From Natural Language to Linear Temporal Logic: Difficulties of Capturing Natural Language Specifications in Formal Languages for Automatic Analysis

Elsa L. Gunter*
New Jersey Institute of Technology

Abstract

To be able to apply tools to verify properties of any system it is necessary to have a formal model of that system, and a formal expression of the properties to be verified. Linear Temporal Logic [5] is a common formalism for expressing properties of systems of concurrent communicating processes, as found for example in embedded systems. To verify these formulae (or attempt to derive test suites from them) for a system, the relevant processes are modeled as automata, and algorithms are applied to the pair of the formulae and the automata to find paths in the automata that satisfy certain properties pertaining to the given specification. However, specifications begin as informal natural language concepts (and maybe documents). It is generally these informal specifications that express the true and fullest system requirements. Therefore, it is necessary to recognize the gap between the informal and the formal specifications. In this paper we concentrate on the gaps that exist between informal specification and formal specifications given in Linear Temporal Logic, and some possibilities for controlling that gap. We also discuss the impact these methods of handling the gap have on the system model.

1 Introduction

In the ideal world, a software programming project would begin with a user's requirements document, which would be expressed in terms that were meaningful to the user and which would form the agreement between the user (or consumer) of the product and the producer [6]. System specifications would then be created by refining the requirements to a description of just the behavior of the system, using (and hopefully documenting) knowledge of the environment in which the system would be deployed to factor out the aspects that rely on phenomena that are not directly input to or output by the system. From here the specification would be decomposed into functional components to provide a framework for the design of the software. Each functional component would be refined to an operational model, possibly such as a finite state machine, and from here coding could begin. After the software has been developed, the specification again would be needed, this time to guide system testing.

Requirements and specifications begin their existence as ideas expressed in natural language. Before automated tools may be applied to the specification, it must be captured in some formal language processable by computers. If an operational model is to be demonstrated to satisfy the specification, then the language capturing the specification must also have a rigorous semantics. However, natural language is highly expressive, and highly ambiguous. Formal languages having a rigorous semantics and admitting automated checking of properties are of limited expressive power. Therefore, just as there is a gap between the implementation and the operational model,

*This research was supported in part by ARO DAAD19-01-1-0473.

and between the operational model and the specification, there is a gap between the informal, natural language specification and the formal specification.

Since the specification is the foundation of the software construction, it needs to satisfy various sanity criteria by itself. In addition, it must be possible to confirm that the composition of the operational models for the system components together with the necessary domain knowledge satisfy the specification, and that the individual code units implement the corresponding operational models. At each step of this confirmation process there are concerns that must be taken into consideration. Efforts to derive code automatically from operational models attempts to address the problem of confirming that individual code units satisfy the operational models. Program and temporal logics, and model checking are a very active area of research for demonstrating that the operational models satisfy formal specifications. In this paper we wish to draw attention to the step of capturing the specifications in a formal language suited to automatic analysis and use in the subsequent stages of development and validation.

2 Examining Natural Language Specifications

As mentioned above, specifications begin there existence in human thought, which is typically not expressed as logical formulae, automata, or mathematical expressions, but rather in natural language. Natural language is at the same time on the one hand imprecise and ambiguous, and on the other, highly expressive. In rendering a natural language specification in a formal language, both aspects are hurdles to be overcome.

Ambiguities are removed by a careful reading of the natural language document by a reader who is prepared to to challenge the precision of every statement, both individually and a component of the whole document. The reader is greatly assisted in this process, if to aid their critical reading, they are attempting to formalize the document as literally as possible in a language with a clear and precise semantics. Eliminating ambiguities is done in an iterative process of recognizing the ambiguity, seeking clarification from the author or authority, attempting to formalize the clarified version, then uncovering new ambiguities which need clarification. This process is inherently a human one; on one end the basic understanding of the meaning of the specification resides in the minds of its authors, and on the other, the recognition of the ambiguity is done by the human performing the translation. With time, the act of recognizing ambiguities may become increasingly assisted by machine translators. However, by making use of a translator, in effect, at the point we submit the "natural language" specification to the translator, the language becomes a machine language and the question of whether the natural language expressions have the same meaning as the meaning given by the translator still remains.

Because of the human factor, both the act of seeking clarification and rendering clarification are subject to non-determinism. On the side of seeking clarification, the reader of the specification may read a statement for which only one meaning occurs to them. However, there may have been a different meaning, possibly the one intended by the author. On the other side, the author, when asked for a clarification, may not understand the differences in the possible interpretations, and may choose a possibility different from the one they truly meant.

