Q: A Sound Verification Framework for Statecharts and Their Implementations

Samuel D. Pollard
Sandia National Laboratories
Livermore, California, USA
spolla@sandia.gov

Robert C. Armstrong
Sandia National Laboratories
Livermore, California, USA

John Bender
Sandia National Laboratories
Livermore, California, USA

Geoffrey C. Hulette
Sandia National Laboratories
Livermore, California, USA

Raheel S. Mahmood
Sandia National Laboratories
Livermore, California, USA

Karla Morris
Sandia National Laboratories
Livermore, California, USA

Blake C. Rawlings
Sandia National Laboratories
Livermore, California, USA

Jon M. Aytac
Sandia National Laboratories
Livermore, California, USA

Abstract

We present Q Framework: a verification framework used at Sandia National Laboratories. Q is a collection of tools used to verify safety and correctness properties of high-consequence embedded systems and captures the structure and compositionality of system specifications written with state machines in order to prove system-level properties about their implementations. Q consists of two main workflows: 1) compilation of temporal properties and state machine models (such as those made with Stateflow) into SMV models and 2) generation of ACSL specifications for the C code implementation of the state machine models. These together prove a refinement relation between the state machine model and its C code implementation, with proofs of properties checked by NuSMV (for SMV models) and Frama-C (for ACSL specifications).

CCS Concepts: • **Theory of computation → Program verification**; **Verification by model checking**; • **Software and its engineering → Formal software verification**; **State based definitions**.

Keywords: formal methods, state machines, C, specification languages, temporal logic, model checking

ACM Reference Format:
Samuel D. Pollard, Robert C. Armstrong, John Bender, Geoffrey C. Hulette, Raheel S. Mahmood, Karla Morris, Blake C. Rawlings, and Jon M. Aytac. 2022. Q: A Sound Verification Framework for Statecharts and Their Implementations. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS '22), December 07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3563822.3568014

FTSCS '22, December 07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9907-4/22/12…$15.00
https://doi.org/10.1145/3563822.3568014

1 Introduction

Sandia National Laboratories develops software for high-consequence digital control systems. With embedded control systems, bugs can have disastrous consequences [26]. And so, the high-consequence nature of our work means that it is worthwhile to spend significant effort to develop relatively complex formal statements about required behavior and verify an implementation against them.

Our approach to verifying implementations is subject to two main design constraints. First, our models are constructed from interacting subsystems with different clock domains, but requirements must apply to the system as a whole. Therefore, we require reasoning about the asynchronous composition of many interacting subsystems via *system-level* temporal properties. Note that here we do not focus on the details of the clock domains, such as those modeled with CCSL [2], only that our systems may be asynchronous.

Second, our approach must integrate into existing code bases and workflows. At Sandia, system designers already write specifications in an informal, but hierarchical, state machine-like graphical language along with English language requirements documents. These specifications are then written in Stateflow [28] and implemented in C. We (the formal methods team or "analysts") have the fortune of close communication with the system designers and software engineers, which allows us to enforce a clear separation for hardware interfacing (via API) and enforce coding standards (such as restricting how functions modify state, or the structure of state machines). We later explain how these restrictions enable our goal of automated verification.

Existing work does not satisfy the full constraints of our problem space. Verifying state machine abstractions of systems in modeling languages such as TLA+ [22] have shown success in academia and industry. However, modeling languages do not establish whether an implementation matches the model. This is not a strong enough correctness argument for our problem domain, especially considering the complexities of C.

Separately, there has been extensive work to check temporal properties directly against implementations [5], but these approaches do not support sound compositional reasoning beyond abstract specifications of external behavior. Lastly, significant work has been done to enable manual proofs of labeled transition system specifications against an implementation [4, 19] but the time-intensive nature of these approaches and their sensitivity to code changes would require more time and resources than we have to dedicate.

To address these gaps in the field we developed Q Framework (Q for short), which compiles Stateflow diagrams corresponding to a static, parallel composition of one or more transition systems into an intermediate representation. From this IR, Q then compiles both to SMV for model checking [14] and Frama-C ANSI C Specification language (ACSL) specifications [16] for static analysis of the C code implementation. If the temporal properties hold for the model and the ACSL proof obligations can be proven by Frama-C, Q provides strong, automated evidence that the C implementation refines the model's behavior and thus satisfies the desired temporal properties.

Our paper is structured as follows. In Section 2, we describe the architecture of Q by way of modeling a coffee maker. We then precisely describe our notion of a refinement relation between the model (state machines) and implementation (C code), the composition of state machines, and provide an argument for why these definitions of compositionality and refinement are sound (Section 3), and last conclude with a discussion on related and future work (Sections 4, 5).

Q Framework is not currently open source, however some examples as well as the formal semantics of QSpec are available here: https://github.com/sampollard/q-supplement.

2 Architecture

We now describe Q Framework at a high level. Figure 1 describes the overall architecture of Q. The workflow of Figure 1 roughly flows from the top-left downwards, where the C source code and Stateflow models are built based on requirements documents (written in English and informal diagrams). From these, we manually write the desired linear temporal logic (LTL) and computation tree logic (CTL) properties. Then, these are passed as input into the various parts of Q (described in this section). The final outputs of Q are then: the C source with ACSL specifications, the proofs that the C code matches the specifications (via the back-ends of Frama-C), and the proofs the state machine models obey the LTL/CTL properties (via NuSMV).

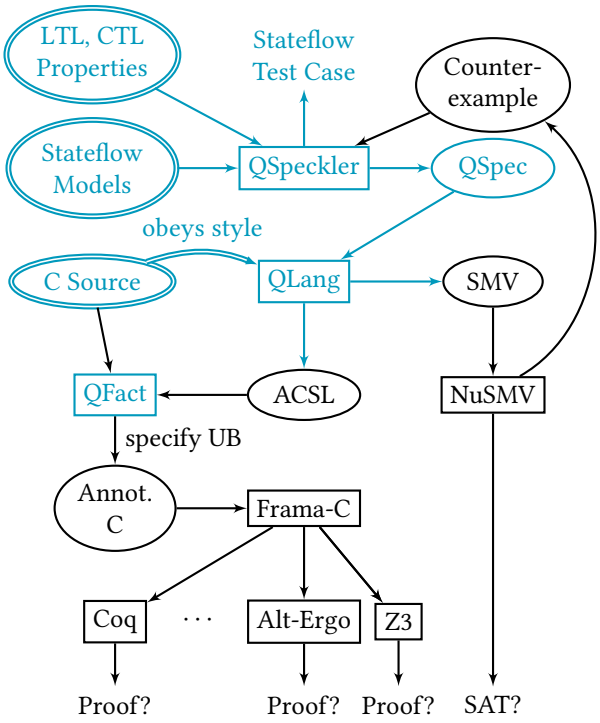

Figure 1. Architectural overview of Q, managed in general by QWorkflow. Ellipses are inputs, rectangles are tools, blue text are developed by Sandia, and double-struck shapes require manual specification or checking. UB refers to both *unspecified behavior* and *implementation-defined behavior.*

