Pragmatic verification for hybrid and real-time designs

Mark R. Greenstreet
Computer Science Department, University of British Columbia
Vancouver, British Columbia, Canada
mrg@cs.ubc.ca

Abstract— **Hybrid and real-time designs are ubiquitous in computing and control systems. There are three prevalent methods for verifying real-time and hybrid systems: simulation, model-checking, and theorem proving. None of these approaches are ideal: simulation can never verify a design for all possible inputs and parameters when these are continuous quantities; model checking methods are typically applicable for only relatively small and inexpressive models; and theorem proving requires substantial manual effort from highly skilled logicians.**

This paper presents a verification approach based on refinement. In our refinement framework, we first establish safety properties for a simplified, abstract model of the design and then show that these are inherited by progressively more detailed models. In this approach, we see a hybrid system as a hybrid *model*: part of the system is modeled with discrete time and state, and other parts are modeled with continuous quantities. This paper illustrates this approach with the design of a simple state-machine for a traffic light.

I. Introduction

Consider the problem of verifying the correctness of a design. Verification is with respect to a mathematical model, M, of the design. This model may be expressed using discrete mathematics, continuous mathematics, or a mixture of the two. In the latter case, we say that M is a *hybrid-model*. Properties that we may wish to verify can be classified as *safety* properties or *liveness* properties. A safety property states that the system does not enter a "bad" state (alternatively, that all reachable states of the system are "good"). Liveness properties state that the system will eventually perform some desired action.

In this paper, I consider the verification of safety properties. The verification of safety properties only requires considering the instantaneous states that a model can reach whereas verification of liveness properties requires considering complete executions. By focusing on safety properties, the verification often problem becomes more tractable allowing the use of more realistic models. Furthermore, for timed systems, many liveness properties are expressed with timing bounds. Bounded-time liveness properties can be expressed as safety properties given that the model guarantees the progress of time (i.e. "non-Zenoness") [AL94]. Although non-Zenoness is a liveness property, for many practical models, it can be established from simple syntactic properties of the model.

Establishing a safety property requires characterizing the set of all reachable states of the model. For a finite, discrete model, this set may be extremely large, growing exponentially with the number of discrete variables in the model. For a continuous model, the set of reachable states is uncountable. Symbolic methods allow tractable representations of these sets for some problems [BC+94], [AHH96]; however, the state explosion problem remains a serious problem for verification techniques. Refinement based reasoning [AL91] is a promising alternative to direct model-checking. In this approach, safety properties are established for simple, abstract models of the design. This model is progressively refined by modeling more and more implementation details until the desired model is obtained. As shown in section III, refinement can often be established by tautology checking which tends to be much less computationally intensive than the fixpoint calculations of model checking. This allows us to verify properties for models that are too detailed for model checking. In [KOTG99] we used this approach to verify a self-timed divider starting with a three-line program for SRT division and applying five levels of refinement to verify a timed, switch-level model of a divider chip [WH+87].

This paper describes the use of refinement between continuous models and discrete ones. This enables reasoning about systems with continuous states to show that they correctly implement discrete specifications. Section II introduces a traffic light that I use as an example throughout the paper. Section III presents the key ideas of refinement. These ideas are employed in section IV to obtain an implementation of the traffic light using synchronous, two-phase logic. Section V extends this refinement to a non-linear ODE model that implements the synchronous version of the light.

