Rigorous Development of Automated Inconsistency Checks for Digital Evidence Using the B Method

Pavel Gladyshev
School of Computer Science and Informatics
University College Dublin, Ireland

Andreas Enbacka
Department of Information Technologies
Abo Akademi University, Turku, Finland

Abstract

Inconsistencies in various data structures, such as missing log records and modified operating system files, have long been used by intrusion investigators and forensic analysts as indicators of suspicious activity. This paper describes a rigorous methodology for developing such inconsistency checks and verifying their correctness. It is based on the use of the B Method – a formal method of software development. The idea of the methodology is to (1) formulate a state-machine model of the (sub)system in which inconsistencies are being detected, (2) formulate consistency criteria for the state of that model, (3) rigorously verify correctness of these criteria using the B Method, and (4) automatically search evidential data for violations of the formulated consistency criteria using ConAlyzer utility developed for this purpose. The methodology is illustrated on an FTP server example.

Section 1: Introduction

Automated checking for inconsistencies in evidential data

Much of advanced digital forensics comes from observations of how the operating system manipulates computer's data storage and how various user activities modify operating systems data. For example, when the user logs into the system, a corresponding log file entry is placed into the system log, and the time of the last access to the user configuration data is modified. If the user logs into the system remotely, some record of the incoming Internet connection may be created (e.g. a record in the firewall log file). Usually user activity leaves multiple "traces" in different data structures. By analysing file and registry timestamps, system log file entries, browser history records, deleted temporary files, and other system and application "artefacts" investigator is able to portray past user activity in great detail.

An obvious defence against digital forensics is to try to hide these traces through deliberate modification of system data - for example by deleting offending entries in the log file. A more systematic approach is taken by the Metasploit Antiforensics Framework project[1], which is developing a collection of tools for hiding user traces in various system data structures.

[1] http://www.metasploit.com/

Although it is possible to modify system data structures, it is currently difficult to modify them in a consistent manner, so that effects of user actions are cancelled across multiple data structures. Discovery of inconsistencies in system data can potentially lead to an investigative breakthrough.

Discovery of inconsistencies is not a straightforward task given the large volume of data normally processed in digital investigations. It is possible, however, to automate this process to some degree by programming automated checks that scan through the evidential data and check it for *violations* of some property that must invariably hold in the system data if the system operates *normally*.

For example, when a sub-directory is created in the FAT file system, the creation time of the '.' and '..' entries within that directory is equal to the creation time of the directory's entry in the parent directory. This condition may be violated, for example, when the directory is restored from a backup tape. In which case the creation time for the sub-directory's entry is taken from the tape, while the creation time of the '.' and '..' entries is the restoration time. A violation of this relation between timestamps, therefore, may indicate that the directory was restored from a backup tape.

If such an automated inconsistency check is implemented, it is desirable to reduce the number of false positives generated by it. Such false positives are unusual system data, which look suspicious to the investigator, but have been generated by the system in the normal course of operation.

Although the investigator usually has a feeling for what kind of property should hold in the evidential data, it may be difficult to see if the violation of that property is really inconsistent with the normal processes in the system.

Contribution of this paper

To make the derivation of evidence inconsistency checks more scientific, we explored the possibility to use the formal B Method for development of such checks. This paper presents the approach and the results of that exploration.

Our approach

The B Method (Abrial, 1996) is a well-known formal method of systems development. It has been used for design and implementation of large safety critical systems in Europe and is gaining popularity in industry (Behm et al., 1999). The B Method possesses several features that make it convenient for specification and analysis of inconsistency checks for digital evidence. These are as follows.

- In the B Method, system functionality is described using a language that resembles the algorithmic language found in computer science textbooks. So, to specify how the system works, you simply write a high-level algorithm of the system. Some parts of the algorithm may be left unspecified using non-deterministic constructs.

- Once the system functionality is specified, the B Method provides a standard way for specifying arbitrary properties of the system and verifying that these properties are *invariant*, that is, that these properties *always* hold in the system, if the system is functioning according to the specification.
- There are several semi-automatic tools that verify invariance of given properties with respect to given system specification. They remove much of tedious mathematical work usually connected with the use of formal methods.

The B Method has traditionally been used for designing safety-critical computer systems. The idea there is to formulate safety requirements as “safety properties” that must always hold in the system. The system is then designed in a controlled fashion that maintains specified safety properties of the design. The automated tools can be used to monitor the continuing conformance of the design to the safety properties.

