A Safety Kernel For Traffic Light Control

Paul Ammann[*]
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030
pammann@gmu.edu

July, 1994
Revised January, 1995

Abstract

The success of kernels for enforcing security has led to proposals to use kernels for enforcing safety. This paper presents a feasibility demonstration of one particular proposal for a safety kernel via the application of traffic light control. The paper begins with the safety properties for traffic light control and specifies a kernel that maintains the safety properties. An implementation sketch of the kernel in Ada is given and use of the kernel is discussed. The contribution of the paper is a demonstration that a kernel is a feasible and desirable technique for software in a realistic, safety-critical application. The paper also illustrates how formal methods aid the software engineer in constructing and reasoning about such software.

[*]Partially supported by the National Aeronautics and Space Administration under grant NAG_1-1123-FDP.

1 Introduction

Software is being used with increasing frequency to control applications that are safety-critical, by which it is meant that certain failures, called safety failures, can have unacceptable costs, such as harm to humans, environmental damage, financial loss, and so on. Computer-controlled, safety-critical systems can and do fail [Neu], and there is intense interest in techniques for improving the software in such systems. Although a variety of techniques have been developed[1], there are significant limitations on what can be achieved [BF93], and there is a need to increase the number of methods available to developers of software for critical systems. In this paper, we work towards this goal by considering the use of kernels, which have enjoyed success in the security area, to achieve safety. The specific contribution of this paper is a feasibility demonstration of Rushby's notion of a kernel [Rus89]; the application chosen for the demonstration is traffic light control.

Safety is purely a system property; software by itself is neither safe nor unsafe. However, as with any other component in a safety-critical system, the software is required to have certain properties for the behavior of the encompassing system to be considered safe. These properties can be expressed as predicates that the software must maintain with respect to its inputs, outputs, and current state. For example, a system safety property for a reactor might be defined as shutting down operation if the reactor overheats. For the reactor to be safe, the software controlling the reactor must satisfy the predicate that a command to shut down the reactor is always issued within some interval after a reactor sensor exceeds a critical threshold. In this example, safety is defined with respect to reactor destruction, and the safety-relevant predicate for the software is the timely relation of inputs from sensors to outputs for actuators.

Showing that the software for a particular system satisfies a given set of safety predicates is a verification problem.[2] In software for systems that are not safety-critical, verification with respect to functional requirements is usually carried out informally via some combination of testing and inspections. Some faults, *i.e.*, design mistakes, escape the informal verification process, and the consequences of these residual faults must be subsequently absorbed as a cost of using the software.

For software in critical systems, the assurance gained by direct, informal verification with respect to safety requirements is generally inadequate, and more rigorous techniques are desirable. In some cases regulatory agencies require more rigorous methods. Making a more formal analysis feasible is difficult and expensive. Since part of the problem is the overall size of the software, one useful strategy is to organize the software so that the most rigorous analysis can be confined to a relatively small portion of the total software, with standard methods applied to the remainder.

Kernels are a useful organization that separates critical parts of the software from noncritical

[1]A recent survey by Rushby [Rus93a] enumerates many of these methods.

[2]The systems engineering process of deriving and documenting the proper set of safety predicates is also crucial, but that activity is beyond the scope of this paper.

parts. In general, kernels must be small, relatively simple units, since a kernel comparable in complexity to non-kernel software has little practical advantage. A kernel must ensure some particular behavior for the overall system without making any assumptions about the trustworthiness or proper functioning of the remaining software. Indeed, designers of security kernels make the worst case assumption that the remaining software is written by a malicious adversary whose goal is to bypass the kernel.

Kernels have a successful history in operating systems as a means of protecting access to computing resources and in security applications as a means of preventing unauthorized information flow. For example, a kernel in a multilevel secure system can prevent a user classified at one security level from directly reading data labeled at a higher or incomparable security level or writing data labeled at a lower or incomparable security level. The key property that a security kernel achieves is that no matter how the software outside the kernel behaves, the kernel maintains a system level security predicate, namely, that all direct information flow is authorized. (Indirect flows, or covert channels, must also be addressed, typically via an extension of a kernel known as a *trusted computing base.*)

The fact that security predicates can be viewed as a special case of safety predicates has prompted various researchers to suggest adapting security techniques for application to other safety-critical software. In particular, the idea of a kernel has been considered for safety [LSST83, Neu86, Rus89, WK94]. Proposals for safety kernels differ in subtle but important ways. Rushby makes a precise and convincing theoretical case for safety kernels [Rus89], and this paper adopts Rushby's model, which from here on is referred to as a *Rushby kernel.*

For a Rushby kernel to enforce safety, two conditions must hold. The first condition is that predicates describing safe operation at the *system* level must be defined at the *kernel* level. In other words, the variables in the safety predicates must be under the control of the kernel. If a safety predicate references variables outside the control of the kernel, the kernel, by itself, cannot guarantee the safety predicate.

The second condition for a Rushby kernel to enforce safety is that arbitrary behavior external to the kernel cannot falsify the safety predicates. Rushby formalizes this condition by stating that to show a kernel maintains a safety predicate P, it is generally necessary to show that

$$\forall \alpha \in op^* \bullet P(\alpha)$$

where each op is an operation either inside or outside the kernel, and op^* is a sequence of such operations. In other words, P holds for any arbitrary sequence of operations, α.

Rushby briefly outlines some examples of how a kernel might be applied in practice [Rus89]. The contribution of this paper is to show the feasibility of a Rushby kernel in a more realistic example. Although traffic light control might appear too simple an application, there is evidence to the contrary. For example, a recent assessment of verification techniques for safety-critical software used a version of the traffic light control problem [GC94]. Most of the variants constructed in that study had latent safety-critical faults, thereby indicating some intrinsic difficulty in traffic light control. Further, serious safety violations can and do occur even in cases where the underlying safety predicates are simple and well understood, *e.g.* the mismatch between beam intensity and turntable position in the Therac-25 accidents [LT93]. Finally, the kernel in this paper is based on actual traffic light specifications [Nat92], and as a result is somewhat more realistic than typical traffic light case studies.