As an example of the need to recognize and eliminate ambiguities, let us consider a sample statement that might be found in a user requirements (or system specification) document of a system that must monitor and regulate a device that includes a pressure reading:

> Once the pump has been turned been on, an initial pressure reading will be taken and displayed, and thereafter the displayed value for the pressure will be updated every 2 seconds.

At first reading, this may seem like quite a precise statement. Let's look at it a bit. What does it mean for the displayed value to be *updated*? The first answer is that the displayed value is made to be the same as the actual pressure value. But since we are talking about time, are we

insisting that the displayed value is made to be the same as the actual value at exactly the same time? Are we insisting that there is *no time* allowed for the data to be read and the display to be changed? If the underlying notion of time is course enough, and our sensors and processors are fast enough, then this may be reasonable. However, this is really just sweeping the issue of time for computation under the rug and assuming that the machine is fast enough to meet whatever the real unwritten requirements are. Thus "updated" really means something more like that the display value is the same as the actual pressure value of not more than some allowed amount of time ago. As the reader has probably also noticed, we can't realistically require the display value to be "the same" as the actual pressure value. We can't measure the actual pressure value, we can only measure an approximate value for it. Another question is what does it mean for the display to be updated *every 2 seconds*? Does it mean that the reading upon which the display is based should be taken exactly every 2 seconds, or that the new display value would be generated exactly every 2 seconds (or perhaps they really wanted to know that the displayed valued was never more than 2 seconds old, which isn't really the same as updating the display every 2 seconds). If the time to read the pressure value and update the display is constant, then the first two options above amount to the same thing. However, if for example, the time to read the pressure is variable, then they are not. And lastly (for now), what should happen if the system fails to be able to update the displayed value at the appropriate time? Saying that this should be impossible means claiming there will never be any faults in the environment or with the system interface to the environment.

The points made above about the given example are relevant regardless of the choice of logic in which to model the specification. However, attempting to formalize this statement in various frameworks would tend to show up at least some of these issues. For example, if we tried to capture the specification with an extended finite state machine, we would likely end up with a guarded transition labeled by something like:

$$\mathsf{seconds}(\mathsf{current\_time} - \mathsf{last\_update\_time}) \geq 2 \Rightarrow$$
$$\mathsf{display\_value} := \mathsf{pressure\_value}$$

If the model is interpreted in a setting where, for each transition, in addition to the actions committed by the machine, the environment is allowed to arbitrarily change all values it supplies (e.g. [7]), then, with each transition, pressure_value potentially will change its value. Therefore, we can not guarantee that there is ever a time when display_value = pressure_value. All we can guarantee is that the current value of display_value is the same as some previous value of pressure_value.

3 From Natural Language to LTL

As discussed above, natural language specifications are subject to considerable ambiguity. However, even if they are rendered with the utmost care and precision, before they can become the grist for the mill of automatic analysis, they must be faithfully translated into a machine-processable formal language, with a rigorous semantics. Moreover, that formal language must admit the kinds of automated analysis desired.

Linear Temporal Logic (LTL) is a common language used for specifying properties that are to hold of concurrent systems. One advantage of LTL is that it has a fairly simple and intuitive syntax with a simple and well-defined semantics. However, the driving motivation behind the use of LTL is that it is possible to effectively check whether a given LTL formula holds of a given system expressed as an automaton. However, these factors are at odds with LTL being a highly expressive language. LTL has a limited ability to express a set of states within a sequence where a property is to hold. Perhaps an even bigger impediment in practice, at least by anecdotal evidence, is the inability in LTL to express desired relations between state components at different points in time.

In LTL, time is not explicit, and thus, if it is necessary to talk about time, this must be done by making time an explicit component of the states tested by the base sub-formulae in the LTL formula. Then time is also added explicitly to any model of the system being checked. (As an example, see [1].) As an example, consider the requirement given earlier concerning the monitoring of the pump pressure. To be able to rendered the clause "updated every 2 seconds" in LTL, requires a variable to monitor the passage of time so that we know when two seconds has passed. However, there are properties of time to which any system model must conform. For example, time can not decrease. (If it could, most implementations of the pump monitor would not provably satisfy the update requirement.) These properties cannot in general be expressed in LTL, so checking that time has been correctly modeled is generally outside the scope of what can be verified by model checking LTL formulae. Just as it is important to document what domain knowledge is used in refining user requirements into a system specification, it is also important to document these extra-logical facts that are required to hold, and upon which the correctness of the system may be depended.

