Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel

Raymond J. Richards

1 Introduction

INTEGRITY-178B is a real-time operating system (RTOS) developed by Green Hills Software [4]. Real-time operation implies that the operating system kernel will schedule tasks as described by a predetermined schedule. System designers depend on the kernel to reliably and faithfully schedule tasks according to the schedule. This is to ensure that the tasks complete their necessary computations before system imposed deadlines.

The initial market for INTEGRITY-178B was safety-critical systems, such as avionics. The FAA accepts the use of DO-178B, Software Considerations in Airborne Systems and Equipment Certification [10], as a means of certifying software in avionics. DO-178B defines five levels of software to describe the impact to aircraft safety should there be a failure. The criticality levels are denoted "A" through "E," "A" is the most critical, and "E" is the least critical.

INTEGRITY-178B is able to host multiple applications of mixed criticality levels. It provides fault containment, preventing faults from cascading to other applications. That is to say, a fault in one application is never noticeable in another application. Recall that this is in a real-time operational environment, meaning that the failure of an application cannot cause any other application to miss a deadline. To achieve this level of fault containment, it is necessary for the kernel to strictly partition not only the time allocated to each application, but also the system memory between the various applications. This stringent time and space partitioning is often referred to as "hard-partitioning."

The high-assurance realms of safety-critical systems and security-critical systems overlap in many interesting ways. In particular, the use of hard partitioning is important in building high-assurance systems in both realms [11]. Mechanisms that provide hard partitioning are often referred to as "separation kernels." In the safety critical realm, separation kernels can integrate functionality of various levels

R.J. Richards (✉)
Rockwell Collins, Inc., Cedar Rapids, IA, USA
e-mail: rjricha1@rockwellcollins.com

D.S. Hardin (ed.), *Design and Verification of Microprocessor Systems for High-Assurance Applications*, DOI 10.1007/978-1-4419-1539-9_10,

of criticality on a single computing platform. Separation provides fault isolation; a fault in a less critical application cannot impact the execution of a more critical function. In the security-critical realm, separation kernels can ensure that there is no unauthorized flow of information between applications. This means that an application cannot inadvertently or maliciously signal another application with which it is not authorized to communicate. Since separation kernels are useful for both safety-critical and security-critical systems, it is reasonable to take a separation kernel that has been certified in one realm and attempt certification in the other realm.

Information assurance products can be certified in accordance with the Common Criteria for Information Technology Security Evaluation [2] or "Common Criteria" for short. In USA, the National Information Assurance Partnership (NIAP) performs Common Criteria evaluations. The Common Criteria defines seven evaluation assurance levels (EALs). The levels are labeled one through seven; EAL 7 is the most stringent level. A study has compared the certification requirements of DO-178B Level A and Common Criteria EAL 7 [1]. This study concluded that a product that is used in a DO-178B certified system could achieve Common Criteria EAL 7 by completing a few missing requirements. The most significant missing requirements are those that pertain to formal analysis.

INTEGRITY-178B was designed to be, and has been used in, systems that have been certified to DO-178B Level A; therefore, it was judged to be a good candidate to be the first separation kernel to obtain a Common Criteria certification. The INTEGRITY-178B analysis effort supported an EAL 6 Augmented (EAL6+) evaluation. EAL6+ means that some of the evaluation requirements were more stringent than prescribed by EAL6.

The Common Criteria defines three levels of rigor in analysis. These three levels are informal, semiformal, and formal. In this context, formal means a precise mathematical treatment with machine-checked proofs. Informal is a natural language-based justification of the security properties. Semiformal is something in between. For the INTEGRITY-178B kernel, this means a mathematical treatment, where some of the proofs are not machine checked.

The INTEGRITY-178B evaluation requirements for EAL 5 and above specify five elements that are either formal or semiformal. These five elements are the Security Policy Model, the Functional Specification, the High-Level Design, the Low-Level Design, and the Representation Correspondence [9]. The level of rigor that was applied to INTEGRITY-178B is as follows:

- Security Policy Model: A formal specification of the relevant security properties of the system.
- Functional Specification: A formal representation of the functional interfaces of the system.
- High-Level Design: A semiformal representation of the system. This representation may be somewhat abstract.
- Low-Level Design: A semiformal, but detailed representation of the system.
- Representation Correspondence: This element demonstrates the correspondence between pairs of the other elements. The Representation Correspondence is

formal when it shows the correspondence between two formal elements; it is semiformal otherwise. The Representation Correspondence shows that:

- The functional specification implements the security policy model.
- The high-level design implements the functional specification.
- The low-level design implements the high-level design.

The Common Criteria explicitly states that one entity may fulfill multiple requirements. For example, a single design specification may fulfill the need for both a high-level and a low-level design; in this case, the correspondence between these two elements is trivial.

INTEGRITY-178B runs on a variety of microprocessors and motherboards. A well-defined hardware abstraction layer with well-defined interfaces facilitates this portability. The formal (and semiformal) analysis was constrained to the hardware-independent portions of the INTEGRIY-178B kernel. A methodical informal analysis was performed on the software in the hardware abstraction layer. This approach allows the formal (and semiformal) analysis to be used as certification evidence on multiple hardware platforms.

