



Andreas Enbacka | Linas Laibinis

Formal Specification and Refinement of a Write Blocker System for Digital Forensics

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 718, November 2005



Formal Specification and Refinement of a Write Blocker System for Digital Forensics

Andreas Enbacka

Åbo Akademi University, Department of Computer Science

Linas Laibinis

Åbo Akademi University, Department of Computer Science

TUCS Technical Report
No 718, November 2005

Abstract

In this paper we present a formal B development of a write blocker system for digital forensics. The field of digital forensics is rapidly expanding as the number of computer security incidents and computer-related crimes is increasing each year. The software tools used to collect and process digital evidence need to exhibit high reliability for the collected evidence to be admissible, e.g., in court proceedings. The purpose of a write blocker (which can be implemented as a part of either software or hardware) is to make sure that the original digital evidence is not altered in any way when making a forensically sound copy of the contents of a storage device. In this paper we present a partial formal development of such a (software) write blocking system using the B Method. The B Method allows us to rigorously demonstrate compliance of the developed software with the original requirements for such systems developed by the American National Institute for Standards and Technology (NIST). The automatic tool support provided by the Atelier-B tool facilitates the entire formal development process.

Keywords: digital forensics, write blocking, formal methods, B Method

TUCS Laboratory
Distributed Systems Design

1. Introduction

During the last years, the number of computer crimes and computer security related incidents has constantly increased. Often these incidents leave the associated digital evidence that might be very valuable to the incident investigators. The reliability and integrity of the digital evidence have to be demonstrated in order for this evidence to be admissible in a court of law. Therefore, the tools used to collect and process the evidence have to be reliable, i.e., they produce trustworthy and accurate results. The procedures for evaluating and testing digital forensics tools have been developed in the United States by the *National Institute for Standards and Technology* (NIST). As a part of the Computer Forensics Tool Testing (CFTT) [15] project at NIST, the informal requirements for disk imaging (i.e., disk copying) and the software/hardware write blocking tools have been developed. Using these requirements a number of commercial digital forensics tools (e.g., the software based write blocker PDBlock [7]) have been evaluated, and the detailed final test reports have been produced. The evaluations were performed on the basis of test cases generated from the informal requirements.

When making a copy of the original digital evidence, it is extremely important that the acquisition process can be guaranteed to be *forensically sound*, i.e., that the original evidence (such as the hard drive content) is not modified in any way, neither intentionally nor unintentionally. In order to guarantee soundness of the disk imaging process (i.e., creation of an exact copy of the disk), investigators usually employ the software or hardware based write blocking tools. The purpose of these tools is to make sure that all writing or other modifying commands issued to the evidence drive are blocked once the protection has been enabled. At the same time, reading and other non-modifying commands are allowed to pass in both protected and unprotected states. As the integrity of the collected digital evidence is of major importance to guarantee a sound crime investigation, reliability of these write blocker tools is critical.

In this paper we outline formal development of a general software-based write blocking system using the B Method [1]. Choosing a formal approach gives us several advantages. First, the initial specification can be rigorously verified to be consistent. Second, using the B Method allows us to obtain a final implementation that is formally proved to satisfy its specification. In addition, commercial automatic tools for the B Method support the entire formal development process providing, e.g., facilities for automatic verification and code generation as well as documentation and project management. The main motivation for using a formal method such as B for specifying and developing a software write blocker system is the growing demand for forensic software tool vendors to improve their tools in order to guarantee that no digital evidence is lost or altered during investigations. In [4], Eoghan Casey notes that “*Conversations among digital investigators are replete with horror stories of tool failures discovered during active investigations, resulting in lost evidence and incorrect information*”. The use of more rigorous methods for development and testing of critical parts of the digital evidence processing tools have been repeatedly called for in [3,4,5].

The rest of the paper is organized as follows. Section 2 presents the introduction into the B Method and the associated development methodology. Section 3 describes the functional principles of the software based write blocker systems and outlines the mandatory requirements that such systems need to satisfy. In Section 4 the formal B development of the write blocker systems is described. Finally, Section 5 presents conclusions, gives an overview of the related work and discusses possible future extensions.

2. The B Method

The B Method [1] is an approach for formally specifying and constructing complex software systems with high dependability (e.g., reliability and safety) requirements. It has been successfully used in the development of several complex real-life applications [6,12]. The B Method has also been applied for formal verification of hardware systems [16].

The development methodology of B is based on the concept of (stepwise) *refinement*. The idea of refinement is that an initial abstract specification is gradually transformed into a final implementation via a series of correctness preserving transformation steps, called refinements. At every refinement step, the corresponding *proof obligations* are generated that need to be proven to demonstrate that the refinement is valid, i.e., it preserves the properties stated in the abstract specification. Both *data refinement* and *algorithmic refinement* can be performed at each step. During data refinement abstract data structures are replaced by more concrete ones, e.g., a set is replaced by an ordered sequence. The sequence can be then implemented as an array structure. The algorithmic refinement is used when we need to reduce or eliminate non-deterministic behaviour present in the abstract specification. The presence of non-determinism in a specification essentially means that there are several execution paths satisfying the specification. The refinement process allows us to incorporate concrete implementation decisions (like choosing some particular algorithm) thus making the specification more deterministic.

The B Method supports structuring of the system architecture by modularisation. Modules are represented in B as *abstract machines*, which can be compared to C++ classes or Ada packages. An abstract machine encapsulates system state (represented by state variables) and operations on the state. B abstract machines can be structured hierarchically by the use of structuring mechanisms.

The B structuring mechanisms allow us to decompose complex systems, which might help to reduce complexity and ease the proof process. The examples of such structuring mechanisms are INCLUDES and SEES. For example, if M1 INCLUDES M2 then the state of machine M2 becomes a part of the state of machine M1. Machine M1 can access the state of M2 directly in the read mode. However, M1 can only update the M2 state via the operations of M2. The INCLUDES structuring mechanism models *exclusive access*, i.e., the master/slave relation. If M1 SEES M2, the machine M1 gets the read access to the state of the machine M2. However, it cannot change the M2 state in any way. Thus, in the latter case the machines M1 and M2 can be viewed as distinct machines. The SEES structuring mechanism models *shared access*. The structure of a basic B abstract machine is shown below.

```
MACHINE M
SETS S
CONSTANTS C
PROPERTIES P
VARIABLES v
INVARIANT Inv(v)
INITIALISATION I

OPERATIONS

out ← Op(param) =
    ...
END
```

The MACHINE clause gives the name of an abstract machine. Sets (local types) are introduced in the SETS clause. Constants are declared using the CONSTANTS clause, and the associated properties and typing information for these are defined in PROPERTIES. Next is the VARIABLES clause, where the state variables of an abstract machine are introduced. These variables are typed in the INVARIANT clause, and additional invariant properties (e.g., the safety-related requirements) can be included as a part of the invariant clause as well. The state variables are initialised in the INITIALISATION clause describing the initial state of the machine. Finally, the state transitions (operations in B) are presented in the OPERATIONS clause.