In addition to concepts such as time, which are possibly implicit in the system or inherited from the system environment, but which may need to be made explicit in the model, it is also often necessary to add "history variables" to our model to enable us to express requirements pertaining to past values of components of the state. LTL is a logic for expressing properties of sequences of states. However, the base formulae are only over a single state. There is no ability to express relations between the values of the state at one point in time and another. We can say that the value of a given variable is always 5, but we cannot say the value of a given variable is constant, or monotonically increasing. To be able to capture the idea that the value of x is monotonically increasing, we would introduce an auxiliary variable, say previous_x and require that

$$\mathsf{Always}\ x \geq \mathsf{previous\_x}.$$

This only captures the notion that x is monotonically growing if we have the additional knowledge that every transition updates previous_x to the previous value of x. This is a fact that can not be expressed in LTL, so it must be verified in the augmented system model by some means other than LTL model checking.

In concurrent and embedded systems, there is an additional difficulty with the use of the temporal operator Next. To be able to prove properties of a system, it is generally necessary to decompose the system into modules and abstract out those components that do not contribute to the property currently being verified. If specifications are given with the Next operator, they may be checked as true in the subcomponent deemed relevant, but the property may be false in the larger system where all the components are composed. Let us consider the LTL specification

$$\begin{aligned}&\mathsf{Always}\ (\mathsf{seconds}(\mathsf{current\_time} - \mathsf{last\_pressure\_reading\_time}) \geq 2 \Rightarrow\\ &\qquad \mathsf{Next}\ (\mathsf{display\_value} = \mathsf{previous\_pressure\_value})).\end{aligned}$$

In natural language, this tells us that anytime the system sees that it's been at least 2 seconds since the last time the display was updated, then the system must update the display to the current pressure value (what will be the previous pressure value, once we complete the transition doing the update). This specification may very well hold for the subsystem composed only of the system (or environment) clock, the pressure monitor, and the display, but if the system is more complex with other values to be monitored and perhaps controlled, when the other components are put in parallel with the particular subsystem (with an interleaving semantics), it is not automatic that the specification still holds, and indeed, unless there are some kinds of locks preventing other parts of the system from progressing, this specification is likely to be false. At the first point at which $\mathsf{seconds}(\mathsf{current\_time} - \mathsf{last\_pressure\_reading\_time}) \geq 2$, it is possible for some other part of the system to be busy doing something. The problem with Next, as well as other constructs such as Always and Until, is that they are not compositional.

Unfortunately, the most obvious alternative to the use of Next is the use of Eventually. This is generally much weaker than is intended by a natural language specification where Next seems

appropriate. If one can identify a property φ that holds of states of the subsystem prior to the transition being taken and that fail to holds as soon as it is taken, then it might be possible to use a formula of the form φ Until ψ in place of Next *psi*. What we would like to use for φ is a statement that the state of the subsystem doesn't change. However, being able to express this directly requires being able to compare values for variables from one state with those from another state (the "successor" state), and as we have said above, this is outside the expressive power of LTL. A non-solution is to add a variable to our state for each transition (or each state) in our subsystem. For each transition, we add the action of setting the corresponding variable to true, and setting all of the variables representing the other transitions (or out-states) to false. Then one can attempt to capture that the subsystem hasn't changed by enumerating all the transitions (or states) and saying that each variable has an explicit fixed value until the property that is to hold next holds. Something like this might be workable in the setting of system models using only plain finite state machines. But this is generally not a solution because it does not allow us to detect when change has occurred in the subsystem through taking a transition that is a loop. In extended finite state machines, where there is a notion of "program states" given by variable values, in addition to the states of the finite state machine, one may enter the same finite state machine state many times having a different program state each time. Thus the very values that we are interested in not changing may change anyway.

Let us return to the example of the display update. Below is a "clarification" of the original specification given.

> Once the pump has been turned been on, an initial pressure reading will be taken and, if the value read is not an error, it is displayed. Thereafter, provided there are no errors in pressure readings, the display will never be more than 2 seconds old.

We can capture much of the intent of this specification with the following LTL formula:

Pump_on $\wedge \neg$error(pressure_reading) $\Rightarrow$
Eventually
(display_value = previous_pressure_reading $\wedge$
 last_display_update_time = previous_current_time $\wedge$
 (error(pressure_reading)
  Releases
  (seconds(current_time $-$ last_display_update_time) ≤ 2
   Until
   (display_value = previous_pressure_reading $\wedge$
    last_display_update_time = previous_current_time))))

Some of the assumptions that are made for this to correctly express the desired specification include that we can define error from just the value of pressure_reading, and that last_display_update_time always contains the time when the display was last updated. These requirements are extra-logical and need verifying by means other than LTL model checking.

