SAND2022-12167C

# 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

Jon M. Aytac*
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 V. Morris*
Sandia National Laboratories
Livermore, California, USA

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

## Abstract

We present Q: a verification framework used at Sandia National Laboratories. The Q framework is a collection of tools used to verify safety and correctness properties of high-consequence embedded systems and is designed to address the issue of scalability which plagues many formal methods tools. 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, Jon M. Aytac, John Bender, Geoffrey C. Hulette, Raheel S. Mahmood, Karla V. Morris, and Blake C. Rawlings. 2022. Q: A Sound Verification Framework for Statecharts and their Implementations. In *Proceedings of Formal Techniques for Safety-Critical Systems (FTSCS '22).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

*These authors contributed equally to this research.



*FTSCS '22, 7 Dec 2022, Auckland, NZ*



ACM ISBN XXX-XXXX-XXXXX/XXX…$15.00
https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Sandia National Laboratories develops software for high-consequence digital control systems. With embedded control systems, bugs can have disastrous consequences [23]. 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.

Second, our approach must integrate into existing engineering 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 [25]) 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 ensure a clean separation of hardware interfacing (via API) and enforce coding standards (such restricting what state functions may modify 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 contstraints of our problem space. Verifying state machine abstractions of systems in modeling languages such as TLA+[20] have shown

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.

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 [4], but these approaches do not support sound compositional reasoning beyond an abstract specification of external behavior.

Lastly, significant work has been done to enable manual proofs of labeled transition system specifications against an implementation but the manual, time-intensive, nature of these approaches and their sensitivity to code changes would require more time and resources than we have to dedicate [3, 19].

To address these gaps in the research we developed the Q Framework, which compiles Stateflow diagrams to an intermediate representation, and then both to SMV for model checking [15] 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 discharged and 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 compositionality of state machines, and some mathematical arguments 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).

## 2 Architecture

We now describe the Q framework at a high level. Figure 1 describes the overall architecture of Q, but before diving into the architecture in detail, we first give some context. The flow 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 with 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 later in this section). This whole workflow is managed by QWorkflow, a set of scripts that run model checking and static analysis tools and compile results describing the coverage of the requirements. The final outputs of the Q Framework 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

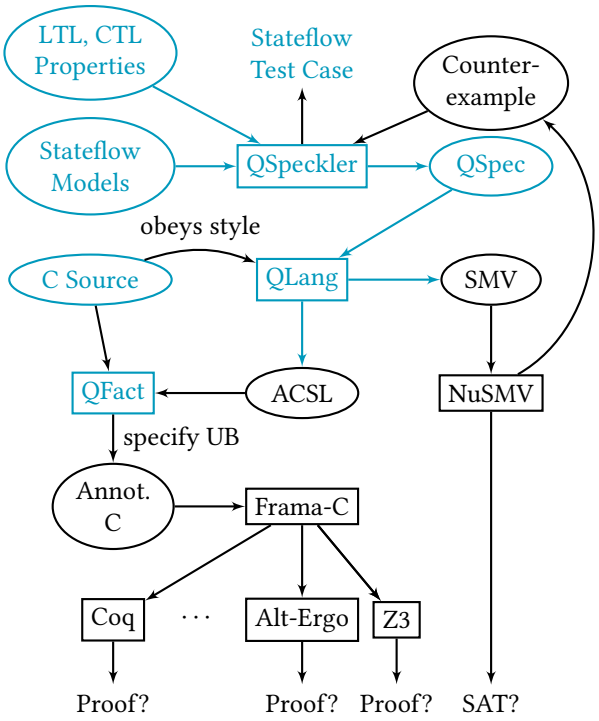


**Figure 1.** Architectural overview of Q, managed in general by QWorkflow. Ellipses are inputs, rectangles are tools, blue text are developed by Sandia, and UB refers to both *unspecified behavior* and *implementation-defined behavior*.

the state machine models obey the LTL/CTL properties (via NuSMV).

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, as well, for if the SMV model does not obey the desired properties, it 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 illustrative model of a "secure coffee maker." At first blush, this example seems somewhat contrived. However, the compositionality of system designs 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 those compatible for QLang.
§ 2.4 LTL and CTL properties.