There are other reasons for working a larger example. First, scalability is a concern for any software engineering technique, and the example in this paper demonstrates scalability to one realistic system. Second, the example presented here begins with a specification of safety predicates for an application, proceeds through a formal specification of a system that satisfies the safety predicates, and finishes with an implementation in Ada (informally) refined from the formal specification. The presentation is not completely formal – different phases correspond to a 1 or a 2 on Rushby's 0 (informal) to 3 (machine theorem proving) scale of formal methods [Rus93b]. However, the author believes that the exercise is accessible, and that it demonstrates an effective use of formal methods easily within the grasp of practicing software engineers. Thus the example in this paper not only serves as a model for constructing a safety kernel, but also as a model for an effective application of formal methods.

The paper is organized as follows. Section 2 motivates a kernel for a traffic light controller and presents the relevant safety requirements, *i.e.* properties that must hold if a traffic light is to be considered safe. Section 3 develops a formal abstract data type (ADT) specification in Z [Spi89] to implement these safety requirements. Section 4 argues that the specification of section 3 is indeed a Rushby kernel with respect to the safety requirements of section 2. Section 5 discusses refining the specification of the kernel into an implementation in Ada, and, via the example in [LCS91], how an implementation of the functional requirements for a traffic light control system can employ the safety kernel. Section 6 concludes the paper.

2 Rationale For A Traffic Light Safety Kernel

Various factors motivate the use of traffic light control as the example for carrying the ideas in this paper. First, traffic light control is safety-critical, because safety failures of traffic signals can lead directly to accidents involving human life and property. Second, although the basic operation of a traffic light is simple and universally understood, actual implementations are quite complex. In this regard, traffic light control is analogous to communication protocols. Even simple communication protocols can have large state spaces that hide subtle, but important errors. Similarly, the list of factors enumerated below complicates the basic traffic light protocol. Third, as in many other systems, the flexibility afforded by powerful microprocessors has led to increased functionality. Traffic light controllers are increasingly more complex, and analyzing the properties of traffic light controllers is correspondingly more demanding.

Traffic light controllers may be implemented as real time, interrupt driven systems.[3] Traffic controllers must communicate with a variety of interfaces, including inputs from traffic sensing devices, messages sent over a network and commands issued directly from the front panel. Functionality and safety requirements may differ depending upon the source of the the message. For example, a request to change a light for a particular direction may be rejected for safety reasons if requested over the network, but be accepted when issued by a human at the panel. Source code for a controller can exceed 10 megabytes, and object code can occupy from 500 kilobytes to 1 megabyte.

Emergency vehicles have devices to demand right-of-way, and thus normal operation of a traffic light may be preempted. Hardware in traffic lights is subject to failure, *e.g.*, via short circuits or lightning strikes; responses to these failures must be safe and should maintain operation if possible, since interruption of service is a safety concern. Although some of these issues are elaborated in traffic light standards, the standards do not address some significant issues. For example, the response to burned out signals is only partially covered by [Nat92]. The combination of the above factors means that assessing the functionality and safety of a given traffic controller, despite the simplicity of the underlying protocol, is a nontrivial task. Although a kernel does not address correctness with respect to function, a kernel does simplify an assessment of correctness with respect to safety.

2.1 Safety Provisions In Traffic Lights

The behavior of a traffic light controller is illustrated in figure 1.[4]

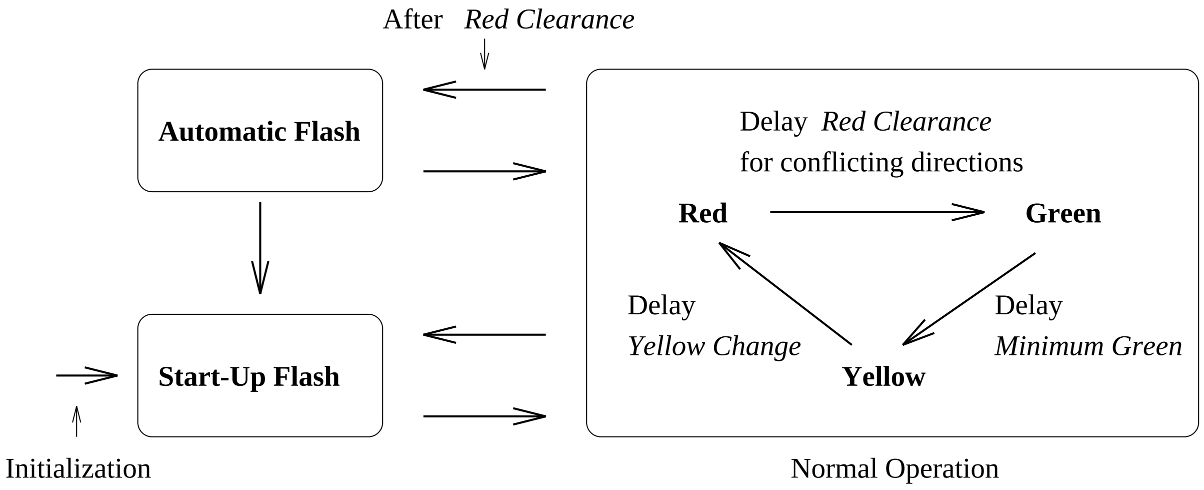

Figure 1: Allowed Traffic Light Transitions

The following safety requirements are derived from the traffic light standard [Nat92]:

[3]Some of the parameters cited here are taken from a standard for traffic light controllers [Nat92]. Others are graciously provided by [Ken94].

[4]The given diagram does not appear explicitly in [Nat92], but rather has been abstracted by the author.