This chapter discusses details of the formal analysis approach taken for the INTEGRITY-178B kernel, including:

- The generalization of the GWV theorem to capture the meaning of separation in a dynamic system.
- A discussion on how the system was modeled including:
  - System state
  - Behavior
  - Information flow
- The proof architecture used to demonstrate correspondence.
- The informal analysis of the hardware abstraction layer.

2 Separation Theorem

Existing formal specifications of separation properties were not expressive enough to state anything meaningful about INTEGRITY-178B. The GWV theorem has been shown to hold for the AAMP7G's hardware-based separation kernel [12]. However, the AAMP7G's kernel is very static. Its execution schedule is set a priori; it is impossible for user-level software to have an impact on the state of the kernel's scheduler. The original GWV theorem is only applicable to such strict static schedulers.

INTEGRITY-178B's scheduling model is much more dynamic. A more general GWV theorem was derived that captures the appropriate system level properties. This theorem is known as GWVr2 [5].

For a GWVr2 proof, the system needs to be modeled as a state transition system (Fig. 1). That is, it receives as inputs the current state of the system, as well as any external inputs. It produces a new system state, as well as any external outputs.

Fig. 1 State transition system

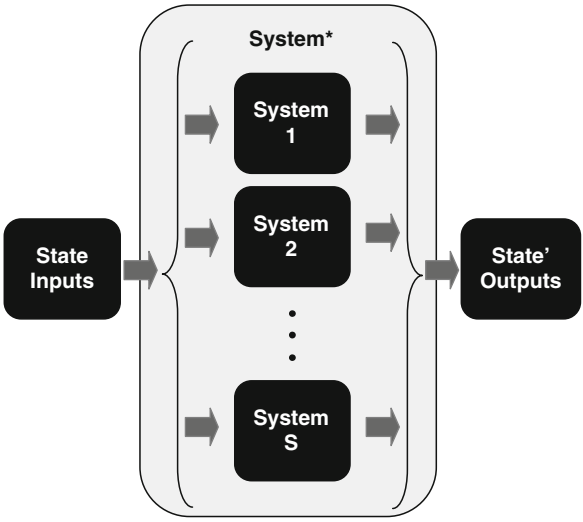

Fig. 2 Modified system model

System execution is a series of these state transitions. As a convenience, we will assume that the external inputs and outputs are contained in the system state structure. This state transition is expressed in the language of the ACL2 theorem prover [7] as:

$$(let\ \text{state}'(system\ \text{state}))$$

The system state, inputs, and outputs can be decomposed into atomic elements. Each of these elements is uniquely identifiable. Let the number of state elements plus the number of output elements be denoted by the symbol S. The system can be represented by S copies of the original system, each producing one element of the next state or output. Each of these S systems can be fed with only the elements of the current state and inputs that are necessary for it to compute its result. A system that takes a current state and input elements, maps the appropriate inputs to S copies of the system, and then maps the S resulting elements into the next state and external outputs is denoted as system* (Fig. 2).

The ACL2 notation for system* is a function that has two inputs. One input is a structure containing current state and external inputs. The other is a graph that specifies how the current state and inputs elements are mapped to the S subsystems. It produces the next system state, expressed in ACL2 as

$$(let\ \text{state}'(system^*\ \text{graph state}))$$

If it can be proven that the system and system* produce identical results for all inputs of interest, it implies that the graph used by system* completely captures the information flow of the system. This is the GWVr2 theorem.

(equal
  (*system* state)
  (*system** graph state))

A trivial graph that satisfies this theorem simply gives each subsystem all inputs and state elements. Conversely, there is a minimal graph, for which removing any element from the input of any subsystem causes the theorem to fail. Elements can be added to the minimal graph, without impacting the correctness of the theorem. This means that the input for one of the subsystems defines all of the data necessary for computing one element of the next state or output.

The GWVr2 Theorem is the Common Criteria Security Policy Model for INTGERITY-178B.

3 Modeling System State

To be consistent with the goal that the formal analysis be platform independent, the model of system state is that of nested abstract data structures. Elements within a data structure can either be a scalar or a nested data structure. A data structure that contains other data structures may be a record of heterogeneous data items or an array of homogenous data items. All elements in a data structure have names that uniquely identify them and distinguish them from their peer elements. This is analogous to a Unix file system containing directories and files. The directories represent nested data structures and the files represent scalar data elements.

In such a file system, one can identify any directory or file resident within a particular directory by specifying a path. The path contains the name of every subdirectory that must be traversed in order to reach the item of interest. Similarly, in the model of state, one can reach any piece of state that is resident in a data structure by specifying a path. Arrays are represented in this model by using the array indices as a specifier in the path.

Paths are considered scalar data items; they can be stored as part of state. This is how C language pointers are modeled. Paths can be references to state locations and can be dereferenced. Dereferencing a path produces the value stored at that location in state.

An example data structure is shown in Fig. 3. Four scalar values are stored in nested data structures. The paths to these for values and the data stored in this structure are shown in Table 1.

