Formalizing EARS – First Impressions

Levi Lúcio, Tahira Iqbal
fortiss GmbH
Guerickestrasse 25
Munich, Germany
{lucio, iqbal}@fortiss.org

***Abstract*—The Easy Approach to Requirements Specification (EARS) has been designed primarily as a set of templates to assist requirements engineers in writing software requirements that are clear and understandable. Its target are thus requirements engineers, software architects and developers. Due to the minimalistic nature of the English sentences that make up an EARS specification, it is reasonable to expect that automated tasks can be performed on EARS specification, among which verification and code synthesis. Given English cannot be directly understood by machines without some degree of ambiguity, EARS requirements can only by automatically processed if they are translated in advance into formal specifications. In this short paper, we explore how a translation from EARS into Linear Temporal Logic can be implemented in practice.**

I. Introduction and Problem Statement

EARS [6] has been designed at Rolls-Royce having in mind helping requirements engineers to specify requirements that are as unambiguous, complete and objective as possible – while being written in natural language. It is well understood that architecting and writing software are expensive tasks. Misunderstandings in communicating requirements to architects and developers amount to using precious software development resources in an unfruitful manner – leading to projects running over budget or simply failing [9]. Mavin *et al.* have shown with their work that, by using a simple set of templated English sentences for specifying requirements, ambiguity, partiality and subjectivity can be significantly reduced in textual requirements. They have shown this holds even for large specifications [4], [5].

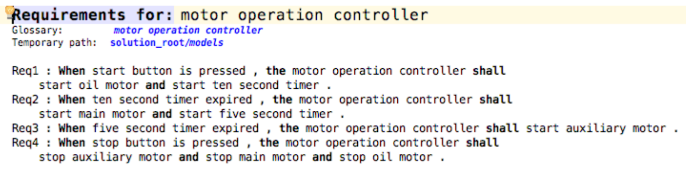
A byproduct of having requirements written in a simple subset of natural language is that automation becomes more possible. The regularity of EARS lends itself well to mathematical treatment and, given the wealth of research in formal methods, it is only reasonable to investigate how EARS specifications can be turned into fully precise formal specifications. Such formal specifications can then be used for purposes such as *code synthesis*, *formal verification* or even *test case generation*. Teufl [8] explicitly states in her PhD thesis that having the means to transform requirements (and in particular EARS requirements) into formal specifications would provide ground truths. These could then be used for formal verification and consistence checking during the requirements gathering process.

The work we present here is motivated by the IETS3 project[1] ran recently by our company on the construction of languages for requirements engineering. With IETS3 we have been able to synthesize software controllers directly from EARS specifications. To that end, we have transformed EARS into Linear Temporal Logic (LTL), a mathematical formalism that allows expressing temporal dependencies between the states of a system. The transformation we present in this paper has been implemented and is part of the EARS-CTRL environment also developed in the context of the IETS3 project. Note that, while the EARS-CTRL environment has been described in [1]–[3], this short paper aims systematically provides an algorithm for the transformation from EARS to LTL and extracts general lessons from the implementation of that transformation.

In this short paper we will elaborate on how this translation has been implemented, identify some of the hurdles found when building such a translation and extrapolate some lessons we have learned into the bigger picture of having a formal counterpart to EARS.

II. Formalizing EARS using Linear Temporal Logic

In this section we will use as running example the simplified requirements for the operation of a controller of an engine system. The system includes a *main* engine, an *auxiliary* engine and an *oil pump* engine. The sequence to start the machine is to first run the old pump engine; after 10 seconds start the main engine; and after 5 more seconds start the auxiliary engine. The EARS requirements for such a controller are depicted in Figure 1.



```
Requirements for: motor operation controller
Glossary:          motor operation controller
Temporary path:    solution_root/models

Req1 : When start button is pressed , the motor operation controller shall
    start oil motor and start ten second timer .
Req2 : When ten second timer expired , the motor operation controller shall
    start main motor and start five second timer .
Req3 : When five second timer expired , the motor operation controller shall start auxiliary motor .
Req4 : When stop button is pressed , the motor operation controller shall
    stop auxiliary motor and stop main motor and stop oil motor .
```

Fig. 1. EARS requirements expressed in the EARS-CTRL environment

Let us now build a set of LTL formulas that reflect the semantics of these requirements. LTL builds on propositional logic by adding temporal operators to it. Formulas in LTL

[1]IETS3 was funded by the German Federal Ministry of Education and Research under code 01IS15037A/B.

