EARS-CTRL: Building and Verifying Controllers for Dummies

Levi Lúcio

Salman Rahman

Saad Bin Abid

Alistair Mavin

In this paper we demonstrate the EARS-CTRL tool for automatically synthesizing and verifying controller software for embedded systems. EARS-CTRL has as starting point requirements written in (English) natural language, more the EARS (Easy Approach to Requirements Syntax) language invented and currently in use at Rolls-Royce. After expressing the requirements in English, the requirements engineer can produce the controller code at the press of a button. EARS-CTRL then provides facilities for verifying the generated controller by allowing step-by-step simulation or test-case generation using MathLab’s Simulink.

# Introduction

The ultimate goal in human-computer interaction is that humans can “explain” to computers their needs, using human-centered languages (ideally natural language). Computers will then perform the actions that will satisfy those needs. This trend can be observed to be on the rise with automated call-centers or personal assistants that, through voice commands, can automatically search for itineraries, restaurants, hotels and even perform online bookings.

In this paper we describe the tool for building and verifying software controllers. has as starting point the EARS (Easy Approach to Requirements Syntax) language which was created at Rolls-Royce to improve gathering natural language requirements . The language can be seen as “gently” constrained English. Through the use of a small number of patterns, or formatted sentences, EARS copes well with large requirements specifications for several domains . It has additionally been shown that using EARS is an effective way for reducing or, in some cases, eliminating many of the problems that plague requirements documents written using unconstrained natural language .

With the tool we attempt a step in the direction of controller construction by using natural language as a central specification artifact. After specifying the vocabulary to be used in the specification, a requirements engineer writes the specification using EARS templates. Then, at the press of a button the controller is synthesized. By using intuitive simulation and test case generation panels the requirements engineer can immediately experiment with and validate the controller by providing inputs and observing the resulting outputs.

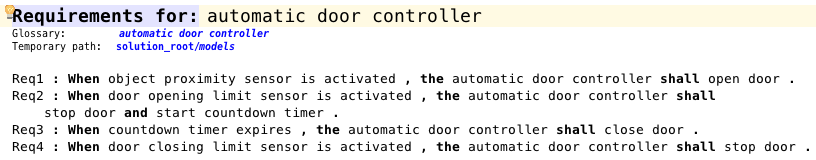
This paper is a follow-up of the article , where we describe a previous version of the tool. Our new contributions are a revision of the requirements language of which is fully aligned with the original EARS – by better covering the semantic gap between EARS and the underlying logical formalism for controller synthesis. We now also offer the possibilities of simulating requirements specifications and of generating test cases.

The tool as well as a set of examples other than the ones we present in this paper is freely available at a GitHub project at ….

# Highlights

## “Real” EARS

EARS was not originally built to describe requirements at a level where they can automatically be transposed into a real system. As such, an effort had to be made in order to overcome the semantic gap between on the one hand the structured but non-formal nature of EARS, and on the other hand the strictly formal nature of the Linear Temporal Logic (LTL) formalism needed by the automated synthesis mechanisms.



Requirements for a sliding door controller

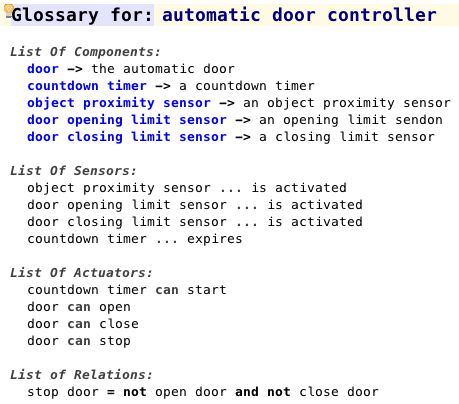
Figure [fig:ears\_reqs] illustrates a set of requirements for the software controller for a sliding door. By remaining as close as possible to the original EARS syntax our editor allows building requirements as correct English sentences that can easily be written and understood by humans. In fact, given the requirements stated in figure [fig:ears\_reqs], no additional explanations are necessary for a human to understand the behavior of the sliding door controller that should be generated.

**When** trigger **then the** system name **shall** response **until** trigger.

In  we have presented a previous version of which included templates that, although not part of the original EARS, had been introduced to simplify translation into LTL. An example of one such templates is presented in figure [fig:ears\_template\_while] – the **while** segment of the requirement is not standard EARS and has been removed in the version of we present in this paper. The work of briging the syntax of closer to “real” EARS while preserving a semantically meaniful translation into LTL was done with together with Alistair Mavin, the author of this paper who is also the main proponent of EARS . Our rationale is that, by remaining as faithful as possible to the original EARS syntax, we: 1) benefit from all the advantages of using EARS already investigated and described in the literature ; and 2) provide to Rolls-Royce and potentially other companies a tool that can immediately be used by engineers trained in the use of EARS.