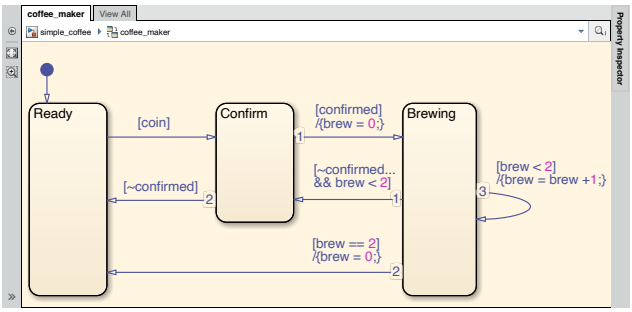


**Figure 2.** Stateflow model of a coffee maker with three states.

§ 2.5 QLang: the compiler from QSpec Statecharts into an SMV model.
§ 2.6 QFact: a clang plugin to add ACSL annotations to C code, as well as perform code transformations to enable verification.
§ 2.7 QWorkflow: scripts use to orchestrate the interaction of the different verification approaches.
§ 2.8.3 : 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 domain knowledge of the system and the C code implementation. While the Stateflow models and LTL/CTL properties require some expertise in which properties can be formalized and proven, in our experience, system analysts need not be formal methods experts to use Q. We provide an example of a Stateflow model in Figure 2. It 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.

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 with different clock rates 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 of compositionality further in Section 3.

### 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 [5], and 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 32. 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 a potential parallel composition construct.

```
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  <sequential>
9    <initial> <!--ready--> </initial>
10   <state id="Ready"> <!--transitions--> </state>
11   <state id="Confirm"> <!--transitions--> </state>
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" expr="(+ brew 1)"/>
18     </transition>
19     <transition label="Brewing_Done"
20                 target="Ready">
21       <guard name="check_confirmed"
22        predicate="(= brew 2)"/>
23       <assign location="brew" expr="0"/>
24     </transition>
25     <transition label="Brewing_Confirm"
26                 target="Confirm">
27       <guard name="check_confirmed"
28        predicate="(/\ (~ confirmed) (< brew 2))"/>
29   </transition>
30   </state>
31 </sequential>
32 <xi:include href="assertions.qi"/>
33 </qspec>
```

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

### 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 our secure coffee maker. 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.

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: the coffee maker should eventually be able to reach the confirm state. In CTL: `EF (state = confirm)`.

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 properites. More information is available from Clarke, Henzinger, and Veith [15].

### 2.5 QLang

QLang is a software tool that soundly 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 (refinement is further discussed in 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 "children"). We omit the formal semantics for space; roughly speaking, 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.

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 this logic is so simple, it is easy to translate to both SMV and ACSL.

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. SMV output, for example, is sometimes 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. We are currently working on adding support within QLang for more efficient ways of encoding the state machine operators within SMV and ACSL, while keeping the semantics equivalent.