In the classical B approach, operations are of the form PRE P THEN S END. The pre-conditioned operation can only be correctly invoked when the predicate P (pre-condition) is true; when P is false, nothing can be guaranteed about the execution of the operation (i.e., the behaviour of the operation can not be predicted). The pre-conditioned operations in B can be implemented as procedures, which are called by the user. In the event-based B approach [13], operations have the form of the guarded substitutions SELECT P THEN S END. In this case the operations model events that can be triggered by the environment. If P is false, operations specified using the guarded substitution will be in the waiting (hibernating) mode until P becomes true. If the condition P is satisfied (i.e., the operation is enabled) the behaviour is the same as for the pre-conditioned operations. A more general case of guarded substitution is of the form ANY vars WHERE P THEN S END, which can be used to introduce parameterized events.

The B operations are specified using the abstract machine notation (AMN), and the operation bodies are given in terms of AMN substitutions (generalised assignments). The B syntax for the AMN substitutions we use in this paper is given below.

$$S ::= x:=e \mid \text{IF condition THEN } S1 \text{ ELSE } S2 \text{ END} \mid S1 ; S2 \mid x:\in T \mid S1 \parallel S2$$

All substitutions except parallel composition (\parallel) and non-deterministic assignment ($:\in$) have the same meaning as in standard imperative programming languages. However, the sequential composition substitution can only be used in refinements, i.e., it is not a specification construct. The parallel composition $S1 \parallel S2$ is used to model parallel execution of $S1$ and $S2$, and the non-deterministic assignment $x:\in T$ is used to assign a variable x an arbitrary value from a set T .

Both the initialisation and the B operations need to preserve the invariant in order for the abstract machine to be *consistent*. For machine operations, the proof obligations that needs to be discharged to prove consistency are of the form

$$\text{Inv} \wedge P \Rightarrow [S]\text{Inv}$$

stating that the invariant is assumed to hold before an operation is executed, and it should still be true after the operation has been completed, assuming that the operation precondition or guard P holds initially. For refinements, certain additional refinement proof obligations need to be discharged to demonstrate that the transformation preserves the properties stated in the initial specification.

The B Method is enhanced by automatic tools (i.e., Atelier-B [18]) providing full support for the entire development process (e.g., syntax and type checking, automatic and interactive provers, and code generation). The availability of automatic B tool support makes the approach more scalable and also means that less mathematical training is required from the user.

3. The Software Write Blocker systems

The purpose of a write blocker tool is to protect against any attempts to alter the hard drive content when performing data acquisition operations. Both software and hardware based write blockers exist; however, in this paper we will focus only on the software based systems.

For any acquired digital evidence to hold up in a court of law (i.e., be admissible), the responsible investigator needs to show the integrity and reliability of the digital evidence. By using a write blocker tool, the investigator can demonstrate to the court that the evidence has indeed not been altered or tampered with in any way. Because of their importance in digital forensics, the reliability of the write blocker tools themselves needs to be guaranteed.

3.1 The Functional Principles of a Software-based Write Blocker

When a device driver or a software application wants to send commands to a hard disk drive, it can do this by using so called *BIOS interrupt calls*. BIOS stands for *Binary Input Output System* and is responsible for handling among other things the start-up of a computer and initialisation of the computer hardware. An *interrupt* can be described as an asynchronous signal (event) that causes the processor of a computer to switch the context, i.e., stop doing what it is currently doing and start doing something else. When an interrupt is triggered, it will transfer control to the associated interrupt service routine (interrupt *handler*). The address of this interrupt handler is stored in the system in the corresponding interrupt vector list.

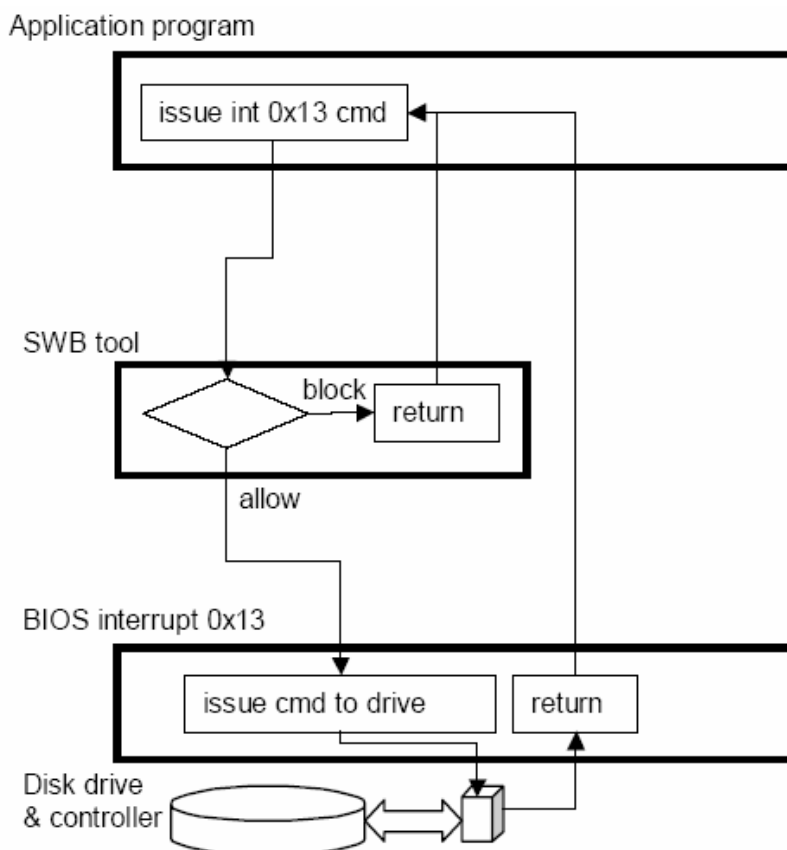


Figure 1. Operation of a software write blocker (SWB) tool.

In order to send a command to a disk drive, an application program should place information about the destination drive and the command to be executed in appropriate hardware registers. Then the program invokes the BIOS interrupt 0x13 to transfer control to the interrupt handler routine in question. The BIOS interrupt 0x13 commands are issued by application programs to perform disk access operations.

When a software write blocker (SWB) is executed, it saves the address of the current interrupt 0x13 handler, and installs a new handler in its place. After this the SWB tool is able to intercept all commands destined for some disk drive and decide whether a particular command should be allowed or blocked. Otherwise, the tool is in idle (hibernating) state waiting for some new command to be processed.

If a command should be allowed to be executed, it is passed to the old interrupt handler. However, if a command is to be blocked, the control is returned back to the application program, i.e., no command is issued to a disk drive. Based on the configuration of the write blocker tool (e.g., the SWB tool can be configured to return success to an application even when a command has been blocked and command execution has actually failed), a command status code of either success or failure is returned to the application program. Fig.1 (from [14]) illustrates the operation of the software write blockers.

3.2 Requirements for the Software Write Blocker systems

In order to be able to assess the reliability of the software-based write blocker tools, US National Institute for Standards and Technology (NIST) has developed an informal requirements document [14] stating the mandatory and optional requirements the SWB tools should adhere to. The developed requirements have been used to generate tests, and several existing commercial write blockers have been tested against them as a part of the CFTT [15] project. The following three high-level requirements need to be satisfied by all write blocker tools [14]:

1. The tool shall not allow a protected drive to be changed.
2. The tool shall not prevent obtaining any information from or about any drive.
3. The tool shall not prevent any operations to a drive that is not protected.

