SAND2019-14897

SANDIA REPORT

Printed

Sandia National Laboratories

A Domain-Specific Language for High-Consequence Control Software

Robert Armstrong, Geoff Hulette

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

While most software development for control systems is directed at what the system is supposed to do (i.e., function), high-consequence controls must account for what the system is *not* supposed to do (i.e., safety, security and reliability requirements). A Domain Specific Language (DSL) for high-consequence digital controls is proposed. As with similar tools for the design of controls, the DSL will have plug-in modules for common controller functions. However, the DSL will also augment these modules with attendant "templates" that aid in the proof of safety, security and reliability requirements, not available in current tools. The object is to create a development methodology that makes construction of high-assurance control systems as easy as controls that are designed for function alone.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Software for unmanned control systems – such as used in defense, energy, and transportation – is prone to failure when created using common programming languages and techniques, due to the difficulty of anticipating all possible behaviors of complex software. We propose developing a controller Domain-Specific Language (DSL) that will make it easier to create correct code and easier to formally verify its correctness for high-consequence controls. Standard development practices encourage programming for function over safety, security and reliability: the programmer concentrates on *what it does* rather than *what it is not supposed to do*. The reason for this is that function is easily verified through testing whereas safety, security and reliability require formal reasoning.

Current approaches to high-assurance correct-by-construction programming are broad and do not target specific properties of controllers that are harder to verify (e.g., real-time requirements). This limits the application of these techniques to very simple systems.

This work has focused on a DSL for high-consequence digital controls that has an unambiguous formal semantics, but also uses a programming metaphor close to that which the designers of NW controls already use. Many designers of control systems use a State Chart analog to conceptualize and test out their designs. The most common programming environment that implements the State Chart metaphor in the US, and indeed at Sandia, is Matlab Simulink/Stateflow [8]. The DSL presented in this work is a formally analyzable language based on State Charts, called Q-Charts , that is compatible with suitably restricted Stateflow. Although Stateflow has no formally defined semantics, we have discovered through this work a suitable restriction on Stateflow's informal semantics that works with our formally correct DSL. This new language can then be analyzed for safety, security, and reliability properties to show the correctness of a specification for high-consequence controls. We will refer to such a specification as an "executable specification".

2. SYNCHRONIZATION, REFINEMENT AND IMPLEMENTATION REFINEMENT IN Q-CHARTS

2.1. INTRODUCTION TO REFINABLE SEMANTICS IN STATECHARTS: Q-CHARTS

A number of formal semantics have been assigned to the original Harel State Charts[2]. Here we present a variant where each composition is also a refinement of the precursor, enabling a divide and conquer strategy for large systems. This language for specifying controllers is common among engineers and the purpose of this work is to make a variant Domain-Specific Language (DSL), called Q-Charts, that is amenible to formal analysis for high-consequence controls but still accessible to engineers. Two important characteristics of this DSL is automation and refinement. Because our target systems are high-consequence controllers, strict verification against a consistent specification that is accessible to engineers is critical.

Refinement not only provides an incremental method of engineering but also providess scalability for formal proofs of the resulting control system. Famously unscalable, naive formal analysis complexity grows exponentially with problem size. Without a divide and conquer scheme for controlling that complexity, even modest-sized designs are out-of-reach. Automation is important because specification and digital system development, though often represented as top-down, are almost uniformly iterative.

State Charts was conceived and remains a primarily graphical language, which nonetheless has a formal semantics. In fact there are a number of different semantics associated with State Charts [11, 4, 7]. Q-Charts preserves these properties, hewing closely to the written language for State Charts: SCXML [9]. Q-Charts adds elements to SCXML that enable and facilitate formal reasoning which have been published separately [10]. In what follows, the semantics of this Q-Chart formalism is laid out, first with an example, then with the generalized Coq-based [1] code and proofs. The Q-Chart formal semantics are interpreted in TLA [6] and underlie the formal reasoning. The Coq rendering of Q-Charts is done in a shallow embedding of TLA in Coq. The Q toolsuite which interprets this extended SCXML and renders it in various languages formal and otherwise can be used for both hardware and software analysis.

2.2. A MOTIVATION WITH AN EXAMPLE IN Q-CHARTS

We hew closely to the TLA formalism for the logic of Q-Charts using the graphical language to guide what the states mean and how composition is done.

We define the semantics Q-Charts in TLA (Temporal Logic of Actions[6]). TLA is particularly useful for formal reasoning about specifications. An important part of this work is to be able to relate specification variants via refinement of the original TLA Q-Chart specification. This includes refinements that are implementable by hardware or software but necessarily lie outside of strict TLA formulae.

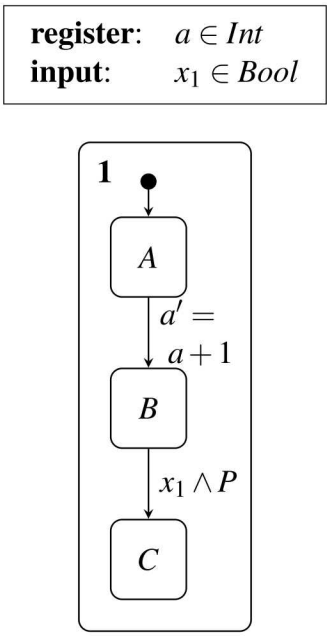

Figure 2-1. Simple Q-Chart that forms the first part (1) of a parallel composition. The input x_1 is identified with a completely nondeterministic choice on every step of the system.

The input variables, over which the chart has no control (x in Figure 2-1), is distinguished from the register variables (a in Figure 2-1) which the machine controls. The input variable, x_1, should be thought of as coming from a completely nondeterministic (Boolean valued) machine. This is important because there will eventually need to put causal, and even implementation constraints on the input so that it remains physical. (x_1 in Figure 2-1) *is* effectively composed with the machine receiving the input (in this case **1**) but we do not consider it part of the machine.

In Q-Charts every composition is a refinement of the machine. This includes both hierarchical and parallel composition (see Chapter 3). As an example, see the Chart in Figure 2-3. The translation into TLA appears in Figure 2-4.

Parallel composition is accomplished by conjoining the *Spec*'s for the machines as well as the variables to be unified between the two. The composed Q-Chart appears in Figure 2-3 where P is with the predicate $in(\mathbf{1}, B)$ ($True$ when machine **1** occupies state B, $False$ otherwise). This has the

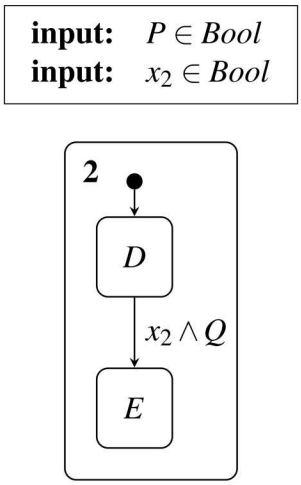

Figure 2-2. Simple Q-Chart that forms the second part (2) of a parallel composition (see Figure 2-3).

effect of "synchronizing" $B \rightarrow C$ and $D \rightarrow E$. Additionally, x_1,x_2 become x so that both machines **1** and **2** receive the same input. The composition's representation in TLA the $NextR$ must be replaced with the conjunction: $Next1 \wedge (x_1 = x) \wedge (P = (st2 = D))$ and $NextL$ is replaced with $Next2 \wedge (x_2 = x) \wedge (Q = (st2 = D))$ and then these are conjoined together.