This process is iterative, since the system designers describe the requirements in English and Stateflow, then pass the designs to the software engineers, who may find inconsistencies or underspecifications. And further, system analysts (users of Q) may find errors or further inconsistencies. This is aided by a feedback loop in Q: for if the SMV model does not obey the desired properties, NuSMV emits a counterexample from which we can then generate a Stateflow test case, in order to further refine our LTL/CTL properties or the Stateflow model itself.

Throughout this section, we use an example of a "secure coffee maker," shown in Figure 2. At first glance, this example seems somewhat contrived. However, the compositionality of state machines allows systems of similar complexity to be used in realistic designs. The structure of this section follows the design of the coffee maker, showcasing the relevant parts of Q. In brief,

§ 2.1 Modeling systems using Stateflow.
§ 2.2 QSpec: a statechart language which evolved from SCXML.
§ 2.3 QSpeckler: A tool to convert Stateflow models into QSpec statecharts compatible for QLang.

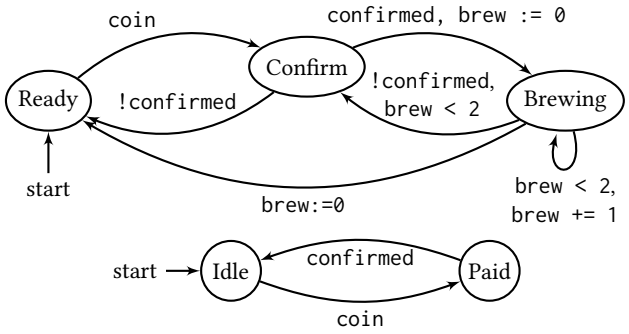

Figure 2. Model of a coffee machine with a coin slot and confirm (`confirmed`) and cancel (`!confirmed`) buttons, along with a payment system.

§ 2.4 LTL and CTL properties.
§ 2.5 QLang: the compiler from QSpec statecharts into an SMV model, which also generates ACSL function contracts.
§ 2.6 QFact: a clang plugin to add ACSL annotations to C code and perform code transformations to enable verification.
§ 2.7 QWorkflow: scripts used to orchestrate the interaction of the different parts of Q.
§ 2.8 Our use of external tools and languages.

2.1 State Machines and Stateflow

Currently, state machine models are designed in Stateflow from the requirements documents provided by system designers along with the C source code and domain knowledge of the system. While the Stateflow models and LTL/CTL properties require some expertise to formalize and prove system requirements, in our experience, analysts need not be formal methods experts to use Q. We now describe the state machines in Figure 2. The top machine begins in the Ready state, inserting a coin puts the machine in the Confirm state, and a toggle button (confirm/cancel) begins or ends the brew process, which takes two ticks of time; coffee is dispensed when the machine transitions from Brewing to Ready. The bottom machine models a payment system (or infinitely thirsty coffee drinker), which continuously pays coins and presses the confirm button and is composed (in parallel) with the top machine, where the transitions `coin` and `confirmed` are matched.