Ubiquitous	**The** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}(response)$
Event-Driven	**When** $\langle$*trigger*$\rangle$ **the** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}(trigger \rightarrow \mathbf{X}(response))$
State-Driven	**While** $\langle$*in state*$\rangle$ **the** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}(\textit{in state} \rightarrow response)$
Option	**Where** $\langle$*feature*$\rangle$ **the** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}(\textit{feature} \rightarrow response)$
Complex	**While** $\langle$*in state*$\rangle$, **when** $\langle$*trigger*$\rangle$, **the** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}((\textit{in state} \wedge \textit{trigger}) \rightarrow \mathbf{X}(response))$
Unwanted	**If** $\langle$*preconditions*$\rangle$ $\langle$*trigger*$\rangle$, **the** $\langle$*system*$\rangle$ **shall** $\langle$*response*$\rangle$	$\mathbf{G}((\textit{preconditions} \wedge \textit{trigger}) \rightarrow \mathbf{X}(response))$

Fig. 2. Translation between EARS templates and LTL

are particularly appropriate to express properties of runs of a reactive system, meaning how the state of that system should evolve over time. For example, one may want to state that always, when a motor is on, a thermostat is measuring its temperature. Or that, while the start button of a car is pressed then the starter engine will assist in firing the main engine. Or even, that eventually in the future the motor will stop. LTL incorporates operators to precisely describe these situations. Due to space limitations we will refrain from formally introduce LTL here, but will rather explain the parts of it that are relevant to our presentation. Note that we do assume basic familiarity with logical notations. LTL was first introduced by Pnueli in 1977 [7].

By looking carefully at the example in Figure 1 it is possible to understand that all requirements are given in the form of EARS *event-driven* templates. Take for instance Req1. A reasonable translation for it into LTL would be as follows:

$$\mathbf{G}(\textit{start button pressed} \rightarrow \mathbf{X}(\textit{start oil motor} \wedge \textit{start 10 sec timer}))$$

The mathematical expression above reads as: *globally*, during a run of the system (temporal operator $\mathbf{G}$), if the system receives the *start button* event, then in the *next* moment (temporal operator $\mathbf{X}$) the motor will be started as well a 10 second timer.

In the table in Figure 2 we present a generic set of rules for transforming the EARS templates presented in [6] into LTL. Note that all the transformation rules follow a similar pattern: for every state in a run of the system some output is triggered, if a state satisfies a given condition. In the *ubiquitous* case, all states in the run have to produce the response (enforced by the $\mathbf{G}$ operator). In the *state-driven* and *option* case, the response is produced only if the state meets a certain condition or a certain feature of the system is enabled. The *event-driven*, *complex* and *unwanted* patterns are different in the sense that the response only appears in the moment following the arrival of the trigger (as imposed by the $\mathbf{X}$ operator).

III. THE IMPORTANCE OF CONTEXT

While the translation we have presented in section II provides a boilerplate to transform EARS into LTL, the rules given in table 2 are naive regarding the interaction between the complete set of requirements in an EARS specification. For example, if one take into consideration the transformation rule for *Event-Driven* requirements, the LTL formlula corresponding to those will only state that in the moment immediately after the *trigger* is received, the *response* will be given. However, the requirement states nothing about the life of the system after that moment. It is reasonable to ask questions such as:

1) after a *trigger* is received, will the *response* keep produced by the system for some period?
2) if so, until when?

While it is challenging to come up with a generic answer for these questions, we will attempt to describe a solution for the case in which such a specification is used for code synthesis, as is done in our EARS-CTRL tool.

As mentioned in section I, EARS-CTRL was built to synthesize software controllers directly from requirements written in EARS. Software controllers receive input from *sensors* and produce output on *actuators*. By orchestrating those signals a software controller controls the operation of a machine, such as the system of motors presented in Figure 1.

Coming back to questions 1 and 2 above, let us attempt to answer then by analysing a concrete example based on our motor controller case study. Part of Req1 states that when the *start button* is pressed then the *oil motor* will start. It is assumed that the motor will continue to be on until it is shutdown[2], which can only happen if Req4 applies (meaning the *stop button* has been pressed). However, this is information that can only be deduced by taking Req4 into consideration.

