EARS-CTRL: Building and Validating Controllers for Dummies

Levi Lúcio

Salman Rahman

Saad Bin Abid

Alistair Mavin

In this paper we present the EARS-CTRL tool for synthesizing and validating controller software for embedded systems. EARS-CTRL has as starting point requirements written in (English) natural language, more specifically in 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 validating the generated controller that allow 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 would then automatically perform the actions that 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 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. EARS was created at Rolls-Royce to improve the gathering of natural language requirements  and can be seen as “gently” constrained English. Through the use of a small number of patterns, or formatted sentences, EARS copes well with large specifications of requirements for several domains . It has additionally been shown that using EARS is an effective way for reducing or even eliminating many of the problems that plague requirements documents written using unconstrained natural language .

With the tool we make 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 simulation and test case generation panels the requirements engineer can the immediately experiment with and validate the controller by providing inputs and observing the resulting outputs.

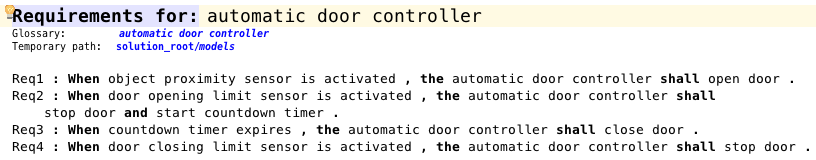
This paper is a follow-up of a previous article . Our new contributions are a revision of the requirements language of which is now fully aligned with the original EARS. We do so by improving the coverage of the semantic gap between EARS and the underlying logical formalism used by the controller synthesizer. We now also offer the possibilities of simulating requirements specifications as well as 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. 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. In particular we had introduced the possibility of adding an *until* segment at the end of requirements which is not standard EARS and which we have removed in the current version of . 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.

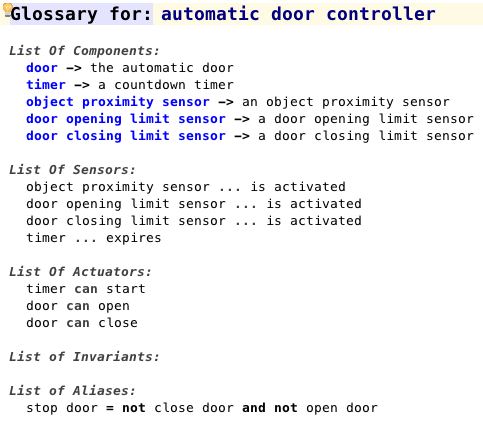
## A Push-Button Approach

can synthesize software controllers directly from EARS requirements, at the push of a button. Such syntheses are produced by the   synthesizer in the form of a synchronous dataflow (SDF) diagram, which our tool 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.