II. A Traffic Light Controller

Consider a simple traffic light at the junction of a north-south road and an east-west road. I'll model the light with two variables:

This work was supported in part by NSERC research grant OGP-0138501.

$$\begin{aligned} &\ll (\mathsf{ns} = \mathsf{R}) \text{ AND } (\mathsf{ew} = \mathsf{R}) \rightarrow \mathsf{ns} := \mathsf{G} \gg \\ \| &\ll \mathsf{ns} = \mathsf{G} \rightarrow \mathsf{ns} := \mathsf{Y} \gg \\ \| &\ll \mathsf{ns} = \mathsf{Y} \rightarrow \mathsf{ns} := \mathsf{R} \gg \\ \| &\ll (\mathsf{ns} = \mathsf{R}) \text{ AND } (\mathsf{ew} = \mathsf{R}) \rightarrow \mathsf{ew} := \mathsf{G} \gg \\ \| &\ll \mathsf{ew} = \mathsf{G} \rightarrow \mathsf{ew} := \mathsf{Y} \gg \\ \| &\ll \mathsf{ew} = \mathsf{Y} \rightarrow \mathsf{ew} := \mathsf{R} \gg \end{aligned}$$

Fig. 1. P_{light}: a Synchronized Transitions program for a traffic light.

$$\mathsf{ns}, \mathsf{ew}\text{: } \{\mathsf{R}, \mathsf{G}, \mathsf{Y}\}$$

If ns has the value Y, this means that the north-south light is yellow in the current state. To model actions of the light, I use the Synchronized Transitions notation [Sta93]. For example, consider the command:

$$\ll \mathsf{ew} = \mathsf{R} \rightarrow \mathsf{ns} := \mathsf{G} \gg$$

The expression to the left of the arrow is the guard. The assignment to the right of the arrow may be performed whenever the guard is enabled. In the example above, this means that the north-south light may turn green whenever the east-west light is red. These guarded commands are called "transitions." Transitions may be combined using the asynchronous combinator, $\|$. For example:

$$\begin{aligned} &\ll \mathsf{ew} = \mathsf{R} \rightarrow \mathsf{ns} := \mathsf{G} \gg \\ \| &\ll \mathsf{ns} = \mathsf{R} \rightarrow \mathsf{ew} := \mathsf{G} \gg \end{aligned}$$

specifies that if the east-to-west light is red, then the north-to-south light may turn green. Likewise, if the north-to-south light is red, then the east-to-west light may turn green. If both transitions are enabled, then *one* is selected non-deterministically and executed. In the example above, if both lights are red, then either light may turn green, but not both. Transitions execute atomically.

Figure 1 shos a Synchronized Transitions program for a simple traffic light. I will write P_{light} to denote this program. To show that this traffic light is safe, I will show that in all states, at least one of the lights is read. More formally, I will show that Q_{light} is a safety property of the program (i.e. Q_{light} holds in all states) where

$$Q_{\text{light}} \equiv (\mathsf{ns} = \mathsf{R}) \text{ OR } (\mathsf{ew} = \mathsf{R})$$

Clearly, Q_{light} must hold in the initial state if it is to be a safety property. I'll write $H_{0,\text{light}}$ to denote the initial state predicate for the program and assume

$$H_{0,\text{light}} \equiv (\mathsf{ns} = \mathsf{R}) \text{ AND } (\mathsf{ew} = \mathsf{R})$$

There are three standard ways to establish a safety property. First, one can start with $H_{0,\text{light}}$ and find all states that are reachable after performing any number of actions. Let $H_{n,\text{light}}$ denote the states that can be reached after at most n steps. We have:

$$\begin{aligned} H_{0,\text{light}} &\equiv (\text{ns} = \text{R}) \text{ AND } (\text{ew} = \text{R}) \\ H_{1,\text{light}} &\equiv \quad (((\text{ns} = \text{R}) \text{ OR } (\text{ns} = \text{G})) \text{ AND } (\text{ew} = \text{R})) \\ &\quad \text{OR } (((\text{ew} = \text{R}) \text{ OR } (\text{ew} = \text{G})) \text{ AND } (\text{ns} = \text{R})) \\ H_{2,\text{light}} &\equiv (\text{ns} = \text{R}) \text{ OR } (\text{ew} = \text{R}) \\ H_{3,\text{light}} &\equiv (\text{ns} = \text{R}) \text{ OR } (\text{ew} = \text{R}) \end{aligned}$$