The ACL2 representation of a path is simply a list of identifiers. The head of a path is the outermost data structure. The tail of a path represents a path that is relative to the head. In this way, paths are analogous to a directory path in a Unix

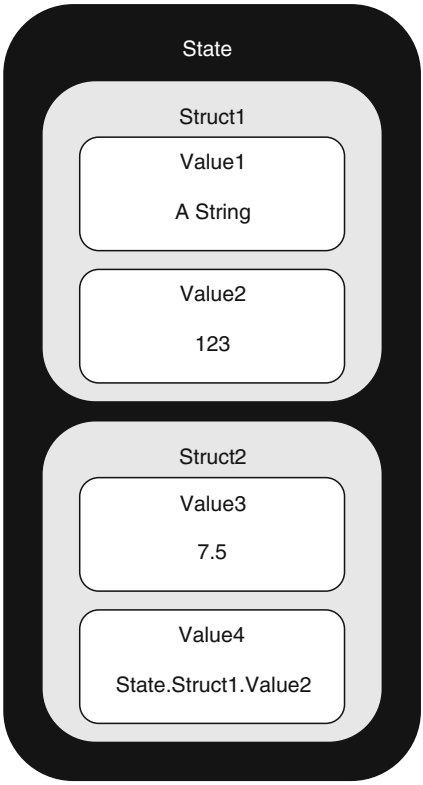

Fig. 3 Example state structure

Table 1 Path examples

Path	Value	Data type
State.Struct1.Value1	"A String"	String
State.Struct1.Value2	123	Integer
State.Struct2.Value3	7.5	Float
State.Struct2.Value4	State.Struct1.Value2	Path

file system. Absolute paths are relative to the root of the file system; relative paths are referenced from the current location.

Operators are defined to update and query the state. Both of these operators use a path to specify which element of the state they are affecting. The query operator "GP," or Get from Path, returns the value stored at the specified location. Its implementation is a recursive function that fetches the data specified by the identifier at the head of the path and recursively calls itself using the tail of the path and the fetched bit of state as the recursive arguments. The signature of the GP operator is:

```
(GP path st)
```

The update operator "SP," or Set Path, returns a new state, where the element specified by the given path is replaced with a new value. Its implementation is

also recursive. SP replaces the element specified by the head of the path with the value returned by its recursive call. The arguments to this recursive call are the tail of the given path and the value found in the state at the location specified by the head of the path. The signature of the SP operator is:

```
(SP path value st)
```

4 Modeling Kernel Behavior

The hardware-independent portion of the INTEGRITY-178B kernel is implemented in C code and formally modeled in ACL2. The Common Criteria explicitly forbids the low-level design specification and the implementation representation (source code) to be one and the same. Furthermore, establishing correspondence between the low-level design specification and the implementation representation is typically a manual, labor-intensive endeavor. This process is sometimes referred to as a "code-to-spec" review.

One of the goals in modeling the system is to capture enough of the implementation details so that a clear and compelling argument can be made that the behavior of the system is captured accurately. In an effort to facilitate that argument, it is useful if the low-level design specification has a one-to-one correspondence with the source code. That is to say, for every action in the source code there is a corresponding action in the model, and for every action in the model there is a corresponding action in the source code. In order to make the correspondences clear, it is useful to make the model greatly resemble the source code.

Two areas where it is not possible to make the ACL2 model closely resemble the C code implementation are in modeling loop constructs, as well as certain types of recursion. Since the formal language (ACL2) used to model the system is a functional language, and the INTEGRITTY-178B implementation Language (C) is an imperative language, it is not always possible to directly represent constructs in the system's implementation in the model. For instance, ACL2 does not have looping constructs. Loops are instead modeled by recursive functions.

ACL2 requires a proof of termination before admitting any function. This implies that a certain style of recursion, known as reflexive recursion, cannot be directly modeled [6]. Reflexive recursion occurs when two successive recursive calls are made, the latter one taking as an argument something calculated by the first. The following is an example of a reflexive recursive function:

```
int reflex (int i){
  int j;
  <function body>
  j=reflex(reflex(i-1));
  <function body>
  return j;
}
```

In this example, the outer call to the function `reflex` depends upon the results of the inner call. This results in the proof of termination depending upon the termination of the function. When this occurs in the INTEGRITY-178B kernel, it is modeled by unrolling the recursion. The recursion encountered in INTEGRITY-178B is controlled by a simple counting variable; when that variable reaches a particular value, the recursion is terminated.

4.1 Reader Macro

A set of ACL2 macros is used to allow the functional model to have an imperative look and feel. These macros are known collectively has the *reader macro*. The reader macro expands statements into a functional form. The reader macro is a form that begins with the symbol "%." This allows a syntax that closely resembles C to expand into native ACL2. The native ACL2 uses the state operators "SP" and "GP" to interact with the system state. The following types of statements are handled by the reader macro:

- Global variable access
- Assignment
- Function invocation
- Conditional early exit from a function

4.1.1 Global Variable Access

In the functional language model, all state information, except for local variables, is stored in the state structure that is passed throughout the model. Local, or stack, variables are modeled by local variables in ACL2, as long as there is no address-based accessing of the variable. A local variable that has its address passed to a subordinate function must be modeled as a state element.

The C language's use of variable identifiers does not distinguish between global and local variables. Since global variables are elements in the state structure, syntax was adopted to indicate when an identifier is an access to a global variable. Preceding an identifier with the symbol "@" indicates that the identifier should be treated as a path to a global variable. Preceding any path, including that of a local variable, with the symbol "*" queries the value stored at that location pointed to by the path.