## Validation

### 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.

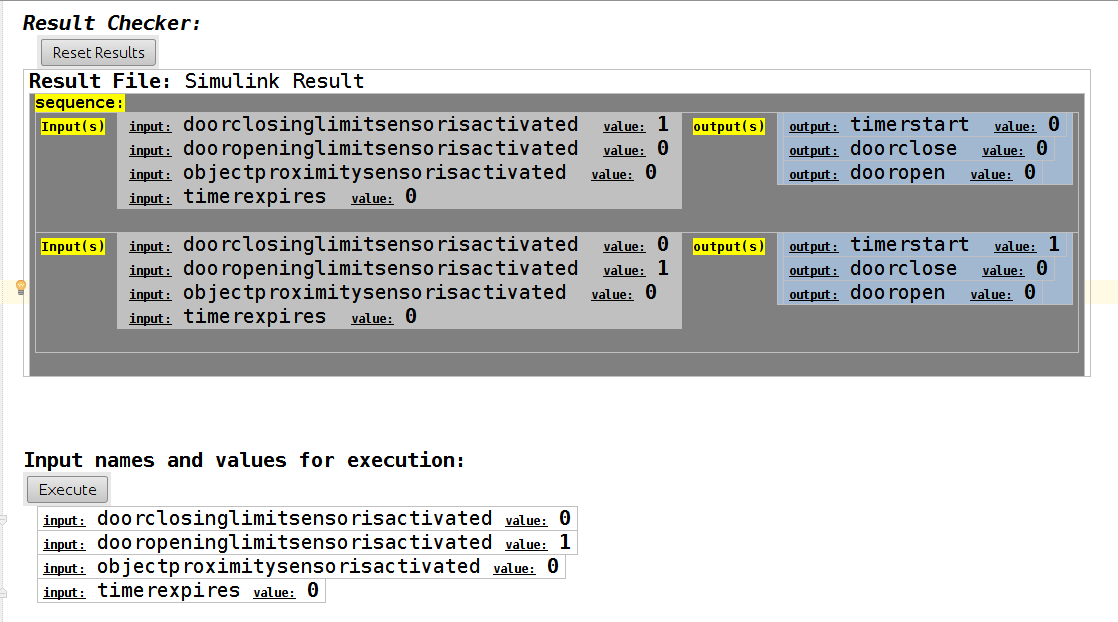


Glossary for sliding door controller

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. An example of such a glossary is depicted in figure [fig:ears\_glossary]. The first section of an 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. To allow for more ease of writing when building requirements, aliases for logical expressions involving sensors or actuators can also be defined in the glossary. More advanced users also have the possibility of defining invariant relationships between sensors or actuators. The vocabulary defined in the glossary is proposed to the requirements engineer by the IDE in order to fill in the placeholders of an EARS template when a new requirement being written.

Because well-formedness is enforced by construction, requirement specifications written in are always syntactically correct.

### 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: or . Test case generation is configured by three parameters:

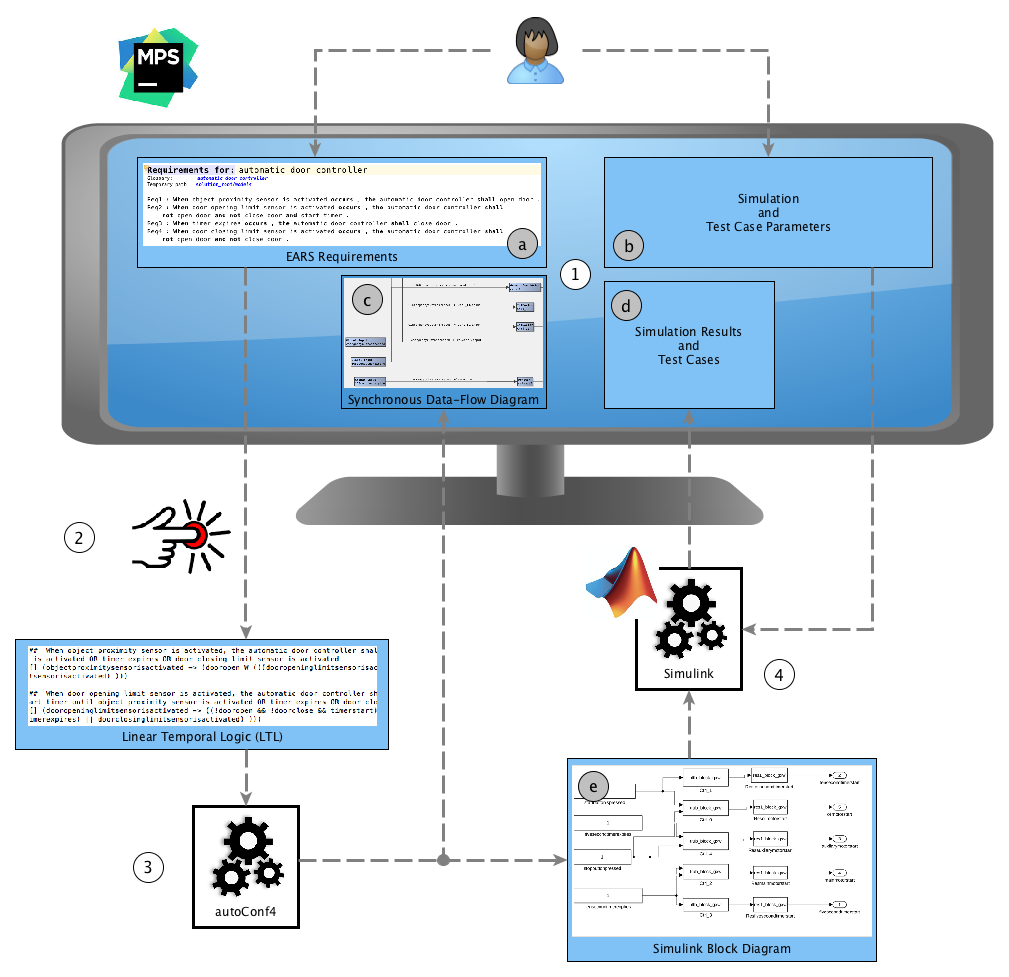
* *Maximum test case length:* defines the maximum length of the pair sequences to be generated.
* *Allow parallel inputs:* enables or disables the possibility of having more than one sensor being active for inputs in the test case.
* *Allow repeated inputs:* enables or disables having repeated inputs 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.