Skip steps, or stuttering steps, are transitions where nothing happens and all of the state for any machine stays the same. As is traditional with TLA specifications the $[Q]_{<vars>}$ notation indicates skip steps over the machine variables: $[Q]_{<vars>}$ is true even if the action Q is not $True$ but set of machine variables, $< vars >$ remains unchanged.

$$[Q]_{<vars>} \triangleq Q \vee Skip_{<vars>} = Q \vee (var1 = var1') \wedge (var2 = var2') ... \wedge (varn = varn') \qquad (2.1)$$

Notably the input variables to the machine are never constrained by the machine in any way except that its type remains constant. This alternate notation will be useful when we further refine the machine closer to something that is less refinable, but more implementable. Implementations, like Verilog or C programs, don't have an idea of unlimited skip steps and do not possess a generic ability to be trace subsetted into further refinements. The idea that implementations lack the ability to be refined is not true in *every* case, it is just not generically true. For example, predicated actions (guards) can always be strengthened without resorting to stuttering, which always yields a refinement. But adding arbitray orthogonal behaviors to the machine by overlaying stuttering is absent in almost all implementations. These connections between skip steps, time, and implementations will be dealt with more systematically later in the this work. For now we will say that Machines 1 and 2 of Figure 2-3 are synchronized when every transition is taken at the same "time": i.e. on the same step.

It is clear that this step synchronization does not exist in the TLA expression of Figure 2-6. There are steps where either or both machines are skipping and, for example, states B and E and C and D can be simultaneously occupied (i.e. not step synchronized as we might like). In order to refine to non-TLA expressions that are closer to the semantics of an implementation, we reformulate the

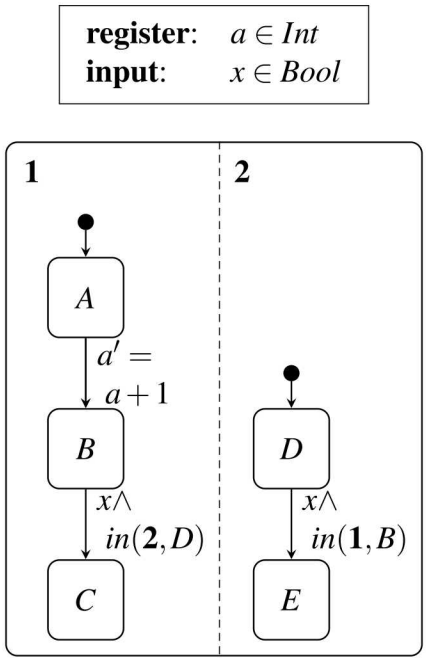

Figure 2-3. Parallel composition of Q-Charts from Figures 2-1 and 2-2. x_1 and x_2 are unified to x, and P is unified to $in(\mathbf{2},D)$, a predicate that is true if Machine (2) is in state D. Similarly $in(\mathbf{1},B)$ is true if Machine 1 occupies B. The effect of this is to synchronize the $B \rightarrow C$ and $D \rightarrow E$ transitions.

specification where *Skip* is an explicit predicate term. Rewriting the specification of Figure 2-6 in the *Skip* syntax:

$$\begin{aligned}[Next1]_{vars1} \wedge [Next2]_{vars2} &= (Next1 \vee Skip_{vars1}) \wedge (Next2 \vee Skip_{vars2}) \\ &= (Next1 \wedge Next2) \vee (Skip_{vars1} \wedge Skip_{vars2}) \\ &\quad \vee (Next1 \wedge Skip_{vars2}) \vee (Next2 \wedge Skip_{vars1})\end{aligned}$$

If we further refine the machine into a subset where both of the machines cannot skip at the same time (i.e. some physical computation is always happening) we can conjoin the above with $\neg(Skip_{vars1} \wedge Skip_{vars2})$:

$$\begin{aligned}([Next1]_{vars1} \wedge [Next2]_{vars2}) \wedge \neg(Skip_{vars1} \wedge Skip_{vars2}) &= (Next1 \wedge Next2) \\ &\quad \vee (Next1 \wedge Skip_{vars2}) \vee (Next2 \wedge Skip_{vars1})\end{aligned}$$

In this case the combined machine is always taking some transition but skips are allowed for one machine or the other. This refinement would conform to C-like semantics where each machine occupies a thread, and scheduling allow one or both machines to take a step. In this case either states B and E or C and D can be simultaneously occupied. Note that this refinement supports an invariant such as $Next1 \vee Next2$ that is not stuttering invariant and thus the new specification:

$$Spec = Init \wedge \Box\big((Next1 \wedge Next2) \vee (Next1 \wedge Skip_{vars2}) \vee (Next2 \wedge Skip_{vars1})\big)$$

```
---------------------------- MODULE Machine1 ----------------------------
  Init1 == st1 = A
  Next1 == \/ (st1 = A) /\ (st1' = B) /\ (a' = a+1)
           \/ (st1 = B) /\ x_1 /\ P /\ (st1' = C)

  vars1 == st1
  TypeInv1 == (st1 \in {A,B,C}) /\ (a \in Int)
           /\ (x_1 \in Bool)/\ (P \in Bool)

  Spec1 == Init1 /\ [][Next1]_vars1 /\ TypeInv
-------------------------------------------------------------------------
THEOREM  Spec1 => ((st1 = C) => [](st1 = C))
=========================================================================
}
```

Figure 2-4. A simple Q-Chart to be composed with that of Figure 2-5.

Because this is expression is not stuttering invariant, it is not a "TLA expression" [6].

Allowing one or the other machine to skip while the other takes a transition allows Machine 2 to delay until both $B \rightarrow C$ and $D \rightarrow E$ transitions can be enabled. Further skipping allows the two transitions to be taken simultaneously but does not require it.

Removing all stuttering would constrain all of the transitions in the two machines to be taken simultaneously, but would not allow for the Machine 2 delay necessary for the $B \rightarrow C$ and $D \rightarrow E$ transitions to be simultaneously enabled, after which the two would be taken synchronously. This completely synchronous semantics is closer to a hardware description where a clock dictates the next transition for all composed machines and there is nothing that could be interpreted as stuttering.

To impose hardware-like synchronous semantics will require refinement of the composition of Figure 2-3 to a completely reactive machine. Because we cannot rely on stuttering to cover for a transition that cannot be taken, the machine must have a transition for every eventuality to avoid a deadlock condition. The most straight forward way to accomplish this is to decorate each state with a self-transition that has a guard with the negation of that state's outgoing transitions.

In Figure 2-8 is this further refinement for the example. The transition relation expression for this Q-Chart is written in Figure 2-9.

The picture of sucessive refinements from a looser TLA specification, most conducive to abstraction/refinement, down to one that more closely adheres to the implementation semantics has advantages at both ends. Abstraction/refinement permits a more facile and scalable environment for proving safety properties and the refinement into an implementation semantics better predicts the behavior of the end product. Refinement guarantees that every property proved at the abstract level still holds at the implementation level. Properties may take on a stronger form

```
---------------------------- MODULE Machine2 ----------------------------
  Init2 == st2 = D
  Next2 == \/ (st2 = D) /\ x_2 /\ Q /\ (st2' = E)


  vars2 == st2

  TypeInv2 == (st2 \in {D,E}) /\ (x_2 \in Bool) /\ (Q \in Bool)

  Spec2 == Init2 /\ [][Next2]_vars2 /\ TypeInv)
-------------------------------------------------------------------------
THEOREM  Spec2 => ((st2 = E) => [](st2 = E))
=========================================================================
```

Figure 2-5. A simple Q-Chart to be composed with that of Figure 2-4.

in refinement, more relevant to the implementation. A property such as:

$$P \Longrightarrow Q \vee Skip_{vars}$$

proved at the abstract level, may take a form:

$$P \Longrightarrow Q$$

at the implementation level.