Most realistic Stateflow models consist of interacting subsystems; for any verification framework of state machine-like designs to be useful, it must support a notion of parallel compositionality between state machines. For example, our systems require parallel composition to account for the different clock domains of the corresponding systems. To accomplish this, we also include *stutter steps* [9], which are self-transitions that do nothing (we elide these in our figures). We explain the intricacies further in Section 3.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <qspec>
3  <datamodel>
4   <data id="brew" type="int" range="(range 1 20)"/>
5   <data id="coin" type="bool" intent="input"/>
6   <data id="confirmed" type="bool" intent="input"/>
7  </datamodel>
8  <state id="System">
9    <parallel>
10     <sequential> <!-- brewer system -->
11       <initial> <!-- Ready --> ... </initial>
12       <state id="Brewing">
13         <transition label="Brewing_Brewing"
14                     target="Brewing">
15           <guard name="check_brewing"
16            predicate="(< brew 2)"/>
17           <assign location="brew"
18            expr="(+ brew 1)"/>
19         </transition>
20         <transition label="Brewing_Done"
21                     target="Ready">
22           <guard name="check_done"
23            predicate="(= brew 2)"/>
24           <assign location="brew" expr="0"/>
25         </transition>
26         <transition label="Brewing_Confirm"
27                     target="Confirm">
28           <guard name="check_confirmed"
29            predicate="(/\ (~ confirmed)
30                           (< brew 2))"/>
31         </transition>
32       </state>
33       ...
34     </sequential>
35     <sequential> <!-- payment system -->
36       ...
37     </sequential>
38   </parallel>
39 </state>
40 <xi:include href="assertions.qi"/>
41 </qspec>
```

Figure 3. The coffee maker state machine modeled in SCXML, with most state transitions elided.

2.2 QSpec

We developed QSpec because of our need for an extensible language to model our particular flavor of state machines. QSpec was inspired by SCXML [6], but has evolved so it is no longer completely compatible. We show an abridged version of the coffee maker SCXML in Figure 3, but remark that in general, QSpec files are not written by hand.

We also use namespaces and file inclusions to manage the complexity of state machines, as shown in Line 40. We do not show the contents of `assertions.qi` (`qi` short for "Q include"), but they are essentially SCXML representations of

LTL/CTL properties. These properties are described further in Section 2.4. Additionally, the `sequential` portion here simply means a "normal" state machine, which is also known as a *region* or *container* within the parallel composition construct.

In a QSpec, transitions are simply relations on states and model variables with syntactic sugar to express operations like assignment and transition guards. Relations are expressed in a simple first-order logic as predicates over model variables. The logic supports a minimal set of data types including booleans, integers, and sets of symbolic constants (we plan to add support for user-defined types like sums and products). Because the logic of QSpec is so simple, it is easy to translate to both SMV and ACSL using QLang.

2.3 QSpeckler

We mentioned that QSpec models are not written by hand: QSpeckler is the tool that generates QSpec from a particular Stateflow model and LTL/CTL properties about it (which both *are* typically hand-written). The challenge of this translation lies in intricacies of Stateflow; for example, one transformation we must perform is from the MATLAB expression language in Stateflow into the S-expressions required for QLang. In actuality, we use a separate tool, but conceptually this occurs alongside QSpeckler.

Another feature of QSpecker is its test case generation: since it understands Stateflow models, provided a counterexample (that is, an execution where the LTL or CTL properties do not hold for a given SMV model), QSpeckler can generate the corresponding Stateflow test case, which allows feedback to system designers of incorrect behavior, or to system analysts to indicate potential specification bugs.

2.4 LTL and CTL Properties

There are many different safety and liveness properties we may want to state for a given system. We state one safety and one liveness property below in English and CTL. We do not describe the translation from CTL into QSpec, but it is straightforward, only requiring an intermediate conversion to an S-expression. For example,

1. Safety: the coffee maker should never go back to the confirm state when coffee is done brewing. In CTL: `AG !(state = confirm & brew = 2)`.
2. Liveness: provided a coin was inserted, the coffee maker should eventually dispense coffee. In CTL: `coin -> EF (state = ready)`.

We next briefly explain these CTL properties. CTL is a branching-time temporal logic that combines *temporal operators* with *path quantifiers*; a temporal operator describes an execution path in terms of the states along that path, and a path quantifier describes a state in terms of the paths that begin in that state. The path operator `G` means "in each state (Globally)" and `F` means "in some Future state". The path quantifier `A` means "for All paths" and `E` means "there Exists a path". Thus, in the preceding examples, `AG` represents *invariance*—a safety property—and `EF` represents *reachability*—a liveness property. We do not focus on the details of model checking, other than we delegate the model checking to NuSMV, which supports both LTL and CTL properties. More information is available from Clarke et al [14].

2.5 QLang

QLang is a tool that transforms a QSpec specification into 1) an SMV model with temporal properties, 2) a C include file with an ACSL-encoded transition system to validate a C implementation, and 3) a set of first-order *proof obligations* that must hold for the model to be self-consistent and also for the SMV and ACSL outputs to be consistent with each other—that is, the ACSL model is a refinement of the SMV model (See Section 3.2). The proof obligations are checked via direct calls to NuSMV or to Frama-C's back-ends (which are typically SMT solvers) and no other output is generated if they cannot be discharged.

Conceptually and in practice, QLang reduces a QSpec's structured state machines to a more universal "flat" transition system representation according to Q's semantics for those operators. This process yields a (much) larger but semantically equivalent state machine that is easy to output directly as an SMV model and ACSL predicates (see Section 3.2).

In QLang, a "flat" state machine (a set of labeled states and transitions without nesting or parallel composition) is called a *Machine*. The model part of a QSpec (the structured state machine) is called a *Chart* and is an inductively-defined structure that is either the parallel composition of two or more Charts or else a nested composition consisting of a *parent* Machine with a map from each state to zero or one Chart (the *child*). We provide the formal semantics of QSpec in the supplementary repository, but informally, parallel composition is (recursively) defined as the product of its child transition systems, while nesting is defined as a (recursive) embedding of the mapped child transition systems into the parent state. In an embedding, transitions into the parent state are composed with the child's initial transitions, self-transitions on the parent are composed with each of the child's inner transitions, and transitions out of the parent are composed with the child's terminal transitions. In addition we support *abort* transitions, which are composed with every transition and can exit the child machine from any of its states.

The "flattening" process used in QLang grows the size of the state machine exponentially and this is often a practical issue, even for relatively small models with more than two or three parallel states. An SMV input file may, for example, be many gigabytes in size. The advantage of this approach is in its simplicity and resulting clarity of QLang's implementation; we are thus confident that transformed models are correct with respect to QSpec's semantics. Conversely,

```
int foo(void){
  printf("foo");
  return 40;
}
int bar(void){
  printf("bar");
  return 2;
}
int sum(int a, int b){
  return a + b;
}
int main(...){
  return sum(foo(),
             bar());
}
```