1. Some traffic directions conflict, and simultaneous traffic flow in conflicting directions may result in an accident. Therefore, if a pair of directions conflict then the light associated with one of the directions must be *Red*. The relation *conflict* enumerates the pairs of the conflicting directions. (The Z notation used here declares the variable above the line and constrains the variable below the line. The constraining predicate $conflict = conflict^{-1}$ asserts that the relation *conflict* is symmetric):

$$\begin{array}{|l} conflict : Direction \leftrightarrow Direction \\ \hline conflict = conflict^{-1} \end{array}$$

2. A light associated with a particular direction may only change according to the shown transition diagram 1. First consider normal operation. From a *safety* perspective, for a light to turn *Red*, the light must be *Yellow* for at least *YellowChange* seconds prior to the transition, thereby giving drivers fair warning of the impending *Red*. Further, for a light to turn *Green*, all lights for conflicting directions must be *Red* for at least *RedClearance* seconds prior to the transition, thereby giving vehicles a chance to clear the intersection. From a *functional* perspective, for a light to turn *Yellow*, the light must be *Green* for at least *MinimumGreen* seconds prior to the transition. We return to the issue of safety vs. functional perspectives later.

   Consider flashing operation. From a safety perspective, the *AutomaticFlash* mode may be entered only if lights for all directions are *Red* and a suitable *RedClearance* period has passed.

3. *StartUpFlash* may be thought of as both an initial state and an error mode. There is no precondition for entering the *StartUpFlash* mode. If *StartUpFlash* is entered due to an error, then the system must remain in *StartUpFlash* for at least the period *MinimumFlash*.

We propose installing a safety kernel *internal* to software in the traffic light controller. Existing traffic lights contain a hardware device *external* to the controller called the Malfunction Management Unit (MMU)[Nat92, Section 4]. The primary purpose of the MMU is safety; the MMU detects potentially unsafe conditions and forces the traffic light into a safe state, *e.g.*, the *StartUpFlash* state, upon detection of an unsafe condition.

The MMU checks for conditions that depend on controller outputs. It is possible for an incorrect controller to request traffic light configurations that violate safety constraints. Specifically, the MMU checks for conflicting signals and also for insufficient delays with respect to *YellowChange* and *YellowChange*/*RedClearance* intervals.

It is possible that conditions other than an incorrect controller result in violations of safety constraints. For example, a short circuit could result in the illumination of conflicting signals even though the controller issues commands in a logically safe sequence. The MMU detects the conflicting signals regardless of the origin of the problem. The MMU also handles safety conditions unrelated to the controller. For example, the MMU handles power supply anomalies such as transient spikes, low voltages, and interruptions. In brief, the MMU enforces safety constraints at the level of the traffic light rather than at the level of the controller unit. Although the MMU is, in effect, a system-level safety kernel for a traffic light, two arguments support the addition of a safety kernel in the controller.

First, it is useful to detect failures as soon as possible. A safety kernel within the controller can detect the impending violation of a safety constraint *before* the outputs are presented at the controller interface. This detection means that the unsafe output can be suppressed, and the controller given a chance to recover from the fault. The source of the intercepted safety violation can be logged as a design defect in the software, and audit trail information can be stored for subsequent analysis. If instead the unsafe outputs are actually generated and the MMU steps in, then the evidence is no longer clear that the software is at fault; for example, a short circuit might be responsible instead. Also, when the MMU intervenes, the driver always views the failure, whereas when the kernel intervenes, the driver might not.

Second, the controller has more detailed information about safety constraints than does the MMU. The MMU enforces uniform *YellowChange* and *YellowChange/RedClearance* delays; both delays must be at least 2.6 seconds [Nat92, Section 4.4.5]. The 2.6 seconds is a lower bound that applies to any traffic light for any traffic direction, and is of necessity relatively weak to accommodate efficiency concerns. In particular, drivers will not tolerate excessively long delays at all intersections. However, some specific intersections require longer delays to be safe, and the required delays generally vary from one traffic light to another, and also for different directions controlled by the same traffic light.

Indeed, a traffic engineer can specify a *YellowChange* delay for each traffic direction of from 3 to 25.5 seconds, and a *RedClearance* delay for each traffic direction of from 0 to 25.5 seconds[Nat92, Section 3.5.3.1]. Thus for example, if the traffic engineer specifies that a particular direction should have a *YellowChange* delay of 4 seconds and a *RedClearance* delay of 2 seconds, then the *YellowChange/RedClearance* delay is 6 seconds before a conflicting traffic light is allowed to *safely* change to *Green*. However, the MMU can only detect a problem if the total delay is 2.6 seconds or less. Thus, for example, the MMU accepts a delay of 5 seconds as safe. However, a safety kernel inside the controller has access to the complete *YellowChange* and *YellowChange/RedClearance* specifications, and, in the example, can detect that a 5 second delay is *not* safe.

The arguments in support of a safety kernel internal to the controller do not diminish the need for the MMU. Even though all safety constraints are *derived* from system properties, safety constraints can be *enforced* at different levels. For the traffic light example, a safety kernel inside the controller can enforce system safety requirements at the controller level, and the MMU, which is, as noted, essentially a system-level safety kernel, can enforce system safety requirements at the traffic light level. Enforcement of safety constraints is useful at both levels.

In summary, despite the existence of the MMU, a safety kernel within the controller is desirable for two reasons. First, a safety kernel can catch potentially unsafe outputs before they are generated. Second, a safety kernel can make a more accurate assessment of whether safety constraints are satisfied.