Our use of the B Method is slightly different. We use the B Method to verify our intuition about forensically significant invariant properties of the system. Our approach is as follows:

1. Express understanding of the system functionality as a high-level, partially specified algorithm in the B formal language.
2. Specify properties (*consistency criteria*) that should always hold in the system in the B formal language.
3. Use the automated verification tools of the B Method (ProB (Leuschel & Butler, 2003) and Atelier-B (Steria, 2001)) to check that the specified properties are, indeed, invariant.
4. Write a program that inspects the evidential data and finds violations of the identified invariant properties[2] (inconsistency checking tool). The formal model is used as a base for deriving the inconsistency checks implemented by the program. The INVARIANT statement of the B model is negated and translated into an SQL query for the special-purpose database engine that views operating system data as a relational database. In this paper, the term *inconsistency check* refers to the negated invariant properties.

This approach has been used to construct an inconsistency checker for analyzing files stored on an FTP server, and finding presence or absence of files whose timestamps are inconsistent with the FTP server’s log files. Such a tool is able to identify files that were manipulated (created / modified / or deleted) by means other than the FTP server. This work is described in Section 3 of this paper. Our findings are two-fold:

1. Despite considerable time spent on the development and analysis of the FTP server model, it was helpful because it helped us to establish the exact mathematical formulation of the consistency properties that we identified, and

[2] It is very important that the inconsistency checking program implements the property check correctly. This issue, however, is not addressed in this paper, as the methods for deriving correct programs from specifications are described elsewhere in the literature (e.g. in (Morgan, 1994)).

2. Once consistency properties were formulated mathematically, if was very *straightforward* to convert them into an executable program. Rather than programming it in a general purpose programming language, such as C, C++, Java, or Perl we used a database management system (SQLite) instead. The use of SQL statements allowed very concise formulation of consistency property checks.

Paper organization

The rest of the paper is organised into three sections.

Section 2 is a brief introduction into the relevant aspects of the B Method. It discusses the use of state machines as models for real systems, covers relevant parts of the B notation, and points out the tools supporting the B Method. Furthermore, a methodology for expressing inconsistency checks using SQL is described. Section 3 then gives an overview of the formal model of an FTP server, and describes the implementation of an inconsistency-checking tool based on the formal B model. The paper concludes with an analysis of advantages and disadvantages of the proposed methodology in Section 4.

Section 2: Formal Background

Introduction to the B Method

Digital systems in general can be modelled as so called *(*finite*) state machines (FSM)*. A state machine consists of a (finite) *state space (*containing all possible states the system can be in*)* and *transitions*. A transition from one state to another (triggered by some particular *event*) corresponds to a state change.

The B Method (Abrial, 1996) is based on the first-order predicate calculus and set theory, and the concept of an abstract machine encapsulating a program state as well as operations on that state. Abstract machines are similar to objects in an object-oriented language like C++. Specification in B is done using the Abstract Machine Notation (AMN), which resembles a programming language. The structure of a basic B abstract machine is shown below.

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

The MACHINE clause gives the name of the abstract machine. It is followed by a number of mandatory and optional sections that define individual aspects of the abstract state machine. For example, necessary data types may be defined in the SETS clause. Symbolic constants may be defined using the CONSTANTS clause, and associated properties and typing information for these constants may be defined in PROPERTIES. State variables (memory elements of the machine) are defined in the VARIABLES clause. The initial values of the memory elements are defined by the INITIALISATION clause.

Table 1 - Description of B AMN substitutions

B AMN substitution	Description	Example
x:=e	*Assignment.* The variable *x* is assigned the value *e*.	x:=2 the new value of x is 2
S1||S2	*Parallel composition.* The substitutions *S1* and *S2* are executed simultaneously (i.e., in parallel).	x:=y|| y:=x swaps the values of x and y (the value of y is assigned to x and simultaneously the old value of x is assigned to y)
s←e	*Add at tail (sequence).* Adds the value *e* to the end of the sequence *s*.	(1,2,3) ← 4 produces (1,2,3,4)
s↓n	*Restrict at tail (sequence).* Removes the *n* first elements from the sequence *s*.	(1,2,3,4) ↓ 2 produces (3,4)

{e}⩤R	*Domain subtraction.* Removes the pairs from the relation *R*, where the first component of the pairs matches the value *e*.	{A}⩤{(B,1),(A,2),(C,3),(A,4)}
		produces
		{(B,1),(C,3)}

The functionality of the machine is specified in a programming-like language in the OPERATIONS clause. It is a sequence of *event* definitions. An event definition is akin to a procedure or function definition found in general purpose programming languages. An event has a name and may have parameters. Each definition describes how the state of the abstract machine must change in response to the occurrence of the event. The idea is that the machine receives a sequence of events from the outer world and changes its state in response to each received event according to the given event definitions. The body of events is expressed using B *AMN* substitutions (generalised assignments). A list of the substitutions used in this paper is given in Table 1.