```
int main(...){
  register int $69;
  register int $68;
  register int $67;
  $67 = foo();
  $68 = bar();
  $69 = sum($67,$68);
  return $69;
  return 0;
}
```

Figure 4. C (left) has unspecified behavior for the order of evaluation of function arguments; Clight (right) specifies this.

the exponential size increase poses an issue with respect to the scalability of our analysis. One solution we employ is to define composition in QSpec as SMV invariants. In effect, this passes the problem onto NuSMV. In theory, this would make no difference, but in practice this sometimes improves performance. More comprehensively addressing the scalability of our approach via, e.g., assume-guarantee reasoning, is future work (Section 5).

2.6 QFact

QFact is a clang tool which annotates a given C program with its ACSL specification. QFact also generates *frame conditions*, which are additional constraints on the transition between two system states and provide further ACSL specifications. One other issue which complicates verification of C code is its large amount of implementation-defined or unspecified behavior (for example, the size of machine integers). Many discrepancies in C are not interesting from a theoretical and optimization sense, and merely complicate the verification process. To address this, we leverage a simplified C language used in the CompCert C compiler, called Clight. A benefit of Clight is it has a formal semantics [7]. And so, we employ a "trick" to more easily analyze C code without requiring extra effort from the software engineers: we convert from C into Clight, and then back into C again, via a modified branch of CompCert.

There are several differences between C and Clight: unspecified or implementation-defined behavior is made explicit. For example, assignments only exist as statements (and not expressions) and integers are fixed sizes (such as the type `int` is always 32 bits). We show an example in Figure 4. Further, the benefit of a clang plugin is our control over the AST of a C program; this is the perfect place to annotate the C program with the ACSL we need to build a correspondence to QSpec. However, the structure of the C source provided to QFact is somewhat restricted; we discuss this further in Sections 3 and 2.8.3.

2.7 QWorkflow

Now that we have outlined the individual parts of Q, we discuss its usage as a tool. QWorkflow is a collection of scripts used to coordinate the interaction between the different verification approaches (e.g. model checking of the state machine models and Frama-C static analysis of the C implementation). The input to QWorkflow is a configuration file with path information for all the different artifacts needed to run the workflow: requirements documents (Microsoft Word and Visio files), QSpec file(s) for the corresponding Stateflow model under analysis, the CTL and LTL properties file(s), and the C code implementation of the design. These are subsequently used to run NuSMV on the model generated by QLang and Frama-C on the C code with ACSL annotations. Each requirement in the Word documents has a unique identifier and a specified labeling convention is used to reference each of the LTL/CTL properties (which are manually generated). The Stateflow models are also annotated with similar labels. Both of these labels are used by QWorkflow to collect the results obtained with NuSMV and Frama-C and report the status of each requirement in the original Word document. This makes coordinating with the many designers feasible and allows cross-referencing all of the parts of Q.

2.8 Tool Usage

We now describe our usage of existing tools and programming languages.

2.8.1 NuSMV. NuSMV [13] is an open source model checking solver that applies symbolic algorithms [11] based on binary decision diagrams (BDDs) [10]. It supports both LTL and CTL model checking. The key limitations with NuSMV (and with BDD-based model checking in general) are that the model must have a finite state space and that the so-called "state-explosion problem" [14] can lead to intractable model checking problems. This is a well-established problem with model checking, and we discuss ways to address this in Sections 2.5 and 5.

2.8.2 Frama-C. Frama-C is a tool for the analysis of C programs. There are many different *plugins* for Frama-C, which range from simple callgraph visualizations, to abstract interpretation, to deductive provers. We focus on the deductive provers, which are realized with the *Weakest Precondition* (WP) plugin. With WP, the ACSL specifications essentially consist of pre-conditions to be verified (*requires* clauses) and post-conditions to be checked (*ensures* clauses).

One powerful feature of Frama-C is its support for multiple provers: all proof obligations are converted to an intermediate language WhyML and are passed into Why3 [8] (elided

in Figure 1 for simplicity). Why3 then attempts to prove the given goal using one or several different provers.

For our use of Frama-C, we treat API contracts as axiomatic. While this is an opportunity for specification bugs, it allows us the necessary separation between the state machine semantics and the low-level C and hardware interfacing that does not map nicely to statecharts.

One feature of Frama-C that Q uses heavily is the notion of *ghost states*. These allow Frama-C to store variables which are not used in the C code, but are updated along with some C function call or statement. Through this, QSpec statecharts can be aligned with their C implementation. QLang automatically adds these ghost states to the C code, matching them with the correct QSpec variables.

2.8.3 C Coding Standards and Considerations. It is worth mentioning the less interesting, but still equally important, coding considerations to achieve the automatic verification provided by Q. For one, we must describe a mapping from Stateflow into C variables. As mentioned previously, any hardware access (via registers or memory-mapped I/O, for example), must be separated into separate API function calls and axiomatized with ACSL. Additionally, each function may have at most one external observable behavior (i.e., a single volatile variable access). Further restrictions with our tool are that pure functions in these APIs must also be annotated with Frama-C annotations. However, for our state machines we only desire the observable behavior, so relaxing this restriction is feasible and part of our future work.

3 Design

Q decomposes the goal of proving system-level temporal properties into two steps. The first is to prove that the temporal safety properties hold for system specifications given as QSpecs. The second is to prove that a given C program implements (refines) a given component of the QSpec, such that temporal safety properties of the system as a whole are preserved.

As described in Section 2, the first step is completed by generating a transition relation over the states and variables of the system-level QSpec, along with initial conditions and other constraints, and encoding this system as an SMV model. We use NuSMV's unbounded model checking to show the model has the desired system-level temporal properties.

In this section we describe how we accomplish the second step. At a high level, we proceed by automatically generating ACSL function contracts from the C code, and then use Frama-C to prove that the C code implements those contracts. The function contracts are carefully constructed so as to witness the desired refinement (Section 3.1). Crucially, we choose our notions of refinement and composition between the implementation and the state machine model to preserve the temporal properties established in the first step (see Section 3.2). Taken together, these steps ensure that temporal safety properties which are shown to hold for a QSpec system-level specification will also hold for an implementation of that specification.