The above high-level requirements are then broken down into the more detailed mandatory and optional requirements, along with the associated assertions. Some examples of the mandatory requirements for the software write blocker systems are given next (the complete list can be found in [14]):

- a)* The tool shall block any commands to a protected drive in the write, configuration or miscellaneous categories.
- b)* The tool shall not block any commands to a protected drive in the read, control or information categories.
- c)* The tool shall report the protection status of all drives.
- d)* The tool shall, if so configured, adjust the return value of any blocked commands to indicate that the operation was carried out successfully even though the operation was blocked.
- e)* The tool shall, if so configured, adjust the return value of any blocked commands to indicate that the operation failed.
- f)* The tool shall not block any commands to an unprotected drive.

Let us explain these requirements in more detail. An application program can send commands (requests) to a disk drive controller (via the BIOS interrupt 0x13 disk interface) in order to perform some operation, e.g., to read some data stored on the disk in question. These commands are represented by the command codes usually specified in the hexadecimal format (for example the command code 0x02 stands for the operation “*read sector(s) into memory*” in some BIOS implementations [14]). In this paper, a *drive* refers to a standard (IDE/ATA interface [19]) hard disk drive, however the SWB tools could also be used with other storage medias such as floppy disks, memory cards etc.

In the SWB requirements the notion of command *categories* is introduced to group the commands based on their type. The commands are grouped into six different categories, namely, the read, write, control, information, miscellaneous and configuration categories. The commands are classified as *modifying* commands in case they belong to the write, configuration or miscellaneous categories, e.g., the write category command 0x03 (*write sectors*). *Non-modifying* commands belong to the read, control, or information categories, e.g., the information category command 0x08 (*read drive parameters*).

The first two requirements (*a*, *b*) are the most critical in order to guarantee reliability of a write blocker system. All modifying commands to a protected drive must be blocked, while non-modifying commands should always be allowed independently of the protection mode of a destination drive. Otherwise, if a modifying command that should be blocked by the write blocker would be passed further to the destination drive, the drive content might be corrupted. As a result, it might not be admissible as the evidence in legal proceedings or security incident investigations.

The requirement *c* specifies that the protection status for all disk drives in the system should be reported by a tool. The requirements *d* and *e* state that it should be possible to adjust the return value for the blocked commands using a SWB tool. Finally, the requirement *f* states that all commands (i.e., both modifying and non-modifying) that are sent to a drive currently not protected by a SWB tool should always be allowed to be executed.

4. Overview of the formal B development

Traditional software development usually begins by formulating detailed functional and non-functional system requirements for the system to be developed. These system requirements are usually *informal*, i.e., they are formulated in natural language. From these requirements an informal system specification is derived. The system design and implementation are then based on this specification. However, natural language is often open to interpretation. As a result, such a system specification might be *ambiguous* and lead to an incorrect implementation.

Formal methods such as B are based on mathematics and logics and, therefore, provide rigor and precision that cannot be achieved by using only natural language. The specifications written using a formal notation can also be more easily analyzed automatically (e.g., by automatic checking of syntax and logical consistency), which is another advantage over traditional informal specifications.

In systems, where the *dependability* requirements are of major importance, it is necessary to guarantee that all system requirements are implemented correctly according to the specification. Formal methods allow us to formally prove that the implementation is correct, i.e., we can demonstrate rigorously that all system requirements are implemented. Using a formal development methodology, we can therefore arrive at a final implementation of the system that is *correct by construction*.

4.1 Organization of the development

The formal write blocker system development is organized hierarchically using the structuring mechanisms of B (i.e., using INCLUDES and SEES). More details are added to the initial (abstract) specification using stepwise refinement. The system structure is shown in Fig.2.

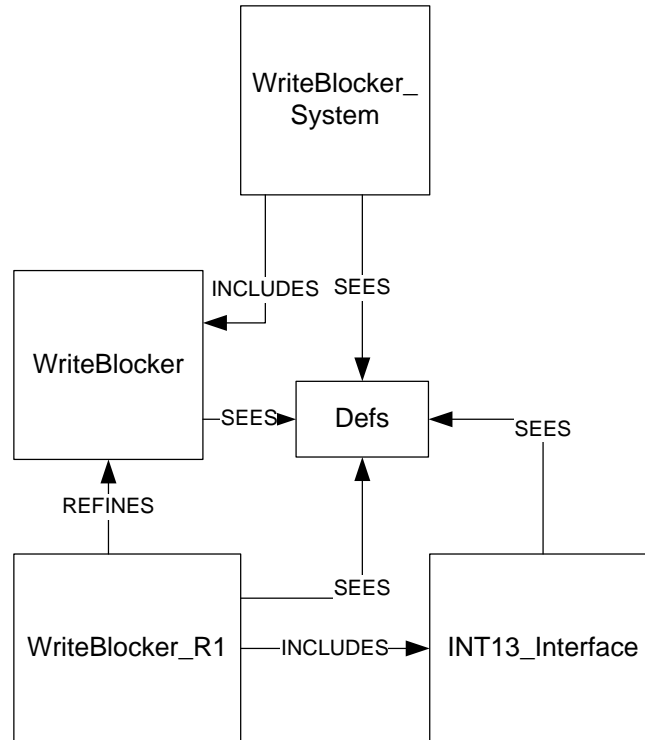


Figure 2. Structure of the formal B development

The abstract specification *WriteBlocker_System* uses the event-based approach to model how the system interacts with its environment. The environment of the system consists of external application programs that generate commands to be processed by the write blocker system. The detailed description of the *WriteBlocker_System* will be presented in the next section.

The specification *WriteBlocker* abstractly models the behaviour of a software write blocker system, and states the properties that the system needs to satisfy based on the informal requirements described in Section 3. During the subsequent refinement step (resulting in the refined specification *WriteBlocker_R1*) more implementation details are added to the abstract B specification (e.g., the details concerning a specific handling of blocked and allowed commands). The subsidiary abstract machine *INT13_Interface* is introduced during the refinement step to model the handling of commands through the BIOS Interrupt 0x13 interface. The INCLUDES structuring mechanism of B is used to incorporate the *INT13_Interface* specification into the refined system.

In the future we are going to concentrate on further refinements of *WriteBlocker* to eventually obtain a detailed B implementation from which executable code can be generated.

4.2 Specification of the top-level WriteBlocker system using event-based B approach

The purpose of the top-level specification *WriteBlocker_System* is to abstractly model the way the system interacts with its external environment. The external environment consists in this case of the external application programs that access disk drives via the interrupt 0x13 interface.

```
MACHINE WriteBlocker_System
INCLUDES WriteBlocker
VARIABLES swb_state, param_drv, in_cmd, in_drv, cur_cmd, cur_drv, out_cmd_status
OPERATIONS
envCmdInput =
SELECT in_cmd = cmd_empty
THEN
  in_cmd := CMD - {cmd_empty}
END;
envDrvInput = ...
readCmd =
SELECT  $\neg(\textit{in\_cmd} = \textit{cmd\_empty}) \wedge \textit{swb\_state} = \textit{swb\_executing}$ 
THEN
  cur_cmd, in_cmd := in_cmd, cmd_empty
END;
readDrv = ...
writeCmdStatus =
SELECT swb_state = swb_cmd_intercepted
THEN
  out_cmd_status, cur_drv, cur_cmd := resp(cur_drv), drv_empty, cmd_empty ||
  swb_state := swb_executing
END;
envReadCmdStatus =
SELECT  $\neg(\textit{out\_cmd\_status} = \textit{emptyval})$ 
THEN
  out_cmd_status := emptyval
END;
enable = ...
disable = ...
installSWB = ...
intercept_cmd =
SELECT  $\neg(\textit{cur\_cmd} = \textit{cmd\_empty}) \wedge \neg(\textit{cur\_drv} = \textit{drv\_empty})$ 
THEN
  processCmd(cur_cmd, cur_drv) || swb_state := swb_cmd_intercepted
END
END
```

Every such access attempt leads to the creation of the corresponding interrupt. When an interrupt occurs, the control is immediately transferred to the corresponding handler routine. We use the event-based approach of B to model interrupt handling by event operations (as we can view interrupt occurrence as triggering of events). The event operations are introduced in the specification using the **SELECT** guarded substitution. The excerpt above outlines the top-level specification.

