Checking LTL properties of application CFGs with LLVM and ITStools

George Hodgkins

December 5, 2021

1 Introduction

This report describes the design of a system to check LTL properties of applications at the control-flow graph (CFG) level using the LLVM compiler infrastructure [5] and the ITStools model-checking suite [4]. The properties checked are composed of the standard set of LTL operators and Boolean connectives, and atomic propositions defined at the basic block level, such as "block writes to memory location x". The underlying infrastructure supports the specification of the property to check, and to some extent even the atomic propositions, at runtime, although due to limitations of the LLVM frontend this is not possible with the system as currently configured. We find that the system reliably produces the correct result for trivial test problems. Section 2 describes the implementation, and section 3 describes limitations of the current system and future work that may be done to improve it.

2 Design

The design of this system (best described as an LLVM plugin) consists of three parts: the atomic proposition subsystem, the function modeling subsystem, and the interfaces between LLVM, the model-checking toolset, and those subsystems. Checking a property would generally proceed as follows. First, the atomic propositions must be defined; currently this is done when compiling the LLVM plugin (but see discussion below). Then, the application is transformed by an LLVM frontend such as clang from some high-level language to a single-static-assignment CFG represented in LLVM's intermediate representation (IR), a kind of target-independent pseudo-assembly language. The plugin takes the intermediate representation of a function as input and creates an abstracted model of the function as a non-deterministic Buchi automaton, which is then passed to the model checker along with the property to check. This process is discussed in detail in the subsections below.

2.1 Atomic Propositions

Generally, atomic propositions for our purposes are defined by "basic block classifiers," any function-like object that maps basic blocks to Boolean values. We say "function-like" because newer C++ standards (LLVM and the plugin are implemented in C++) support callable objects beyond traditional functions [1], including closures and class objects, permitting a wide range of behaviors to be implemented behind the classifier interface. Thus, although there must exist some statically-compiled code defining a classifier, it is straightforward to implement meta-classifiers for common properties which can be parameterized at runtime to produce a final classifier. For instance, for our tests we defined a "calls function" meta-classifier which takes the name of a function as an argument and produces a closure which checks blocks for a call to that specific function.

As implemented, atomic propositions are referenced in the user-specified LTL property by their index ("atp0", "atp1", etc), but the underlying tools would support arbitrary names for propositions. When building the model as described in the next section, labellings of basic blocks with atomic propositions (Boolean vectors) are computed on-demand when a block is added to the model, and the results are cached, since the labelling will likely be requested multiple times when adding transitions.

2.2 Property Specification

The plugin takes as an argument an LTL property specified in PSL syntax [2] (a superset of LTL) which is parsed using the SPOT formula-parsing library [6]. After parsing, the property is checked to confirm that it is valid LTL and that it contains only defined atomic propositions. The plugin is designed to allow the property to be specified at runtime, but due to limitations of the LLVM frontend used to run the plugin, it cannot be passed as a command-line option and so must be specified at compile-time.

2.3 Function Modeling

At runtime, the plugin creates a model of each function of interest in Guarded Action Language (GAL), a specification language created by the designers of ITStools [3]. States are not explicitly specified in GAL, but are defined by a set of state variables with initial values, and transitions enabled by a predicate which mutate those variables; the states of the produced graph are all unique combinations of state variables reachable from the initial state.

Our model defines a variable **state** which represents the "current" basic block. Basic blocks are uniquely identified by pointers to their internal representation in LLVM; we transform those to linear indices when building the model because the data types used internally by the model checker are too small to hold 64-bit pointers. In addition, we define variables **atp1**, **atp2**, etc, to represent the labelling of each basic block with atomic propositions – these are integer variables (GAL does not have a Boolean type for variables), but are only ever assigned values of 0 or 1 for false or true respectively. The initial value of **state** is the linear index of the function's entry block, which is always zero, and the initial values of the atomic propositions are the labelling of that block.

Then, beginning from the entry block, we iterate over each block in the function and add its transitions to its successors. For all transitions, the enabling predicate is simply [state == <predecessor block ID>]. The action of the transition is a synchronous assignment which assigns the successor block's ID to **state** and assigns any differing atomic proposition labels their new values (i.e. if **atp1** is true of block 0 but not of block 1, a transition from block 0 to block 1 would include the assignment atp1 := 0). The final model represents a non-deterministic Buchi automaton with labellings and states corresponding to our labeled CFG. Once the model is generated, it can be passed directly to the model checker, along with the property to check, and the model checker will return a Boolean value indicating whether the property is satisfied by the given structure.

3 Limitations and Future Work

As it stands, this design is a prototype of a concept which has wide-ranging and interesting extensions. This section discusses some of those extensions and what would be required to achieve them.

3.1 More Atomic Propositions

As mentioned previously, the interface for specifying atomic propositions is very flexible, especially when combined with the feature-rich LLVM API for manipulating and analyzing basic blocks. Classifiers could check pretty much any statically-decidable property of an application, even to the absurd extent of running another model-checker instance within the classifier. If the system were better integrated with LLVM (as it stands it would be difficult to keep LLVM integrated with the model checker as they are developed simultaneously), one could imagine that it would offer new opportunities for simplifying the design and implementation of new and existing program analyses.

3.2 Better Integration with Toolset

The utility of this system would be improved by better integration with both LLVM and ITStools. With respect to LLVM, which has a well-developed plugin infrastructure, the main improvement would be allowing dynamically-loaded plugins to define command-line options. Currently this is supported by the LLVM command-line option subsystem, but not by the frontend used to run individual passes (opt). This issue could

be circumvented by compiling the plugin into LLVM directly, but that is undesirable from a development perspective because rebuilding LLVM is slow - even just recompiling a single small source unit will require 15-20 minutes to re-link the large executables.

With respect to ITStools, the utility of the system could be improved by more closely integrating the model checking with the primitives provided by libITS, and getting rid of the dependency on ITS-LTL, which is a somewhat cumbersome application to build, is not maintained as well as the libraries, and is not designed to be used as a library.

3.3 More Expressive Function Modeling

GAL is a quite general system modeling language; its use here is somewhat like forcing it into a box that it was designed to escape from (explicit specification of states). It is likely that more expressive abstract models of function CFGs could be developed than the basic one used here, for instance taking into account data flow, that would make this system useful for solving a wider range of problems.

References

[1] *C++ Named Requirements: Callable*. URL: `https://en.cppreference.com/w/cpp/named_req/Callable`.

[2] Alexandre Duret-Lutz. *Spot's Temporal Logic Formulas*. URL: `https://spot.lrde.epita.fr/tl.pdf`.

[3] Laboratoire d'Informatique de Paris 6. *Guarded Action Language*. URL: `https://lip6.github.io/ITSTools-web/gal`.

[4] Laboratoire d'Informatique de Paris 6. *ITS-tools*. URL: `https://lip6.github.io/ITSTools-web/`.

[5] The LLVM Project. *The LLVM Compiler Infrastructure*. URL: `https://llvm.org/`.

[6] Laboratoire de Recherche et de Développement de l'EPITA. *Spot: a platform for LTL and ω-automata manipulation*. URL: `https://spot.lrde.epita.fr/index.html`.