2.2 Safety Requirements for a Traffic Light

Let the type *Light* enumerate possible traffic light settings and let the type *Direction* enumerate possible traffic directions for some particular intersection, *e.g.*, 'Through Traffic On Route 50 West', 'Left Turn Onto Route 123 North', etc. The specification here can be applied to any intersection, regardless of the intersection's complexity, and so we omit the specifics of the type *Direction*:[5]

$Light ::= Red \mid Yellow \mid Green \mid Dark$

$Direction ::= \ldots$

$[Time]$

$clock : Time$

Time is informally specified in seconds, although the required resolution is in milliseconds. For our present purposes, we need not be concerned with the details of how time is implemented, only that we can state time intervals such as '0 seconds' and '2.7 seconds'.[6]

Let the type *Mode* enumerate the three modes described in figure 1. The type *Status* indicates whether operations are proceeding normally, or whether some safety-related error has been detected. A status of *Error* is always associated with the mode *StartUpFlash*.

$Mode ::= NormalOperation \mid AutomaticFlash \mid StartUpFlash$

$Status ::= Ok \mid Error$

We define the safety kernel as an ADT. The schema *TrafficLightKernel*, shown below, defines the state of the ADT. Explanations for each component of *TrafficLightKernel* (components appear above the middle line) and each constraint on the components in *TrafficLightKernel* (constraints appear below the middle line) appear after the schema.

[5]Definitions here use the Z syntax [Spi89], although various liberties are taken with standard Z style to make the specification match the Ada implementation more directly. In the implementation, the Z type *Light* translates to Ada as `TYPE Light IS (Red, Yellow, Green, Dark)`. The type *Direction* is handled similarly.

[6]In the implementation it turns out to be helpful to distinguish the notion of of absolute time from that of an interval between two absolute times. Towards this end, the implementation uses the Ada types `Time` and `Duration` from the standard Ada package `Calendar`. The distinction can be omitted in the specification.

$$
\begin{array}{|l}
\textit{TrafficLightKernel} \\
\hline
lights : Direction \rightarrow Light \\
mode : Mode \\
status : Status \\
faulttime : Time \\
lastchanged : Direction \rightarrow Time \\
\hline
mode \in \{NormalOperation,\ AutomaticFlash\} \Rightarrow \\
\quad (\forall\, d_1, d_2 : Direction \mid d_1 \mapsto d_2 \in conflict \bullet \\
\quad\quad lights(d_1) = Red \lor lights(d_2) = Red) \\
status = Error \Rightarrow mode = StartUpFlash
\end{array}
$$

The component *lights* assigns a signal setting to each direction in *Direction*. For the modes *NormalOperation* and *AutomaticFlash*, the first invariant ensures at least one of the lights for each pair of conflicting directions is *Red*. Conflicting lights are not an issue in *StartUpFlash* mode. (A possible signal configuration for *StartUpFlash* is a solid *Red* combined with a flashing *Yellow*.)

The *mode* and *status* components of *TrafficLightKernel* are constrained by the second invariant such that a status of *Error* requires the mode to be *StartUpFlash*. The component *faulttime* records the value of *clock* from the most recent occurrence of setting status to *Error*. (See the operations *RegisterError* and *ClearError* below.)

To aid in evaluating timing constraints, the time of the last update to each light is recorded in the component *lastchanged*. Timing constraints with respect to the component *lastchanged* are used in the preconditions of the operations defined below.

To handle timing requirements, we formalize minimum signal periods as a function of traffic directions. The timing constraints on the intervals are taken from a traffic light standard [Nat92, Section 3.5.3.1 and Section 4.4.2].

$$
\begin{array}{|l}
YellowChange : Direction \rightarrow Time \\
RedClearance : Direction \rightarrow Time \\
MinimumGreen : Direction \rightarrow Time \\
MinimumFlash : Time \\
\hline
\forall\, d : Direction \bullet \\
\quad 3 \leq YellowChange(d) \leq 25.5 \land \\
\quad 0 \leq RedClearance(d) \leq 25.5 \land \\
\quad 1 \leq MinimumGreen(d) \leq 255 \land \\
6 \leq MinimumFlash \leq 16
\end{array}
$$

We define the safety predicate P, the predicate the kernel must maintain for any sequence of operations, as four separate predicates: $P = P_1 \land P_2 \land P_3 \land P_4$, where P_1, P_2, P_3, and P_4 are shown below.

P_1: If the mode is *NormalOperation* or *AutomaticFlash*, then at least one light for each pair of conflicting directions is always *Red*. This predicate appears as an invariant in *TrafficLightKernel*; we repeat it below:

$$\begin{array}{l} mode \in \{NormalOperation,\ AutomaticFlash\} \Rightarrow \\ (\forall\, d_1, d_2 : Direction \mid d_1 \mapsto d_2 \in conflict \bullet \\ lights(d_1) = Red \vee lights(d_2) = Red) \end{array}$$

P_2: Except for the mode *StartUpFlash* (where light settings are unimportant), there is a delay of *RedClearance* on conflicting directions before the light for any direction is allowed to change from *Red* to some other color. This predicate must be a precondition on any operation that can change a *Red* light. The required predicate is:

$$\begin{array}{l} mode \neq StartUpFlash \wedge mode' \neq StartUpFlash \Rightarrow \\ (\forall\, d_1, d_2 : Direction \mid d_1 \mapsto d_2 \in conflict \wedge lights(d_1) = Red \wedge lights'(d_1) \neq Red \bullet \\ clock - lastchanged(d_2) \geq RedClearance(d_2)) \end{array}$$

(In Z variables with primes ($'$) denote a state component after an operation. Variables without primes denote state components before an operation.)

P_3: If the mode is *NormalOperation*, then there is a delay of *YellowChange* before a light for any direction is allowed to change from *Yellow* to some other color. This predicate must be a precondition on any operation that can change a *Yellow* light. The required predicate is:

$$\begin{array}{l} mode = NormalOperation \wedge mode' = NormalOperation \Rightarrow \\ (\forall\, d : Direction \mid lights(d) = Yellow \wedge lights'(d) \neq Yellow \bullet \\ clock - lastchanged(d) \geq YellowChange(d)) \end{array}$$

P_4: If the mode *StartUpFlash* is entered as the result of an error, then the mode cannot be changed until a period *MinimumFlash* has expired.

$$\begin{array}{l} status = Error \wedge status' = Ok \Rightarrow \\ clock - faulttime \geq MinimumFlash \end{array}$$

3 Kernel Specification

As indicated in figure 1, the initial mode is *Start Up Flash*. The operation *TrafficLightKernelInit* defines the initial state of *TrafficLightKernel*.