Fig.3 presents possible execution states of the top-level system and the associated state transitions (e.g., from the initial state the system can enter the executing state etc). The execution flow is specified in B by using the *interleaving* semantics.

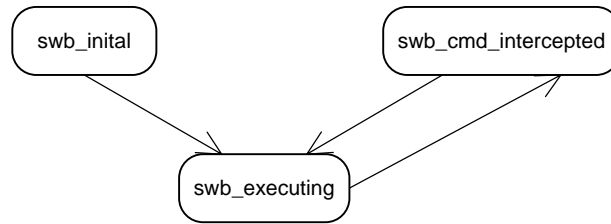


Figure 3. States and state transitions for the top-level WriteBlocker system

The interleaving is achieved by the appropriate operation guards that control the order in which events are invoked. This is important, for instance, to specify that some command needs to be available for processing before the event responsible for command handling is enabled. The specification *WriteBlocker_System* INCLUDES the main specification *WriteBlocker* described in the next section. Therefore, event operations can contain calls to the corresponding operations of the specification *WriteBlocker*.

Communication with the environment is abstractly specified by introduction of the variables *in_cmd* and *in_drv* modelling *communication channels* [9] used by the external applications for supplying command and drive parameters to the system. For example, the event *envCmdInput* models the environment input of commands to be processed. The operation guard specifies that the event is only enabled when there is not already some command waiting for processing in the input channel modelled by *in_cmd*. The constant *cmd_empty* models the absence of a supplied command. Here we assume that the input channel will be read and emptied by the write blocker system almost immediately after it is written to. In later refinements, these abstract channels will be replaced by some concrete communication protocol.

When some command is available in the input channel buffer (and the write blocker system is in executing state), the event *readCmd* becomes enabled. The operation assigns the content of the input channel to the variable *cur_cmd*, which models the current command available for handling by the write blocker. After that the input channel is emptied so that a new command can be supplied by the environment. The handling of the drive input parameters is specified similarly.

When some command and drive are available (i.e., *cur_cmd* and *cur_drv* are non-empty), the event operation *intercept_cmd* becomes enabled. The operation specifies triggering of the disk access interrupt 0x13. The event body contains a call to the operation *processCmd* of the included machine *WriteBlocker*. In addition, the variable *swb_state* of the write blocker system is set to *swb_cmd_intercepted* to indicate that some command has been intercepted. Setting the state to *swb_cmd_intercepted* has the effect of enabling the event *writeCmdStatus*, which writes the command execution code returned by the write blocker system into the output channel buffer *out_cmd_status*. The operation simultaneously empties the variables *cur_cmd* and *cur_drv* so that new command and drive parameters can be read from the input buffer.

Finally, when a command execution code is available in the output buffer, the event operation *envReadCmdStatus* is enabled. The operation *envReadCmdStatus* models the environment access to the output of the write blocker system. As a result of the operation, the constant *emptyval* is assigned to *out_cmd_status*, i.e., the output buffer is emptied.

The event operations *enable* and *disable* specify enabling and disabling of write protection for a given drive. They are specified by calls to the corresponding operations of *WriteBlocker*. The initial installation and activation of the write blocker system, i.e., loading the system into

memory (as a memory-resident application), is modelled by the event *installSWB*, which calls the operation *installSWBHandler* of *WriteBlocker*. The full B specification of the top-level *WriteBlocker_System* can be found in the Appendix.

4.3 Abstract WriteBlocker specification

In the previous section we presented the top-level specification *WriteBlocker_System* describing the system interface to the environment. Next, we will specify the core part of a SWB system responsible for handling of intercepted commands. The informal requirements for the software write blocker systems were presented in Section 3. We are going to incorporate all of these requirements into a specification as the corresponding INVARIANT clauses that should be preserved by all system operations.

Below is an excerpt from the abstract WriteBlocker specification with intuitive text descriptions of the operations. The complete specification can be found in the Appendix.

```

MACHINE WriteBlocker

SETS CMD;
      CMD_CATEGORY = {no_cmd_category, write, configuration, miscellaneous, read,
control, information};
      DRIVE

VARIABLES mode, cur_cmd_category, action, ret_success, resp, swb_active

INVARIANT
  <Typing of state variables and formulation of the system requirements>

OPERATIONS

enableWB(drv) =
  <Enables write protection for a specified drive>

disableWB(drv) =
  <Disables write protection for a specified drive>

setRetSuccess(drv, val) =
  <Set the operational mode configuration for a drive, i.e. whether to return success even though a
command has been blocked by the system>

processCmd(cmd, drv) =
  <Process a command destined for a given drive, i.e. decides whether the command should be
blocked or allowed based on the command category and the drive protection state.>

stat ← getWBMode(drv) =
  <Returns the current protection state for a drive>

installSWBHandler =
  <Models installation and activation of the write blocker system>

END

```

The sets *CMD* and *DRIVE* are used to model commands and drives, respectively. They are abstract (or deferred) sets, i.e., their exact definition is postponed until some later refinement step. The command categories are modelled using the enumerated set *CMD_CATEGORY*.

The state variable *mode* models the protection status of a given drive, i.e., whether it is currently *protected* or *unprotected* by the write blocker. The state variable *action* abstractly models the current action taken by the write blocker system, which can be either allowing or blocking an intercepted command. The variable *ret_success* models the configuration mode of

the write blocker, i.e., whether the system is configured to return the failure or success status after a command has been blocked.

The variable *cur_cmd_category* variable stores the category of the currently processed command. The state variable *resp* is used to model the return value passed by the write blocker system to the calling application (the environment) after an intercepted command has been processed. Finally, the state variable *swb_active* abstractly models activation of the write blocker system, i.e., initial loading of the system into memory.

In the INVARIANT clause we formally state the (mandatory) requirements that the write blocker system need to satisfy. These properties need to hold for all drives in the system. For example, the invariant conjunct

$$\text{mode}(\text{drv}) = \text{unprotected} \wedge \text{cur_cmd_category}(\text{drv}) \in \{\text{write, configuration, miscellaneous}\} \Rightarrow \text{action}(\text{drv}) = \text{allow}$$

formally states that when the mode of a drive is unprotected and the current command belongs to either the write, configuration or miscellaneous categories, the write blocker should allow to execute the command. The variable *drv* is universally quantified inside the invariant, which means that the invariant properties should be true for any drive of the system.