The conclusion is that the abstract level provides a generic ability to refine specifications and a scalable way to prove properties about it. The more implementation refinement allows an understanding of what effect those properties have on a physical device that can be built.

```
---------------------- MODULE Composed Machine ----------------------
Init == (st1 = A) /\ (st2 = D)
Next1 == \/ (st1 = A) /\ (st1' = B) /\ (a' = a+1)
         \/ (st1 = B) /\ x /\ (st2 = D) /\ (st1' = C)
Next2 == \/ (st2 = D) /\ x /\ (st1 = B) /\ (st2' = E)


vars2 == st2
vars1 == st1

TypeInv == /\ (st1 \in {A,B,C}) /\ (x \in Bool)
           /\ (st2 \in {D,E})

Spec == Init /\ [][[Next1]_vars1 /\ [Next2]_vars2 /\ TypeInv)
-----------------------------------------------------------------
=================================================================
```

Figure 2-6. The denoted TLA expression for the composition of Figure 2-3. Here $in(\mathbf{i},S) \triangleq (sti = S)$: the state variable sti in the TLA rendering occupies the state S in Machine i.

```
---------------------- MODULE Composed Partial Skip Machine ----------------------
  Init == (st1 = A) /\ (st2 = D)
  Next1 == \/ (st1 = A) /\ (st1' = B) /\ (a' = a+1)
           \/ (st1 = B) /\ x /\ (st2 = D) /\ (st1' = C)

  Next2 == \/ (st2 = D) /\ x /\ (st1 = B) /\ (st2' = E)


  vars2 == st2
  vars1 == st1

  TypeInv == /\ (st1 \in {A,B,C}) /\ (x \in Bool)
             /\ (st2 \in {D,E}) /\ (a \in Int)

  Spec == Init /\ [](Next1 /\ Next2 \/ (Next1 /\ Skip_vars2) \/ (Next2 /\ S
/\ TypeInv)
-----------------------------------------------------------------------------
=============================================================================
```

Figure 2-7. Expression removing the ability for both machines to stutter at once, similar to threaded C semantics. This is not a *valid* TLA expression in the sense that it is no longer stuttering invariant.

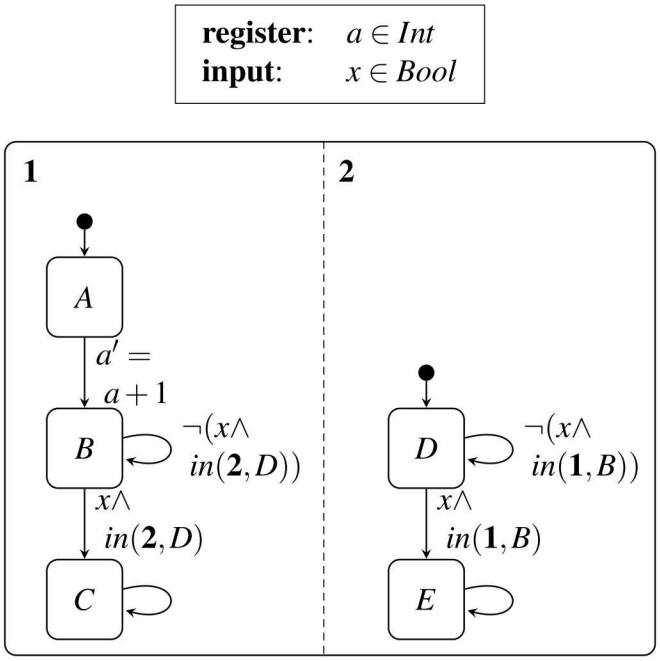

Figure 2-8. Parallel composition of Q-Charts from Figure 2-3 but now rendered as a reactive synchronous machine. No stuttering steps are required but refinement becomes more difficult.

```
---------------------------- MODULE Composed Reactive Machine ----------------------------
Init == (st1 = A) /\ (st2 = D)
Next1 == \/ (st1 = A) /\ (st1' = B) /\ (a' = a + 1)
         \/ (st1 = B) /\ x /\ in(2,D) /\ (st1' = C)
         \/ (st1 = B) /\ \neg (x /\ in(2,D)) /\ (st1' = B)
         \/ (st1 = C) /\ (st1' = C)
Next2 == \/ (st2 = D) /\ x /\ in(1,B) /\ (st2' = E)
         \/ (st2 = D) /\ \neg (x /\ in(1,B)) /\ (st2' = D)
         \/ (st2 = E) /\ (st2' = E)


vars1 == st1
vars2 == st2,a

TypeInv == /\ (st1 \in {A,B,C}) /\ (a \in Int) /\ (x \in Bool)
           /\ (st2 \in {D,E})

Spec == Init /\ [](Next1 /\ Next2 /\ TypeInv)
-------------------------------------------------------------------------------
===============================================================================
```

Figure 2-9. The denoted TLA expression for the reactive machine refinement of the Q-Chart of Figure 2-8. Because the only modification is self-transitions, these can subset stuttering in either expressions of Figures 2-6 or 2-7 and thus is itself a refinement of both of those machines. This transition relation coheres more with the semantics of hardware, where at every step some transition must be taken.

3. FORMAL REASONING ABOUT Q-CHARTS

3.1. MOTIVATION

Specifications are an essential part of the engineering process for high-consequence digital systems. They bridge the design gap between the *requirements*, which capture desired behavior at the highest and most domain-specific level, and the *implementation*, in the form of some computer program or hardware description, which realizes the requirements. Specifications capture design and engineering decisions in a way that is still accessible to customers or other stakeholders and allows for feedback and discussion, but are more detailed and useful than the raw requirements for engineers and implementers. Traditional specifications are informal, captured in prose or diagrams which appeal to intuition, but often elide important details. Moreover, because of their informal nature, they generally cannot be subjected to rigorous testing or verification and thus may hide mistakes which may then be propogated to the implementation. *Formal* specifications complement informal specification with mathematically precise language for describing desired digital system behavior. They are unlike programming languages, in that they describe what constitutes an acceptable behavior of the system without necessarily describing, step-by-step, how to achieve that behavior.

Statecharts [3] have inspired numerous graphical languages, some designed for writing specifications and some for regular programs, all of which are based on the idea of articulating a system as a connected set of state machines. Differences in the nature of the connections, as well as different notions of variables, input, events, etc., give rise to variants. In this work we present yet another variation on the theme, a version of Statecharts called Q-Charts, which is designed for the design and specification of high-consequence digital systems, including the ability to *refine* specifications from high-level, abstract models down to executable code. In particular, a refined specification is guaranteed to preserve all stuttering-invariant safety properties (including inductive invariants) of its abstraction. Crucially, this design facilitates independent development of separate components, which can be composed after the fact with assurance that safety properties are maintained.

In this paper we describe a formal syntax for Q-Charts and assign to it a denotational semantics using Lamport's Temporal Logic of Actions (TLA) [5]. We also give a syntactic definition of refinement in Q-Charts that follows its from Statechart-like composition operators, and prove that this definition does indeed yield refinement in the denoted TLA domain. We chose TLA for the denotational domain because it has a simple and useful definition of refinement which preserves stuttering-invariant safety properties, and this in turn allows for simpler proofs. We have

formalized and machine-checked our definitions and proofs in the Coq theorem prover environment.