Because $H_{2,\text{light}} = H_{3,\text{light}}$, an induction argument shows $\forall n \geq 2. H_{n,\text{light}} \equiv (\mathsf{ns} = \mathsf{R}) \text{ OR } (\mathsf{ew} = \mathsf{R})$. Thus, all reachable states of P_{light} satisfy Q_{light} as required. This approach is called forward reachability analysis in [BC+94] or $sin(P_{\text{light}}, H_{0,\text{light}})$ (i.e. strongest invariant) in [Lam87]. In general, if $sin(P, A_0) \Rightarrow Q$, then Q is a safety property of P.

The second approach is to start with the desired safety property and find all states that satisfy the property and are guaranteed to satisfy it for the the next n steps. The fixpoint of this iteration is also an invariant. This approach is called backward reachability analysis in [BC+94] and Lamport calls this fixpoint $win(P, Q)$. If $A_0 \Rightarrow win(P, Q)$, then Q is a safety property of P. In the above example, Q_{light} is an invariant; accordingly, $win(P_{\text{light}}, Q_{\text{light}}) = Q_{\text{light}}$.

Forward and backward reachability analysis involve a fixpoint computation which can be computationally intensive. This limits these methods to systems with relatively simple models. Reachability analysis derives invariants from the program's transitions and the initial state predicate or the desired safety property. In general, these two invariants are not the same. If Q is a safety property, then $sin(P_{\text{light}}, A_0) \Rightarrow win(P_{\text{light}}, Q)$ but it is not necessarily the case that $sin(P_{\text{light}}, A_0) = win(P_{\text{light}}, Q)$. These observations motivate the third approach to verifying a properties.

To show that Q is a safety property of P, it is sufficient to find *any* predicate I, such that I is an invariant, $A_0 \Rightarrow I$, and $I \Rightarrow Q$. To show that I is an invariant of a program, one can show that for each transition of the program, I holding before performing the transition implies that I will continue to hold after performing the transition. More formally, let $\ll G \rightarrow M \gg$ be a transition with guard G and action (i.e. multi-assignment) M. We can view M as a function from states to states. A predicate I is maintained by a transition $\ll G \rightarrow M \gg$ if and only if $(I \text{ AND } G) \Rightarrow I \circ M$, where $\circ$ denotes function composition. A predicate I is an invariant of a program if it is maintained by all transitions of the program.

Unlike forward and backward reachability analysis, establishing that a given predicate is an invariant does not require a fixpoint calculation. This allows the method of testing invariants to be applied to larger systems than the reachability approaches. The challenge with this approach is finding the required invariant. The refinement techniques presented in this paper simplify this task.

III. Refinement

Let P and P' be two programs. Program P' is a refinement of program P if everything that P' can do is permitted by program P. In refinement based design, a designer starts with a highly abstract program with the required safety properties. As the designer makes design choices, he or she produces progressively more specific programs. Each program in this sequence is a refinement of its predecessors. A refinement based approach allows the designer to verify key properties of a design at a high-level of abstraction where the verification is relatively simple. By showing that each more detailed programs is a refinement of the original one, all programs, including the final design, are guaranteed to inherit the properties of the original, high-level abstraction.

Definition 1 (Refinement) Let P and P' be two programs. Let R be the state transition relation for P: $(s_1, s_2) \in R$ if there is a transition of P that is enabled in state s_1 such that performing this transition leads to state s_2. Likewise, let R' be the state transition relation for P'. Let A be a function from states of P' to states of P. Program P' is a refinement of program P if for every reachable state s'_1 of P', and every state s'_2 with $(s'_1, s'_2) \in R'$, either $(A(s'_1), A(s'_2)) \in R$ or $A(s'_1) = A(s'_2)$.

In the first case, P has a state transition corresponding to the state transition of P'. In the second case, the state transition of P' is invisible in the state space of P. Such a state transition is called a stuttering move.