The requirement that all modifying commands to a protected drive should be blocked is expressed in the invariant conjunct

$$\text{mode}(\text{drv}) = \text{protected} \wedge \text{cur_cmd_category}(\text{drv}) \in \{\text{write, configuration, miscellaneous}\} \Rightarrow \text{action}(\text{drv}) = \text{block}$$

The operations *enableWB* and *disableWB* enable and disable write protection for a drive, respectively. The operation *getWBMode* is simply an enquiry operation (i.e., it does not modify the system state in any way) that returns the current protection status for a drive. The initial activation of the write blocker is specified by the operation *installSWBHandler*. The operation *setRetSuccess* sets the current configuration mode for a drive.

The main operation of the system is specified by the operation *processCmd*. Any intercepted command is classified into the category to which it belongs (e.g., write or read) by the abstract function *cmd_category*. The result is assigned to the variable *cur_cmd_category* as outlined below.

```

processCmd(cmd,drv) =
  PRE cmd ∈ CMD ∧ drv ∈ DRIVE
  THEN
    cur_cmd_category(drv) := cmd_category(cmd) ||...

```

Next, based on the protection status of the destination drive and the determined command category, the write blocker either blocks or allows an intercepted command. In case the processed command is blocked, the system returns the execution code of either *fail* or *success* according to the write blocker configuration, as shown in the excerpt below.

```

IF (mode(drv) = protected) ∧ (cmd_category(cmd) ∈ {write, configuration, miscellaneous})
  THEN
    IF ret_success(drv) = TRUE
      THEN
        action(drv) := block || resp(drv) := success
      ELSE

```

```

action(drv) := block || resp(drv) := fail
END

```

If the processed command is allowed, it is passed further to the drive for execution. Since we do not know whether command execution will succeed or fail, we model the command execution by a non-deterministic assignment to the variable *resp*, as shown below.

```

...
ELSE
  action(drv) := allow ||
  ANY val WHERE val ∈ SWB_STATUS-{emptyval}
  THEN
    resp(drv) := val
  END
END

```

The initial abstract specification can be seen as a B specification pattern, because it can be used to specify concrete software write blocking systems by instantiating the abstract data types and functions with concrete data. Moreover, specification process of the hardware-based write blocking systems could be based on the same abstract specification pattern, since the requirements for hardware and software-based write blockers are essentially the same.

4.4 The first refinement of WriteBlocker

The general purpose of refinement process is to gradually add implementation details to an abstract specification. In our first refinement step we specify in more detail the handling of blocked and allowed commands.

In the initial specification of a write blocker system we modelled the action undertaken by the system (i.e., allowing or blocking of intercepted commands) in a very abstract way. This was done by the corresponding assignment to the variable *action*. The value assigned to the variable *action* directly depended on the category of an intercepted command.

In the first refinement step we decompose our write blocker specification by introducing a subsidiary abstract machine *INT13_Interface*. This machine abstractly specifies a specific handling of intercepted commands via the interrupt 0x13 interface. The machine *WriteBlocker* is then refined (by *WriteBlocker_R1*) in such a way that the assignments to the variable *action* are replaced by activation of the corresponding interrupt handlers defined in *INT13_Interface*.

<pre> MACHINE <i>INT13_Interface</i> SETS <i>ADDRESS</i>; <i>SWB_STATUS</i> = {<i>success</i>, <i>fail</i>, <i>emptyval</i>} VARIABLES <i>old_handler_addr</i>, <i>swb_handler_addr</i>, <i>handler_invoke</i>, <i>drv_modified</i> INVARIANT <Typing of state variables> OPERATIONS installHandler = <Models the installation of the write blocker interrupt handler> setHandler(<i>drv</i>,<i>handler</i>) = <Determines which handler routine should be invoked, i.e., the old handler or the replacement write blocker handler> <i>ret</i> ← int_handler(<i>drv</i>, <i>cmd</i>) = <Abstractly models invocation of the interrupt handler > END </pre>

Interrupt 0x13 interface. As mentioned in Section 3, application programs can communicate with the hard disk controllers via the BIOS interrupt 0x13. The write blocker systems replace the standard interrupt handler in order to be able to intercept all commands and take appropriate actions depending on their category. By specifying the interrupt 0x13 interface very abstractly, we do not impose any serious implementation restrictions on the externally provided interface. In other words, the interrupt handler mechanism can be implemented using any external code that satisfies the abstract B specification. The correctness of the externally supplied code needs to be verified separately. The description of our abstract specification of the interrupt 0x13 interface is given above.

The installation and activation of the write blocker system is specified by the operation *installHandler*, in the following way:

```

ANY addr WHERE addr ∈ ADDRESS - {null_value,sys_handler}
THEN
    swb_handler_addr := addr
END

```

The handler addresses are modelled by introducing the deferred type *ADDRESS*. The state variable *swb_handler_addr* gets non-deterministically assigned some distinct address, which can be any address except null or the replaced system handler address (modelled by *sys_handler*).

The operation *setHandler* sets which an interrupt handler is to be invoked. The operation assigns the value *old_handler* to the variable *handler_invoke* in case the standard handler should be invoked, or the value *swb_handler* in case the write blocker system should handle the command.

The operation *int_handler* specifies invocation of the interrupt handler routine by the write blocking system. The operation updates the state variable *drv_modified* to reflect the modification state of a drive (i.e., whether the drive is modified or not by the interrupt invocation). The operation returns the command execution code, i.e., the values *success* or *fail* (of the type *SWB_STATUS*).

The first part of the operation checks whether the current handler points to the old (standard) BIOS 0x13 handler or, alternatively, the replacement handler installed by the SWB tool. In case the current handler to be invoked is the old system handler, the returned execution code can be any non-empty value, which is modelled by the corresponding non-deterministic assignment to the return parameter *ret*. The reason for assigning the return value non-deterministically is that we do not know whether the command execution will succeed or fail. During later refinements this non-deterministic assignment will be replaced by assigning the result of some external procedure call. At the same time, the modification status of a drive, modelled by the variable *drv_modified*, is also updated non-deterministically expressing the fact that the drive may or may not have been modified. This is presented in the excerpt below.

```

IF handler_invoke(drv) = old_handler
THEN
    ANY mod_state WHERE mod_state ∈ BOOL
    THEN
        drv_modified(drv) := mod_state END ||
        ret := ∈ SWB_STATUS-{emptyval}
    END

```

If the write blocker interrupt handler handles the command, we model it by assigning some non-deterministic value to the return parameter *ret*, and setting *drv_modified* to *FALSE* (i.e., the drive is not modified in this case), as shown below.

```

...
ELSE
  drv_modified(drv) := FALSE ||
  ret :∈ SWB_STATUS-{emptyval}
END

```

The WriteBlocker refinement. The state of the abstract machine *INT13_Interface* becomes a part of the state of the refinement machine *WriteBlocker_R1* by the use of the **INCLUDES** structuring mechanism. The excerpt from the refinement machine (with intuitive textual descriptions) is given below.

```

REFINEMENT WriteBlocker_R1

REFINES WriteBlocker

INCLUDES INT13_Interface

VARIABLES mode, ret_success, cur_cmd_category, resp

INVARIANT
  <Formulation of the gluing invariant on the refinement state and the
  refined state of the abstraction>

OPERATIONS
...
processCmd(cmd,drv) =
  <Refinement of the handling of blocked and allowed commands, e.g.
  allowed commands are passed further for execution to the BIOS
  interrupt 0x13 handler routine. In case of a blocked command the
  replacement handler will just return>

installSWBHandler =
  <Installation and activation of the write blocker is refined by calls to the
  installHandler operation in INT13_Interface to model how the write
  blocker is installed at some (non-deterministically) determined
  address>

END