TrafficLightKernelInit

$$\begin{array}{|l} TrafficLightKernel' \\ \hline mode' = StartUpFlash \\ status' = Ok \end{array}$$

(Initialization is viewed as an operation that only has a state after the operation; hence the primes in *mode'* and *status'*. The components *faulttime'*, *lights'*, and *lastchanged'* are not constrained in *TrafficLightKernelInit*, except that they must satisfy the invariants in *TrafficLightKernel′*, and hence these components could have a variety of values in the initial state.)

We now turn to specifying operations on the kernel. For brevity, we omit specifying error cases, *i.e.* what to do if an operation is requested when its precondition is not satisfied. Standard error handling methods apply.

In the event that an external error condition, such as a short circuit, is detected, the operation *RegisterError* can be applied at any time. The *clock* time at which *RegisterError* is called is recorded in the component *faulttime′*. (Δ*TrafficLightKernel* defines two states, the state before an operation is applied and the state after an operation is applied, and asserts that the invariants hold on both states.) *ClearError* is called to recover from a call to *RegisterError*.

```
── RegisterError ──────────────────────────────
│ ΔTrafficLightKernel
├──────────
│ mode′ = StartUpFlash
│ status′ = Error
│ faulttime′ = clock
└──────────────────────────────────────────────
```

```
── ClearError ─────────────────────────────────
│ ΔTrafficLightKernel
├──────────
│ mode = StartUpFlash
│ status = Error
│ clock − faulttime ≥ MinimumFlash
│ mode′ = StartUpFlash
│ status′ = Ok
└──────────────────────────────────────────────
```

Transition to the mode *NormalOperation* is as follows. Lights for all directions become *Red* when the mode switches to *NormalOperation*, and the component *lastchanged* is set to reflect the current *clock* time for each direction.

```
── BeginNormalOperation ───────────────────────
│ ΔTrafficLightKernel
├──────────
│ (mode = StartUpFlash ∧ status = Ok) ∨ mode = AutomaticFlash
│ mode′ = NormalOperation
│ lights′ = {d : Direction • d ↦ Red}
│ lastchanged′ = {d : Direction • d ↦ clock}
└──────────────────────────────────────────────
```

Operations for the *NormalOperation* mode are defined to accommodate the arcs in figure 1. All three operations, *ToGreen*, *ToRed*, and *ToYellow*, have similar structure, as captured in the schema *NormalOp*. *NormalOp* has the precondition that the current mode is

NormalOperation and the postcondition that *lastchanged* is updated for direction $d?$. (The decoration '?' indicates that variable $d?$ is an input.)

$$\begin{array}{|l}
\hline \mathit{NormalOp} \\
\Delta\, \mathit{TrafficLightKernel} \\
d? : \mathit{Direction} \\
\hline
\mathit{mode} = \mathit{NormalOperation} \\
\mathit{lastchanged}' = \mathit{lastchanged} \oplus \{d? \mapsto \mathit{clock}\} \\
\mathit{mode}' = \mathit{mode} \\
\mathit{status}' = \mathit{status} \\
\hline
\end{array}$$

The specifications for the three operations *ToGreen*, *ToRed*, and *ToYellow* are given below:

$$\begin{array}{|l}
\hline \mathit{ToGreen} \\
\mathit{NormalOp} \\
\hline
\mathit{lights}(d?) = \mathit{Red} \\
\forall\, d : \mathit{Direction} \mid d? \mapsto d \in \mathit{conflict} \bullet \\
\qquad \mathit{lights}(d) = \mathit{Red} \land \\
\qquad \mathit{clock} - \mathit{lastchanged}(d) \geq \mathit{RedClearance}(d) \\
\mathit{lights}' = \mathit{lights} \oplus \{d? \mapsto \mathit{Green}\} \\
\hline
\end{array}$$

The preconditions for *ToGreen* are that the light for direction $d?$ is *Red*, that lights for all conflicting directions are *Red*, and that the *RedClearance* interval for all conflicting directions has passed.

$$\begin{array}{|l}
\hline \mathit{ToRed} \\
\mathit{NormalOp} \\
\hline
\mathit{lights}(d?) = \mathit{Yellow} \\
\mathit{clock} - \mathit{lastchanged}(d?) \geq \mathit{YellowChange}(d?) \\
\mathit{lights}' = \mathit{lights} \oplus \{d? \mapsto \mathit{Red}\} \\
\hline
\end{array}$$

The preconditions for *ToRed* are that the light for direction $d?$ is *Yellow* and that the light has been *Yellow* for at least the period specified by *YellowChange* for light $d?$.

It is tempting to specify the operation *ToYellow* in a similar manner:

$$\begin{array}{|l}
\hline \mathit{ToYellowDraftSpecification} \\
\mathit{NormalOp} \\
\hline
\mathit{lights}(d?) = \mathit{Green} \\
\mathit{clock} - \mathit{lastchanged}(d?) \geq \mathit{MinimumGreen}(d?) \\
\mathit{lights}' = \mathit{lights} \oplus \{d? \mapsto \mathit{Yellow}\} \\
\hline
\end{array}$$

The preconditions for *ToYellowDraftSpecification* are that the light for direction $d?$ is *Green* and that the light has been *Green* for at least the period specified by *MinimumGreen* for light $d?$.