3.2. SYNTAX OF Q-CHARTS

The syntax of Q-Charts is intentionally quite simple – much simpler than most Statechart-like languages. In particular, we do not allow for certain constructions, such as inter-machine transitions or signals, which complicate the semantics while, in our view, adding little expressivity. We opt instead to define a "core language" for Q-Charts, with those extra constructions defined as "sugar," i.e., in terms of the core. This simple structure has two benefits. First, since Q-Charts are intended to serve as a formal specification language, it keeps the meaning of a Q-Chart very clear and hopefully avoids confusion or disagreement over the meaning of a given Q-Chart. Second, we are interested in proving refinement relations between certain Q-Charts based on their compositional structure, and the simplicity of those structures facilitates that goal.

Syntactically, a Q-Chart is defined as a composition over simple state machines with variables. First, we describe the (core) syntax for the constituent state machines. Second, we describe the compositional structure of Q-Charts.

We write `Bool` to denote the set of Boolean values `False` and `True`. We assume two sets as parameters: a set S of state labels, and a set of E of "environments" (see below). We also define symbols $C \triangleq E \to \texttt{Bool}$, i.e., predicates over E, as well as $A \triangleq E \times E \to \texttt{Bool}$, i.e., relations on E (think of C and A as mnemonics for "Condition" and "Action", respectively).

A machine is a tuple $\langle I, T, U, N \rangle$, where $I \subseteq S \times C$ is the set of initial machine conditions, $T \subseteq S \times C$ is the set of the terminal conditions, $U \subseteq S \times A$ is the set of "inner steps" (see below), and $N \subseteq S \times S \times A$ is the set of steps.

The set S of state names should be thought of as the labels on the boxes of a state machine diagram. For example, in Figure 3-1, it would be the case that $A, B, C \subseteq S$. Note that S may contain any number of "unused" labels, so, in particular, it is convenient to think of S a large set of finite-length alphanumeric identifiers.

In this setting, we treat the set E of environments abstractly. Intuitively, an element of E can be thought of as a maps from variable names to values. In the core language presented here it suffices to elide the details of variable names and values, as well as the structure of the map, as they do not impact the refinement argument.

The sets I and T are both subsets of $S \times C$ and represent initial and terminal states of the machine, respectively. Intuitively, an initial/terminal state consists of a state label (an element of S) along with a set of possible configurations of the environment (an element of C, which "picks out" acceptable initial/terminal environments).

The set N represents the "steps" that the machine can take. It is a subset of $S \times S \times A$ where the first term can be understood as the "source" state label, the second as the "target" state label, and the third as the "action" which relates the sources environement to the target environment.

The set U represents steps which may change the environment but which stay in the same state (i.e., where the element of S remains unchanged). It is a subset of $S \times A$ where the first term is the state label in question and the second is the action which descibes how the environment may evolve. A machine by itself makes no semantic distinction between steps in U and steps in N which happen to have the same source and target state; however, in composition they have different meanings and the distinction is important. Roughly speaking, U can be thought of as "stay steps" which describe how the environment may evolve while time is passing in a state. Self-transitions in N are, by contrast, understood as peers with any other (non-self) step (see Section 3.3).

Given a set of machines M, we define the syntax of χ-terms inductively, as follows:

1. `Unit` is a χ-term;
2. If C_1 and C_2 are χ-terms, then $\texttt{Par}(C_1, C_2)$ is a χ-term;
3. If $m \in M$ is a machine and f is a function from S to χ-terms, then $\texttt{Node}(m, f)$ is a χ-term;

and any χ-term is a Q-Chart. These terms correspond to compositions of state machines, as described below in Section 3.3.

Finally we need a syntax for "configurations," which we will call γ-terms. Intuitively, a γ-term is a composition of state labels in S that describes the composite state of a Q-Chart. We define γ-terms inductively as follows:

1. `U` is a γ-term;
2. If G_1 and G_2 are γ-terms, then $\texttt{P}(G_1, G_2)$ is a γ-term;
3. If $s \in S$ is a state and G is a γ-term, then $\texttt{N}(s, G)$ is a γ-term.

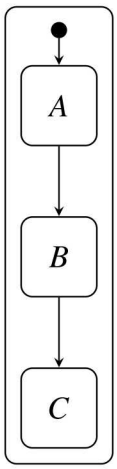

Figure 3-1. Example machine with states A, B, and C.

3.3. SEMANTICS OF Q-CHARTS

Intuitively, a χ-term representing a Q-Chart denotes a set of *behaviors* that are allowed by any implementation of the Q-Chart. Here, we assume that an "implementation" of a Q-Chart specification is just another (probably more detailed) Q-Chart. In practice it could also be a C program, hardware description, or other description of a digital system. A behavior may be thought of as a valid "trace" of the Q-Chart, that is, a sequential list of pairs of configurations and environments. To formalize this notion, we appeal to the existing notion of a behavior or trace in TLA, and we define our semantics in terms of a denotation of χ-terms to TLA formulae.

We have formalized the following mathematics, including machine-checked proofs, in the Coq theorem prover, see Section 3.4.

To define our denotational semantics, we must first define a few auxillary definitions. We begin with an inductive definition of a relation `initial` which relates χ-terms to a pair consisting of a γ-term and a condition in C.

$$\frac{p \in C}{\langle \mathtt{U}, p\rangle \in \mathtt{initial}(\mathtt{Unit})} \qquad \frac{\langle g_1, p\rangle \in \mathtt{initial}(c_1) \qquad \langle g_2, q\rangle \in \mathtt{initial}(c_2)}{\langle \mathtt{P}(g_1, g_2), p \wedge q\rangle \in \mathtt{initial}(\mathtt{Par}(c_1, c_2))}$$

$$\frac{m = \langle I, T, U, N\rangle \qquad \langle s, p\rangle \in I \qquad \langle g, q\rangle \in \mathtt{initial}(f(s))}{\langle \mathtt{N}(s, g), p \wedge q\rangle \in \mathtt{initial}(\mathtt{Nest}(m, f))}$$

We also need a `terminal` relation, which is defined similarly:

$$\frac{p \in C}{\langle \mathtt{U}, p\rangle \in \mathtt{terminal}(\mathtt{Unit})} \qquad \frac{\langle g_1, p\rangle \in \mathtt{terminal}(c_1) \qquad \langle g_2, q\rangle \in \mathtt{terminal}(c_2)}{\langle \mathtt{P}(g_1, g_2), p \wedge q\rangle \in \mathtt{terminal}(\mathtt{Par}(c_1, c_2))}$$

$$\frac{m = \langle I, T, U, N\rangle \qquad \langle s, p\rangle \in T \qquad \langle g, q\rangle \in \mathtt{terminal}(f(s))}{\langle \mathtt{N}(s, g), p \wedge q\rangle \in \mathtt{terminal}(\mathtt{Nest}(m, f))}$$

And now we can give the definition of `step`, which relates between a chart to a set of triples, where each triple consists of a pair of γ-terms and an action (an element of A, i.e., a relation on environments) that define the next-state behavior of the chart.