```

The invariant of the refinement machine serves as a *gluing* invariant, tying the state of the refinement to that of its abstraction. Particularly, the *gluing* invariant states the connection between the state variable *action* of the abstract specification and the state variable *handler_invoke* introduced in the refinement. For example, the first conjunct of the invariant

$$\text{action}(\text{drv}) = \text{block} \Rightarrow \text{handler_invoke}(\text{drv}) = \text{swb_handler}$$

requires that, when the action of the write blocker is blocking in the abstract specification, the command in question should be taken care of by the write blocker handler, and not passed further to the destination drive. The state variable *swb_active* of the abstraction is replaced by the refinement variables *old_handler_addr* and *swb_handler_addr* modelling the addresses of the old system handler and the write blocker handler, respectively. The following invariant conjunct

$$\begin{aligned} \text{swb_active} = \text{TRUE} \Rightarrow & \text{old_handler_addr} = \text{sys_handler} \wedge \\ & \text{swb_handler_addr} \neq \text{null_value} \wedge \\ & \text{swb_handler_addr} \neq \text{sys_handler} \end{aligned}$$

simply states that when the write blocker is active, the address of the replaced standard interrupt handler points to the system handler address, while the address of the replacement (the write blocker) handler points to some non-null address distinct from the system handler address. A non-null address value for the replacement handler means that the write blocker has been successfully loaded into memory.

The operation *processCmd* is refined by replacing assignments to the variable *action* with the calls to operations of the included machine. These operation calls model a specific way (using the interrupt 0x13 interface) the write blocker system handles blocked and allowed commands. The excerpt of the refined operation is given below.

```

IF (mode(drv) = protected) ∧
    (cur_cmd_category(drv) ∈
    {write, configuration, miscellaneous})
THEN
    setHandler(drv,swb_handler);
    VAR val IN
        val ← int_handler(drv,cmd)
    END;
    IF ret_success(drv) = TRUE
    THEN
        resp(drv) := success
    ELSE
        resp(drv) := fail
    END
END

```

If an intercepted command should be blocked, the call to *setHandler* specifies that the write blocker is set to handle the command. In this case, the write blocker returns the execution code of the command to the calling application. The actual value of the execution code (*success* or *failure*) depends on the current configuration of the write blocker.

The operation *int_handler* updates the variable *drv_modified*, modelling the modification state of the drive in question, and will return an execution code of *success* or *failure*. The VAR substitution is used here for introducing a local variable in the operation body.

In the case when the command should be allowed to be executed, we model the transfer of control to the old interrupt handler by setting the current handler to the system handler (by the operation *setHandler*), and then forwarding the command to it (by the operation *int_handler*).

```

...
ELSE
    setHandler(drv,old_handler);
    VAR val IN
        val ← int_handler(drv,cmd);
        resp(drv) := val
    END
END

```

Note that the parallel substitutions used in the abstract specification were replaced by sequential composition in the refinement machine.

Abstract installation of the write blocker is refined in the operation *installSWBHandler* by a call to the operation *installHandler* of *INT13_Interface*. The complete refined specification can be found in the Appendix.

5. Conclusions

In this paper we presented a formal specification and refinement of a write blocking system (SWB) using the B formal method [1]. The initial formal B specification has been obtained by translation of the informal NIST requirements for SWB systems given in [14]. The system refinement described in the paper demonstrates how more concrete implementation details can be added later, in order to specify a more detailed handling of blocked and allowed commands (in this specific case, via the interrupt 0x13 interface). Moreover, by modelling the top-level system using the Event-B [13] approach, we specified interaction of the write blocking system with its external environment. The presented work shows how formal methods such as B can benefit the digital forensics discipline by increasing the reliability of software tools used for critical stages of digital investigation.

All formal development steps have been proved by using the Atelier-B [18] tool. The proofs were complete, i.e., all associated proof obligations have been discharged using the automatic and interactive provers provided by the tool. Some statistics for the B development are shown in Table 1.

<i>B component</i>	<i>Number of proof obligations</i>	<i>% of proof obligations proved automatically / interactively</i>
<i>WriteBlocker_System</i>	1	100 / 0
<i>WriteBlocker</i>	74	62 / 38
<i>INT13_Interface</i>	12	100 / 0
<i>WriteBlocker_R1</i>	56	63 / 37

Table 1. Proof obligation statistics for the B SWB development

In [11], Lyle and Black described a methodology for testing (the BIOS interrupt 0x13 based) software write blocker tools. Their approach is based on deriving relevant test cases from the informal requirements document [14] developed at NIST. The test cases were used for verification of the SWB tools using software utilities specifically developed for this purpose. Our proposed approach is focused on development of a formal specification for the SWB tools from which a final executable implementation could be obtained by using the concept of stepwise refinement in B. It can be seen as a specification pattern since the initial specification abstractly models the working of a general SWB system. During later refinement steps, specific interface dependent implementation details can be added. However, the relevant test cases for testing existing write blocking tools could also be systematically derived from the formal B specification in order to achieve the maximum test coverage. Investigation of this is a part of our future work.

Gladyshev [8] proposed a state machine formalisation of the digital event reconstruction theory. The objective of his research is to develop a theory for formal reasoning about incidents in digital systems. The main idea is to describe the system under investigation as a finite state machine, and, by using algorithms based on transition back-tracing, remove the incident scenarios that disagree (contradict) with the available evidence. The MAC times attached to files (i.e., the last *modification*, *access* and *creation* times) are the useful attributes that are

frequently used for performing temporal reconstruction of events that have occurred during an incident. These time attributes might be tampered, thus invalidating the collected digital evidence. The purpose of employing the write blocking tools during disk acquisition operations is to avoid such intentional or accidental modification of evidence. The tools for processing and reasoning about the MAC times of files could be formally specified and implemented based on the state machine reconstruction approach, and this is another subject of our future research.

Our future work and extensions will involve further refinement of the presented write blocker B specification to eventually obtain an executable implementation. Moreover, it would be interesting to investigate the possibility of incorporating the *fault tolerance* [2] mechanisms into our B specifications to cope with system failures or unexpected events (e.g., a command belonging to an unrecognised category). Introducing such fault tolerance mechanisms into the write blocking systems would further strengthen their reliability.

Acknowledgements

We would like to thank Dr. Paul E. Black of the US National Institute for Standards and Technology for his very helpful comments on our formal specification of the write blocker. Also, we would like to thank Dr. Pavel Gladyshev of University College Dublin for his valuable comments and suggestions on the contents of this paper.