4.1.2 Assignment

Assignment statement syntax depends on the impact of the assignment. That is, assignments to local variables have a different syntax than assignments that change the persistent state.

Assignment statements can be generalized as an lvalue, an assignment operator, and an rvalue.

```
lvalue assign_op rvalue
```

The lvalue denotes where the assigned value is stored. This can be a local variable or a location in state. Local variables are modeled by local ACL2 variables. This means when the scope in which the local variable has been declared is exited, its value is lost. The syntax for local variable assignment uses the variable identifier as the lvalue, followed by an equal sign "=," followed by the rvalue.

```
lvalue = rvalue
```

Assignments to local variables are transformed into *let bindings*. The body of the let binding is the scope where that assignment is valid.

In assignment statements that change state, the lvalue must evaluate to a path. Rvalues are evaluated and the results are stored in the location indicated by the lvalue. Assignments to state are transformed into state updates. The lvalue of an assignment that impacts the state must evaluate to a path into the state. The syntax for such statements is as follows:

```
  (path) @= rvalue
global_var @= rvalue
```

4.1.3 Functions

C language functions may or may not return a value. When modeling in ACL2, functions need to at least return the state that is a result of their invocation. The reader macro transforms function invocations that appear to not return a value into a function call that returns the new state, catching it in the appropriate variable. Model functions are declared using a form called "defmodel," which is similar to the ACL2 defun form.

Functions that return a value are modeled using a multivalued return. That is, it returns a list of items. The return list has a length of two; the second item is always the state returned from the function.

4.1.4 Conditional Early Exit

It is a common coding practice for functions to perform checks on the validity of their inputs. If the input checks do not pass satisfactorily, the function is exited, often returning an error code.

```
if (conditional){
  error handling;
  return;
}
```

ACL2 functions only exit at the end of the function body. The reader macro recognizes conditional early exits, translating them into an *if* statement whose *then*

clause includes whatever error handling is needed. The *else* clause contains the remainder of the function. The syntax for conditional early exist is:

```
(ifx (conditional)
  error handling)
```

4.2 Model Example

The following example will be used to illustrate the various parts of this analysis. The example is a function that operates on a circular, doubly linked list. This function removes one element from the list, maintaining a well-formed linked list. This function is passed two arguments. The first is a pointer to a structure that contains a pointer to the head of the list. The second is a pointer to the element that is removed from the list. It is assumed that the element pointed to by the second argument is a member of the list pointed to by the first argument. How this assumption is captured in the analysis will be discussed later in this chapter. The example's C language implementation is:

```
void RemoveFromList (LIST *TheList, ELEMENT *Element){
   ELEMENT *NextInList, *PrevInList;

   NextInList = Element -> next;

   if(NULL == NextInList)
     return;

   /* Update list */
   if (Element == NextInList){
      /* only element in the list */
      TheList->First = NULL;
   } else {
      /* not only element in the list */
      if (TheList->First == Element){
          /* Element is first in list */
          TheList->First) = NextInList;
      }

      PrevInList = Element->prev;
      PrevInList->next = NextInList;
      NextInList->prev = PrevInList;
   }
      /* clear this element's links */
      Element->next = NULL;
      Element->prev = NULL;
}
```

The formal model for this function is defined as follows:

```
defmodel RemoveFromList (TheList Element st)
  (%
   (NextInList = (* Element -> next))

      (ifx (NULLP NextInList)
         st)

   (if (equal Element NextInList)
       (%

       ;; only element in the list
       ((TheList -> First) @= (NULL)))

    ;; else
    (%
     ;; not only element in the list
     (if (equal (* TheList -> First) Element)

         ;; Element is first in list
         (%
          ((TheList -> First) @= NextInList))

    ;; else
        st)

      (PrevInList = (* Element -> prev))
      ((PrevInList -> next) @= NextInList)
      ((NextInList -> prev) @= PrevInList)))

    ;; clear this element's links
    ((Element -> next) @= (NULL))
    ((Element -> prev) @= (NULL))))
```

4.3 Model Syntax Summary

The following table describes how various C language constructs are modeled

C	ACL2	Lisp/ACL2 Notes
Variable reference		
x	X	Value of local variable x
	(* @ x)	Value of global variable x
$*x_p$	($*x_p$)	Value pointed to by local variable x_p
	(* (* @ x_p))	Value pointed to by global variable x_p
&x	(@ x)	Address of global y variable