In order to incorporate this contextual information in an LTL specification, static analysis of the EARS requirements becomes necessary. Such a static analysis is embodied in the algorithm in Figure 3. The algorithm calculates which triggers exist in other EARS requirements that should be taken into consideration during the translation and accumulates them in the $untilClause$ set. The body of the algorithm goes through all other requirements in the specification and checks whether the responses overlap, in which case their *triggers* are added to the $untilClause$ set. Such a static analysis needs to be done for all EARS templates in table 2, except for the *Ubiquitous* template – in which case the response should in any case always be present in the system.

If we now revisit the translation of Req1 to include contextual information calculated by the static analysis algorithm, we can now produce the following LTL formula:

[2]Note that we assume that *start motor oil* $= \neg$(*stop motor oil*). In EARS-CTRL this is explicitely expressed in a glossary accompanying the requirements in Figure 1.

```
$untilClause = \emptyset$
$currentReq$ = requirement being transformed
$contextReq$ = first requirement different from $currentReq$
while not all context requirements have been processed do
  if $currentReq.responses \cap contextReq.responses$ then
    $untilClause = untilClause \cup contextReq.trigger$
  end if
end while
```

Fig. 3. Algorithm to calculate contextual information

$$\mathbf{G}(\textit{start button pressed} \rightarrow (\mathbf{X}(\textit{start oil motor} \wedge \textit{start 10 sec timer}) \ \mathbf{U}\ \textit{stop button pressed}))$$

The **U** operator in the formula enforces that the *start motor on* actuator will be left on **U**ntil the *stop button* is pressed. Note that the LTL formula contains only one proposition after the until operator. This is in general not the case for the algorithm Figure 3, as more than one proposition can be accumulated in the $untilClause$ set – in which case the propositions after the until operator appear disjuncted in the corresponding generated LTL formula.

IV. Open Issues and Discussion

In this paper we have provided an algorithm for translating EARS requirements into LTL. The algorithm operates in two phases: it first translates every EARS requirement present in the specification taking into consideration only local information (section II), while a second pass adds some of the contextual information present in the totality of the requirements (section III).

We do not presume our translation if generalizable for all other code synthesizers or other tools that consume LTL specifications (such as model checkers). Nonetheless, we are convinced that the *local* and *context* translation phases we have identified in our translation are generally necessary. Being aimed at requirements engineers that deal mostly with natural language, it is expectable that some of the information exposed in an EARS specification is implicitly stated. While for humans this does not pose a major problem, it is a showstopper for computers. By breaking down the translation into phases and bringing that implicit knowledge to the foreground while encoding into LTL formulas, we believe a part of this problem is solvable in practice.

Note that other contextual information could be retrieved from an EARS specification and made explicit. Such information could include e.g. finding similar terms that denote the same entity, automatically deriving a notion of state of the system and even explicitly deriving the behavior (e.g. as state machines) from EARS requirements. Such state machines could e.g. be partially built using the static analysis algorithm we have presented in section III.

From a more abstract viewpoint, the translation we present here is also potentially also an enabler for the formal verification of and test case generation from EARS specifications. Which kind of model-checker or test-case generator specific information would be required to be added to the LTL we currently produce is beyond the scope of this paper. Research in this direction could marry the increasing power of formal verification tools with the proposals of the EARS community.

References

[1] EARS-CTRL Model Development Environment. https://github.com/levilucio/EARS-CTRL.git.

[2] L. Lúcio, S. Rahman, S. bin Abid, and A. Mavin. EARS-CTRL: generating controllers for dummies. In *Proceedings of MODELS 2017 Satellite Event: FlexMDE, 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA*, volume 2019 of *CEUR Workshop Proceedings*, pages 566–570. CEUR-WS.org, 2017.

[3] L. Lúcio, S. Rahman, C.-H. Cheng, and A. Mavin. Just formal enough? automated analysis of ears requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 427–434. Springer, 2017.

[4] A. Mavin and P. Wilkinson. Big Ears (The Return of "Easy Approach to Requirements Engineering"). In *RE*, pages 277–282. IEEE, 2010.

[5] A. Mavin, P. Wilkinson, S. Gregory, and E. Uusitalo. Listens Learned (8 Lessons Learned Applying EARS). In *RE*, pages 276–282. IEEE, 2016.

[6] A. Mavin, P. Wilkinson, and M. Novak. Easy Approach to Requirements Syntax (EARS). In *RE*. IEEE, 2009.

[7] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[8] S. Teufl. *Seamless Model-based Requirements Engineering: Models, Guidelines, Tools.* PhD thesis, Technical University Munich, Germany, 2017.

[9] The Standish Group. Software Chaos. https://www.projectsmart.co.uk/white-papers/chaos-report.pdf, 2014.