Deriving Formal Specifications from Natural Language Requirements Using Arsenal 2

Daniel Elenius, Eric Yeh, Stéphane Graham-Lengrand, Shalini Ghosh (Samsung Research), Patrick Lincoln, and Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

May 1, 2019

1Supported by DARPA Contract No. N66001-18-C-4002 and SRI International. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of SRI International, DARPA, or the U.S. Government.
Specifications for the requirements of systems, protocols, procedures, and standards are written in unstructured text or semi-structured formats for human consumption.

Arsenal is a semantic parser for translating informal specifications into analyzable formal models, e.g.,

- Finite state machines, e.g., models in Symbolic Analysis Laboratory (SAL) language
- Grammars
- Requests for Comments (RFCs)
- Data formats
- Protocols
- Real-time and Hybrid systems
- Standard Operating Procedures
- Pseudocode
- Code Documentation
- User Manuals and man pages
- Policies
- Contracts
Arsenal aims to assist stakeholders in developing precise and correct specifications that are humanly readable and amenable to formal modeling, analysis, and synthesis.

It does not process legacy requirements documents — this is an extremely hard problem due to lack of coherent organization and stylistic uniformity.

Our focus is on providing authors with an IDE for editing requirements together with analyzable (and synthesizable) formal content.
Some Motivating Examples

- **APIs:** The `fopen()` function opens the file whose name is the string pointed to by `path` and associates a stream with it.

- **Architecture:** Event-triggered, time-triggered, synchronous, asynchronous, publish/subscribe architectures.

- **Data Formats:** An IPv4 address is a 32-bit value. A UTF-8 character is encoded in one to four bytes.

- **Protocols:** TCP/IP, sendmail, FTP, HTTP, WebDAV, XML-RPC.

- **State Machines:** Services, Cars/planes/elevators/microwave ovens, middleware (TTEthernet)

- **Functional Properties:** The elevator must stop at the next requested floor.

- **Security Properties:** Comments for a paper are only accessible to reviewers.

- **Non-Functional Properties:** The system must recover from a device crash within two seconds.
The Challenge of Requirements

- Requirements (system behaviors that shall/must or shall/must not occur) are typically constructed using a mix of unstructured or loosely structured natural language, textual notation, and graphical representations like class diagrams, message sequence charts, and control flow graphs.\(^2\)

- Natural language (with tables and diagrams) is typically the only representation accessible to all the stakeholders for a system under design.

- Natural language is inherently incomplete, ambiguous, vague, imprecise, and sometimes inconsistent.

- Software requirements can be quite complex with both functional and non-functional properties, APIs, data representations, protocols, and properties.

- Natural language requirements description can rely on a great deal of contextual and domain knowledge.

\(^2\)IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications
Microsoft Open Specifications

From https://docs.microsoft.com/en-us/openspecs/

Natarajan Shankar

Arsenal 2
Arsenal 1 used a pipeline to generate a type dependency parse from the NL input.

General and domain ontologies are used to produce a intermediate logic representation (IR).

The IR is then used to generate formal representations in SRI’s SAL language.
Arsenal 1 was applied to a number of requirements documents: FAA Isolette (with LTL realizability and synthesis) and TTEthernet.
With Arsenal 2, we dispensed with the ontologies and intermediate representation to focus on end-to-end neural network models for semantic parsing exploiting recent advances in neural machine translation.

**Challenge:** Lack of large training corpora annotated with ground truth.

Existing documents are not organized for natural language processing: assume domain knowledge, lack uniformity and precision.

**Key insight:** Use synthetic training data from randomized generators (generative models) of language/ground-truth pairs.

**Caveat:** The quality of the training depends on the quality of the generator.

**Added Bonus:** The generator can be tuned for specific domains and natural language styles.
The Arsenal 2 approach is generic with respect to the target formal domain; current targets are SAL state machines and regular expression grammars.

Each formal domain is characterized by a Concrete Syntax Tree (CST) representation of formal syntax with compositional generation of (randomized) natural language renderings.

System architecture given by a collection of web services:
- WebIDE: For requirements developers and domain experts to interact with the system
- NL2CST Semantic Parser: Transforms natural language sentences to concrete syntax trees.
- CST2SAL: Transforms the concrete syntax tree
- CST2NL generates random CSTs and randomized readings of the CST in natural language.
The Arsenal 2 Pipeline

Arsenal processes sentences such as:

*If the Monitor Mode is set to INIT and the Monitor Status is set to True, the Monitor Mode shall be set to NORMAL.*

It translates them to SAL models with variables, types, definitions, initializations, transitions, and properties:

\[
\text{Monitor\_Mode} = \text{INIT} \land \text{Monitor\_Status} \\
\rightarrow \text{Monitor\_Mode}' = \text{NORMAL}
\]
The Arsenal 2 Pipeline

- **NL2CST** generates a JSON representation of the concrete syntax tree (CST) representing the target formal language.
- **CST2NL** generates (randomized) natural language from (randomly) generated CST.
- **CST2SAL** performs type inference to produce analyzable SAL models from the CST.
The pipeline can be reversed to produce CST and natural language starting from SAL.

- This is useful for producing training data sets.
- It also allows existing legacy modules to be integrated.
- Importantly, it allows the author to check that the input has been correctly parsed.
Generating Training Data

NL2CST

Designer

Formal Grammar for AST

Formal Grammar for CST, with NL pretty-printer & random generator biases

Generator

Arbitrary large synthesized data \((CST + NL\, printout)_{i=1..N}\)

Training

Seq2Seq model

text w/ entity placeholders

CST
Problem: Given an input sentence $x = x_1, \ldots, x_m$, construct output $y = y_1, \ldots, y_m$ by finding the maximal $y$ such that $p(y|x)$.

We want to model the probability $p(y|x)$ as the probability of the $j$'th token given the input and the preceding output: $\prod_{j=1}^{M} p(y_t|y_{<t}, x)$.

This probability is modeled by an encoder/decoder pair. The encoder (RNN) produces $e_i = E(x_i, e_{i-1}) \in \mathbb{R}^k$. 

Attention Models for Alignment

- Using the full sentence encoding $e$ to generate the output makes it difficult to handle long sentences.
- It also makes it difficult to realign the order of the output relative to the input as in deriving *zone economique europeenne* from *european economic zone*.
- The attention vector $\alpha_j$ is computed as a softmax of another encoding $a(s_{j-1}, h_i)$ which is correlated to the importance of input $i$ to output $j$ given the prior output encoding state $s_{j-1}$.
The user interacts with an IDE that interacts with a number of web services for preprocessing entities, parsing, and interacting with the SAL model checker.
The IDE can be used to enter the text.
A preprocessor replaces entities and numerals with place-holder tokens.
The NS2CST service parses the input sentences into CST form.
A SAL model is generated from the CST.
The SAL model can be model checked to generate success or a counterexample that can be explored.
Future Work

- More extensive end-to-end training for NL2CST – the CST2NL currently generates SAL models in a limited ontology.
- Handling structured and semi-structured input in the form of grammars, tables, and diagrams supported by natural language descriptions.
- Exploring variants of seq2seq such as seq2tree and Transformer.
- More features in the IDE to support debugging and revision.
- User studies.
- Generative models for other domains, and better generative models of existing domains.

Questions?