## A Push-Button Approach

Controllers can be fully automatically synthesized from EARS requirements into controllers, at the push of a button. Note that in order to build requirements model illustrated in figure [fig:ears\_reqs] it is necessary to, as a first step, build a glossary for the controller. Such a glossary identifies the components of the system to be controlled. Each one of those components contains actuators and (possibly) sensors that will be used by the controller logic as, respectively, inputs from and outputs to the real system. The vocabulary defined in the glossary is appropriately proposed by the IDE to fill in the placeholders when a new requirement is built.



Glossary for sliding door controller

We present in figure [fig:ears\_glossary] the glossary for the automatic door controller specification. Note that the glossary allows defining invariants that will be taken into consideration during controller synthesis: for example in the last line of glossary in figure [fig:ears\_glossary] we state that if the motor is , it cannot be or .

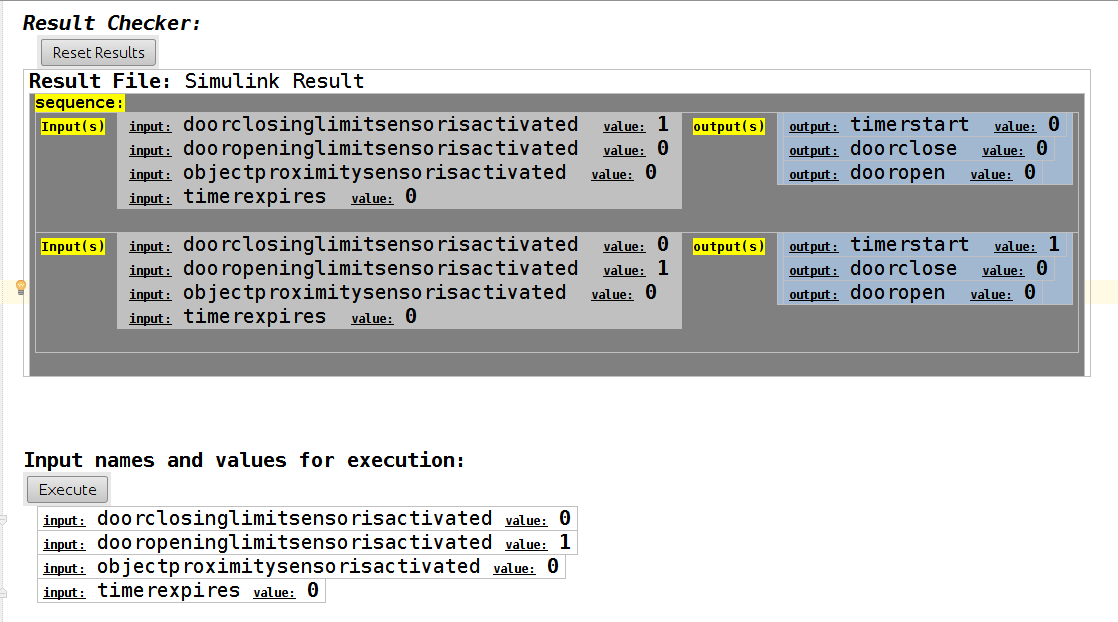
## Verification

### Well-Formedness by Construction

Well-formedness by construction is enforced in two different ways by : firstly, only valid EARS requirement patterns can be added to a requirements specification. When the requirements engineer picks an EARS template for her new requirement, the corresponding sentence is displayed by the IDE as a structure with placeholders. Such structures provide a first level of well-formedness, as only correctly formed EARS patterns can be added to the specification. Secondly, only valid sensors or actuators can be picked to fill in the placeholders in an EARS requirement.

Because well-formedness is enforced by construction, requirement specifications written in are always syntactically correct. The semantics of such specifications is then given by the   synthesizer in the form of a synchronous dataflow (SDF) diagram, which can display graphically. In the cases where synthesis is not possible the error code from the tool is lifted such that the requirements that prevent the controller from being generated are pointed out.

### Simulation



specification simulator

Once a controller has been synthesized from a set of EARS requirements, it becomes important to understand whether it behaves as expected. In order to do so we have used the Simulink engine  as a simulation back-end. In figure [fig:ears\_simulator] we display the panel that allows “playing” the controller by providing a sequence of inputs manually. Outputs are incrementally added to the panel as new inputs are provided by the requirements engineer. Note that, because controllers have internal state, the order in which the commands influences the controller’s output. A “Reset” button in the panel allows resetting the controller to its initial state.