$$\frac{a \in A}{\langle \mathtt{U}, \mathtt{U}, a \rangle \in \mathtt{step}(\mathtt{Unit})} \qquad \frac{\langle g_1, g_1', a_1 \rangle \in \mathtt{step}(c_1) \qquad \langle g_2, g_2', a_2 \rangle \in \mathtt{step}(c_2)}{\langle \mathtt{P}(g_1, g_2), \mathtt{P}(g_1', g_2'), a_1 \wedge a_2 \rangle \in \mathtt{step}(\mathtt{Par}(c_1, c_2))}$$

$$\frac{m = \langle I, T, U, N \rangle \qquad \langle s, a_1 \rangle \in U \qquad \langle g, g', a_2 \rangle \in \mathtt{step}(f(s))}{\langle \mathtt{N}(s, g), \mathtt{N}(s, g'), a_1 \wedge a_2 \rangle \in \mathtt{step}(\mathtt{Nest}(m, f))}$$

$$\frac{\begin{array}{c} m = \langle I, T, U, N \rangle \qquad \langle s, s', a \rangle \in N \\ \langle g, p \rangle \in \mathtt{terminal}(f(s)) \qquad \langle g', q \rangle \in \mathtt{initial}(f(s')) \\ r(e, e') \triangleq p(e) \wedge a(e, e') \wedge q(e') \end{array}}{\langle \mathtt{N}(s, g), \mathtt{N}(s', g'), r \rangle \in \mathtt{step}(\mathtt{Nest}(m, f))}$$

Now we can define the denoted TLA formula in a schematic way. A χ-term X denotes a TLA formula $\mathtt{Init} \wedge \Box[\mathtt{Next}]_{\mathtt{vars}}$ where

$$\begin{aligned} \mathtt{Init} &\triangleq \bigvee_{\langle x, p \rangle \in \mathtt{initial}(X)} \mathtt{st} = x \wedge p \\ \mathtt{Next} &\triangleq \bigvee_{\langle x, y, a \rangle \in \mathtt{step}(X)} \mathtt{st} = x \wedge \mathtt{st}' = y \wedge a \\ \mathtt{vars} &\triangleq \langle st, \text{variable names in environment} \rangle \end{aligned}$$