The invariant properties of the machine are specified in the INVARIANT clause. These properties are, essentially, logical conditions specifying allowed combinations of values in the state variables.

The B Method allows hierarchical composition of abstract machines, so that more complex abstract machines can be built by combining simpler state machines. The B Method has well refined rules that extend the user defined invariant properties of the composite state machine with additional properties that preserve the invariant properties of the component state machines. A good introduction to the B Method can be found in e.g. (Schneider, 2001).

Verifying consistency properties for digital evidence using the B Method

One of the main benefits of constructing a formal B model is that it allows us to rigorously verify the correctness of consistency properties for the digital evidence. This is achieved by (1) formulating consistency properties of the model in the INVARIANT clause and then (2) verifying the *validity* of the B model. *Validity* here means that any state reachable by the B model satisfies the INVARIANT clause. The B method provides a rigorous method for proving the *validity* of the model. That method is supported by a number of automated tools, such as Atelier-B and ProB.

What does validity of the B model mean from a forensics point of view? To answer this question, let us analyse the logical meaning of the term "inconsistent evidential data" and how it relates to the B model.

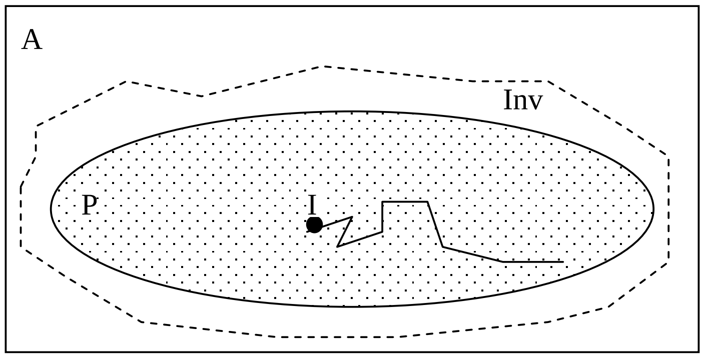
In digital forensics, an investigator considers evidential data "consistent" when it agrees with (1) the investigator's understanding of how the evidential device works, and (2) with the investigator's hypothesis of the incident. If either of the two is violated, the evidential data is considered by the investigator as "inconsistent". Thus, evidential data can be inconsistent in two senses:

(1) Evidential data may contradict the way digital device works, and/or
(2) It may contradict the hypothesis of the incident.

In a way, the second case is weaker than the first case, because if the evidential data is inconsistent in the first case, it is also inconsistent in the second case. However, the inverse is not always true. This paper discusses evidential data inconsistencies only of the first type – those that contradict the way the device works.

The investigator collects data from the evidential device and tries to see if it is consistent. Under the above restriction, evidential data is consistent if it corresponds to a "consistent" (i.e. reachable) state of the device's volatile memory, registers, and non-volatile data storage. To see the distinction between consistent and inconsistent states, consider Fig. 1 below. The big rectangle "A" in Fig. 1 represents *all* possible combinations of bits in the device's memory and data storage (some of these combinations may never arise during the device's operation).

Figure 1 - Illustration of B model validity. P is a set of model states reachable from the initial state I. Inv is the set of states that obey INVARIANT clause. B model is valid if P is contained in Inv.



The small circle labelled "I" represents the initial state of the device (the content of its memory, registers, and data storage when the device is powered up). The curved line that starts at "I" represents an example "path" of the device through the state space as it switches from state to state during operation. The oval "P" represents all "consistent" states – that is, states that may be reached by the device during its operation. All states outside oval P are, therefore, "inconsistent", because they cannot be reached by the device during its normal operation.

The dashed-line polygon "Inv" that surrounds the oval P represents all states that satisfy the INVARIANT properties. In the B Method, the model is *valid*, if for any state reachable from the initial state of the model (i.e. for all states in the oval P), the specified INVARIANT properties hold. In other words, the B method verifies only that the states in P possess the INVARIANT properties, and no claim is made about the states outside of P – they may or may not possess the INVARIANT properties.