### Generation of Test Cases

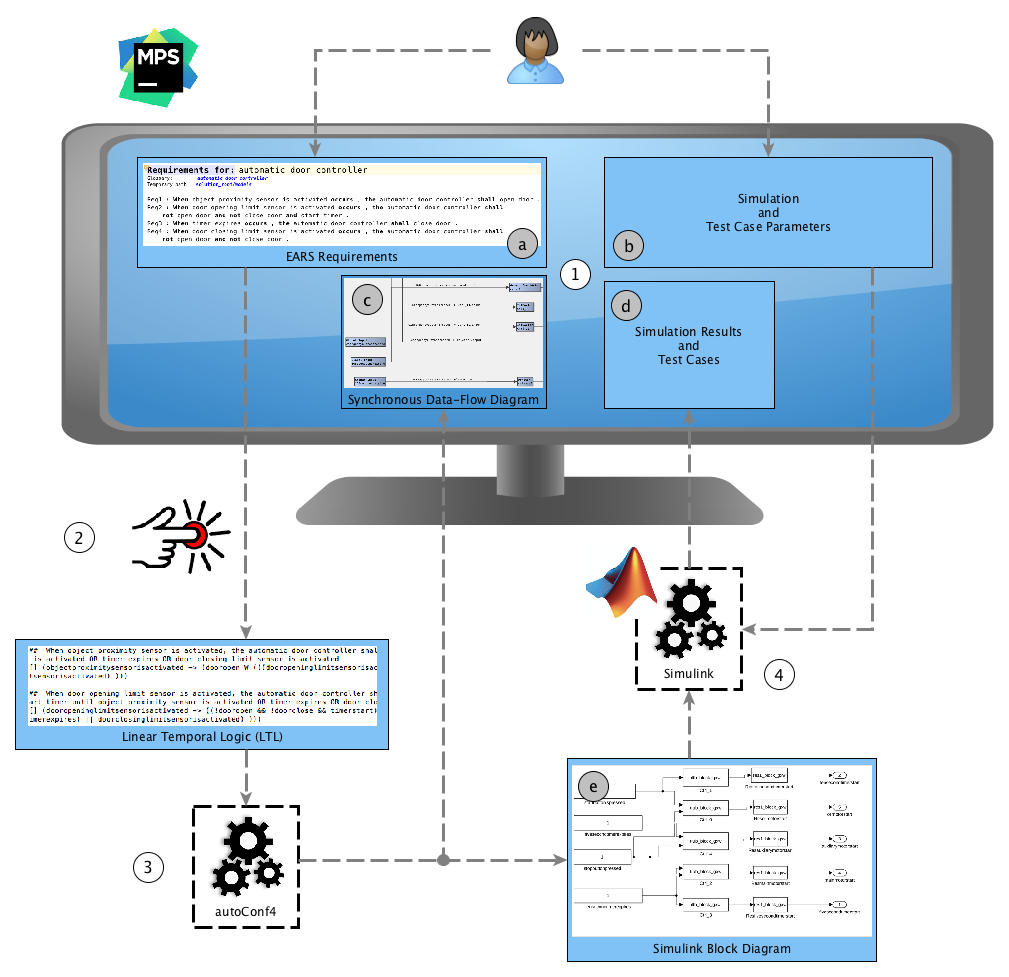
allows generating test cases directly from the EARS requirements. A test case consists of a sequence of pairs, where each input is a vector of sensor states and each output a vector of actuator states. Note that individual sensors and actuators can assume two states: “0”: off; “1”: on. Test case generation is parameterized by providing the following values:

* *Test Sequence Length:* defines the maximum length of the pair sequences to be generated.
* *Allow parallel inputs:* enables or disables the possibility of having more than one sensor being active for inputs in the test case.
* *Allow repeated inputs:* enables or disables having repeated inputs in the same test sequence. When enabled this parameter makes it such that an input vector cannot occur more than once in a test sequence – thus limiting the length of test sequences to the number of possible input vectors.

## Code Generation

Generation of C code can be achieved by running Simulink’s code generator on the Simulink model that is generated from the requirements EARS expressed in .

# Architecture



The Tool Chain

In figure [fig:ears\_ctrl\_toolchain] we depict the architecture of the tool. In the following paragraphs we will provide the reader a brief description of the main components of the tool’s architecture, how those components have been implemented as well as the artifacts they exchange. The paragraphs are numbered such that each descriptions can be matched with the process-related components of the tool depicted in figure [fig:ears\_ctrl\_toolchain]. Additional letter-labels are used in figure [fig:ears\_ctrl\_toolchain] to refer to data artifacts.