### 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

```c
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());
}
```

```c
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.

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, for example, assignments only exist as statements (and not expressions), and all unspecified or implementation-defined behavior is made explicit; 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 C source input 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 use 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 subsequesntly 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 (not shown in Figure 2). 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 [14] is an open source model checking solver that applies symbolic algorithms [12] based on binary decision diagrams (BDDs) [11]. 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" [15] can lead to intractable model checking problems even when only relatively few components are combined in the system to be analyzed.

**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 systems-level C and hardware interfacing that does not map nicely to statecharts.

There are more features of ACSL, such as user-defined functions, assertions, and axioms, but these all help towards the goal of proving post-conditions hold given a set of pre-conditions. One feature that Q uses heavily is Frama-C's support for *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. Thus, 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. Further restrictions with our tool is 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, which are hierarchical compositions of state machines (see Section 2.2). The second is to prove that a given C program implements (refines) a given component within the QSpec, called the "program component," 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 bounded model checking to show that the model has the desired system level temporal properties.

In this section we focus on how we accomplish the second step. At a high level, we proceed by automatically generating ACSL function contracts from the program component, 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 such that the system composed of the program component and the rest of the system preserves 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

We show that a QSpec model refines the program component inductively through a proof of simulation. A simulation proof requires a relation between abstract source and concrete target states; the user must provide this relation and Q provides a syntax for doing so.

Simulation is defined on transition systems and we cannot analyze a C program as a transition system directly. Instead, we synthesize separation logic conditions to "lift" the C program's behavior with respect to observables at function boundaries into a transition system. These conditions are designed to be discharged by Frama-C's weakest precondition (WP) plugin [6]. Effectively, we view the implementation semantics via a predicate transformer semantics, in which states are a pair of a program location and sets of execution states of the C program. These sets of states are the atomic propositions of ACSL predicates, while the program locations are the entry and exit points of function calls.

More precisely,

$$ProgState = EnvProp \times PLoc, \tag{1}$$

where

$$EnvProp \simeq \mathcal{P}(ExecState) \tag{2}$$

is a predicate on the execution state of the C program (that is, an ACSL predicate) and *PLoc* is a program location.

Because we annotate terms in C which may represent more than one state transition in the C semantics (practically, the terms are the whole of a function body due to the limitations of Frama-C) *the C term may visit many intervening states where the model would make a single transition*, so proof of the ACSL contracts must demonstrate a *stuttering backward simulation* [10] between the implementation and the model:

$$\begin{aligned} &Impl \preceq Model ::= \\ &\quad \forall o \in Obs, t_1\, t_2 \in Imp, \exists\, s_1\, s_2 \in Model, \\ &\qquad t_1 \xrightarrow{o} t_2 \\ &\qquad \Rightarrow s_1\, R\, t_1 \\ &\qquad \Rightarrow s_1 \xrightarrow{\star} \xrightarrow{o} \xrightarrow{\star} s_2 \wedge s_2\, R\, t_2, \end{aligned}$$

where and $t_1, t_2$ are states in the implementation *Imp* and $s_1, s_2$ states in the model. Note that both the *Imp* and model transitions must share the same member $o$ from the set of observables *Obs*. The source is allowed "wait" for the related C term, complete its transition and then wait further for the C term to complete. As we will discuss in Section 3.2, separation logic over program states alone cannot express the notion of observable, so Q uses "ghost state" to capture externally observable behaviors and match them with the model.

In contrast to prior efforts in this vein (see Section 4), we aim to minimize the proof effort required of the user. To achieve this we require the user to explicitly articulate the following aspects of the simulation relation $R$:

1. A mapping between the states of the model and states of the implementation, a *simulation relation*, $R \subseteq State \times ProgState$ (the same *ProgState* from (1)).
2. The behaviors that the state machine performs which are considered observable, *Obs*
3. A mapping between observables and terms, $map : Obs \longrightarrow EnvProp$.

The user provides this information to Q in a simple JSON format.

We fold over this structure to construct a map of states $\varphi_{State} : State \to \mathcal{P}(ProgState)$ and a map of observables $\varphi_{Obs} : \mathsf{Exp}(Obs) \to EnvProp$. [1]

Thus we obtain a map from the transition relation to separation triples:

$$State \times \mathsf{Exp}(Obs) \times State \to \mathcal{P}(ProgState \times ProgState) \quad (3)$$

of the form

$$(s, l, s') \mapsto \Big\{ (\phi_{pre}, \phi_{Obs} \wedge \phi_{post}) \;\Big|\; \phi_{pre} \in \varphi_{State}(s), \; \phi_{Obs} = \varphi_{Obs}(l), \; \phi_{State} \in \varphi_{State}(s') \Big\}.$$