Test cases generated by can serve two purposes: on the one hand they are traces of execution of the synthesized controller and can be used to make sure that controller behaves as expected; on the other hand one may consider that the synthesizer is not trusted to generate controllers used in production: in this case the synthesized controller can behave as an oracle to generate test cases for a production controller implemented using alternative means.

## Code Generation

Although it is not possible to generate C code for the controller directly from , this can be achieved by directly running Simulink’s code generator on the Simulink model obtained from an requirements specification.

# Architecture



The Tool Chain

In figure [fig:ears\_ctrl\_toolchain] we depict the architecture of the tool. In the following paragraphs we will provide to 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 description can be matched to the process-related components of the tool depicted in figure [fig:ears\_ctrl\_toolchain]. 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 panell 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 the 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 change will immediately be reflected in any requirements that refer to that component name. This automated change is an off-the-shelf feature of any editor defined using MPS and 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 to 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 in such a straightforward manner. 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. 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* (denoted by the “**W**” operator) the door sensor is activated, *or* other events stated in door-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 a 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 as output signals. Note that 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 SDF diagrams onto Simulink models. Given that the SDF formalism is very similar to the Simulink formalism, the structural translation is 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 stateful Simulink blocks had to be built by us to mimic the semantics of some of the blocks present in SDF specifications.

The Simulink model is generated by as a Mathlab simulink script that programmatically builds the model (label () in figure [fig:ears\_reqs]). Communication between the IDE and Simulink is achieved programmatically through the Java API.

# Related Work

Given the recent fast-paced development of Artificial Intelligence relying on increasigly powerful hardware, a number of projects have devoted effort to the generation of controllers from requirements. The ARSENAL project  has as starting point specifications written in arbitrary natural language and uses the GR-1  synthesizer for automatically building controllers. In  the authors also use the GR-1 synthesizer to automatically build robot controllers. The work of Yan et. al.  takes as inputs full LTL specifications and includes features such as the use of dictionaries for automatically derive relations between terms, or guessing the I/O partitioning that allow detecting inconsistencies in the specifications. The commercial argosym STIMULUS tool , while not based on AI algorithms from controller synthesis, is a commercial platform that allows specifying requirements in a formal language using a close-to natural language syntax. Requirements expressed in STIMULUS can be simulated and test cases can also be directly generated from the requirements.

Our approach differs from the GR-1-based projects mentioned above in the sense that we do not aim at applying pure natural language parsing to arbitrary requirements. Using EARS allows us to provide the readability of the English language while gently contraining it to fit the domain of requirements gathering. Also, rather than using the full expressiveness of LTL, we have restricted our approach to the subset of LTL which is handled by the tool. Using this subset it is possible to directly generate controllers as SDF diagrams, which are easy to inspect and to simulate. Tools that are based on GR-1 or bounded synthesis  typically produce controllers as BDD or explicit state machine structures that can be very large and difficult to inspect or simulate.

Regarding the STIMULUS tool, our approach was conceptually though of starting from an opposite direction – while STIMULUS essencially uses as central formalism state machines wrapped by a syntactic-sugar English-like specification language, uses a constrained version of the English language. We have purposefully placed EARS at the center on our tool – the goal has been to adapt the subset of LTL used by to EARS and to remain unbiased towards the formalisms “under the hood”. Unlike in our work, STIMULUS relies on the state machines underlying the requirements to allow simulation as in fact the approach’s goal is to verify requirements and not to synthesize usable controllers.

# 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.