Turning a light *Yellow* when the precondition $clock - lastchanged(d?) \geq MinimumGreen(d?)$ is not satisfied does *not* constitute a safety failure. In fact, there are cases, such as preemption of a signal to yield right-of-way to an emergency vehicle, when violating the (draft) precondition with respect to *MinimumGreen* is *required.* Hence, the timing requirement with respect to *MinimumGreen* does *not* belong in the safety kernel, and *ToYellow* is specified as follows:

$$\begin{array}{|l}
\textit{ToYellow} \\
\hline
\textit{NormalOp} \\
\hline
\textit{lights}(d?) = \textit{Green} \\
\textit{lights}' = \textit{lights} \oplus \{d? \mapsto \textit{Yellow}\} \\
\hline
\end{array}$$

Automatic flashing operation employs a configuration of flashing *Red* and *Yellow* lights, and perhaps also some *Dark* lights [Nat92, Section 3.9.1.2], as described by *autoflashconfiguration*:

$$\begin{array}{|l}
\textit{autoflashconfiguration} : \textit{Direction} \nrightarrow \textit{Light} \\
\hline
\textit{Green} \notin \operatorname{ran} \textit{autoflashconfiguration} \\
\forall d_1, d_2 : \textit{Direction} \bullet d_1 \mapsto d_2 \in \textit{conflict} \Rightarrow \\
\qquad \textit{autoflashconfiguration}(d_1) = \textit{Red} \lor \textit{autoflashconfiguration}(d_2) = \textit{Red}
\end{array}$$

We specify the operation *BeginAutomaticFlash* to enter the *AutomaticFlash* mode. The mode *AutomaticFlash* can be entered from *NormalOperation* mode. (Leaving *AutomaticFlash* is accomplished via the schema *BeginNormalOperation*.)

The precondition on entering the *AutomaticFlash* mode is that the current mode is *NormalOperation*, that all signals are *Red*, and that a suitable *RedClearance* interval has passed with respect to each signal.

$$\begin{array}{|l}
\textit{BeginAutomaticFlash} \\
\hline
\Delta \textit{TrafficLightKernel} \\
\hline
\textit{mode} = \textit{NormalOperation} \\
\forall d : \textit{Direction} \bullet \\
\qquad \textit{lights}(d) = \textit{Red} \land \textit{clock} - \textit{lastchanged}(d) \geq \textit{RedClearance}(d) \\
\textit{mode}' = \textit{AutomaticFlash} \\
\textit{lights}' = \textit{autoflashconfiguration} \\
\textit{lastchanged}' = \textit{lastchanged} \\
\textit{status}' = \textit{status} \\
\hline
\end{array}$$

4 Analyzing the Safety Kernel

The purpose of this section is to verify that the kernel specification developed in Section 3 maintains the safety predicates developed in Section 2. There are two aspects to analyzing the kernel. One aspect is to apply the standard analysis methods of formal methods to the kernel. The other aspect is to consider the effect of an arbitrary sequence of operations on the safety predicates P_1 through P_4.

In Z, the standard analysis procedures relevant to the example may be broken into three categories: initialization checks, precondition investigation, and totality analysis. Initialization checks ensure that the initial state satisfies the invariant. It is also necessary to ensure that remaining operations maintain the invariant. In Z, the invariant is explicitly asserted on the 'after' state, and so precondition investigation focuses instead on ensuring that operations are 'honest', by which it is meant that preconditions are explicit. Precondition investigation also aids the implementor. An implementor confronted with a 'dishonest' schema is likely to omit checks for some part of the precondition, and hence produce an implementation that is an invalid refinement of the specification. Finally, totality analysis shows that an operation is defined for all states that satisfy the invariant.

The details of initialization checks, precondition investigation, and totality analysis are not shown the traffic light kernel, since they are not a primary focus of this paper. However, it is interesting to note that the formal proofs do require some revision of the safety predicates. For example, the simplification of the precondition on the operation *ToGreen* turns out to require the additional constraint that *conflict* be irreflexive, since a light for a direction that conflicts with itself must always be *Red*.

An important aspect of completing the standard analysis procedures is that doing so simplifies the analysis of whether the kernel maintains the safety predicates. Specifically, these procedures provide assurance that the invariant is, indeed, invariant. Initialization establishes the invariant, kernel operations explicitly maintain the invariant, and nonkernel operations, which are assumed to not have update access to the variables in the invariant, cannot affect the invariant.

Consider P_1. P_1 is an invariant in *TrafficLightKernel*, and so the standard analysis procedures guarantee that P_1 is always satisfied. Formally:

$$\forall \alpha \in op^* \bullet P_1(\alpha)$$

Consider P_2. It suffices to consider the last operation in some arbitrary sequence op^*. To show that a suitable *RedClearance* period passes before any light is allowed to change from *Red* to some other color, we find relevant operations, namely, operations that satisfy the antecedents in P_2. Specifically, relevant operations apply when the mode is not *StartUpFlash*, and the light for some direction changes from *Red* to another color. Inspection reveals two such operations: *ToGreen* and *BeginAutomaticFlash*.

Therefore, it must be shown that *ToGreen* and *BeginAutomaticFlash* maintain P_2. The formal proof statements are:[7]

$$\begin{array}{l} \mathit{ToGreen} \vdash P_2 \\ \mathit{BeginAutomaticFlash} \vdash P_2 \end{array}$$

In the case of *ToGreen*, the only light being changed is for direction $d?$, and so P_2 need only be considered in the case where $d_1 = d?$. P_2 reduces to:

$$\begin{array}{l} \mathit{lights}(d?) = \mathit{Red} \wedge \mathit{lights}'(d?) \neq \mathit{Red} \Rightarrow \\ (\forall d_2 : \mathit{Direction} \mid d? \mapsto d_2 \in \mathit{conflict} \bullet \\ \qquad \mathit{clock} - \mathit{lastchanged}(d_2) \geq \mathit{RedClearance}(d_2)) \end{array}$$

which is implied by the predicates in *ToGreen*.