The difficulties we have mentioned here are not specific to Linear Temporal Logic, but extent to similar, more expressive logics such as CTL* (see [2]) and the μ-Calculus (see [4]). Allowing branching time constructs (along some path, or along all possible paths) does not address the issues raised here. The additional expressiveness of the μ-Calculus does allow greater ability to describe a set of states for which a proposition is to hold. However, in all these systems, there is still a lack of ability to relate values from different points in time, and a lack of compositionality.

4 Reflecting Assumptions in the System Model

In the previous section we discussed methodology for how we can use LTL formulae to indirectly capture the meaning of natural language specifications. These methods have limitations

and come at an expense of imposing restrictions on the system models to be checked. These restrictions often involve the addition of new variables and alterations to the set of transitions to reflect the needed behavior of these variables. For example, when adding history variables, there is the extra-logical (i.e., not expressible in Linear Temporal Logic) requirement that the history values record the appropriate values at the appropriate times, and never take on any irrelevant values. Not only must these extra variables appear in the LTL specifications, but they must also occur throughout the system model to be verified with respect to the LTL specification, in such a way as to guarantee the extra-logical requirements. Cluttering the system model by modifying it everywhere needed with this additional information that is not explicitly present in the actual implementation reduces the trustworthiness of the verification, and the increases the gap between the system model and the implementation.

We propose that, where possible, the system model be left intact, and the model that is verified be an automatically generated product of the original system model together with a separate model capturing the added variables and transitions. Augmenting the model with history variables, for example seem a prime candidate for such automation. For some notions of automata and products, deriving the model needed for verification from the initial system model via product may not be an appropriate thing to do. However, for extended finite state machines (EFSM) with labeled transitions, this is possible. Briefly, an extended finite state machine with labeled transitions is a finite state machine where the edges are labeled from a given alphabet, but where we extend with a function mapping letters of the alphabet to guarded actions over a fixed set of variables. In this setting, if the original system specification is deterministic to the extent that no edges from the same start have the same label, then if we take the product of the system EFSM with an EFSM with the same labeled finite state machine, but extended by different guarded actions, the result will yield an extended finite state machine that has the same underlying finite state machine, but where the guarded actions are now the pairwise composition of the guarded actions from the two machines.

Let us see what this means in the example with the pressure display. Possible (pseudo-) code for the pressure display is as follows:

```
if Pump_on then if not(pressure_value = error_value)
then {display_value := pressure_value;
      last_display_time_update := current_time;}
else pressure_alarm := true;
while (pressure_alarm = false)  do
{if not(pressure_value = error_value)
 then (if seconds (current_time - last_display_time_update) >= 1.8
       then {display_value := pressure_value;
             last_display_time_update := current_time;})
 else pressure_alarm := true;}
```

For reasons of space, we omit a detailed description of the EFSM that captures this process. Briefly, the code can be compiled into a control flow graph [3] and then automatically translated into an EFSM with ten states. Each state corresponds to either a conditional or an assignment. All transitions are labeled distinctly. Each conditional state has two out transitions, each with a guard but no real action, and each assignment state has one out transition with no guard, but an action corresponding to the assignment. A more compact EFSM can undoubtedly be created, but there is a simple algorithm to automatically generate this one. To add the information about the previous_ variables, we create a new EFSM with exactly the same states, edges, and transition labels, but now every transition has the same unguarded action associated with it:

```
previous_pressure_reading := pressure_reading;
previous_current_time := current_time
```

If we had more previous_ variables to be kept track of we would add them in. The product of this EFSM with the system EFSM yields an EFSM with the same states, edges, and labels, but where the action component of the guarded action associated with each transition is augmented by the additional assignments. By decomposing the problem this way, we separate out the task of of verifying the "extra-logical" properties, so that they only need to be shown for the EFSMs that were constructed solely to make them true. For the example of the history variables, by this factorization, it is immediate that for each transition the value of each previous_ variable is the same after the transition as the value of corresponding original variable was before the transition. Moreover, by this factorization, we don't have to worry about having accidentally changed the system model in some unpredictable way since the system model makes no mention of the previous_ variable and the history EFSM effects only them. At present, what is proposed here is only methodology. However, it seems clear that at least in some limited but common cases, this process of factoring and constructing auxiliary EFSMs, such as the history EFSMs, could be automated.