3.4. SEMANTICS AND PROOFS IN COQ

Here we include two listings of Coq source code – definitions and proofs – for the syntax, semantics, and proofs.

```
(* FILE: Process.v *)
Require Import Coq.Lists.List.
Require Import Chart.Util.
Set Implicit Arguments.
Import ListNotations.

Section Process.

  Variable A : Type.

  Record Process := process
    {
      initial : A -> Prop;
      step : A -> A -> Prop
```

```
  }.

Variable m : Process.

Inductive reachable : A -> Prop :=
| reachable_initial : forall x,
    initial m x -> reachable x
| reachable_step : forall x y,
    reachable x -> step m x y -> reachable y.

Definition deterministic :=
  (forall x x', initial m x -> initial m x' -> x = x') /\
  (forall x, reachable x ->
             forall y y', step m x y -> step m x y' -> y = y').

Definition reactive :=
  forall x, reachable x -> exists y, step m x y.

Definition stuttering :=
  forall x, reachable x -> step m x x.

Lemma reactive_reachable :
  reactive ->
  forall x, reachable x -> exists x', reachable x'.
Proof.
  unfold reactive.
  intros Hreact x Hreach.
  specialize (Hreact x Hreach).
  inversion Hreact as (y,Hy).
  exists y.
  apply reachable_step with x; assumption.
Qed.

Inductive trace_to : A -> list A -> Prop :=
| trace_to_initial : forall x,
    initial m x -> trace_to x []
| trace_to_step : forall x x' t,
    trace_to x t -> step m x x' -> trace_to x' (x::t).

Definition trace (t : list A) :=
  match t with
  | [] => True
  | (x::t') => trace_to x t'
  end.
```

```
Lemma initial_trace :
  forall x, trace [x] -> initial m x.
Proof.
  intros x Ht.
  inversion Ht.
  assumption.
Qed.

Lemma step_trace :
  forall x y t, trace (y::x::t) -> step m x y.
Proof.
  intros x y t Ht.
  inversion Ht; subst.
  assumption.
Qed.

Theorem reachable_trace :
  forall x, reachable x -> exists t, trace (x::t).
Proof.
  unfold trace.
  intros x Hreach.
  induction Hreach.
  exists []; constructor; assumption.
  inversion IHHreach as (t,Ht).
  exists (x::t).
  apply trace_to_step; assumption.
Qed.

Lemma trace_reachable :
  forall x t, trace (x::t) -> reachable x.
Proof.
  unfold trace.
  intros x t Htrace.
  induction Htrace.
  constructor; assumption.
  apply reachable_step with x; assumption.
Qed.

Lemma trace_cons :
  forall x t, trace (x::t) -> trace t.
Proof.
  intros x t Ht.
  destruct Ht.
  simpl; constructor.
  assumption.
```

```
Qed.

Theorem in_trace_reachable :
  forall t x, trace t -> In x t -> reachable x.
Proof.
  intros t x Ht H.
  induction t; inversion H; subst.
  eauto using trace_reachable.
  apply IHt; eauto using trace_cons.
Qed.

Lemma trace_app :
  forall t1 t2, trace (t1++t2) -> trace t2.
Proof.
  intros t1 t2 H.
  induction t1 as [|x t1].
  assumption.
  apply IHt1.
  simpl in H.
  apply trace_cons with x.
  assumption.
Qed.

Theorem trace_postfix :
  forall t1 t2, postfix t1 t2 -> trace t2 -> trace t1.
Proof.
  intros t1 t2 Hpost Ht2.
  induction Hpost.
  assumption.
  apply IHHpost.
  apply trace_cons with x.
  assumption.
Qed.

Definition invariant (P : A -> Prop) :=
  forall x, reachable x -> P x.

Definition trace_property (P : list A -> Prop) :=
  forall t, trace t -> P t.

Theorem invariant_trace_property :
  forall P, invariant P <-> trace_property (Forall P).
Proof.
  unfold invariant, trace_property.
  split; intros H.
```

```
    intros t Ht.
    apply Forall_forall.
    intros x Hx.
    apply H.
    apply in_trace_reachable with t; assumption.

    intros x Hx.
    apply reachable_trace in Hx.
    inversion Hx as [t Ht].
    specialize (H (x::t) Ht).
    rewrite Forall_forall in H.
    apply H.
    apply in_eq.
  Qed.

  Theorem unreachable_invariant :
    forall x, ~(reachable x) <-> invariant (fun st => st <> x).
  Proof.
    unfold invariant, not.
    split; intro H; intros; subst;
      eapply H; eauto.
  Qed.

End Process.


Section Safety.

  Variable A : Type.

  Definition safety (P : list A -> Prop) :=
    forall l1 l2, P l2 -> postfix l1 l2 -> P l1.

  Theorem trace_safety :
    forall (m : Process A), safety (trace m).
  Proof.
    unfold safety.
    eauto using trace_postfix.
  Qed.

End Safety.


Section Refinement.
```

```
Variable A : Type.            (* "Abstract" states *)
Variable C : Type.            (* "Concrete" states *)
Variable ma : Process A.      (* Abstract machine *)
Variable mc : Process C.      (* Concrete machine *)
Variable R : C -> A -> Prop. (* State relation *)

Definition refinement :=
  forall tc, trace mc tc -> exists ta, list_rel R tc ta /\ trace ma ta.

Section RefinementProperties.

  (* It would be nice if our definitions allowed us to derive that R
  must be a function or Galois connection or whatever.  In fact, at
  least for the properties below, we don't seem to need unique
  existance, merely existance. *)
  (* Hypothesis R_func : function R. *)
  Hypothesis Href : refinement.

  Lemma refinement_reachable :
    forall c, reachable mc c -> exists a, R c a /\ reachable ma a.
  Proof.
    intros c Hc.
    apply reachable_trace in Hc.
    inversion Hc as [tc Htc]; subst; clear Hc.
    specialize (Href (c::tc) Htc).
    inversion Href as (ta, (HR,Hta)); subst; clear Href.
    destruct ta as [|a ta].
    inversion HR.
    exists a.
    split.
    inversion HR; subst; clear HR; assumption.
    apply trace_reachable with ta; assumption.
  Qed.

  Theorem refinement_trace_property :
    forall (Pa : list A -> Prop) (Pc : list C -> Prop),
      (forall ta tc, list_rel R tc ta -> Pa ta -> Pc tc) ->
      trace_property ma Pa ->
      trace_property mc Pc.
  Proof.
    unfold trace_property.
    unfold refinement in Href.
    intros Pa Pc HP Ha tc Htc.
    specialize (Href tc Htc).
```

```
    inversion Href as (ta, (HR,Hta)).
    specialize (HP ta tc HR).
    apply HP.
    apply Ha.
    assumption.
  Qed.

  Theorem refinement_invariant :
    forall (Ia : A -> Prop) (Ic : C -> Prop),
      (forall c a, R c a -> Ia a -> Ic c) ->
      invariant ma Ia ->
      invariant mc Ic.
  Proof.
    intros Ia Ic.
    repeat (rewrite invariant_trace_property).
    intros H.
    apply refinement_trace_property.
    intros tc ta HRl.

    induction HRl as [| c tc a ta HRl Hi HR].
    intros; constructor.
    specialize (H c a HR).
    intros Ha.
    inversion Ha; subst; clear Ha.
    constructor; auto.
  Qed.

End RefinementProperties.

Section Simulation.

  Definition simulation :=
    (forall c, initial mc c -> exists a, R c a /\ initial ma a) /\
    (forall c c' a, step mc c c' -> R c a -> exists a', R c' a' /\ step m

  Theorem simulation_refinement :
    simulation -> refinement.
  Proof.
    unfold simulation, refinement.
    intros (Hinit,Hstep) tc Htc.

    destruct tc as [| c tc].
    exists [].
    split; constructor.
```

```
      induction Htc.

      rename x into c.
      specialize (Hinit c H).
      inversion Hinit as (a,(HR,Ha)).
      exists [a].
      split; repeat constructor; auto.

      rename x into c.
      rename x' into c'.
      rename t into tc.
      inversion IHHtc as (ta, (HR, Hta)); clear IHHtc.
      inversion HR; subst; clear HR.
      rename y into a.
      rename ys into ta.
      rename H into Hsc.
      rename H4 into HR.
      specialize (Hstep c c' a Hsc HR).
      inversion Hstep as (a', (HR',Ha)); clear Hstep.
      exists (a'::a::ta).
      split; repeat constructor; auto.
    Qed.

  End Simulation.

End Refinement.


Section BiRefinement.

  Variable A : Type.
  Variable B : Type.
  Variable ma : Process A.
  Variable mb : Process B.
  Variable R : A -> B -> Prop.

  Definition birefinement :=
    refinement ma mb (transp R) /\ refinement mb ma R.

End BiRefinement.

(* FILE: Semantics.v *)
Require Import Chart.Process.
Require Import Chart.Util.
Require Import Coq.Logic.ProofIrrelevance.
```

```
Require Import Coq.Logic.Decidable.
Set Implicit Arguments.

Section Chart.

  Variable S : Set.
  Variable E : Set.

  Record Machine :=
    { machine_initial : (S * E) -> Prop;
      machine_terminal : (S * E) -> Prop;
      machine_inner : S -> E -> E -> Prop;
      machine_step : (S * E) -> (S * E) -> Prop
    }.

  Inductive Chart :=
  | Unit : Chart
  | Par : Chart -> Chart -> Chart
  | Nest : Machine -> (S -> Chart) -> Chart.

  Inductive Config :=
  | U : Config
  | P : Config -> Config -> Config
  | N : S -> Config -> Config.

  Inductive chart_config : Chart -> Config -> Prop :=
  | CC_Unit : chart_config Unit U
  | CC_Par : forall chl chr sl sr,
      chart_config chl sl ->
      chart_config chr sr ->
      chart_config (Par chl chr) (P sl sr)
  | CC_Nest : forall x m cs s,
      chart_config (cs x) s ->
      chart_config (Nest m cs) (N x s).

  Theorem dec_chart_config :
    forall ch cfg, decidable (chart_config ch cfg).
  Proof.
    induction ch.
    destruct cfg.
    left. constructor.
    right; intros Hcc; inversion Hcc.
    right; intros Hcc; inversion Hcc.

    destruct cfg.
```

```
  right; intros Hcc; inversion Hcc.
  specialize IHch1 with cfg1.
  specialize IHch2 with cfg2.
  destruct IHch1, IHch2.
  left; constructor; assumption.
  right; intros Hcc; inversion Hcc; subst; contradiction.
  right; intros Hcc; inversion Hcc; subst; contradiction.
  right; intros Hcc; inversion Hcc; subst; contradiction.
  right; intros Hcc; inversion Hcc.

  destruct cfg.
  right; intros Hcc; inversion Hcc.
  right; intros Hcc; inversion Hcc.
  specialize (H s cfg).
  destruct H.
  left; constructor; assumption.
  right; intros Hcc; inversion Hcc; subst; contradiction.
Qed.

Fixpoint chart_config_bool (ch : Chart) (s : Config) : bool :=
  match (ch,s) with
  | (Unit, U) => true
  | (Par chl chr, P sl sr) => chart_config_bool chl sl && chart_config_bo
  | (Nest m cs, N x s) => chart_config_bool (cs x) s
  | _ => false
  end.

Theorem chart_config_bool_correct :
  forall ch s, chart_config ch s <-> chart_config_bool ch s = true.
Proof.
  intros ch s.
  split; generalize dependent s.

  induction ch; intros s Hcc;
    inversion Hcc; subst; clear Hcc; simpl; intuition.

  induction ch; intros s Hcc;
    destruct s; inversion Hcc; subst; clear Hcc;