Variable assignment		
x = . . .;	(% .. (x = . . .) ..)	Assign value of local variable x
	(% .. (x @= . . .) ..)	Assign value of global variable x
	(% .. ((x$_p$) @= . . .) ..)	Assign value pointed to by local variable x$_p$
*x$_p$ = . . .;	(% .. ((* @ x$_p$) @= . . .) ..)	Assign value pointed to by global variable x$_p$
Simple structure references		
x.y	(* (@ x) |.| y)	Value of field y of the structure instance at global variable x
x$_p$− > y	(* x$_p$− > y)	Value of field y of the structure pointed to by local variable x$_p$
	(* (* @ x$_p$)− > y)	Value of field y of the structure pointed to by global variable x$_p$
x[y]	(∗ (@ x) [y])	Value of element at index y in the array instance at global variable x
x$_p$[y]	(* x$_p$ [y])	Value of element at index y of the array to which local variable x$_p$ points
	(* (* @ x$_p$) [y])	Value of element at index y of the array to which global variable x$_p$ points
&x.y	(& (@ x) |.| y)	Address of field y of the structure instance at global variable x
&x$_p$− > y	(& x$_p$− > y)	Address of field y of the structure pointed to by local variable x$_p$
	(& (* @ x$_p$)− > y)	Address of field y of the structure pointed to by global variable x$_p$
&x[y]	(& (@ x) [y])	Address of element at index y in the array instance at global variable x
&x$_p$[y]	(& x$_p$ [y])	Address of element at index y of the array to which local variable x$_p$ points
	(& (@*x$_p$) [y])	Address of element at index y of the array to which global variable x$_p$ points
Complex structure references		
*x.y	(* (* (@ x) |.| y))	Value pointed to by field y of the structure instance at global variable x
*x$_p$− > y	(* (*x$_p$− > y))	Value pointed to by field y of the structure pointed to by local variable x$_p$
	(* (* (* @ x$_p$)− > y))	Value pointed to by field y of the structure pointed to by global variable x$_p$
*x[y]	(* (* (@ x) [y]))	Value pointed to by element at index y in the array instance at local variable x

$*x_p[y]$	(* (*x_p [y]))	Value pointed to by element at index y of the array to which local variable x_p points
	(* (* (*@ x_p) [y]))	Value pointed to by element at index y of the array to which global variable x_p points
Simple structure assignments		
x.y = . . . ;	(% .. (((@ x) |.| y) @= . . .) ..)	Assign value of field y of the structure instance at global variable x
x_p− > y = . . .;	(% .. ((x_p− > y) @= . . .) ..)	Assign value of field y of the structure pointed to by local variable x_p
	(% .. (((*@ x_p)− > y) @= . . .) ..)	Assign value of field y of the structure pointed to by global variable x_p
x[y] = . . . ;	(% .. (((@ x) [y]) @= . . .) ..)	Assign value of element at index y in the array instance at global variable x
x_p[y] = . . . ;	(% .. ((x_p [y]) @= . . .) ..)	Assign value of element at index y of the array to which local variable x_p points
	(% .. (((*@ x_p) [y]) @= . . .) ..)	Assign value of element at index y of the array to which global variable x_p points
Complex structure assignments		
*x.y = . . . ;	(% .. (((* (@ x) |.| y)) @= . . .) ..)	Assign value pointed to by field y of the structure instance at global variable x
*x_p− > y = . . .;	(% .. (((* x_p− > y)) @= . . .) ..)	Assign value pointed to by field y of the structure pointed to by local variable x_p
	(% .. (((*(*@ x_p)− > y)) @= . . .) ..)	Assign value pointed to by field y of the structure pointed to by global variable x_p
*x[y] = . . .;	(% .. (((* (@ x) [y])) @= . . .) ..)	Assign value pointed to by element at index y in the array instance at global variable x
*x_p[y]. . .;	(% .. (((* x_p [y])) @= . . .) ..)	Assign value pointed to by element at index y of the array to which local variable x_p points
	(% .. (((* (*@ x_p) [y])) @= . . .) ..)	Assign value pointed to by element at index y of the array to which global variable x_p points
Arithmetic operators		
X + y;	(+ x y)	Addition
X – y;	(− x y)	Subtraction
x* y;	(ACL2::* x y)	Multiplication
x/y;	(/ x y)	Division
Logical operators		
x == y	(equal x y)	Equal
x != y	(not (equal x y))	Not-equal (Negation)
x < y	(< x y)	Less-than
x > y	(> x y)	Greater-than
x <= y	(<= x y)	Less-than or equal-to
x_p == NULL	(NULLP x_p)	Null pointer test
!x_p		

$x_p == 0$		
x_p	(NNULL x_p)	Non-Null pointer test
Conditional control structures		
if (x) y; else z;	(if x y z)	If expression x is *true* do y otherwise do z.
if (x) y;	(if x y st)	If expression x is *true* do y otherwise to nothing. State (st) as the else component is analogous to do-nothing or the absence of as else-component.
{. . .}	(% . . .)	Groups a series of sequential statements into a single global structure or block
If (x) {A_{stmt}; B_{stmt}; } else { C_{stmt}; }	(if x (% $A_{stmt}B_{stmt}$. . .) (% (C_{stmt}) . . .))	If example with a code block.
x ? y : z	(if x y z)	C alternation
Function Declarations		
type foo . . .	(defun foo . . .	Return type not specified in function signature
. . . foo (int x) . . .	. . . foo (x st)	No parameter type declarations State (st) parameter added for access to global state
. . . foo (int *x) . . .	. . . foo (x_p st)	No parameter type declarations State (st) parameter added for access to global state
return;	(return st)	Function return when function application only changes state
return x;	(return x)	Function returns a single value x when function application does not change state

4.4 Kernel Boundaries

There are several important boundaries to the kernel model that could not be modeled as a straightforward translation of the system source code. These boundaries include asynchronous interactions with the world outside of the kernel, including interrupt processing and execution of application code. Another boundary is the interaction with the portion of the kernel that is specific to the hardware platform.