References

- [1] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] T. Anderson, P.A. Lee, *Fault Tolerance: Principles and Practice*. Dependable Computing and Fault Tolerant Systems, Vol. 3, Springer Verlag, 1990.
- [3] B. Carrier, *Open Source Digital Forensics Tools: The Legal Argument*, @stake research report, 2002. Available at http://www.atstake.com/research/reports/acrobat/atstake_opensource_forensics.pdf.
- [4] E. Casey, Editorial in *Digital Investigation*, Vol. 2, Issue 2, June 2005, Elsevier Science Ltd.
- [5] E. Casey, *Digital Evidence and Computer Crime*, Elsevier Academic Press, 2004.
- [6] P. Behm, et al., *METEOR: A successful application of B in a large project*. In Wing et al (Eds.), Proc. of the World Congress on Formal Methods. LNCS 1709, 1999.
- [7] Digital Intelligence, *PDBlock*. Available at <http://www.digitalintelligence.com>
- [8] P. Gladyshev, *Formalising Event Reconstruction in Digital Investigations*. Ph.D. dissertation, Department of Computer Science, University College Dublin, 2004.
- [9] L. Laibinis, E. Troubitsyna, S. Leppänen, J.Lilius, Q. Malik, *Formal Service-Oriented Development of Fault-Tolerant Communicating Systems*. In M. Butler, C. Jones, A. Romanovsky, E. Troubitsyna (Eds.), Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005), June 2005.
- [10] M. Leuschel and M. Butler. *ProB: A Model Checker for B*. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli (Eds.), Proceedings FME 2003, Pisa, Italy, LNCS 2805, pages 855–874. Springer, 2003.
- [11] James R Lyle, Paul E. Black, *Testing BIOS Interrupt 0x13 Based Software Write Blockers*. In Proc. of ECCE 05, Monaco, March 2005.
- [12] *MATISSE (Methodologies and Technologies for Industrial Strength Systems Engineering)*. IST Project 1999-11435. Available at <http://www.matisse.qinetiq.com/>

- [13] C. Métayer , J.-R. Abrial, L. Voisin, *Event-B Language*, RODIN (Rigorous Open Development Environment for Complex Systems) Project IST-511599 Deliverable 3.2, May 2005. Available at <http://rodin.cs.ncl.ac.uk>.
- [14] NIST, *Software Write Block Tool Specification & Test Plan*, 2003. Available at http://www.cftt.nist.gov/documents/SWB-STP-V3_1a.pdf.
- [15] *NIST Computer Forensics Tool Testing Project CFTT*. Available at <http://www.cftt.nist.gov>
- [16] *PUSSEE (Paradigm Unifying System Specification Environments for proven Electronic design)*. IST Project 2000-30103. Available at <http://www.keesda.com/pussee/>
- [17] E. Sekerinski, K. Sere (Eds.), *Program Development by Refinement – Case Studies Using the B Method*. Springer Verlag, 1999.
- [18] Steria, Aix-en-Provence, France. *Atelier B. User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index_uk.html.
- [19] T13 Technical Committee, *AT Attachment Standards*. Available at <http://www.t13.org/#Standards>.

Appendix

Complete specifications of the B SWB development

MACHINE *WriteBlocker*

SEES *Defs*

SETS

$CMD_CATEGORY = \{no_cmd_category, write, configuration, miscellaneous, read, control, information\};$
 $SWB_MODE = \{protected, unprotected\};$
 $SWB_ACTION = \{allow, block, idle\}$

VARIABLES *mode, cur_cmd_category, action, ret_success, resp*

INVARIANT

$mode \in DRIVE \rightarrow SWB_MODE \wedge$
 $cur_cmd_category \in DRIVE \rightarrow CMD_CATEGORY \wedge$
 $action \in DRIVE \rightarrow SWB_ACTION \wedge$
 $ret_success \in DRIVE \rightarrow \mathbf{BOOL} \wedge$
 $resp \in DRIVE \rightarrow SWB_STATUS\{-emptyval\} \wedge$
 $(\forall drv.(drv \in DRIVE \Rightarrow ($
 $(mode(drv) = unprotected \wedge$
 $cur_cmd_category(drv) \in \{write, configuration, miscellaneous\} \Rightarrow$
 $action(drv) = allow) \wedge$
 $(mode(drv) = unprotected \wedge$
 $cur_cmd_category(drv) \in \{read, control, information\} \Rightarrow$
 $action(drv) = allow) \wedge$
 $(mode(drv) = protected \wedge$
 $cur_cmd_category(drv) \in \{write, configuration, miscellaneous\} \Rightarrow$
 $action(drv) = block) \wedge$
 $(mode(drv) = protected \wedge$
 $cur_cmd_category(drv) \in \{read, control, information\} \Rightarrow$
 $action(drv) = allow) \wedge$
 $((cur_cmd_category(drv) = no_cmd_category) \Leftrightarrow (action(drv) = idle)))))) \wedge$
 $swb_active \in \mathbf{BOOL}$

INITIALISATION

$mode := DRIVE \times \{unprotected\} \parallel$
 $ret_success := DRIVE \times \{\mathbf{FALSE}\} \parallel cur_cmd_category := DRIVE \times \{no_cmd_category\} \parallel$
 $action := DRIVE \times \{idle\} \parallel resp := DRIVE \times \{success\} \parallel swb_active := \mathbf{FALSE}$

OPERATIONS

enableWB(*drv*) =
 PRE $drv \in DRIVE \wedge action(drv) = idle$
 THEN
 $mode(drv) := protected$
 END;

```

disableWB(drv) =
PRE drv ∈ DRIVE
THEN
    mode(drv) := unprotected ||
    action(drv) := idle ||
    cur_cmd_category(drv) := no_cmd_category
END;

setRetSuccess(drv, val) =
    PRE drv ∈ DRIVE ∧ val ∈ BOOL
    THEN
        ret_success(drv) := val
    END;

processCmd(cmd, drv) =
    PRE cmd ∈ CMD ∧ drv ∈ DRIVE
    THEN
        cur_cmd_category(drv) := cmd_category(cmd) ||
        IF (mode(drv) = protected) ∧ (cmd_category(cmd) ∈ {write, configuration, miscellaneous})
        THEN
            IF ret_success(drv) = TRUE
            THEN
                action(drv) := block || resp(drv) := success
            ELSE
                action(drv) := block || resp(drv) := fail
            END
        ELSE
            action(drv) := allow ||
            ANY val WHERE val ∈ SWB_STATUS-{emptyval}
            THEN
                resp(drv) := val
            END
        END
    END;

stat ← getWBMode(drv) =
    PRE drv ∈ DRIVE
    THEN
        stat := mode(drv)
    END;

installSWBHandler =
BEGIN
    swb_active := TRUE
END

END

```

REFINEMENT *WriteBlocker_RI*

REFINES *WriteBlocker*

SEES *Defs*

INCLUDES *INT13_Interface*

VARIABLES *mode, ret_success, cur_cmd_category, resp*

INVARIANT $\forall drv.(drv \in DRIVE \Rightarrow$
 $((action(drv) = block) \Rightarrow$
 $(handler_invoke(drv) = swb_handler)) \wedge$
 $((action(drv) = allow) \Rightarrow$
 $(handler_invoke(drv) = old_handler)) \wedge$
 $((action(drv) = idle) \Rightarrow$
 $(handler_invoke(drv) = no_handler)))) \wedge$
 $((swb_active = \mathbf{FALSE}) \Rightarrow$
 $(old_handler_addr = sys_handler \wedge swb_handler_addr = null_value)) \wedge$
 $((swb_active = \mathbf{TRUE}) \Rightarrow$
 $(old_handler_addr = sys_handler \wedge swb_handler_addr \neq null_value \wedge$
 $swb_handler_addr \neq sys_handler))$

INITIALISATION *mode := DRIVE* \times *{unprotected} ||*
 ret_success := DRIVE \times *{FALSE} ||*
 cur_cmd_category := DRIVE \times *{no_cmd_category} ||*
 resp := DRIVE \times *{success}*