      match goal with
      | [ H : andb _ _ = true |- _ ] =>
        rewrite Bool.andb_true_iff in H
      | _ => idtac
      end; constructor.
  apply IHch1; intuition.
  apply IHch2; intuition.
```

```
  apply H; assumption.
Qed.

Inductive chart_initial : Chart -> (Config * E) -> Prop :=
| CI_Unit : forall env,
    chart_initial Unit (U, env)
| CI_Par : forall ch1 cfg1 ch2 cfg2 env,
    chart_initial ch1 (cfg1, env) ->
    chart_initial ch2 (cfg2, env) ->
    chart_initial (Par ch1 ch2) (P cfg1 cfg2, env)
| CI_Nest : forall m cs x cfg env,
    machine_initial m (x, env) ->
    chart_initial (cs x) (cfg, env) ->
    chart_initial (Nest m cs) (N x cfg, env).

Lemma chart_initial_config :
  forall ch cfg env, chart_initial ch (cfg,env) -> chart_config ch cfg.
Proof.
  intros ch cfg env.
  generalize dependent cfg.
  induction ch; intros cfg Hi; inversion Hi; subst; clear Hi; constructor
  apply IHch1; assumption.
  apply IHch2; assumption.
  apply H; assumption.
Qed.

Inductive chart_terminal : Chart -> (Config * E) -> Prop :=
| CT_Unit : forall env,
    chart_terminal Unit (U, env)
| CT_Par : forall ch1 cfg1 ch2 cfg2 env,
    chart_terminal ch1 (cfg1, env) ->
    chart_terminal ch2 (cfg2, env) ->
    chart_terminal (Par ch1 ch2) (P cfg1 cfg2, env)
| CT_Nest : forall m x cs cfg env,
    machine_terminal m (x, env) ->
    chart_terminal (cs x) (cfg, env) ->
    chart_terminal (Nest m cs) (N x cfg, env).

Lemma chart_terminal_config :
  forall ch cfg env, chart_terminal ch (cfg,env) -> chart_config ch cfg.
Proof.
  intros ch cfg env.
  generalize dependent cfg.
  induction ch; intros cfg Ht; inversion Ht; subst; clear Ht; constructor
  apply IHch1; assumption.
```

```
  apply IHch2; assumption.
  apply H; assumption.
Qed.

Inductive chart_step : Chart -> (Config * E) -> (Config * E) -> Prop :=
| CS_Unit : forall env env',
    chart_step Unit (U, env) (U, env')
| CS_Par : forall ch1 cfg1 cfg1' ch2 cfg2 cfg2' env env',
    chart_step ch1 (cfg1, env) (cfg1', env') ->
    chart_step ch2 (cfg2, env) (cfg2', env') ->
    chart_step (Par ch1 ch2) (P cfg1 cfg2, env) (P cfg1' cfg2', env')
| CS_Nest_outer : forall m cs x cfg env x' cfg' env',
    machine_step m (x, env) (x', env') ->
    chart_terminal (cs x) (cfg, env) ->
    chart_initial (cs x') (cfg', env') ->
    chart_step (Nest m cs) (N x cfg, env) (N x' cfg', env')
| CS_Nest_inner : forall m cs x cfg env cfg' env',
    machine_inner m x env env' ->
    chart_step (cs x) (cfg, env) (cfg', env') ->
    chart_step (Nest m cs) (N x cfg, env) (N x cfg', env').

Lemma chart_step_src_config :
  forall ch cfg1 env1 cfg2 env2,
    chart_step ch (cfg1,env1) (cfg2,env2) -> chart_config ch cfg1 /\ char
Proof.
  intros ch cfg1 env1 cfg2 env2 Ht.
  split;
    generalize dependent cfg2;
    generalize dependent cfg1;
    induction ch; intros cfg1 cfg2 Ht;
      inversion Ht; subst; clear Ht; constructor.
  specialize (IHch1 _ _ H2); assumption.
  specialize (IHch2 _ _ H6); assumption.
  apply chart_terminal_config with env1; assumption.
  specialize (H _ _ _ H7); assumption.
  specialize (IHch1 _ _ H2); assumption.
  specialize (IHch2 _ _ H6); assumption.
  apply chart_initial_config with env2; assumption.
  specialize (H _ _ _ H7); assumption.
Qed.

Inductive chart_refines : Chart -> Chart -> Prop :=
| CR_Unit : forall ch, chart_refines Unit ch
(* | CR_Par_l : forall ch1 ch2, chart_refines ch1 (Par ch1 ch2) *)
(* | CR_Par_r : forall ch1 ch2, chart_refines ch2 (Par ch1 ch2) *)
```

```
| CR_Par : forall ch11 chr1 ch12 chr2,
    chart_refines ch11 ch12 ->
    chart_refines chr1 chr2 ->
    chart_refines (Par ch11 chr1) (Par ch12 chr2)
| CR_Nest : forall m cs1 cs2,
    (forall x, chart_refines (cs1 x) (cs2 x)) ->
    chart_refines (Nest m cs1) (Nest m cs2).

Inductive config_refines : Config -> Config -> Prop :=
| CFR_U : forall cfg, config_refines U cfg
| CFR_P : forall cfgl1 cfgl2 cfgr1 cfgr2,
    config_refines cfgl1 cfgl2 ->
    config_refines cfgr1 cfgr2 ->
    config_refines (P cfgl1 cfgr1) (P cfgl2 cfgr2)
| CFR_N : forall x cfg1 cfg2,
    config_refines cfg1 cfg2 ->
    config_refines (N x cfg1) (N x cfg2).

(* ch1 - the abstract chart *)
(* cfg2 - the concrete config *)
Fixpoint config_proj_opt (ch1 : Chart) (cfg2 : Config) : option Config :=
  match (ch1, cfg2) with
  | (Par ch1 chr, P cfgl cfgr) =>
    match config_proj_opt ch1 cfgl with
    | None => None
    | Some cfgl' =>
      match config_proj_opt chr cfgr with
      | None => None
      | Some cfgr' => Some (P cfgl' cfgr')
      end
    end
  | (Nest _ cs, N x cfg) =>
    match config_proj_opt (cs x) cfg with
    | None => None
    | Some cfg' => Some (N x cfg')
    end
  | (Unit, _) => Some U
  | (_, _) => None
  end.

Section Refinement.

  Variables ch1 ch2 : Chart.
  Hypothesis Href : chart_refines ch1 ch2.
```

```
Lemma config_refines_exists_unique :
  forall s2,
    chart_config ch2 s2 ->
    exists! s1, chart_config ch1 s1 /\ config_refines s1 s2.
Proof.
  induction Href; intros s2 Hcc2.

  (* Unit *)
  exists U; split.
  split; constructor.
  intros s (Hcc1, Huniq).
  inversion Hcc1.
  reflexivity.

  (* Par *)
  inversion Hcc2 as [|chl chr sl sr Hl Hr|]; subst; clear Hcc2.
  assert (exists! s, chart_config chl1 s /\ config_refines s sl) as Hl'
      by (apply IHc1; assumption).
  assert (exists! s, chart_config chr1 s /\ config_refines s sr) as Hr'
      by (apply IHc2; assumption).
  inversion Hl' as (sl', ((Hc1a, Hc1b), Hl_uniq)).
  inversion Hr' as (sr', ((Hc2a, Hc2b), Hr_uniq)).
  exists (P sl' sr').
  split.
  split; constructor; assumption.
  intros s' (Hcc, Huniq).
  inversion Hcc; subst; clear Hcc.
  inversion Huniq; subst; clear Huniq.
  specialize (Hl_uniq sl0).
  specialize (Hr_uniq sr0).
  rewrite Hl_uniq by intuition.
  rewrite Hr_uniq by intuition.
  reflexivity.

  (* Nest *)
  inversion Hcc2 as [|x m' cs s Hcs|]; subst; clear Hcc2.
  assert (exists !s', chart_config (cs1 x) s' /\ config_refines s' s) as
      by (apply H0; intuition).
  inversion Hs' as (s', ((Ha, Hb), Huniq)).
  exists (N x s').
  split.
  split; constructor; assumption.
  intros s'' (Hcc, Huniq').
  inversion Hcc; subst; clear Hcc.
  inversion Huniq'; subst; clear Huniq'.
```

```
  specialize (Huniq s0).
  rewrite Huniq by intuition.
  reflexivity.
Qed.

Definition config_proj_rel (s2 : {s | chart_config ch2 s}) (s1 : {s | c
  config_refines (proj1_sig s1) (proj1_sig s2).

Theorem config_proj_rel_func :
  function config_proj_rel.
Proof.
  unfold function, config_proj_rel.
  intros (s2, Hcc2).
  assert (exists! s1, chart_config ch1 s1 /\ config_refines s1 s2) as H
      by (apply config_refines_exists_unique; assumption).
  inversion Hc as (s1, ((Hcc1,Hcr), Huniq)).
  exists (exist (chart_config ch1) s1 Hcc1).
  split; try assumption.
  simpl.
  intros (s1', Hcc1') H.
  simpl in H.
  specialize (Huniq s1').
  assert (s1 = s1') by (apply Huniq; intuition). subst.
  assert (Hcc1 = Hcc1') by apply proof_irrelevance. subst. (* Yuck *)
  reflexivity.
Qed.

Lemma config_proj_correct1 :
  forall s1 s2,
    chart_config ch2 s2 ->
    config_proj_opt ch1 s2 = Some s1 ->
    chart_config ch1 s1 /\ config_refines s1 s2.
Proof.
  induction Href as [ | | m cs1 cs2 Hr IH]; intros s1 s2 Hcc Hp.

  (* Unit *)
  inversion Hp; subst; clear Hp.
  split; constructor.

  (* Par *)
  inversion Hcc; subst; clear Hcc.
  simpl in Hp.
  destruct (config_proj_opt ch11 s1) as [s1'|] eqn:Hl.
  destruct (config_proj_opt ch1r sr) as [sr'|] eqn:Hr.
  inversion Hp; subst; clear Hp.
```

```
  specialize (IHc1 c1 sl' sl).
  specialize (IHc2 c2 sr' sr).
  split; constructor; tauto.
  inversion Hp.
  inversion Hp.

  (* Nest *)
  inversion Hcc; subst; clear Hcc.
  inversion Href; subst; clear Href.
  inversion Hp; subst; clear Hp.
  destruct (config_proj_opt (cs1 x) s) eqn:Hc.
  inversion H1; subst; clear H1.
  specialize (IH x (H0 x) c s).
  split; constructor; tauto.
  inversion H1.
Qed.

Lemma config_proj_correct2 :
  forall s1 s2,
    chart_config ch2 s2 ->
    chart_config ch1 s1 ->
    config_refines s1 s2 ->
    config_proj_opt ch1 s2 = Some s1.
Proof.
  induction Href; intros s1 s2 Hcc2 Hcc1 Hcr.

  (* Unit *)
  inversion Hcc1; subst; clear Hcc1.
  reflexivity.

  (* Par *)
  inversion Hcc1; subst; clear Hcc1.
  inversion Hcc2; subst; clear Hcc2.
  inversion Hcr; subst; clear Hcr.
  rename sl0 into sl'.
  rename sr0 into sr'.
  simpl.
  destruct (config_proj_opt chl1 sl') eqn:Hl.
  destruct (config_proj_opt chr1 sr') eqn:Hr.

  specialize (IHc1 c1 sl sl').
  specialize (IHc2 c2 sr sr').
  rewrite IHc1 in Hl by tauto.
  rewrite IHc2 in Hr by tauto.
  inversion Hl; subst; clear Hl.
```

```
    inversion Hr; subst; clear Hr.
    reflexivity.

    specialize (IHc2 c2 sr sr').
    rewrite IHc2 in Hr by tauto.
    inversion Hr.

    specialize (IHc1 c1 sl sl').
    rewrite IHc1 in Hl by tauto.
    inversion Hl.

    (* Nest *)
    inversion Hcc1; subst; clear Hcc1.
    inversion Hcc2; subst; clear Hcc2.
    inversion Hcr; subst; clear Hcr.
    rename x0 into x.
    rename cs1 into cs.
    rename cs2 into cs'.
    rename s0 into s'.
    simpl.
    assert (config_proj_opt (cs x) s' = Some s) by (apply H0; try assumpt
    destruct (config_proj_opt (cs x) s') eqn:Hc; inversion H1; subst; ref
  Qed.

  Theorem config_proj_correct :
    forall s1 s2,
      chart_config ch2 s2 ->
      config_proj_opt ch1 s2 = Some s1 <-> chart_config ch1 s1 /\ config_
  Proof.
    split; intros.
    apply config_proj_correct1; tauto.
    apply config_proj_correct2; tauto.
  Qed.

End Refinement.


(* Refinement *)

Definition chart_process (ch : Chart) : Process (Config * E) :=
  {| initial := chart_initial ch;
     step := chart_step ch
  |}.

Theorem chart_refines_refinement :
```

```
    forall ch1 ch2,
      chart_refines ch1 ch2 ->
      refinement (chart_process ch1) (chart_process ch2) eq.
  Proof.
    intros ch1 ch2 Href.
    unfold refinement.
(*    intros tc ta Hlr.
    replace ta with tc by (apply list_rel_eq__list_eq; assumption).
*)
  Abort.

End Chart.
```