To better see the forensic implications of this definition, the definition of B model validity can be re-phrased according to a well known Boolean tautology (*contrapositive*) $P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$ as follows: In a valid B model, *if the invariant property does not hold for a given state, the state cannot be reached from the initial state*. That is:

1. if the evidential data does not satisfy the INVARIANT properties of the B model, it represents a state that cannot be reached from the initial state of the model and, therefore, such evidence is inconsistent (in Fig. 1 such a state corresponds to a point outside of "Inv");
2. if the evidence does satisfy the INVARIANT properties, nothing can be inferred about consistency or inconsistency of the evidence. In Fig. 1 this case corresponds to a point inside "Inv", which may or may not be inside the oval P.

Recall that our method of inconsistency check verification is as follows: (a) create a B model of the device being forensically analysed; (b) add evidence consistency properties into the INVARIANT clause of the created B model, and (c) check the resulting B model is still valid. Once all of the steps are successfully performed we know that

(a) The inconsistency check is *correct*. Any inconsistency that it finds is indeed, an inconsistency.
(b) However, the inconsistency check may be *incomplete*. It may miss some evidence inconsistencies. By strengthening the invariant of our formal model (i.e., by adding more properties) we can make the model more complete.

Note, that the incompleteness of the inconsistency check is not really a problem. Many of the existing digital forensic tools are incomplete in the above sense. Indeed, it is rather presumptuous to expect a digital forensic tool to find *all* possible inconsistencies in the digital evidence. Much more important is the fact that the verified inconsistency check is correct, and may be safely used in the forensic analysis process.

Tools supporting the B Method

There are several tools that automate checking of invariant properties of the abstract state machines. Two such tools – ProB (Leuschel & Butler, 2003) and Atelier-B (Steria, 2001) – were used in this work.

ProB is an integrated development environment (IDE) for the B Method. It provides a text editor, a syntax and type checker for verifying syntactical and type correctness of abstract state machine specifications, temporal and constraint-based model checkers, and an abstract state machine simulator.

The ProB tests invariant properties of the abstract state machine by subjecting the machine to a random sequence of events and verifying that the specified invariant properties hold after every event. Multiple random sequences can be employed to make testing more thorough. Obviously, ProB testing is not exhaustive.

Atelier-B is another integrated development environment for the B Method. In addition to syntax and type checking, it can verify invariant properties of abstract state machines using semi-automated theorem proving. The advantage of verification by theorem proving is that verification is exhaustive. This works as follows.

First, several so-called “proof-obligations” are generated. These proof-obligations are logical formulae that need to be “discharged” (i.e. mathematically proved to be true) to guarantee that the invariant properties always hold. Atelier-B (Steria, 2001) can generate these proof-obligations automatically from a B specification. Atelier-B can then assist with the discharging of them using the built-in automatic and interactive provers.

Unlike ProB, Atelier-B can guarantee that invariant properties always hold in the model. This ability, however, comes at a price. The random testing offered by ProB is completely automatic and requires minimal user involvement. The theorem proving, however, is rarely fully automatic. User input is occasionally required to suggest non-trivial proof-rules. The costs in terms of human labour required to figure a correct proof may be quite high (in the order of days).

It is important to note that the B Method offers several different verification techniques that provide a trade-off between the degree of automation and the degree of assurance in the correctness of the invariant properties. This allows the digital forensic analyst to choose which verification technique to use, depending on the importance of the case and the stage of model development. In particular, we found that ProB is very useful at the initial stages of formal development as it quickly finds major problems in the formal model. Once the development of the formal model is complete, higher degree of assurance can be achieved using theorem proving.

Implementing inconsistency checks using SQL

The formalised consistency properties of a B model (the INVARIANT clause) are written in the first-order logic extended with elements of the set theory. Structured Query Language (SQL) is a popular database manipulation language, which is based on the same theoretical basis.

To see how inconsistency checks can be implemented using SQL, observe first that every invariant property (consistency criterion) is of the form

$$\forall(x, y, z, \ldots).\ (condition_1 \wedge condition_2 \wedge \ldots \wedge condition_n \Rightarrow test) \qquad (\dagger)$$

In plain English, this property means that logical condition *test* must hold for all combinations of values of variables x, y, z, … that satisfy logical conditions $condition_1$, $condition_2$, … $condition_n$. To find violations of this property we need to search for combinations of values of x, y, z, … that satisfy logical conditions $condition_1$, $condition_2$, … $condition_n$, but DO NOT satisfy the *test*.

