler. User-centered requirements engineering in record - an overview. In Proceedings of Nordic Workshop on Programming Environment Research (NWPER), 1996.

11. N. Boyd. Using natural language in software development. Journal of Object Oriented Programming, 11(9), 1999.

12. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. Form. Methods Syst. Des., 19(1):7–34, July 2001.

13. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. Model checking. MIT Press, Cambridge, MA, USA, 1999.

14. M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In In Proc. Intl. Conf. on language resources and evaluation (LREC), pages 449–454, 2006.

15. R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille. Completeness-driven development. In International Conference on Graph Transformation, 2012.

16. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An automatic quality evaluation for natural language requirements. In Proceedings of International Workshop on RE: Foundation for Software Quality, 2001.

17. V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. ACM Trans. Softw. Eng. Methodol., 14, July 2005.

18. D. Gildea and D. Jurafsky. Automatic labeling of semantic roles. Computational Linguistics, 28:245–288, 2001.

19. P. Goddefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2008.

20. J. A. Goguen. Formality and informality in requirements engineering. In Proceedings of International Conference on Requirements Engineering, 1996.

21. I. G. Harris. Extracting design information from natural language specifications. In Proceedings of the 49th Annual Design Automation Conference, pages 1256–1257, 2012.

22. H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating structured english to robot controllers. Advanced Robotics, pages 1343–1359, 2008.

23. O. Kupferman. Recent challenges and ideas in temporal synthesis. In SOFSEM 2012: Theory and Practice of Computer Science, volume 7147 of Lecture Notes in Computer Science, pages 88–98. Springer Berlin Heidelberg, 2012.
24. D. L. Lempia and S. P. Miller. Requirements engineering management handbook. Final Report DOT/FAA/AR-08/32, Federal Aviation Administration, June 2009.
25. W. Li, L. Dworkin, and S. Seshia. Mining assumptions for synthesis. In 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pages 43–50, 2011.
26. J. T. Malin. Automated tool and method for system safety analysis: 2009 progress report. Technical Report NASA/TM-2010-214800, NASA, 2009.
27. Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. 1992.
28. G. A. Miller. Wordnet: A lexical database for english. Communications of the ACM, 38:39–41, 1995.
29. A. Nikora and G. Balcom. Automated identification of LTL patterns in natural language requirements. In 20th International Symposium on Software Reliability Engineering (ISSRE), 2009.
30. O. Ormandjieva, L. Kosseim, and I. Hussain. Toward a text classification system for the quality assessment of software requirements written in natural language. In European Conference on Software Quality Assurance, 2007.
31. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, Computer-Aided Verification, CAV ’96, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
32. N. Piterman and A. Pnueli. Synthesis of reactive(1) designs. In In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), pages 364–380. Springer, 2006.
33. K. Rozier and M. Vardi. LTL satisfiability checking. In Model Checking Software, volume 4595 of Lecture Notes in Computer Science, pages 149–167. Springer Berlin Heidelberg, 2007.
34. R. Schwitter and N. E. Fuchs. Attempto controlled English (ACE) a seemingly informal bridgehead in formal territory. In JICSLP, 1996.
35. W. Scott, S. Cook, and J. Kasser. Development and application of context-free grammar for requirements. In System Engineering Test and Evaluation Conference (SETE), 2004.
36. K. Shimizu. Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs. PhD thesis, Department of Electrical Engineering, Stanford University, August 2002.
37. R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: An approach supporting property elucidation. In 24th International Conference on Software Engineering, 2002.
38. M. Soeken, R. Wille, and R. Drechsler. Assisted behavior driven development using natural language processing. In Objects, Models, Components, Patterns, volume 7304 of Lecture Notes in Computer Science, pages 269–287. 2012.
39. W. Steiner and B. Dutertre. SMT-based formal verification of a TTEthernet synchronization function. In FMICS, 2010.
40. X. Xiao, A. M. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In SIGSOFT FSE, page 12, 2012.
41. D. Z., J. M., and P. J. From textual use cases to service component models. In Proceedings of 3rd International Workshop on Principles of Engineering Service-Oriented Systems, pages 8–14, 2011.
42. D. Zowghi, V. Gervasi, and A. McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2001.