4.4.1 Breaking the Kernel Loop

The GWVr2 theorem requires that steady-state operation be defined as a step that can be performed repeatedly. Since the steady-state execution of most operating

systems is implemented with an infinite loop, it is a natural inclination to have one step in the model represents a single iteration of this loop. We will refer to this as the scheduling loop.

Great care needs to be taken to determine where this loop is broken. The cut point of the loop must represent a well-defined state in the execution. This is the point where the next entity to execute can be identified from the state, and the previous entity has been removed from execution and its state saved. This is the point where all assumptions about the state hold. These assumptions include well formedness and necessary constraints on the data. We refer to the state between two steps as a secure state. It is assumed that the system reaches a *secure initial state* as a result of the system initialization. Evidence justifying this assumption has been generated and is beyond the scope of this chapter. The generalization of the GWV theorem to the GWVr2 theorem does allow for the execution of a previous system step to influence which is the next entity to execute. The successive execution of steps is shown in Fig. 4.

The INTEGRITY-178B scheduling loop picks which thread of the current partition is to execute next and does not necessarily represent a context switch or a change in the logical partition.

4.4.2 Hardware-Dependent Layer

As mentioned earlier, only the hardware-independent portion of the kernel is formally modeled. The hardware-dependent portion of the kernel is present in the formal model only in an abstract form. The functional interface to this portion of the kernel is modeled by ACL2 *encapsulations*. In an encapsulation, the function signatures are defined, as well as key properties, but no implementations are modeled. The properties given in these encapsulations are the properties necessary to prove either termination of a higher level model function or the GWVr2 Theorem.

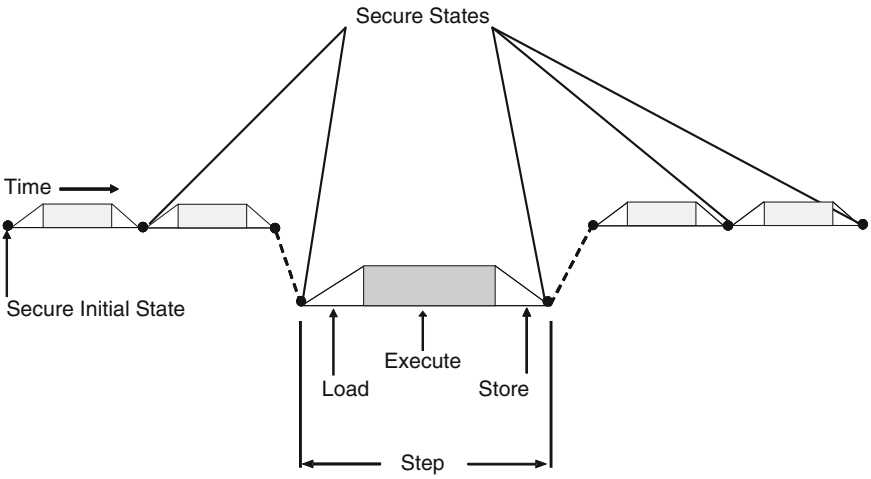

Fig. 4 Dividing Kernel execution into discrete steps

Justification of these properties in the actual implementation occurred as part of the rigorous manual analysis of that code. This analysis is discussed later in this chapter.

4.4.3 Interrupt Processing

Asynchronous events typically interact with INTEGRITY-178B via the interrupt mechanism. Interrupt processing is implemented as functions that are invoked from the platform-specific interrupt handling software. We model all such interrupt processing at the level of the scheduling loop. That is, before the step concludes, it processes all of the interrupts that occurred during execution of the step.

A logical mechanism known as an *oracle* is used to determine the number and type of interrupts that have arrived. An oracle is modeled by an unspecified function that produces the information that is needed. Properties may be stated about the oracle function, which may be as simple as the type of information it produces. The oracle function takes as an input an undefined piece of the system state. This gives us a convenient way to reason about an arbitrary number of arbitrary events.

Interrupts are modeled by a loop whose number of iterations is controlled by one oracle; in the body of the loop, a second oracle determines what type of interrupt has arrived.

4.4.4 Application Software Execution

The purpose of a separation kernel is to orchestrate the execution of multiple partitions. From time to time the kernel relinquishes control of the hardware to application software. However, the actions of the application cannot be modeled, since they are unknown. What is modeled are the application software's interactions with the kernel. These interactions come in the form of system calls. We model an arbitrary number of arbitrary system calls in the same manner in which we model interrupts. One oracle determines how many systems calls are going to be processed, and a second oracle determines the type of each system call.

5 Modeling Information Flow

Information flows are modeled by defining for a given element in system state; what are the state elements that can influence its next value. This can be thought of as a graph with vertices representing state elements and edges representing dependencies. For each function of the model, a new function is defined that calculates its graph, given its set of inputs, including the input state. The naming convention for these graph-computing functions (often referred to as graph functions) is to append "-graph" to the model function's name. The graph function's parameters are identical to that of the model function. The graph function returns a data structure that contains all of the graph edges for each state element that is updated by the model function.