3.1 Refinement to C

Q Framework is designed to allow compositional reasoning about the observable temporal properties of asynchronously communicating software and hardware components. This entails unifying two very different sorts of compositionality. To reason effectively about asynchronously communicating components, we require a proof relating abstract source and concrete target states; we call this a *refinement*, and require these proofs of refinement themselves to be compositional over asynchronous parallel composition. At the same time, to reason effectively about software components, we need a logic which is compositional over the *sequential* composition of functions and statements for C implementations. In this section we outline some of the key arguments for soundness of our refinement, which is somewhat similar to CompCert's proof of semantics preservation [25] in that it is a weak simulation proof designed to capture refinement of observable behaviors. However the notion of *event* for state machines is quite different from C, so our refinement must be structured differently.

3.1.1 Hoare Logic from Transition System Specifications. Consider a C program fragment f. The behavior of this fragment is defined by its sequence of observable behaviors, and f acts on a program state `ProgState`, which we define as the state of a C program (stack, heap, variable bindings, etc.). Some of these variables may be unbound; these (open) variables in C are realized as `volatile`, which we describe in more detail in this section.

We first provide a definition of *partial correctness* of a program fragment f. Provided a predicate p, we say $s \models p$ if, provided state s, p holds on that state. And now, given fragment f and predicates p, q we define a partial correctness assertion over all program states as:

$$\{p\}f\{q\} := \forall s \in \texttt{ProgState}. \quad s \models p \implies (\forall s' \in \texttt{ProgState}.\, s[\![f]\!]s' \implies s' \models q), \tag{1}$$

where $[\![\cdot]\!]$ is the predicate transformer semantics for f, or its nontermination (hence the *partial* correctness).

Put another way, a proof of $\{p\}f\{q\}$ witnesses that any execution of f, should it terminate, maps the set of states supporting precondition p, $\texttt{supp}(p)$, to the set of states supporting postcondition q, where

$$\texttt{supp}(p) := \{s \in \texttt{ProgState} \mid s \models p\} \in \mathcal{P}(\texttt{ProgState}). \tag{2}$$

Specifically, Frama-C's WP plugin tries to prove whether partial correctness assertions $\{p\}f\{q\}$ are entailed by their weakest precondition $\{\mathrm{WP}(q)\}f\{q\}$. This Hoare logic is compositional only over *sequential* composition of program fragments.

On the other hand, the abstract specifications of our systems are labeled transition systems (LTS). We use the typical definition of LTS as triples $P := (S_P, O_P, \rightarrow_P)$ where S_P is the set of states, O_P the set of labels, and $\rightarrow_P \subseteq S_P \times O_P \times S_P$ the transition relation with the label written above the arrow. We use the letter O for labels to indicate they are the *observables* of the system. A *trace* of P is a sequence of O_P allowed by $\rightarrow_P$. Given an LTS P, we might be interested in a simpler LTS $Q = (S_Q, O_Q, \rightarrow_Q)$ (*not* short for Q Framework) whose behavior *subsumes* P; in this case, any LTL temporal property satisfied by P is also satisfied by Q.[1]

If P subsumes Q, then P is a strict refinement of Q, which we write as $P \preceq_{strict} Q$. The motivation here is it is sometimes easier to prove a temporal property on the simpler model Q and then prove strict refinement. However, a proof of strict refinement requires constructing a *simulation relation* $R \subseteq S_P \times S_Q$ such that any transition in P corresponds to at least one transition in Q, with the same label. The simulation R corresponds to a multifunction—post-composing with $\bigvee$ (join), we obtain a function which we call a *simulation map* from states in the LTS to predicates over states in P, which we denote

$$\varphi_{[R]} : S_P \rightarrow (\texttt{ProgState} \rightarrow \texttt{Prop}).$$

In Q Framework, φ is a JSON file describing the relation of Stateflow variables to predicates over C variables.

It is also convenient to define the support of the simulation map from (2), which are the states that model the predicate $f_{[R]}$:

$$\hat{\varphi}_{[R]} := \text{supp} \circ f_{[R]} : S_P \rightarrow \mathcal{P}(\texttt{ProgState}).$$

By using Frama-C, strict refinement is too strong of a condition: the Frama-C WP plugin cannot prove partial correctness for arbitrary program fragments f, but instead only for functions (this is required for Frama-C's modularity). But our simulation relation R, as defined above, only relates the pre and postconditions and not the intermediate steps the C program (and in turn, the compiled binary) may take.

More precisely, given a transition $p \rightarrow_P^{\alpha} p'$, there may be any number of intermediate program states which have been visited by the program fragment f: that is, we must prove $\{\varphi_{[R]}(p)\} f \{\varphi_{[R]}(p')\}$. These program states do not have a corresponding LTS label, so any refinement must also include the silent transition τ.

Thus, we can only hope to obtain *weak* refinements witnessed by *weak* simulation relations, i.e., some $R \subseteq S_P \times S_Q$ such that

$$P \preceq_{weak} Q := \forall (p, q) \in R, \alpha \in O_P, p' \in S_P. \tag{3}$$
$$p \xrightarrow{\alpha}_P p' \implies \exists q' \in S_Q. \left(q \xrightarrow{\tau^\star}\xrightarrow{\alpha}_Q\xrightarrow{\tau^\star} q' \wedge (p', q') \in R \right),$$

where we concatenate the silent observations τ to our collection of observations, and $\tau^\star$ is a composition of an arbitrary number of transitions under the silent transition.

3.1.2 Observables in Hoare Logic Using Ghost State.

At this point, we cannot yet prove a weak simulation between LTS and C using Q Framework, since the Hoare logic we defined in (1) does not capture any notion of observations. Suppose our program interacts with its environment through some memory-mapped input/output (I/O) port or a value accessed by an asynchronously interrupting function—by the C standard, such interactions should be through variables declared `volatile`. The C standard specifies that volatile variable accesses are *observable events*, or *side effects*, and as such, like termination (or lack thereof), must be preserved for any semantics-preserving transformation.

However, at every sequence point, (semicolon in C) the value of a volatile variable may be modified by unknown factors, so the value of a volatile variable upon exit of a function tells us nothing about the value observed at the time of the volatile variable access—this is observable behavior of the function not reflected in program states. The underlying problem is that volatile variables in embedded systems correspond to *open* variables and so refinement proofs must take place in a context.

The solution to this problem is accomplished through ghost state, whose evolution can be specified through Hoare logic annotations of functions thanks to how CompCert renders observable side effects into events [24]. Specifically, the `event` type in CompCert is constructed from system calls, variable loads, variable stores, and annotations. When compiling C into CompCert's Clight, volatile accesses are compiled into system calls.

For example, suppose we have declared a global variable `volatile uint8_t fgetCVal` which our program accesses through the global pointer `volatile uint8_t *fgetC`. Then the assignment `uint8_t c = *fgetC;` gets compiled into Clight as