A program P' is a refinement of P if each transition of P' is a refinement of P. This means that refinement can be verified one transition of P' at a time. Likewise, P' is a refinement of P if it is a refinement of any subset of the transitions of P. Often, it is clear which transitions of P "explain" a transition of P' (for example, those that write corresponding variables). By using these rules, refinement can be verified without ever constructing the full state transition relations of either program. This is similar to the disjunctive partitioning method for model checking described in [BC+94].

An important property of refinement is that if program P' is a refinement of P, then P' inherits all of the safety properties of P. More formally:

Theorem 2 (Inheritance of Safety Properties) Let P and P' be two programs such that P' and A be an abstraction function such that P' is a refinement of P under abstraction function A. Let Q be a safety property of program P. Then $Q \circ A$ is a safety property of program P'.

Inheritance of safety properties provides a convenient way to verify refinement.

Theorem 3 (Verifying Refinement) Let P and P' be two programs and A be an abstraction mapping from states of P' to states of P. Let Q be a safety property of P, and let I' be a proposed invariant of P' (i.e. the invariance of I' must be verified). Program P' has invariant I' and is a refinement of P if $H'_0 \Rightarrow (H_0 \circ A)$ is a tautology and

```
ns, ew: ARRAY[0..1] OF BOOLEAN;
H_{0,light'} ≡      ns[1] AND (NOT ns[0])
                AND ew[1] AND (NOT ew[1]);
I' ≡        (NOT (ns[1] AND ns[0]))
            AND (NOT (ew[1] AND ew[0]))
   ≪ ns[1], ns[0] := ns[0] OR (NOT ew[1]),
                     NOT (ns[0] OR ns[1]) ≫
|| ≪ ew[1], ew[0] := ew[0] OR (NOT ns[1]),
                     NOT (ew[0] OR ew[1]) ≫
```

Fig. 2. $P_{\text{light}'}$: a refinement of the traffic light program.

$$\begin{array}{l}\forall (s'_1, s'_2) \in R'.\ (Q(A(s'_1))\ \mathsf{AND}\ I'(s'_1)) \Rightarrow \\ \quad (((A(s'_1), A(s'_2)) \in R)\ \mathsf{OR}\ (A(s'_1) = A(s'_2))) \\ \quad \mathsf{AND}\ I'(s'_2)\end{array}$$

Theorem 3 says that when verifying refinement, one may assume that safety properties of the abstract program hold in the "before" state of a state-transition. Using this theorem, a designer can establish invariants for highly abstracted version of the design (where model checking and/or manual methods are practical). These invariants can be used when verifying refinement. The verification step amounts to tautology checking. In practice, tautology checking can be applied to much larger models than the fixpoint computations of model checking. Thus, theorem 3 provides a practical basis for verifying large designs.

One may observe that the tautology checks of theorem 3 are essentially the steps required to directly verify a proposed invariant. From this perspective, the refinement methodology is simply a convenient way to represent invariants of the implementation program in terms of invariants of the specification.

To demonstrate refinement, figure 2 shows refinement of the traffic light program from figure 1 where each light is modeled by two boolean variables. Let $P_{\text{light}'}$ denote the program of figure 2. The mapping from states of $P_{\text{light}'}$ to P_{light} is:

$P_{\text{light}'}$.ns[1]	$P_{\text{light}'}$.ns[0]	P_{light}.ns
false	false	G
false	true	Y
true	false	R
true	true	X

and likewise for P_{light}.ew and $P_{\text{light}'}$.ew. Let A denote this mapping from states of program $P_{\text{light}'}$ to states of program P_{light}. This program uses *multi-assignments*: a transition of the form $\ll g \rightarrow a, b := x, y \gg$ specifies that from states where g is satisfied, a may receive the value of x and b may receive the value of y. A multi-assignment is performed as a single atomic action; conceptually, all expressions on the right side of the assignment are evaluated first, and then all variables on the left side are updated. A transition without a guard is always enabled.