5.1 Crawlers

In order to define a graph, it is necessary to be able to articulate what elements in state belong to which data structures. That is to say, for an operation upon a data structure, we need to be able to create sets of elements that may be updated by the operation and sets of elements whose values are used to create the new values. To do this, we have created a construct known as a *crawler*. Crawlers are used to create collections of state elements. Each member of the collection is identified by its path. Once a collection is created, it can be transformed into a collection of subelements by appending the subelement identifier to each path. We call the action of appending the same identifier to each path *decorating* the paths in the collection.

Let us consider the doubly linked circular list example. Each list element contains a next and previous pointer in order to connect it to the list. A crawler over this data structure might create a collection containing all elements in the list. That is, it creates a set of paths, one representing each element in the list. A decorate operation might refine these paths to refer to the next and previous fields of each element in the list.

5.2 Graphs

A graph describes, for a set of state elements that may have their value changed by an operation, what are the sources of information that are used in calculating the new values. Using our circular linked list example, let us consider a graph for either a sort or remove-element operation. In each case, the state elements that may be changed are the previous and next fields of all of the elements of the list. The new values that may be stored in these locations are the values that are stored in these same locations before the operation. Therefore, the graph states that the new values of the previous and next fields in the list depend upon what is currently stored in the previous and next fields in the list. More precisely, the graph contains an entry for each previous and next pointer as a location that may be updated. Each entry defines a dependency on the set of previous and next pointers as the source of information for the updated values.

Several functions and macros are defined to assist in developing graph functions. Chief among these is "defgraph," which takes four arguments. The first argument is the name of the function whose graph is being defined. The second argument is the list of names of the function's parameters. Any parameter that is a pass-by-value structure has its name in parentheses. The third argument is the list of variable names whose values are returned by this function. Again, pass-by-value structures have their names in parentheses. The last argument is the body of the graph defining function.

The functions "du," "du*," and "merg-u2" are used to create dependencies or use lists. The functions "su" and "su*" are used to associate a dependency set with an

element that has its value defined by the function. The function "mvg" returns the graph and associates variables with returned values.

The graph for the RemoveFromList example function is defined as follows:

```
(defgraph RemoveFromList (:TheList :Element (:st))
                         ((:st))
 (%
   ;; determine the nodes in the list
   (list-nodes = (crawl-list TheList st))

   ;; define the state elements whose values
   ;; might change
   (list-ass =
      `(,@(decorate-list list-nodes
                         (ElementStruct_Kstr))
        ,@(decorate-list TheList
                         (ListStruct_Kstr))))

    ;; create the dependencies set for the things
    ;; that might change
    (u2 = (merg-u2 :TheList :Element
                           (du* :st list-ass)))

    ;; define the dependencies for things that
    ;; might change
    (g = (su* :st list-ass u2 g))

    (mvg (:st) st)))
```

5.3 Graph Composition

In the formal analysis, graphs are created for each function in the kernel. A graph for a function that calls other functions must be no smaller than the graphs of the subordinate functions. That is, the dependencies defined by any graph of a called function must exist in the graph of the calling function.

In the circular linked list example, consider an Add operation. The state elements that may be updated are not only the previous and next fields of the existing list, but also the previous and next fields of the element being added to the list. The sources of new values for these elements are not only the previous and next pointers of the existing list, but also the parameter to the function pointing to the new element.

The graph of any function calling the Add operation must relate the previous and next fields of the current list members and the previous and next fields of any elements that might be added to the list to the sources of possible new values. The

sources of possible new values are, of course, the previous and next fields of the current members of the list and the locations that could supply the new elements to the Add operation.

6 Proof of Separation

In order to prove the GWVr2 theorem, it is useful to first prove two lemmas with respect to the function being analyzed. These lemmas are referred to as the Workhorse Lemma and the ClearP Lemma. We will discuss these lemmas with respect to the circular linked list example. Before we discuss these lemmas, we will define functions needed to support them.

- *RemoveFromList-Hyp*. For every model function "foo" a function "foo-hyp" is defined. This function states the hypothesis that is needed in order to have the model function work appropriately. The hypothesis function takes the same arguments as the model function. Recall that for the RemoveFromList function, it was assumed that the element given to the function is a member of the list; the RemoveFromList-Hyp function is where that assumption is stated.
- *Keys*. The Key function is passed a dependency graph and returns the set of state elements that may be updated, according to the graph.
- *DIA*. The direct interaction allowed (DIA) function takes a state element and a graph. It returns the set of state elements that the passed-in element has dependencies on, as defined by the graph.
- *CP-Set-Equal*. CP-Set-Equal is a predicate that takes a set of state elements and two states. It evaluates to *True* if the two states have the same value for each member of the set. It does not say anything about portions of the state that are not in the set. Therefore, the two states may be different in the parts of the state not defined in the set.
- *CLRP-Set*. CLRP-Set takes a set of state elements and a state. It returns a state that is a copy of the passed-in state, but the elements of the state specified in the set have been cleared. In this case, cleared means that their values have been replaced with *nil*.