5 Conclusions

In this paper we have discussed the transition from an informal natural language specification to a formal specification in Linear Temporal Logic and the construction of a model suited to the specification, but still capturing the system design so that the system design can be verified to have the desired properties. To formalize informal specifications, there is an iterative process of clarification and rendering the specification more precise. Following, or interleaved with this, is the process of trying to express the informal specification in a formal language. The more expressive the language, the closer the formal specification can be to expressing exactly the intent of the informal specification. However, the more expressive the language, the less likely there is to be effective fully automated support for proving that the properties hold of the system model. Since Linear Temporal Logic is a dominant language for formal specification, owing to the ability to effectively model check LTL properties of finite models, we focus on translating specifications into LTL. We discuss some of the ways LTL lacks expressive power, and we discuss techniques for circumventing some of the difficulties resulting from this deficiency. A method for capturing aspects of the environment, such as time, and information about past state involves augmenting the state with additional variables. The properties to which these auxiliary variables must adhere are typically not expressible in LTL. These auxiliary variables have no existence in the system design (or only an existence for purpose of being read, if they represent input to the system from the environment). To capture the additional needed restrictions on these variables it is necessary for it to be added to the system model. We propose a methodology for this addition that involves creating separate models capturing the needed behavior of just these auxiliary variables, and then automatically composing these models with the existing system model to generate the model used for model checking. This discipline allows us to isolate the aspects of the model expressing the auxiliary information so that it can be verified by other methods (such as informal proof through inspection). It also allows us to retain the system model intact so that we can have confidence that the results of the model checking applied to the composite model imply the intended meaning for the original system specification.

6 Hopes for the Future

In this paper we have identified three problems with transitioaning from natural language specifcations to formal language specifications:

- Natural language specifcations are generally more ambiguous than they first appear. Therefore, there is a need to identify ambiguiutes and disambiguate. The process of rendering

the specifcations in a formal language can serve as a useful tool in identifying ambiguities.

- To date, formal langagues suitable for such automation as model checking are not adequately expressive enough to directly capture the natural langague specifications. We have proposed some methodologies for indirect capture through a combination of auxiliary variables and extra-logical requirements. This methodolgy handle common cases moderately well, but at heart is only a collection of heuristics.
- Adding auxiliary notions to the system specification requires adding them to the system model as well. Rather than clutter the system model with this auxiliary information not explicit in the implemention, we propose that the auxiliary information be captured in a auxiliary model constructed solely for this purpose, and then composed with the system model to obtain the model used for verification.

The problem that we have indicated the best solution for is that of the augmenting the system model. For certain methods of augmenting, it is clear that it is posible for this to be performed autmatically. So do it. And apply it. See how well we can handle real world problems this way.

We have discussed some heuristics/methodologies for for capturing natural language concepts in a language such as LTL. However, clearly we are contorting ourselves to fit the language. More work needs to be done on constructing langagues with the expressive power to capture the ideas needed in specifcations. Simply making the language more expressive, in the mathematical sense, isn't the answer. I do believe that a language such as Zermelo-Frankel set theory, or higher-order logic, which are expressive enough to capture basic mathematics, are expressive enough to capture system specifcations, and even user requirements. However, such languages cannot admit the decision procedures necessary for system verification. It is not clear, however, that for "real" system specification, the full expressive power of such langagues is required. One area of research that is needed is to find languages, probably similar to LTL, but with expressivity extended in directions that are those needed for capturing most commonplace specifcation constructs. An example of this might be adding the notion of the next-state value for a variable, in addition to the current value. Such constructs need to be chosen with consideration for both the extent to which they will facilitate rendering specifications, and the abilty to automatically check that they hold of system models.

Another possible partial answer is to translate the natural language into a highly expressive formal language, and then derive approximaitons to these requirements in a less expressive language suitable for automation such as model checking. In this scenario, at first derived formulae that were stronger than the original would be checked. If they succeeded, then the original requirement would be known to hold. If they failed to check, then counter-examples to the derived formulae could be generated and if these

With experience of what kinds of auxiliary information can be a

References

[1] Nikolaj Bjrner, Anca Browne, Eddie Chang, Michael Coln, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Toms E. Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In *International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418, 1996.

[2] E. A. Emersom and J. Y. Hapern. 'Sometimes' and 'Not Never' revisited: On branching versus linear time temporal logic. *JACM*, 33(1):151–178, Jan. 1986.

[3] Elsa Gunter and Doron Peled. Path exploration tool. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, volume 1579 of *Lecture Notes in Computer Science*, pages 405–419, Amsterdam, The Netherlands, 1999. Springer.

[4] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, Dec. 1983.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[6] C. Mazza, J. Fairclough, B. Melton, D. de Pablo, A. Scheffer, and R. Stevens. *Software Engineering Standards*. Prentice Hall, 1994.

[7] M. A. Ozols, K. A. Eastaughffe, A. Cant, and S. Collignon. DOVE: A tool for design modelling and verification in safety critical systems. In *Proceedings of the 16th International System Safety Conference*, Seattle, USA, September 1998.