I will verify that $P_{\text{light}'}$ is a refinement of P_{light}. First, $H_{0,\text{light}'} \Rightarrow H_{0,\text{light}} \circ A$ is clearly a tautology. To verify the first transition of program $P_{\text{light}'}$ requires showing that

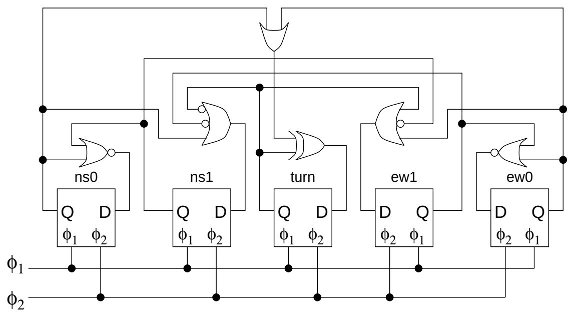

Fig. 3. A Synchronous Implementation of the Traffic Light

(((ns[1] AND (NOT ns[0])) OR (ew[1] AND (NOT ew[0]))) AND (NOT (ns[1] AND ns[0])) AND (NOT (ew[1] AND ew[0]))) $\Rightarrow$ ((((ns[1] AND (NOT ns[0])) AND ((NOT (ns[0] OR (NOT ew[1]))) AND (NOT (NOT (ns[0] OR ns[1]))))) OR (((NOT ns[1]) AND (NOT ns[0])) AND ((NOT (ns[0] OR (NOT ew[1]))) AND (NOT (ns[0] OR ns[1])))) OR (((NOT ns[1]) AND ns[0]) AND ((ns[0] OR (NOT ew[1])) AND (NOT (NOT (ns[0] OR ns[1]))))) OR (((ns[0] OR (NOT ew[1])) = ns[1]) AND ((NOT (ns[0] OR ns[1])) = ns[0]))) AND ((NOT ((ns[0] OR (NOT ew[1])) AND (NOT (ns[0] OR ns[1])))) AND (NOT (EW[1] AND EW[0]))))

This can be readily and shown to be a tautology (computers are handy for this kind of tedium). Verifying the second transition is similar. Thus, $P_{\text{light}'}$ is a refinement of P_{light}: one light is red in every state reached by $P_{\text{light}'}$, and the lights in $P_{\text{light}'}$ cycle from red through green to yellow and back to red onlyonly in sequences allowed by P_{light}.

IV. A Synchronous Traffic Light

Figure 3 shows a synchronous implementation of the traffic light from figure 2. Two-phases latches hold the state for the traffic light. For example, the latch labeled ns0 holds state for variable ns[0] in program $P_{\text{light}'}$. Synchronous circuits are deterministic; therefore, the synchronous design refines the non-determinism of the previous program with a deterministic implementation. In this design, I introduced a variable called turn that enforces alternation of granting of green to the two directions. For simplicity, I omitted many details that are important for real traffic lights. For example, there are no timers; thus, the green and yellow states have equal duration.