#### 1. Editors and Control Panels

The requirements editor, the glossary editor, the simulation and test generation control panel, the test generation control panel and the synchonous data-flow diagram visualizer (respectively noted (), (), () and () in figure [fig:ears\_ctrl\_toolchain]) have all been built as domain-specific languages (DSLs) in the Meta Programming System (MPS) tool. MPS is both a projectional editor and a domain-specific language workbench. Domain-specific languages in MPS are composed of an abstract syntax (also known as meta-model) and a concrete syntax. The concrete syntax allows displaying and/or editing the information present in a model (as depicted for instance in figures [fig:ears\_reqs], [fig:ears\_glossary] and [fig:ears\_simulator]). Note that because MPS is a projectional editor, the abstract syntax is directly edited which avoids an explicit or implicit intermediate step where the concrete syntax is parsed. A direct consequence of this is for example the fact that when a component’s name is updated an glossary, that update will immediately be reflected in any requirements that refer to that component name. This automatic update is an off-the-shelf feature of any editor defined using MPS and is an easy way to guarantee that references between the several parts of an editor always remain consistent.

#### 2. From EARS to Lineal Temporal Logic

Let us consider the requirement which is part of the specification of the sliding doors controller in figure [fig:ears\_reqs]:

**When** *object proximity sensor is activated* **then the** *automatic door controller* **shall** *open door*.

This requirement, taken in isolation, translates into the following LTL formula:

which, if one takes into consideration the semantics of the operator as “implies”, is the expected logical meaning of . All EARS templates, when taken in isolation, can be directly translated into LTL and propositional logic operators in such a straightforward manner. If, however, one translates the whole set of requirements for the automatic door in [fig:ears\_reqs] into LTL, the result for will be as follows:

This is due to the fact that the requirements specify behaviors that are interwined during execution. For example, from in figure [fig:ears\_reqs] we know that if the is activated, the doors will open. We also know from that, when the sensor is activated, the doors will stop. Without additional information, the synthesis tool identifies a contradition in these two requirements since, if the two sensors are activated during the same execution, the doors will logically simultaneously open and close. In order to avoid such contradictions it becomes necessary to establish a temporal dependency between the behaviors specified by the requirements. In order to achieve this our tool performs a static analysis of the requirements in order to identify such dependencies and to add this information to the generated LTL specification. This additional contextual information in the generated LTL is clear from the second translation above: the door will only open, *until* (the “**W**” operator) the door sensor is activated, *or* other events stated in related requirements occur.

#### 3. Synthesizing a Controller using

Controller synthesis is achieved via ’s Java API. The LTL specification obtained as explained in section [sec:ears\_LTL] is passed into the synthesizer which returns a synchronous data-flow (SDF) diagram as an Java object instance. The SDF diagram is then parsed and rebuilt as a visual model which is an instance of the synchonous data-flow diagram visualizer DSL (identified by label () in figure [fig:ears\_reqs]). Such a visual model provides the requirements engineer with a graphical and technical view of the synthesized controller as a set of blocks and wires which can be used as a debugging artifact.

#### 4. Simulation and Test Generation using Simulink

The SDF diagram obtained from the consists, for short, of a set of synchronized blocks that perform arithmetic, logical or other functions on input signals and return the result on output signals. The controller’s inputs and outputs are also themselves represented as blocks. The fashion in which blocks are synchronized is declared by connecting those blocks’ inputs and outputs via wires. In order to simulate specifications we have built a translator from such data-flow diagrams onto Simulink models. Given that the SDF formalism is very similar to the Simulink formalism, the structural translation is essentially one-to-one. However, only a subset of all blocks present in the SDF specifications that are produced by is available off-the-shelf in Simulink. As such, a number of Simulink blocks had to be built by us to accommodate the semantics of SDF specifications. In particular….

The Simulink model is generated by as a Mathlab simulink script that builds the model programmatically (label in figure [fig:ears\_reqs]). Communication with simulink from the IDE for simulation and test case generation is also programatically achieved through the use of the Java API.

# Related Work

# Limitations and Future Work

Due to the boolean representation of sensors and actuators in it is currently not possible for to express or analyze states of the system that involve numerical data. This naturally limits our approach to controllers for systems where sensors or actuators can be represented using boolean types. For instance, expressing a state such as “the throttle is pushed to 1/4 of its capacity” using would at best involve modelling four different sensors in order to partition the input space of a single sensor. Such an approach may prove to be infeasible and/or impractical in the real world and other code synthesizers for might be considered in the future.

Future work will concentrate on exploring the usability of for larger case studies. In particular we expect to continue the collaboration with Rolls-Royce in order obtain real requirements such that the synthesis and verification mechanisms explained in this paper can be put to the test in the field. It of particular interest to understand not only how based-synthesis will scale, but also to which extent the verification and debug mechanisms we propose are helpful in practice.