```
$1 = volatile_load_uint8_t_(fgetC);
```

We give an axiomatic model of the sequence of observations (`obs`) at volatile memory location `fgetC`. Frama-C annotations are indicated with a comment beginning with the `@` symbol.

```
/*@ghost int obs_t;
axiomatic model {
  type obs;
  logic obs obs_at(integer t);
  logic uint8_t fgetCObs(obs o); } */
```

We then axiomatize the sequence of observations through a Hoare triple for `volatile_load_uint8_t` with `obs_at` representing a sequence of values read from `fgetC`:

[1] Q subsumes P here means any acceptable trace in P can also be a trace in Q, provided a correspondence between observables in P (such as volatile variable reads and writes) and observables in Q (the O_Q).

```
/*@
requires \valid(unsigned char volatile *v);
requires fgetC == v;
ensures obs_t == \old(obs_t) + 1;
ensures \result \in (0 .. 255);
ensures \result <==>
  fgetCObs(obs_at(\old(obs_t))); */
uint8_t *volatile_load_uint8_t_(uint8_t *v);
```

And so, the predicates from propositions over `fgetCObs` are no longer strictly over `ProgState`, but are now over `ProgState` × `GhostState`.

So far, we have have equipped our LTS $Q = (S_Q, O_Q, \to_Q)$ with a simulation map from LTS states to predicates over the `ProgState` of a C program. That is, Q has support $\hat{\varphi}_{[R_{S_Q}]} : S_Q \to \mathcal{P}(\texttt{ProgState})$. Henceforth, we refer to the C progrm as P_C and the LTS derived from the QSpec as Q. Suppose we are given, in addition, such a map from specification observations O_Q to predicates over `GhostState`, i.e., taking support, $\hat{\varphi}_{[R_{O_Q}]} : O_Q \to \mathcal{P}(\texttt{GhostState})$.

With these, we can now generate from $\to_Q$ the partial correctness assertions which could prove $P_C \preceq_{\text{weak}} Q$. Let `EnvProp` be propositions over `ProgState`×`GhostState`. Then

$$\begin{aligned}(\to_Q) &\to (\texttt{EnvProp},\ \texttt{EnvProp})\\ (s, o, s') &\mapsto \left(\varphi_{[R_{S_Q}]}(s),\ \varphi_{[R_{O_Q}]} \wedge \varphi_{[R_{S_Q}]}(s')\right).\end{aligned}$$

We call the product of a predicate (`EnvProp`) and a program location (a `PLoc` matching clang's notion of program location) an *execution context*: `ExecCtxt` = `EnvProp` × `PLoc`. We ask of the user, alongside their specification of the abstract model Q, the relations R_{S_Q} and R_{O_Q} to relate observables in Q and observables in `ExecCtxt`. From our simulation map $\varphi_{[R_{O_Q}]}$, we select the `EnvProp` and `PLoc` from them, which we denote via a superscript. Then the partial correctness assertion associated with $(s, o, s') \in \to_Q$, is

$$\begin{aligned}&(s, o, s') \mapsto \qquad (4)\\ &\left\{\varphi^{\texttt{EnvProp}}_{[R_{S_Q}]}(s)\right\}\left(\varphi^{\texttt{PLoc}}_{[R_{S_Q}]}(s)\right)\left\{\varphi^{\texttt{EnvProp}}_{[R_{O_Q}]}(o) \wedge \varphi^{\texttt{EnvProp}}_{[R_{S_Q}]}(s')\right\}.\end{aligned}$$

In this way, we compile from the simulation map, for every function, the Frama-C annotations. Since we moreover assert that these are the only behaviors (using the Frama-C annotation `complete behaviors`) for each function, if Frama-C succeeds in proving all these Hoare triples, we have obtained a proof that

$$\begin{array}{ccc} O_Q & \xrightarrow{\quad\to_Q\quad} & \mathcal{P}(S_Q \times S_Q) \\ {\scriptstyle \hat{\varphi}_{[R_{O_Q}]}}\downarrow & \subseteq & \downarrow{\scriptstyle \hat{\varphi}_{[R_{S_Q}]}} \\ \mathcal{P}(\texttt{GhostState}) & \xrightarrow[\to_{P_C}]{} & \mathcal{P}(\texttt{ProgState} \times \texttt{ProgState}) \end{array}$$

That is, the proofs of all the Hoare triples in (4) shows the ghost state-indexed program state transformer semantics $\to_{P_C}$ defined by the Hoare triples given by $(\varphi_{[R_{O_Q}]}, \varphi_{[R_{S_Q}]})$, thought of, itself, as a transition system, witnesses a simulation, as $(\hat{\varphi}_{[R_{O_Q}]} \circ \to_{P_C}) \subseteq (\to_Q \circ \hat{\varphi}_{[R_{S_Q}]})$. This amounts to (3), if we can think of $\to_{P_C}$ as a transition system. But the labels in a transition system are discrete sets, whereas already in Figure 3 Line 23, `(= brew 2)`, is atomic, while the `check_brewing` predicate `(< brew 2)` *is not atomic*, as `(< brew 1)` $\implies$ `(< brew 2)`.

So how *can* we think of $\to_{P_C}$ as a transition system? At this point, we should remind ourselves that the observations specified in Figure 3 are *predicates*, so our map $\varphi_{[R_{O_Q}]}$ isn't a map amogst discrete sets of labels, but amongst lattices of predicates. This extra structure puts some constraints on $\to_Q$, namely

$$\frac{\beta \implies \alpha \quad p \xrightarrow{\alpha} p'}{p \xrightarrow{\beta} p'} \qquad \frac{p \xrightarrow{\alpha} p' \quad p \xrightarrow{\beta} p'}{p \xrightarrow{\alpha \vee \beta} p'} \qquad \frac{}{p \xrightarrow{\bot} p'}.$$

Thus $\to_Q$ must be a Galois connection from the lattice O_Q to the lattice $\mathcal{P}(S_Q \times S_Q)$, and $\varphi_{[R_{O_Q}]}$ must be monotonic for $\to_{P_C}$ to inherit this property.

And so, we encourage the user to give R_{O_Q} as a relation between *atomic* predicates in the specification and arbitrary predicates over C program and ghost state. In Q Framework, we first do a syntactic check on the given simulation relations to detect whether any specification predicates are not atomic. When they fail to be atomic, we test for implications amongst predicates via Z3, and generate ACSL side obligations which, if discharged by Frama-C, show implication of their images along $\varphi_{[R_{O_Q}]}$, i.e. $\forall \alpha, \beta \in O_P.\ (\beta \implies \alpha) \implies (\varphi_{[R_{O_Q}]}(\beta) \implies \varphi_{[R_{O_Q}]}(\alpha))$.

We therefore ask the user to specify the simulation relation with the following data:

1. A mapping between the states of the model and states of the implementation, called the *simulation relation*, $R \subseteq S_Q \times \texttt{ProgState}$;
2. the behaviors that the state machine performs which are considered observable, $Obs \subseteq O_Q$; and
3. a mapping between observables and terms in the C implementation, $Obs \to \texttt{EnvProp}$.

If Frama-C can discharge these side obligations along with the Hoare logic obligations synthesized from the abstract specification are true and complete, then we have a simulation map with the requisite structure.

Now, we have this simulation map for the two implementation details, the `EnvProp` and `PLoc`. Provided

$$\begin{aligned}&\forall (s, o, s') \in S_Q \times O_Q \times S_O.\\ &\quad (s, s') \in \xrightarrow{o}_Q \implies\\ &\quad \left\{\varphi^{\texttt{EnvProp}}_{[R_{S_Q}]}(s)\right\}\left(\varphi^{\texttt{PLoc}}_{[R_{S_Q}]}(s)\right)\left\{\varphi^{\texttt{EnvProp}}_{[R_{O_Q}]}(o) \wedge \varphi^{\texttt{EnvProp}}_{[R_{S_Q}]}(s')\right\},\end{aligned}$$

we have shown that the specification *weakly* simulates the implementation, i.e. $P_C \preceq_{\text{weak}} Q$. We obtain inclusion of

observable traces, although up to the relation amongst observations given by R_{O_Q}.

3.2 Refinement and Composition

In the preceding section, we saw that the Hoare logic for simulation cannot be synthesized without first building the accompanying ghost state which axiomatizes the behavior of `EnvProp`. We therefore axiomatize the behavior of volatiles to reflect their *external* non-determinism.

Moreover, volatile reads in C lack any guarantee of freshness. This means it is possible for the C program to observe *stale* values. Such behavior is modeled by the *asynchronous* composition of components. Given $P = (S_P, O_P, \rightarrow_P)$ and $Q = (S_Q, O_Q, \rightarrow_Q)$, $P \| Q = (O_P \cup O_Q, S_P \times S_Q, \rightarrow_{P\|Q})$, $\rightarrow_{P\|Q}$ is the smallest closure of