OPERATIONS

enableWB(*drv*) =
 PRE *drv* \in *DRIVE* \wedge *cur_cmd_category*(*drv*) = *no_cmd_category*
 THEN
 mode(*drv*) := *protected*
 END;

disableWB(*drv*) =
 PRE *drv* \in *DRIVE*
 THEN
 mode(*drv*) := *unprotected*;
 cur_cmd_category(*drv*) := *no_cmd_category*;
 resetDrvState(*drv*);
 setHandler(*drv*, *no_handler*)
 END;

setRetSuccess(*drv*, *val*) =
 PRE *drv* \in *DRIVE* \wedge *val* \in **BOOL**
 THEN
 ret_success(*drv*) := *val*
 END;

```

processCmd(cmd,drv) =
  PRE cmd ∈ CMD ∧ drv ∈ DRIVE
  THEN
    cur_cmd_category(drv) := cmd_category(cmd);
    IF (mode(drv) = protected) ∧ (cur_cmd_category(drv) ∈ {write, configuration, miscellaneous})
    THEN
      setHandler(drv,swb_handler);
      VAR val IN
        val ← int_handler(drv,cmd)
      END;
      IF ret_success(drv) = TRUE
      THEN
        resp(drv) := success
      ELSE
        resp(drv) := fail
      END
    ELSE
      setHandler(drv,old_handler);
      VAR val IN
        val ← int_handler(drv,cmd);
        resp(drv) := val
      END
    END
  END;

stat ← getWBMode(drv) =
  PRE drv ∈ DRIVE
  THEN
    stat := mode(drv)
  END;

installSWBHandler =
BEGIN
  installHandler
END

END

```

MACHINE *INT13_Interface*

SEES *Defs*

SETS *ADDRESS*;

HANDLER = {*no_handler*, *old_handler*, *swb_handler*}

VARIABLES *old_handler_addr*,
swb_handler_addr,
handler_invoke,
drv_modified

CONSTANTS *null_value*,
sys_handler

PROPERTIES *null_value* ∈ *ADDRESS* ∧ *sys_handler* ∈ *ADDRESS* ∧
sys_handler ≠ *null_value*

INVARIANT

old_handler_addr ∈ *ADDRESS* ∧
swb_handler_addr ∈ *ADDRESS* ∧
handler_invoke ∈ *DRIVE* → *HANDLER* ∧
drv_modified ∈ *DRIVE* → **BOOL**

INITIALISATION *old_handler_addr* := *sys_handler* ||
swb_handler_addr := *null_value* ||
handler_invoke := *DRIVE* × {*no_handler*} ||
drv_modified := *DRIVE* × {**FALSE**}

OPERATIONS

installHandler =

BEGIN

ANY *addr* **WHERE** *addr* ∈ *ADDRESS* - {*null_value*,*sys_handler*}

THEN

swb_handler_addr := *addr*

END

END;

setHandler(*drv*,*handler*) =

PRE *drv* ∈ *DRIVE* ∧ *handler* ∈ *HANDLER*

THEN

handler_invoke(*drv*) := *handler*

END;

resetDrvState(*drv*) =

PRE *drv* ∈ *DRIVE*

THEN

drv_modified(*drv*) := **FALSE**

END;

```

ret ← int_handler(drv, cmd) =
  PRE drv ∈ DRIVE ∧ cmd ∈ CMD
  THEN
    IF handler_invoke(drv) = old_handler
    THEN
      ANY mod_state WHERE mod_state ∈ BOOL
      THEN
        drv_modified(drv) := mod_state
      END ||
      ret := ∈ SWB_STATUS-{emptyval}
    ELSE
      ret := ∈ SWB_STATUS-{emptyval} ||
      drv_modified(drv) := FALSE
    END
  END
END

```

MACHINE *WriteBlocker_System*

SEES *Defs*

INCLUDES *WriteBlocker*

SETS *STATE* = {*swb_initial*, *swb_executing*, *swb_cmd_intercepted*}

CONSTANTS *cmd_empty*, *drv_empty*

PROPERTIES *cmd_empty* \in *CMD* \wedge *drv_empty* \in *DRIVE*

VARIABLES *swb_state*, *param_drv*, *in_cmd*, *in_drv*, *cur_cmd*, *cur_drv*, *out_cmd_status*

INVARIANT *swb_state* \in *STATE* \wedge *in_cmd* \in *CMD* \wedge *cur_cmd* \in *CMD* \wedge
in_drv \in *DRIVE* \wedge *cur_drv* \in *DRIVE* \wedge *param_drv* \in *DRIVE* \wedge
out_cmd_status \in *SWB_STATUS*

INITIALISATION *swb_state* := *swb_initial* ||
in_cmd := *cmd_empty* ||
cur_cmd := *cmd_empty* ||
in_drv := *drv_empty* ||
cur_drv := *drv_empty* ||
param_drv := *drv_empty* ||
out_cmd_status := *emptyval*

OPERATIONS

envCmdInput =
SELECT *in_cmd* = *cmd_empty*
THEN
in_cmd : \in *CMD* - {*cmd_empty*}
END;

envDrvInput =
SELECT *in_drv* = *drv_empty*
THEN
in_drv : \in *DRIVE* - {*drv_empty*}
END;

readCmd =
SELECT \neg (*in_cmd* = *cmd_empty*) \wedge *swb_state* = *swb_executing*
THEN
cur_cmd, *in_cmd* := *in_cmd*, *cmd_empty*
END;

readDrv =
SELECT \neg (*in_drv* = *drv_empty*) \wedge *swb_state* = *swb_executing*
THEN
cur_drv, *in_drv* := *in_drv*, *drv_empty*
END;

```

writeCmdStatus =
SELECT swb_state = swb_cmd_intercepted
THEN
  out_cmd_status,cur_drv,cur_cmd := resp(cur_drv),drv_empty,cmd_empty ||
  swb_state := swb_executing
END;

envReadCmdStatus =
SELECT  $\neg$ (out_cmd_status = emptyval)
THEN
  out_cmd_status := emptyval
END;

installSWB =
SELECT swb_state = swb_initial
THEN
  installSWBHandler || swb_state := swb_executing
END;

enable =
ANY drv WHERE
drv ∈ DRIVE ∧ ¬(param_drv = drv_empty) ∧ swb_state = swb_executing ∧ drv = param_drv ∧
action(drv) = idle
THEN
  enableWB(drv) || param_drv := drv_empty
END;

disable =
ANY drv WHERE
drv ∈ DRIVE ∧ ¬(param_drv = drv_empty) ∧ swb_state = swb_executing ∧ drv = param_drv
THEN
  disableWB(drv) || param_drv := drv_empty
END;

intercept_cmd =
SELECT  $\neg$ (cur_cmd = cmd_empty) ∧  $\neg$ (cur_drv = drv_empty)
THEN
  processCmd(cur_cmd,cur_drv) || swb_state := swb_cmd_intercepted
END

END

```

MACHINE *Defs*

SETS

CMD;

CMD_CATEGORY = {*no_cmd_category*, *write*, *configuration*, *miscellaneous*, *read*,
control, *information*};

SWB_STATUS = {*success*, *fail*, *emptyval*};

DRIVE

CONSTANTS *cmd_category*

PROPERTIES *cmd_category* \in *CMD* \rightarrow *CMD_CATEGORY*-{*no_cmd_category*}

END

TURKU
CENTRE *for*
COMPUTER
SCIENCE

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research



Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1624-7
ISSN 1239-1891