Since $ProgState = EnvProp \times PLoc$, this gives a set of pre-condition/post-condition pairs of predicates and program locations. As WP is aimed at the predicate transformer semantics of functions, this will result in the two clauses

$$\texttt{assumes envProp}(\phi_{pre});$$
$$\texttt{ensures envProp}(\phi_{Obs} \wedge \phi_{post});$$

in a named behavior for the function at program location $\texttt{pLoc}(\phi_{pre})$. We clarify here our notation: the typewriter font

$$\texttt{envProp} : ProgState \to EnvProp \text{ and}$$
$$\texttt{pLoc} : ProgState \to PLoc$$

simply refer to the canonical projections out of the product type *ProgState*.

These annotations are merged into ACSL specifications for the behavior of every function in the C program. QFact then absorbs these specifications and the C program's Makefile, normalizes the code via the C to Clight to C transpilation, annotates the normalized code with the synthesized specifications, does some simple static analysis to generate the necessary `assigns` and `requires` annotations. Every function is annotated with the ACSL `complete behaviors;` annotation, so that proof of the ACSL obligations gives a proof that every observed behavior of every function in the C program corresponds, up to stuttering, via the simulation relation, to a behavior allowed by the specification, completing our proof of stuttering simulation.

### 3.2 Refinement and Composition

The proofs of the full QSpec's temporal properties combined with the proof that the C program refines the program component to together yield a proof of the temporal properties for the system implementation. There is a subtlety in this argument, however. The C program is shown to meet its specification *as a sequential program*, and the system correctness properties are correctness *as a distributed system*.

Recall in Section 3.1, the Hoare triple $\{P\}f\{Q\}$ (and their corresponding separation triple (3))) witnesses only that, in a state satisfying precondition $P$, the function call may visit arbitrarily many states before returning in a state satisfying postcondition $Q$, or else it may never terminate at all. Thus Hoare triple partial correctness assertions can only support *stuttering* simulation relations[2].

So proof of the Hoare triples witnesses a stuttering refinement between the observable behavior of the abstract C specification as a labeled transition system $C_A$ and the observable behavior of the C program $C_C$; we denote this refinement relation $C_A \succeq C_C$. However, while the alphabet of observables is part of the data of $C_A$ as a labeled transition system, it is not directly present in Hoare logic. Q plays a key role in bridging this gap.

We described above how we map from transitions into Hoare triples factored through a map from expressions over observables $\varphi_{Obs} : \mathsf{Exp}(Obs) \to EnvProp$ into pairs of program states $\mathcal{P}(ProgState \times ProgState)$. This map into ACSL expressions alone does not suffice, however, since the notion of external *observables*, fundamental to the specification as a labeled transition system, has no corresponding notion in the Hoare Logic, where we have only predicates over the *state* of the C program. This problem is solved by writing predicates over an extension of the state of the C program by *ghost state*, which is used to axiomatize the behavior of these observables.

For instance, a C function may interact with memory-mapped I/O through a volatile variable. The C standard specifies the semantics of volatile variable accesses to be *completely non-deterministic*. As every interaction with a volatile variable is observable and side-effectful, normalization into Clight will replace every volatile variable access with a function call. The user must annotate this function call with pre-conditions and post-conditions modeling the effect of accessing the volatile as changes to global ghost state. For volatile variables, the only suitable axiomatization is the totally non-deterministic relation. Thus, the map $\varphi_{Obs}$ used to rewrite transitions into Hoare triples is more fully understood as a map into *predicates over the product of the C program's execution state and the ghost state* $\varphi_{Obs} : \mathsf{Expr}(Obs) \to (ExecState \times GhostState \to Prop)$. Thus our Hoare triples are never over the program state alone. We consider now what this means for our simulation proof.

The non-deterministic model of observables in global ghost state amounts to composing $C_C$ with the most abstract

[1] Since our refinement argument factors through a map of an alphabet constructed inductively from the data of a mapping from expressions over observables to ACSL predicates, we have a side obligation, discharged automatically via Z3 [17], that the map is indeed a Galois connection of term languages.

[2] In fact, divergence-blind stuttering simulation relations, and our $\preceq = \preceq_{dbss}$. Proof-carrying C compilers [22] preserve *termination improving* stuttering simulation $\preceq_{tiss}$, which is subsumed by divergence-blind stuttering simulation $\preceq_{tiss} \Rightarrow \preceq_{dbss}$, so our soundness extends beyond the C to the actual behavior of the compiled binaries.

transition system over the interface, that is, the completely non-deterministic transition system $1_A$ with state $\star$ and maximally non-deterministic transition relation $\star \times A \times \star$. Since this transition system is terminal in the refinement pre-order of transition systems, we refer to $1_A$ as the *terminal machine* for observables $A$. We choose as our parallel composition the asynchronous composition, where the asynchronous composition $C$ and $D$ is the most abstract common stuttering refinement of components, which we write $C \|_a D$.

Thus our WP proof witnesses that $C_A \|_a 1_{D_A} \succeq C_C \|_a 1_{D_C}$. Since, for any $D_A$ with the same interface alphabet, $1_{D_A} \succeq D_A$.