4. CONCLUSION

Currently Q-Charts are being actively used in NW development. A translator from Simulink/Stateflow has been written (outside of the scope of this work) and engineers are using Q-Charts to verify their designs. Beyond this, proofs of implementation code for these systems are translated from the same Q-Chart SCXML. This has enabled Sandia to give evidence that safety, security and reliability properties are upheld by our NW controllers. Previously, the only assurance for these properties was a "best effort" by the engineers. Beyond the reach of testing, there was no mechanism to demonstrate definitively why these properties held.

It is our hope that this is just the beginning. There are indications that the way NW control systems are engineered and verified is changing to a more evidenced-based methodology. The work is not nearly done. While we have formal evidence for every layer of abstraction from the systems executable specification down to the implementation binary (alas not the processor yet). Even so, it is not the *best* evidence. There are seams between the abstraction layers where the semantics appears to line up but it is not yet proven to be so. Currently it is just assumed axiomatically. Eventually we intend to have a semantic representation of every abstraction layer in a theorem prover (probably Coq) and be able *prove* consistency between abstraction layers. In the end we expect to have "one QED" for all of the software and hardware that goes into a nuclear weapon. We are not there yet, and it may take a decade or so, but we can now see a clear path.

In the meantime we are not taking the usual ASC throw it over the fence approach. We are contributing and having an impact on the weapons programs currently underway, in spite of imperfections. We believe that this approach, though slower and less efficient, increases the chance that the output of this program will be useful and used. The ultimate goal is safer, more secure and more reliable nuclear weapons.

REFERENCES

[1] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[3] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[4] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[5] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

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

[7] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In Paolo Ciancarini, Alessandro Fantechi, and Robert Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347, Boston, MA, 1999. Springer US.

[8] MathWorks. Stateflow documentation, 2016. Available at `https://www.mathworks.com/help/stateflow`.

[9] Scott McGlashan, Rafah Hosn, Marc Helbing, James Barnett, Jerry Carter, Klaus Reifenrath, RJ Auburn, Johan Roxendal, Rahul Akolkar, Noam Rosenthal, Torbjörn Lager, Daniel Burnett, T. V. Raman, and Michael Bodell. State chart XML (SCXML): State machine notation for control abstraction. W3C recommendation, W3C, September 2015. http://www.w3.org/TR/2015/REC-scxml-20150901/.

[10] Karla Morris, Colin Snook, Thai Son Hoang, Robert Armstrong, and Michael Butler. Refinement of statecharts with run-to-completion semantics. In *The Sixth International Workshop on Formal Techniques for Safety-Critical Systems (16/11/18)*, November 2018.

[11] Clayton Shepard, Hang Yu, Narendra Anand, Erran Li, Thomas Marzetta, Richard Yang, and Lin Zhong. Argos: Practical many-antenna base stations. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 53–64, New York, NY, USA, 2012. ACM.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email– [redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

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.