Figure 4 shows an abridged version of a Synchronized Transitions program for the synchronous traffic light. The transitions are divided into three groups. The first group models the combination logic of the design. For brevity, I've only shown the transitions for generating the ns0 signals. The others are similar. The second group shows the generation of the two, non-overlapping clocks, ϕ_1 and ϕ_2. These transitions assume an internal timing generator that divides each clock period into four, consecutive intervals. The last group of transitions models this timing generator. The guards of these transitions assert that

```
   (* modeling the combinational logic *)
   ≪ ns0.d := NOT (ns0.q OR ns1.q) ≫
|| ≪ φ1 →ns0.e := ns0.d ≫
|| ≪ φ2 →ns0.q := ns0.e ≫
|| ... (* transitions for ns1, ew0, ew1, and turn *)
   (* generating the two-phase clocks *)
|| ≪ (phase = 0) → φ2 := FALSE ≫
|| ≪ (phase = 1) → φ1 := TRUE ≫
|| ≪ (phase = 2) → φ1 := FALSE ≫
|| ≪ (phase = 3) → φ2 := TRUE ≫
   (* the internal clock phase generator *)
|| ≪ (phase = 0) AND (NOT φ2) → phase := 1 ≫
|| ≪     (phase = 1) AND φ1 AND (ns0.e = ns0.d)
         AND (ns0.d = NOT (ns0.q OR ns1.q))
         AND ...
      → phase := 2
   ≫
|| ≪ (phase = 2) AND (NOT φ1) → phase = 3 ≫
|| ≪     (phase = 3) AND φ2 AND (ns0.q = ns0.e)
                 AND ...
      → phase := 0
   ≫
```

Fig. 4. $P_{\text{light}''}$: the synchronous traffic light.

each clock phase is long enough for the combinational logic to settle before the next phase begins. Again, I've only shown the clauses related to ns0.

The abstraction mapping from states of $P_{\text{light}''}$ to states of $P_{\text{light}'}$ is based on the observation that the .q signals are stable when phase $\in \{1, 2\}$, and the .e signals are stable when phase $\in \{0, 3\}$. This observation is formalized by the invariant:

$$\begin{array}{rl} & (\mathsf{phase} \in \{1,2\}) \Rightarrow \mathsf{NOT}\ \phi_2 \\ \mathsf{AND} & (\mathsf{phase} \in \{0,3\}) \Rightarrow \mathsf{NOT}\ \phi_1 \end{array}$$

This invariant is easily verified. The abstraction function for ns[0] is

$$\mathsf{ns[0]} \;=\; \begin{cases} \mathsf{ns0.q}, & \text{if } \mathsf{phase} \in \{1,2\} \\ \mathsf{ns0.e}, & \text{if } \mathsf{phase} \in \{0,3\} \end{cases}$$

The abstraction functions for ns[1], ew[0], and ew[1] are equivalent. With the above invariant and abstraction mappings, it is straightforward to show that program $P_{\text{light}''}$ is a refinement of program $P_{\text{light}'}$.

V. An ODE model for the traffic light

This section presents a simple ODE model that implements the synchronous traffic light. For brevity, I focus on the implementation of the ns0 variable. The implementations of the of the other variables are similar. Furthermore, I use simple, polynomial models. These do not reflect the actual dynamics of a realistic implementation technology such as MOSFET transistors. The models that I use here are amenable to simple analysis.

In the models used here, variables with values near +1 correspond to logical "true" values in a discrete model. Conversely, continuous variables with values near −1 correspond to logical "false" values in the discrete model. The NOR gate that produces ns0.d in figure 3 can be effected by:

$$\frac{d}{dt} ns0.d \quad = \quad \frac{1}{2}(1 - ns0.q)(1 - ns1.q) - 1 - ns0.d$$

If either $ns0.q$ or $ns1.q$ in near $+1$, then

$$\frac{d}{dt} ns0.d \quad \approx \quad -1 \ - \ ns0.d$$

This system has an attractor near -1, and the basin of attraction is the entire space. Interpretted in the logic abstraction, if either ns0.q or ns1.q is true, then the NOR gate drives ns0.d to false. Likewise, if both $ns0.q$ and $ns1.q$ have values near -1, then

$$\frac{d}{dt} ns0.d \quad \approx \quad +1 \ - \ ns0.d$$

which has an attractor near $+1$ and a basin of attraction of the entire space. In the logical abstraction, if both ns0.q and ns1.q are false, then the NOR gate drives ns0.d to true. Thus, the model given here has the qualitative properties of a NOR gate.