In fact, it is the product in the category of transition systems and divergence-blind stuttering simulations, and so we obtain the inference

$$\frac{C_A \|_a 1_A \succeq C_c \|_a 1_A \; 1_A \succeq D_A}{C_A \|_a D_A \succeq C_c \|_a D_A}$$

from this argument, if we were moreover given a proof of refinement of the rest of the design by its implementation, we would obtain a system-level refinement proof

$$\frac{C_A \succeq C_c \; D_A \succeq D_c}{C_A \| D_A \succeq C_c \| D_c}$$

and for any *safety* property $P_{\text{safe}}$, we can use a proof of $P_{\text{safe}}(C_A \|_a D_A)$ to infer $P_{\text{safe}}(C_C \|_a D_C)$, as

$$\frac{C_A \succeq C_c \; D_A \succeq D_c \; P_{\text{safe}}(C_A \|_a D_A)}{P_{\text{safe}}(C_C \|_a D_C)}.$$

Q guarantees this congruence by construction, since it interprets specifications and simulation maps consistently in both SMV and ACSL. The asynchronous composition is accomplished by rendering components into SMV with additional external non-determinism, such that any state in a component may transition into itself under any letter in the alphabet not owned by the component.

## 4 Related Work

Model checking has a long history in formal verification of software systems [1, 12, 13, 18]. Well-known industrial uses of model checking gain value with models that are divorced from implementation [24]. We consider these use cases a good start, 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 (e.g., in C).

Tools like SLAM [4] 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 [21]. These approaches assume the behavior of the larger system is encoded soundly in assumptions of their specification. 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) and assume the composed environment Q to be fully unbound in its behavior. In practice this means that the state machine model uses variables that are unbound within their type that are conceptually the interface between the specification and its environment. At the C level these are volatile variables that are used as a communication medium by other system components and do not have unbounded behavior. Moreover, our models exist outside the process of verifying their refinement to C so it is an important feature of our approach that we be able to take a model distinct from the C and verify properties against it and then demonstrate a refinement to the C implementation.

Several works have explicitly aimed to bridge the gap between state-machine-like specifications and real implementations. Broadly they have focused on generality where the user can arrive to the tool with a program and a spec and eventually derive a proof of simulation. 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, [26] 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 [2].

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 foundational and they are carried all the way down to the point of 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

The 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.

One limitation of Q is its strict requirement on the structure of the C implementation. As mentioned in Section 3, the current state of Q poses somewhat strict limitations on the state machines and C code. However, we are interested in using the the Verified Software Toolchain's (VST) [2] 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, a verification framework to verify the correctness of digital control systems. Q works by linking together state machines (expressed in Stateflow) with a source code implementation (in C), and proving that 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. 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.

## 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. XXXX-XXXXXXX.

## 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. *STTT* 12, 6 (2010), 447–466.

[2] 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

[3] 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

[4] 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*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[5] 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/.

[6] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. 2022. WP Plug-in Manual. Available at http://frama-c.com/download/frama-c-wp-manual.pdf.

[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] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical computer science* 59, 1-2 (1988), 115–131.

[11] 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

[12] 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

[13] 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*, Ana Cavalcanti, David Deharbe, Marie-Claude Gaudel, and Jim Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–44.

[14] 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.

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

[16] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods (SFEM (LNCS 7504))*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer, Thessaloniki, Greece, 233–247.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Proofs and Refutations, and Z3. In *7th International Workshop on the Implementation of Logics at the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (IWIL LPAR 2008)*. Doha, Qatar. http://ceur-ws.org/Vol-418/paper10.pdf

[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] 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.

[21] Rustan Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *16th International Conference, LPAR-16, Dakar, Senegal* (16th international conference, lpar-16, dakar, senegal ed.). Springer Berlin Heidelberg, 348–370. https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/

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

[23] 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

[24] 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 (mar 2015), 66–73. https://doi.org/10.1145/2699417

[25] 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.

[26] 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)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32