EARS-CTRL: Generating Controllers for Dummies

Levi Lúcio
Model-Based Software Engineering
fortiss GmbH
Munich, Germany
lucio@fortiss.org

Salman Rahman
Technische Universität München
Munich, Germany
salman.rahman@tum.de

Saad Bin Abid
Model-Based Software Engineering
fortiss GmbH
Munich, Germany
abid@fortiss.org

Alistair Mavin
Rolls-Royce
Derby, UK
alistair.mavin@rolls-royce.com

Abstract—**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 at Rolls-Royce and currently in use at numerous organizations around the world. 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 MATLAB Simulink.**

Index Terms—**Software Controllers, Natural Language, Code Synthesis, MathLab Simulink**

I. INTRODUCTION

In this paper we describe the EARS-CTRL tool for building and verifying discrete-event software controllers. EARS-CTRL has as starting point the EARS (Easy Approach to Requirements Syntax) language. EARS was created at Rolls-Royce to improve the expression of natural language requirements [12] and can be seen as a way to "gently" constrain English. The application of EARS produces requirements in a small number of patterns. EARS copes well with large specifications of requirements for several domains [10], [11]. EARS is also an effective way of reducing many of the problems that plague requirements documents written using unconstrained natural language [12].

With the EARS-CTRL tool we make a step in the direction of controller construction using natural language as the central specification. 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. Simulation and test case generation panels allow the requirements engineer to immediately experiment with and validate the controller.

This paper builds on a previous article [9]. Our new contributions are a revision of the requirements language of EARS-CTRL, which is now has exactly the same syntax as the EARS language presented in [12]. 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 EARS-CTRL tool is freely available as a GitHub project at [1].

II. HIGHLIGHTS

A. "Real" EARS

EARS was not originally built to describe requirements at a level where they can automatically be transposed into a computer application. As such, an effort had to be made in order to overcome the semantic gap between the structured but non-formal nature of EARS and the strictly formal nature of the Linear Temporal Logic (LTL) formalism needed by the automated synthesis mechanisms. In particular, our work uses the GXW subset of LTL as that subset is supported by the autoCode4 tool we use for synthesizing controllers.

Figure 1 illustrates a set of EARS-CTRL 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 1, no additional explanations are necessary for a human to understand the behavior of the sliding door controller that should be generated. In [9] we have presented a previous version of EARS-CTRL 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 tool. The work of matching the syntax of EARS-CTRL closer to "real" EARS while preserving a semantically meaningful translation into LTL was done together with Alistair Mavin, the co-author of this paper who is also the lead author of EARS [12]. 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 [11], [12]; and 2) provide to Rolls-Royce and

Fig. 1: EARS-CTRL Requirements for a sliding door controller

potentially other companies a tool that can immediately be used by engineers trained in the use of EARS.

B. A Push-Button Approach

EARS-CTRL can synthesize software controllers directly from EARS requirements, at the push of a button. Such syntheses are produced by the autoCode4 [5] tool in the form of a synchronous dataflow (SDF) diagram which our tool can display graphically.

C. Validation

1) Well-Formedness by Construction: In order to build requirements model illustrated in figure 1 it is necessary to firstly build a glossary for the controller. Glossaries are specific to EARS-CTRL and are not part of the EARS language described in [12]. An EARS-CTRL 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 outputs to and inputs from the real system. The vocabulary defined in the glossary is proposed to the requirements engineer by the EARS-CTRL IDE in order to fill in the placeholders of an EARS template when a new requirement is being written. Because well-formedness is enforced by construction, requirement specifications written in EARS-CTRL are always syntactically correct. 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 [2] as a simulation back-end. In figure 2 we display the EARS-CTRL 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 the simulation panel dynamically displays the sensors of the controller being simulated, as can be seen in figure 2 for the sliding doors example.

2) Generation of Test Cases: EARS-CTRL allows generating test cases directly from the EARS requirements. A test case consists of a sequence of $\langle input, output \rangle$ 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: ON or OFF. Test case generation is configured by three parameters:

- *Maximum test case length:* defines the maximum length of the $\langle input, output \rangle$ 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 a test.