6.1 Workhorse Lemma

The Workhorse Lemma states a relationship between the results of two invocations of a function. These two invocations operate on different states, but on the same parameters. For the circular linked list example, the two states satisfy the following constraints:

- Both states satisfy the RemoveFromList-Hyp assumptions. This means we are only considering invocations of this function, where the element to be removed is a member of the list.
- The two states have the same values for all elements that are in one of the dependency sets defined by the RemoveFromList graph.

These constrains are specified in ACL2 as:

```
(AND
   (RemoveFromList-Hyp List Element St1)
   (RemoveFromList-Hyp List Element St2)
   (Member Path (Keys
             (RemoveFromList-Graph List Element St1)))
   (CP-Set-Equal
      (DIA Path
          (RemoveFromList-Graph List Element St1))
           St1 St2))
```

This lemma concludes that having these criteria satisfied implies that the two states resulting from the two RemoveFromList invocations have the same values for all state elements that are defined by the RemoveFromList-Graph function. The ACL2 statement of this lemma is:

```
(DEFTHM RemoveFromList-Workhorse
   (IMPLIES
      (AND
         (RemoveFromList-Hyp List Element St1)
         (RemoveFromList-Hyp List Element St2)
         (Member Path
                 (Keys (RemoveFromList-Graph List
                                             Element
                                             St1)))
       (CP-Set-Equal
          (DIA Path
               (RemoveFromList-Graph List
                                     Element
                                     St1))
          St1
          St2))
    (IFF (EQUAL
          (GP Path (RemoveFromList List Ele ment St1))
         (GP Path (RemoveFromList List Element St2)))
        T)))
```

The Workhorse lemma demonstrates that the function's graph sufficiently captured the dependencies in the data flows of the function.

6.2 ClearP Lemma

The ClearP Lemma demonstrates that all of the changes to state performed by a function are captured by the function's graph. In the circular linked list example, for a list, element, and a state that satisfy the function's hypothesis function

```
(RemoveFromList-Hyp List Element St1)
```

the ClearP lemma establishes that the operation of the function does not change state in a manner that is not captured by its graph. The lemma considers two states, the input state and the output state the function. If all elements that the graph says might be updated are removed from both states, then satisfying ClearP means that the remaining states are identical. This demonstrates that the footprint of state changed by the function is captured by the graph.

Once these two lemmas are proven, it is straightforward to prove GWVr2 for the function.

7 Hardware-Dependent Code Analysis

In the INTEGRITY-178B evaluation, the small layer of hardware-dependent code was subjected to a rigorous by hand review. NIAP provided a list of source code characteristics that when present tend to promote correctness, stability, and understandability. These characteristics were collected from industry's best practices for real-time high-assurance software. An example of these characteristics is the absence of pointer arithmetic. Importantly, the assumptions stated in the abstract model of the code are validated by this analysis.

All of the source code in this layer (C code and assembly language) was examined to determine if it conformed to the characteristics list. For each function, justification for each characteristic was documented. This documentation was provided as part of the certification evidence.

8 Conclusion

After a thorough review of all of the certification evidence, including the formal, semiformal, and informal analysis described herein, NIAP granted a Common Criteria Certificate for the INTEGRITY-178B kernel at the EAL6+ level on September 1, 2008. The "home page" for the certification documentation can be found online [8]; a summary of the formal verification activities can be found in the Security Target document [3].

References

1. Alves-Foss J, Rinker B, Taylor C (2002) Towards common criteria certification for DO-178B compliant airborne systems, Center for Secure and Dependable Systems, University of Idaho
2. Common Criteria for Information Technology Security Evaluation (CCITSE) (1999). Available at http://www.radium.ncsc.mil/tpep/library/ccitse/ccitse.html

3. Common Criteria Testing Laboratory. Green Hills Software INTEGRITY-178B Security Target, Version 1.0, May 30, 2008. http://www.niap-ccevs.org/cc-scheme/st/st_vid10119-st.pdf
4. Green Hills Software, Inc. INTEGRITY real-time operating system. http://www.ghs.com/products/rtos/integrity.html
5. Greve D (2010) Information security modeling and analysis. In: Hardin D (ed) Design and verification of microprocessor systems for high-assurance applications. Springer, Berlin, pp 249–299
6. Greve D, Wilding M (2002) Dynamic data structures in ACL2: a challenge. Available at http://www.hokiepokie.org/docs/festival02.txt
7. Kaufmann M, Manolios P, Moore JS (2000) Computer-aided reasoning: an approach. Kluwer, Dordrecht
8. NIAP CCEVS. Validated Product – Green Hills Software INTEGRITY-178B Separation Kernel. http://www.niap-ccevs.org/cc-scheme/st/vid10119/index.cfm
9. Richards R, Greve D, Wilding M, Vanfleet M (2004) The common criteria, formal methods, and ACL2. In: Proceedings of ACL2'04, Austin, TX, November 2004
10. RTCA, Inc (1992) Software considerations in airborne systems and equipment certification, RTCA/DO-178B, December 1, 1992
11. Rushby J (1981) Design and verification of secure systems. In: Proceedings of the eighth symposium on operating systems principles, vol 15, December 1981
12. Wilding M, Greve D, Richards R, Hardin D (2010) Formal verification of partition management for the AAMP7G microprocessor. In: Hardin D (ed) Design and verification of microprocessor systems for high-assurance applications. Springer, Berlin, pp 175–191