$$\frac{p \xrightarrow{\alpha}_P p'}{(p,q) \xrightarrow{\alpha}_{P\|_{\text{async}}Q} (p',q)} \qquad \frac{q \xrightarrow{\alpha}_Q q'}{(p,q) \xrightarrow{\alpha}_{P\|_{\text{async}}Q} (p,q')}.$$

For $\|_{\text{async}}$, we are interested in an even weaker notion of refinement. The relations given by the graphs of the projections $R_{\pi_{P,Q}} = \text{graph}(\pi_{P,Q}) \subseteq ((R_P \times R_Q) \times R_Q)$ witness neither strict nor weak simulation relations, unless we have the necessary self-transitions:

$$\forall \alpha \in O_P \cup O_Q, p \in S_P, q \in S_Q.\ p \xrightarrow{\alpha}_P p \text{ and } q \xrightarrow{\alpha}_Q q.$$

The most natural of these, which we sketch out the proof of here, is the refinement of TLA specifications [21]. If S_L and S_R are stuttering invariant properties of LTSs L and R stipulating, at a given state, which actions are enabled, then the property applied to $S_L \|_{\text{async}} S_R$ is simply the conjunction $S_L \wedge S_R$. Moreover, this means refinements are implications, up to the existence of traces of hidden variables and refinement mappings. Put more precisely, given abstraction Q, refinement P, and hidden variables H, we say $P \implies \exists H.\ Q$ when $P \preceq_{\text{TLA}} Q$.[2] Then the compositionality of refinement results from the universal property of conjunction, (namely, $S_L \wedge S_R$ is the most abstract common refinement of S_L and S_R). Fortunately, $\preceq_{\text{weak}} \implies \preceq_{\text{TLA}}$, so we can use the refinement proof from the preceding section to reason compositionally about system-level properties of implementations.

The key insight here is we model all external observables in C as a sort of "external" transition system, with a single state and (potentially) many actions, which we denote $\mathbf{1}$; from this, we can prove (via Frama-C) that $\mathbf{1}_{O_Q}$ is a refinement of the external observables in the C (e.g., volatile reads and writes), which we illustrate in Figure 5. Equipped with refinements of $\mathbf{1}_{O_{P_C}} \preceq_{\text{weak}} \mathbf{1}_{O_Q}$ and $P_C \preceq_{\text{weak}} Q$, all that remains is to prove their composition is also a refinement. The proof relies on the universal property of $\|$, the universal property of $\mathbf{1}$, and a Galois connection of the lattices of observables O_{P_C} and O_Q which gives $\mathbf{1}_{O_{P_C}} \sim \mathbf{1}_{O_Q}$. It is simple to extend this refinement proof to, e.g., the SMV back-end, by equipping all basic transition systems with self-transitions under all of the other transition system's labels.

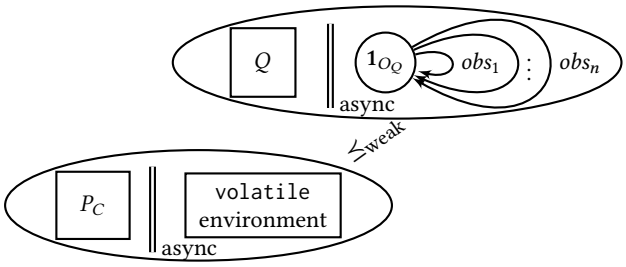

Figure 5. Asynchronous composition of an LTS Q with $\mathbf{1}$ to show an abstraction of the C implementation.

This refinement allows us to prove safety properties of the C program and its interaction with the (`volatile`) environment. Though we can use NuSMV to prove liveness properties of the model, to prove liveness in the implementation we need some notion of *fairness* with asynchronous composition. Proving systems without an assumption of fairness, or proving that asynchronous systems are fair is future work. For a more complete discussion, we refer to the supplementary material linked in Section 1.

4 Related Work

Model checking has a long history in formal verification of software systems [1, 11, 12, 18]. Well-known industrial uses of model checking gain value with models that are divorced from implementation [27]. These use-cases help write correct code, but in our setting we aim to go one step further and take invariant properties proven for the model and ensure they apply to their implementation.

Tools like SLAM [5] have had significant impact in industrial uses by checking for proper integration of device drivers with the Windows kernel. More broadly, model checking programs directly is a well studied technique [23]. These approaches assume the behavior of the larger system is encoded soundly in assumptions of their specifications. For example, in the case of SLAM's driver verification tool SDV, they specify a set of API usage rules that can be seen as approximating environmental behavior and constraints. By contrast in our approach we use our theory of refinement (Section 3.1) make no assumptions in the behavior of the environment. In practice this means that our state machine model uses variables that are unbound and conceptually act as the interface between the specification and its environment. In C, these are volatile variables which are used as a communication medium by other system components and do not have unbounded behavior. From the perspective of state machines, there has been recent work verifying properties about only the Simulink models via SMT [20]. Conversely, our models exist separately from their implementations, so

[2] The $\exists$ refers to the temporal existential operator.

an important feature of Q Framework is the link between models and implementation.

The work most similar to Q Framework is Trillium [29], which permits refinement proofs between a higher-order distributed separation logic (TLA) and an concrete implementation language (AnerisLang). Trillium has the benefit of having a fully mechanized proof of correctness in Coq, but its implementation language is not a general purpose. Therefore, the tradeoff of Q Framework is less formalization (using NuSMV and Frama-C) in exchange for the flexibility of C as the concrete implementation language.

Several works have explicitly aimed to bridge the gap between state-machine-like specifications and real implementations. Broadly they have focused on generality, where it is up to the user to build a simulation proof between the program and its specification. As a consequence they require a large amount of user intervention. In the case of Ironfleet [19] a separate intermediate refinement in the form of a protocol must be designed and proved. In the case of DeepSpec, [30] a "linear" specification is designed along with an intermediate "implementation" specification. The coinductive ITree specifications are infinite state while ours are infinite-state with finite representation as practical matter for checking temporal properties against our model. Similar to Ironfleet, refinement is demonstrated through the intermediate specification but here the proof takes place in the Coq proof assistant and the final refinement to C is demonstrated using the Hoare logic at the heart of the Verified Software Toolchain [3]. The foundational nature of proofs in DeepSpec are notable because semantics underlying VST for C come from the CompCert compiler and are verified in Coq. As a result, the proofs are carried all the way down to assembly generation.

By contrast, we have aimed to facilitate automation of refinement proofs for programs fitting a particular form. With respect to DeepSpec, the key ideas and the architecture of our tool are such that we can produce VST obligations to provide similar foundational guarantees via a new back-end and this is planned as future work.

5 Future Work

Q Framework is a mature enough project that it sees industrial use-cases at Sandia today. However, it is just one part in our ultimate goal (similar to the DeepSpec project), to have "One Q.E.D."—a single proof of correctness, from the functional (or state-machine) specifications, to the high-level programming language implementation, to the generated binary, all the way down to the hardware being executed. To this end, we wish to extend the Q Framework for hardware verification, instead of treating access to the hardware (or ISA) as axiomatic in ACSL.