The latch models are similar. To model a transparent latch with data input d, data output q and enable signal ϕ, I use the equation:

$$\frac{d}{dt} d \quad = \quad (1 + \phi)(d - q) \ + \ (1 - \phi)(q - q^3)$$

If ϕ is near $+1$, then

$$\frac{d}{dt} d \quad \approx \quad d - q$$

which has an attractor when q is near d. In the logical abstraction, the the latch drives its output to the value of its data input when the enable input is true. Conversely, if ϕ is near -1, then

$$\frac{d}{dt} d \quad \approx \quad q - q^3$$

which has two attractors: $q = +1$ and $q = -1$ separated by an unstable equilibrium at $q = 0$. In the logical abstraction, the the latch maintains its output at its last logical value when the enable input is false.

The other components of the synchronous traffic light can be implemented in the same style.

To directly verify that the non-linear ODE model for a traffic light implements the synchronous version requires reasoning about a system with eighteen variables: three per latch, two for the clocks, and one for the output of the OR gate that drives the XOR gate that drives turn.d. Here, the refinement approach provides a simpler approach: section IV established the correct behavior for the synchronous program; to establish refinement, it is only necessary to verify individual transitions of the ODE model as justified by theorem 3. I will use the following specification for the clock signals: (1) each high interval for each clock phase is at least 1.35 time units wide; (2) each non-overlap interval (both clocks negative) is at least 0.15 time units wide; (3) low-to-high clock transitions have a slope of at least 40; (4) high-to-low clock transitions have a slope of at most -40. The first two requirements impose a minimum clock period of 4 time units.

Note that the $.q$ and $.d$ signals for each latch change when ϕ_2 is high, and the $.e$ signals change when ϕ_1 is high. To separate the proof of refinement into a separate proof for each logical signal, I will show for each latch that if all the $.q$ signals "settle quickly" for the current cycle, then the $.q$ output of the latch will settle quickly as well. A signal **settles quickly** if its absolute value is always at most 1, and if its difference from the value prescribed by the synchronous model is at most $1.9e^{-\tau}$ where τ is the amount of time that has elapsed since it was enabled to change. For example, τ is measured from the rising edge of ϕ_2 for changes of $ns0.q$.

Monotonicity arguments based on the ODE model show that the slowest transitions of $ns0.d$ occur when $ns0.q$ is initially at ± 1 and the $ns1.q$ changes as slowly as possible. This in turn leads to the slowest change of $ns0.e$ and $ns0.q$. Numerically integrating the ODE model for this worst-case shows that $ns0.d$ always settles quickly. Similar arguments apply for the other signals. These arguments can be formalized to show that the non-linear ODE model correctly implements the traffic-light state machine.

VI. Conclusions

This paper has presented a method for showing that a continuous system has a desired discrete behavior. The method is based upon refinement. With this method, global, safety properties are first established for a simple, abstract model of the system. More detailed models are verified by showing that each transition of the detailed model is allowed by the abstract model. In this approach, discrete actions are understood as abstractions of continuous behaviors. The approach was demonstrated by refining a simple state machine for a traffic light into a system on non-linear ODEs.

References

[AHH96] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. on Soft. Eng.*, 22(3):181–201, 1996.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM TOPLAS*, 16(5):1543–1571, September 1994.

[BC+94] J.R. Burch, E.M. Clarke, et al. Symbolic model checking for sequential circuit verification. *IEEE Transactions on CAD*, 13(4):401–424, April 1994.

[KOTG99] C. Kern, T. Ono-Tesfaye, and M.R. Greenstreet. A light-weight framework for hardware verification. In *TACAS'99*, pages 330–344, Amsterdam, March 1999. Springer. LNCS 1579.

[Lam87] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, DEC Systems Research Center, Palo Alto, CA, May 1987.

[Sta93] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1993.

[WH+87] T.E. Williams, M. Horowitz, et al. A self-timed chip for division. In *Proc. ARVLSI'87*.