In the case of *BeginAutomaticFlash*, and we can consider a strengthened version of P_2:

$$\begin{array}{l} \forall d_2 : \mathit{Direction} \bullet \\ \qquad \mathit{clock} - \mathit{lastchanged}(d_2) \geq \mathit{RedClearance}(d_2) \end{array}$$

which is an explicit predicate in *BeginAutomaticFlash*.

Thus we have established:

$$\forall \alpha \in op^* \bullet P_2(\alpha)$$

Consider P_3. Inspection reveals that the only operation relevant to P_3 is *ToRed*. We must show that *ToRed* maintains P_2. The formal proof statement is:

$$\mathit{ToRed} \vdash P_3$$

The proof follows directly from the predicate $\mathit{clock} - \mathit{lastchanged}(d?) \geq \mathit{YellowChange}(d?)$ in *ToRed* since $d = d?$ is the only d of interest in P_3. Thus we have established:

$$\forall \alpha \in op^* \bullet P_3(\alpha)$$

Finally, consider P_4, for which the only relevant operation is *ClearError*. The following two theorems are immediate.

$$\mathit{ClearError} \vdash P_4$$

[7] For these theorems to be convincing, we need to establish (perhaps by inspection), that the component *lastchanged* is updated to the current clock value each time the light for some direction is changed.

$$\forall \alpha \in op^* \bullet P_4(\alpha)$$

We conclude that the specification given in Section 3 constitutes a Rushby kernel with respect to the specified safety predicate $P = P_1 \wedge P_2 \wedge P_3 \wedge P_4$. We have only presented sketches of informal proofs. In general, Z specifications are better for generating informal proofs intended for human readers than for generating formal proofs amenable to machine manipulation. However, formal proofs are by no means precluded.

As a separate verification exercise, we should show that that operations described by the Z specifications conform to the state transition diagram in figure 1. Verification requires showing correspondence in both directions between the state diagram and the operations on *TrafficLightKernel*. Since we did not supply formal semantics for the state transition diagram used in this example, this verification must be carried informally, *e.g.*, via an inspection process.

5 Implementing and Using the Safety Kernel

The safety kernel can be implemented in a variety of ways. A formal approach is to perform data and operation refinement, in which more abstract data structures are systematically replaced with ones closer to an implementation. For example, sets are typically implemented as (injective) sequences. The refinement approach generates a set of proof obligations, specifically 'initialization' obligations, 'applicability' obligations, and 'correctness' obligations, that collectively ensure that the refined specification is a desirable implementation of the original specification.

For our purposes, the safety kernel specification is fairly simple, and we proceed directly to an implementation in Ada. For traffic light control, supporting concurrency is appropriate – actual traffic lights are implemented with concurrency [Ken94] – so we choose an implementation with tasking over one with procedures. The state described by *TrafficLightKernel* is implemented with an Ada package, and each operation is implemented with an entry call in a task, *TrafficLightKernelTask*. An excerpt of the Ada specification part of an implementation is shown below.

```
PACKAGE TrafficLightKernel IS
    ...
    TYPE Direction IS (NorthSouth, EastWest); -- for example
    ...
    TASK TrafficLightKernelTask IS
        ENTRY RegisterError;
        ENTRY ClearError;
        ENTRY BeginNormalOperation;
        ENTRY ToGreen (d :  IN Direction);
        ENTRY ToRed (d :  IN Direction);
        ENTRY ToYellow (d :  IN Direction);
        ENTRY BeginAutomaticFlash;
     END TrafficLightKernelTask;
  END TrafficLightKernel;
```

An excerpt of the Ada body part of an implementation is shown below.

```
  PACKAGE BODY TrafficLightKernel IS
     TYPE Light IS (Red, Yellow, Green, Dark);
     TYPE ModeType IS (Ok, Error);
     TYPE StatusType IS (NormalOperation, AutomaticFlash, StartUpFlash);
     ...
     mode :  ModeType;
     status :  StatusType;
     lights :  ARRAY (Direction) OF Light;
     lastchanged :  ARRAY (Direction) OF Time; -- Time is define in package CALENDAR
     ...
     TASK TrafficLightKernelTask IS
     ...
        LOOP
           SELECT
           ...
           OR ACCEPT ToRed (d :  IN Direction) DO
              IF PreToRed (d) THEN
                 lights (d) := Red;
                 lastchanged (d) := Clock;
              ELSE -- raise exception / produce audit trail / return error code
              END IF;
           END ToRed;
           ...
           OR TERMINATE;
           END SELECT;
        END LOOP;
     ...
     END TrafficLightKernelTask;
  BEGIN
     mode := StartUpFlash; -- Implement TrafficLightKernelInit
     status := Ok;
  END TrafficLightKernel;
```

The code excerpts in figures above illustrate several aspects of the translation from Z to

Ada. Ada enumerated types directly implement Z data type definitions, *i.e.*, *Direction*, *Light*, *Mode*, and *Status*. Initialization, specificed by the schema *TrafficLightKernelInit*, is implemented with initialization code for the package body of `TrafficLightKernel`. Preconditions are checked after the entry is accepted; violations result in some unspecified action, such as an exception being raised, an audit trail being generated, and/or an error code being returned to the caller.

In the example, the precondition of `ToRed` is checked by the function `PreToRed`. `PreToRed` is a direct implementation of *PreToRed*, calculated during precondition investigation and shown below.

PreToRed

$$\begin{array}{|l}
TrafficLightKernel \\
d? : Direction \\
\hline
mode = NormalOperation \\
lights(d?) = Yellow \\
clock - lastchanged(d?) \geq YellowChange(d?) \\
\hline
\end{array}$$

The implementation of *PreToRed* is as follows. The additional structure in `PreToRed` accommodates the fact that `lights` and `lastchanged` might not be defined if `mode` is not `NormalOperation`.

```
FUNCTION PreToRed (d :  IN Direction) RETURN BOOLEAN IS
BEGIN
   IF mode = NormalOperation THEN
      RETURN (lights(d) = Yellow AND Clock - lastchanged(d) >= YellowChange(d));
   ELSE
      RETURN (FALSE);
   END IF;
END PreToRed;
```

We note again that it is particularly important for specifications such as *ToRed* to be honest, so that an implementation `ToRed` can be derived in a straightforward way and still be assured to be a valid refinement of *ToRed*.