Test cases generated by EARS-CTRL can serve two purposes: firstly, they are traces of execution of the synthesized controller and can be used as witnesses of correct/incorrect behavior; secondly, it may be that the synthesizer is not trusted for generating controllers used in production: in this case the synthesized controller can behave as an oracle to generate test cases for a controller implemented using alternative means.

D. Code Generation

Although it is not possible to generate C code for the controller directly from EARS-CTRL, this can be achieved by directly running Simulink's code generator on the Simulink model obtained from an EARS-CTRL requirements specification. This possibility is particularly important for our next steps in our collaboration with Rolls-Royce as we wish to experiment with running the synthesized controllers in real or simulated execution environments.

III. ARCHITECTURE

In figure 3 we depict the architecture of the EARS-CTRL tool, its main components and the artifacts those components they exchange. The paragraphs below are numbered such that each description can be matched to the process-related components of the tool depicted in figure 3. Letter-labels are used in figure 3 to refer to data artifacts.

*a) **Editors and Control Panels:*** The requirements editor, the glossary editor, the simulation and test generation control panels and the synchronous data-flow diagram visualizer (respectively noted (a), (b), (c) and (d) in figure 3) have all been built as domain-specific languages (DSLs) in the Meta Programming System (MPS) tool [4]. 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 1 and 2. Note that because MPS is a projectional editor, the abstract syntax is directly edited which avoids the explicit or implicit intermediate step where
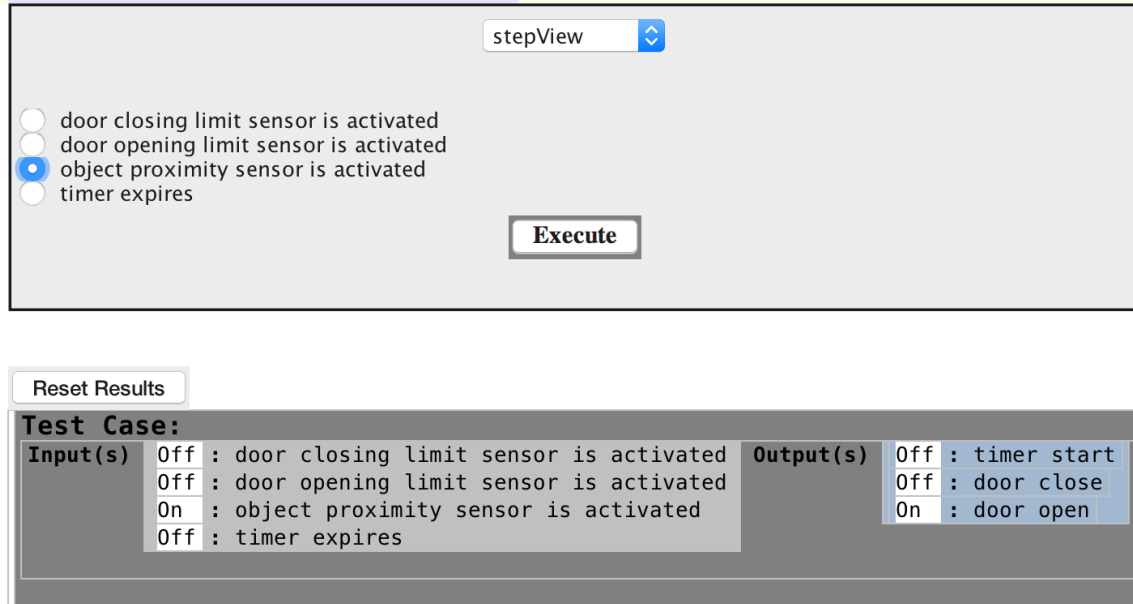
Fig. 2: EARS-CTRL specification simulator

the concrete syntax is parsed. A consequence of this is for example the fact that when a component's name is updated an EARS-CTRL glossary, that change will immediately be reflected in any requirements that refer to that component name.

b) From EARS to Lineal Temporal Logic: Let us consider the requirement Req1 which is part of the specification of the sliding doors controller in figure 1:

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:

$$[](object proximity sensor is activated \rightarrow door open)$$

which, if one takes into consideration the semantics of the $\rightarrow$ operator as "implies", is the expected logical meaning of Req1. All EARS templates, when taken in isolation, can be directly translated into LTL and propositional logic in such a straightforward manner. However, when one translates the whole set of requirements for the automatic door in 1 into LTL, the result for Req1 will be as follows:

$$[](object proximity sensor is activated \rightarrow (door open \, W$$
$$(door opening limit sensor is activated \lor timer expires$$
$$\lor door closing limit sensor is activated)))$$

This is due to the fact that the requirements specify behaviors that are interwined during execution. For example, from Req1 in figure 1 we know that if the object proximity sensor is activated, the doors will open. We also know

from Req2 that, when the opening limit reached sensor is activated, the doors will stop. Without additional information, the autoCode4 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 opening limit reached sensor is activated, *or* other events stated in door-related requirements occur.

c) Synthesizing a Controller using **autoCode4**: Controller synthesis is achieved via autoCode4's Java API. The LTL specification obtained as explained in section III-0b is passed into the synthesizer, which returns a synchronous data-flow (SDF) diagram as a Java object instance. The SDF diagram is then rebuilt as a visual model which is an instance of the synchonous data-flow diagram visualizer DSL (identified by label (e) in figure 1). 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.

d) Simulation and Test Generation using Simulink: The SDF diagram obtained from autoCode4 consists, for short, of a set of synchronized blocks that perform arithmetic,
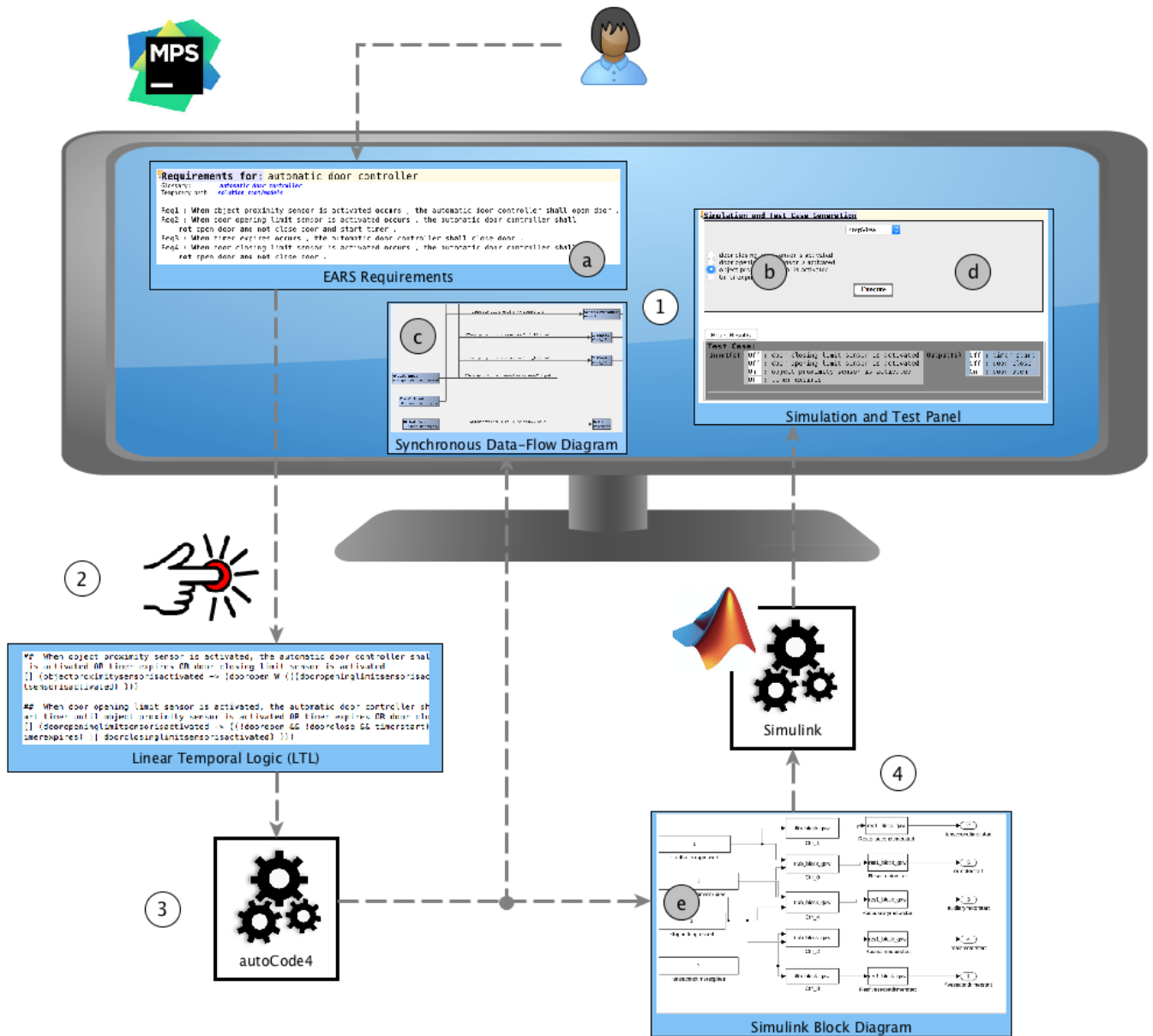
Fig. 3: The EARS-CTRL Tool Chain