As mentioned in Section 2.5, flattening models can grow QSpec (and their corresponding SMV) to be too large to check. We are currently working on adding support within QLang for more efficient ways of encoding the state machine operators within SMV and ACSL, while preserving their semantics. Beyond this, one manual part of Q Framework is the decomposing and tracking the assumptions of each interacting component. For large models, we have manually deconstructed the systems to be able to use assume-guarantee reasoning [15]. To automate this process, we are investigating circular assume-guarantee [17] reasoning for Q, which would automatically build a set of assumptions required for compositional model checking.

As mentioned in Section 3, one limitation of Q is its strict requirements on the structure of the C implementation and the ACSL annotations Q expects. However, we are interested in using the Verified Software Toolchain's (VST) [3] symbolic executor to automatically generate the ACSL specifications to allow more complex functions to be annotated automatically with ACSL. Lastly, we plan to extend our notion of modularity one step futher: we plan to extend Q to allow verification of both nested and parallel composition of state machines. This would further expand the class of state machines, and corresponding C code, that can be verified.

6 Conclusion

We presented Q Framework, a verification framework to verify the correctness of digital control systems. Q was designed around the idea that high-consequence embedded control software has complex requirements, and that it is worth significant effort to ensure the software upholds these requirements.

Q works by linking together state machines (expressed in Stateflow) with a source code implementation (in C), and proving an implementation is a refinement of the model and that it obeys some set of requirements expressed as temporal properties. This allow us to verify deep temporal properties about systems and their concrete implementations, provided that these implementations are written in a restrictive coding style that matches very closely the Stateflow model. Q is used at Sandia by our team of approximately 10 people, who work with several small groups of system designers and software developers for several embedded system. We found that state machines elicit modularity from digital designers; this modularity, combined with our formal analysis, can often be translated into opportunities for refinement proofs, and in turn, better scalability of analysis.

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND No. SAND2022-12167 C.

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12, 6 (2010), 447–466. https://doi.org/10.1007/s10009-010-0145-y

[2] Charles André and Frédéric Mallet. 2009. Specification and Verification of Time Requirements with CCSL and Esterel. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Dublin, Ireland) *(LCTES '09)*. Association for Computing Machinery, New York, NY, USA, 167–176. https://doi.org/10.1145/1542452.1542475

[3] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems (ESOP/ETAPS (LNCS 6602))*. Springer-Verlag, Saarbrücken, Germany, 1–17. http://dl.acm.org/citation.cfm?id=1987211.1987212

[4] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: The Science of Deep Specification. In *Verified Trustworthy Software Systems (Philosophical Transactions of the Royal Society A)*. The Royal Society, London, UK. http://doi.org/10.1098/rsta.2016.0331

[5] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/b96106

[6] Jim Barnett, Rahul Akolkar, R. J. Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T. V. Raman, Klaus Reifenrath, No'am Rosenthal, and Johan Roxendal. 2015. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Technical Report Version 1.0. WC3: The World Wide Web Consortium. Available at https://www.w3.org/TR/scxml/.

[7] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43 (Oct. 2009), 263–288. Issue 3. https://doi.org/10.1007/s10817-009-9148-3

[8] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64. https://hal.inria.fr/hal-00790310.

[9] M.C. Browne, E.M. Clarke, and O. Grümberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59, 1 (1988), 115–131. https://doi.org/10.1016/0304-3975(88)90098-9

[10] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[11] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98 (1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

[12] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. The TLA+ Proof System: Building a Heterogeneous Verification Platform. In *Theoretical Aspects of Computing (ICTAC 2010)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 44. https://doi.org/10.1007/978-3-642-14808-8_3

[13] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*. Springer-Verlag, Berlin, Heidelberg, 359–364. https://doi.org/10.1007/3-540-45657-0_29

[14] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. 2018. Introduction to Model Checking. In *Handbook of Model Checking*. Chapter 1, 1–26. https://doi.org/10.1007/978-3-319-10575-8_1

[15] E. M. Clarke, D. E. Long, and K. L. McMillan. 1989. Compositional Model Checking. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 353–362. https://doi.org/10.1109/LICS.1989.39190

[16] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods (SEFM (LNCS 7504))*. Springer, Thessaloniki, Greece, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[17] Karam Abd Elkader, Orna Grumberg, Corina S. Păsăreanu, and Sharon Shoham. 2015. Automated Circular Assume-Guarantee Reasoning. In *FM 2015: Formal Methods*. Springer International Publishing, Cham, 23–39. https://doi.org/10.1007/978-3-319-19249-9_3

[18] E. Allen Emerson. 2008. *The Beginning of Model Checking: A Personal Perspective*. Springer Berlin Heidelberg, Berlin, Heidelberg, 27–45. https://doi.org/10.1007/978-3-540-69850-0_2

[19] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

[20] Daisuke Ishii, Takashi Tomita, Toshiaki Aoki, The Quyen Ngo, Thi Bich Ngoc Do, and Hideaki Takai. 2022. SMT-Based Model Checking of Industrial Simulink Models. In *Formal Methods and Software Engineering*. Springer International, Cham, 156–172. https://doi.org/10.1007/978-3-031-17244-1_10

[21] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923. https://doi.org/10.1145/177492.177726

[22] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 364 pages.

[23] Rustan Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *16th International Conference for Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[24] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[25] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[26] N. G. Leveson and C. S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (July 1993), 18–41. https://doi.org/10.1109/MC.1993.274940

[27] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[28] The MathWorks, Inc. 2022. Stateflow: Model and Simulate Decision Logic Using State Machines and Flow Charts. Available at https://www.mathworks.com/products/stateflow.html.

[29] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2021. Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic. arXiv. Available at https://arxiv.org/abs/2109.07863.

[30] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32