Using The Kernel

The kernel may be used by the remainder of the software, where the functional requirements on traffic lights are implemented. For example, the implementation may service requests generated by cars depressing pressure pads, synchronize phases with other traffic lights, or satisfy preemption requests by emergency vehicles. In all cases, any changes to a particular light requires a call to the relevant entry in `TrafficLightKernelTask`.

With the kernel, the safety predicates are maintained no matter how complicated the remainder of the implementation. Without the kernel, it can be quite difficult to detect whether even simple implementations maintain the safety properties. For example, Leveson, Cha, and Shimeall present an implementation in Ada for servicing requests generated by cars tripping detection sensors [LCS91, figure 18]. Leveson, Cha, and Shimeall use the example to illustrate the Software Fault Analysis (SFTA) technique for tasking implementations.

The example in Leveson, Cha, and Shimeall is a traffic light where the type *Direction* in our model has two values, *EastWest* and *NorthSouth*. In the example from [LCS91], one task entry services requests made by *EastWest* traffic, and another task entry services requests made by *NorthSouth* traffic. Viewed from the kernel specified here, the failure scenario discovered via SFTA in [LCS91] appears as the following sequence of entry calls. (*EastWest* lights are *Green* and *NorthSouth* lights are *Red* at the beginning of this sequence.)

```
A. ToYellow (EastWest); delay(1.0)            – [LCS91, figure 18, line 34]
B. ToRed (EastWest); ToGreen (NorthSouth);    – [LCS91, figure 18, line 35]
C. ToRed (NorthSouth); ToGreen (EastWest);    – [LCS91, figure 18, line 33]
```

As discovered via SFTA in [LCS91], Step C presents a problem. In particular, the precondition of *ToRed* in step C is violated since the lights for direction *NorthSouth* are never set to *Yellow*. (Also, the precondition of *ToGreen* in steps B and C is violated if *RedClearance* is nonzero for directions *EastWest* and *NorthSouth*. However, this safety constraint is not considered in [LCS91].)

SFTA does not have any completeness guarantee, so it is possible that some safety-related faults slip past the analysis. We note that if the direct manipulation of the traffic lights in the Leveson, Cha, and Shimeall example is replaced by calls to the entries of the safety kernel specified here, then *all* manifestations of safety faults as defined by the predicate P from Section 2.2 are guaranteed to be detected at execution time. The above comments do not mean that SFTA should not be applied, but rather that kernels are a complementary approach.

6 Conclusion

To demonstrate that the feasibility of Rushby kernels we have exhibited a safety kernel for traffic light control. Safety requirements were abstracted from a traffic light standard

[Nat92], specified with Z as an ADT, and implemented in Ada. Analysis showed that the kernel maintained four predicates necessary for safe traffic light control. We concluded with a discussion of how the kernel could be used by the remaining software in the controller. The exercise shows by example that a kernel is indeed a feasible technique for ensuring safety and that formal methods aid the software engineer in constructing a kernel. The intent is to thereby inspire the use of both safety kernels and formal methods in other applications.

Acknowledgements

The author is pleased to acknowledge improvements to this paper inspired by Timothy Gorgos, John Knight, Melissa Moore, Jeff Offutt, Michael Rittler, and Kevin Wika. The author is also pleased to acknowledge feedback from students in the 1994 fall semester of *Formal Methods and Models for Software Engineering*, in which the traffic light kernel was used as a term project.

References

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[GC94] Lon D. Gowen and James S. Collofello. Assessing traditional verification's effectiveness on safety-critical software systems. *The Journal of Systems and Software*, 26:103–115, 1994.

[Ken94] Bud Kent, 1994. Personal communication.

[LCS91] Nancy G. Leveson, Stephen S. Cha, and Timothy J. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, 8(4):48–59, July 1991.

[LSST83] Nancy G. Leveson, Timothy J. Shimeall, Janice L. Stolzy, and Jeffrey C. Thomas. Design for safe software. In *Proceedings AIAA 21st Aerospace Sciences Meeting*, pages 1–5, Reno, NV, January 1983. American Institute of Aeronautics and Astronautics.

[LT93] Nancy Leveson and Clark Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Nat92] National Electrical Manufacturers Association, 2101 L Street N.W., Washington, D.C. 20037. *Traffic Controller Assemblies*, 1992. NEMA Standards Publication No. TS 2-1992.

[Neu] Peter Neumann. Risks to the public. Regular column in *Software Engineering Notes*. Also published electronically in `comp.risks`.

[Neu86] Peter G. Neumann. On hierarchical design of computer systems for critical applications. *IEEE Transactions on Software Engineering*, SE-12(9):905–920, September 1986.

[Rus89] John Rushby. Kernels for safety? In Tom Anderson, editor, *Safe And Secure Computing Systems*, pages 210–220. Blackwell Scientific Publications, 1989. Proceedings of a Symposium held in Glasgow, UK, October 1986.

[Rus93a] John Rushby. Critical system properties: Survey and taxonomy. Technical Report CSL-93-01, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993. Available via anonymous ftp from `ftp.csl.sri.com` in `/pub/reports/csl-93-1.dvi.Z`.

[Rus93b] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-07, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1993. Available via anonymous ftp from `ftp.csl.sri.com` in `/pub/reports/csl-93-7.dvi.Z`.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, New York, 1989.

[WK94] Kevin G. Wika and John C. Knight. A safety kernel architecture. Technical Report CS-94-04, Computer Science Department, University of Virginia, Charlottesville, VA, February 1994.