logical or other functions on input signals and return the result as 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 EARS-CTRL specifications we have built a translator from such SDF diagrams onto Simulink models (label (e) in figure 1). 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 autoCode4 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 EARS-CTRL IDE communicates with Simulink via the matlabcontrol [3] Java API.

IV. RELATED WORK

Given the recent fast-paced development of Artificial Intelligence relying on increasingly powerful hardware, a number of projects have devoted effort to the generation of controllers from requirements. The ARSENAL project [6] has as starting point specifications written in arbitrary natural language and uses the GR-1 [13] synthesizer for automatically building controllers. In [8] the authors also use the GR-1 synthesizer to automatically build robot controllers. The work of Yan et.

al. [15] 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 [7], 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 constraining it to fit the domain of expressing requirements. Also, rather than using the full expressiveness of LTL, we have restricted our approach to the GXW subset of LTL which is handled by the autoCode4 tool. We then directly generate controllers as SDF diagrams, which are easy to inspect and to simulate. GR-1- or bounded synthesis [14]-based tools 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 essentially uses as central formalism state machines wrapped by a syntactic-sugar English-like specification language, EARS-CTRL 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 autoCode4 to EARS and to remain unbiased towards the formalisms "under the hood". 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.

REFERENCES

[1] Ears-ctrl github project. https://github.com/saadbinabid1/EARS-CTRLReqAnalysis/.
[2] Matlab simulink. https://de.mathworks.com/products/simulink.html/.
[3] matlabcontrol java api. https://code.google.com/archive/p/matlabcontrol/.
[4] Meta Programming System. https://www.jetbrains.com/mps/.
[5] C.-H. Cheng, E. Lee, and H. Ruess. autoCode4: Structural Reactive Synthesis. In TACAS'17. Tool available at: http://autocode4.sourceforge.net.
[6] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. ARSENAL: Automatic Requirements Specification Extraction from NL. In *NFM*, 2016.
[7] B. Jeannet and F. Gaucher. Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.
[8] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating Structured English to Robot Controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.
[9] L. Lúcio, S. Rahman, C. Cheng, and A. Mavin. Just formal enough? automated analysis of EARS requirements. In *NASA Formal Methods - 9th Intl. Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, 2017.
[10] A. Mavin and P. Wilkinson. Big Ears (The Return of "Easy Approach to Requirements Engineering"). In *RE*. IEEE, 2010.
[11] A. Mavin, P. Wilkinson, S. Gregory, and E. Uusitalo. Listens Learned (8 Lessons Learned Applying EARS). In *RE*. IEEE, 2016.
[12] A. Mavin, P. Wilkinson, and M. Novak. Easy Approach to Requirements Syntax (EARS). In *RE*. IEEE, 2009.
[13] N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive (1) designs. In *VMCAI*. Springer, 2006.
[14] S. Schewe and B. Finkbeiner. Bounded Synthesis. In *ATVA*. Springer, 2007.
[15] R. Yan, C. Cheng, and Y. Chai. Formal Consistency Checking Over Specifications in Natural Languages. In *DATE*, 2015.