Observe also that the values of variables are not arbitrary. One of the purposes of conditions $condition_1$, $condition_2$, … $condition_n$ is to specify the type and source of values for variables used in the formula. For example, the condition

$$mtime \in TIMESTAMP$$

means that the variable *mtime* must hold a date/time value.

Therefore, to find violations of property (†) using SQL we must (1) import all sources of values for variables x, y, z, … into a relational database as tables and (2) search for combinations of values that satisfy logical conditions $condition_1$, $condition_2$, … $condition_n$, but do not satisfy the *test* condition. The second step can be performed by a SELECT statement of the following general form:

```
SELECT * FROM table_1 , table_2 , … , table_m
WHERE
  condition_1 AND
  condition_2 AND
  …
  condition_n AND
  NOT test ;
```

Note that conditions that specify type and source of values for variables do not need to be included into this SELECT statement, because such conditions are implicitly satisfied by construction of database tables and by semantics of the SELECT statement.

Note also that when some of the conditions $condition_1$, $condition_2$, … $condition_n$ are of the form (†) themselves, they can be implemented using nested SELECT statements. In some cases, the use of nested SELECT statements can be replaced by appropriate use of aggregate functions, such as MAX and MIN, in combination with GROUP BY clause.

Finally, instead of using a single complex SELECT statement, an inconsistency check can be implemented by a sequence of simpler SELECT statements, where the output of one SELECT statement is stored in an SQL *view* and processed by subsequent SELECT statements.

Section 3: Development of an inconsistency-checking tool for FTP servers using the B Method

Illegal file sharing using FTP servers

Uploading and downloading files on an FTP server is a very common way of file sharing. Most operating systems have built-in FTP client software and setting-up an FTP server is relatively uncomplicated. The access to an FTP server can be restricted by user name and passwords. It makes FTP servers an attractive tool for sharing illegal file material.

When a computer containing an FTP server with illegal material is seized during an investigation, it is important to establish who is responsible for placing the illegal material on the server. This may not be easy, because although it could be a user of the computer, it is not uncommon to find poorly configured FTP servers exploited by hackers and used by them to store illegal material (AusCERT, 2002). Nevertheless, the FTP server does offer many clues to the identity of the perpetrator. Some of the clues may be quite obvious, such as the user ID of the owner of file with illegal content. Other clues are less obvious.

One such clue is obtained by checking timestamps on entries in the FTP log file and comparing them against file timestamps in the FTP repository. Intuitively, we would expect that

if some file is uploaded onto the FTP server, the timestamp (†)

of that file in the FTP repository would be the same as the

timestamp on the upload event in the FTP log file.

A violation of this check seems to indicate that the file was created by writing directly into the FTP repository, bypassing the FTP server.

As shown in the rest of this paper, this intuitive perception is not always correct. As explained in the following sub-sections, the creation of a formal model of an FTP server using the B Method helped us to clarify validity conditions for that check, and to discover other related checks, which were subsequently implemented in an inconsistency checking tool.

Overview of the FTP server model

As the first step in the formal development of the inconsistency checking tool, a formal model of an FTP server was developed. It is schematically depicted in Figure 2. Since we are only interested in the relationship between timestamps on files and on log file entries, a number of assumptions have been made to simplify the formal model. These are as follows:

- Only the components relevant to the making and storage of timestamps are modelled. This includes: the FTP file repository (called the fileStore), the system clock, and the FTP log file.

- *Files have numerical IDs (slot numbers) instead of textual names*, because we only use file names to distinguish files from each other. The name of a file has no effect on its timestamps.

- *There are no sub-directories in the FTP repository model*, because the place of a particular file in the file tree hierarchy does not affect its modification timestamp[3].

- *Deleted files are left outside of the model scope*. To formulate and analyse consistency properties described in this paper the evidence available in unallocated file entries was not necessary. For the sake of formal simplicity, such evidence has been left outside of the model scope. Note, however, that such evidence can be easily added into the model, if it is required for proving some additional consistency property.

- *Only the last modification time (mtime) of every file is modelled*. Again, only the last modification time was needed for formulating and reasoning about the consistency properties described in this paper, and for the sake of formal simplicity all other timestamps have been left outside of the model scope.

According to the above assumptions, the FTP file repository is modelled as an array of file entries, where each entry stores information about one file attribute, namely the last modification time (*mtime*). The elements of the array are referred-to by their unique file IDs called slot numbers or *slots*. Note that it is easy to extend this model to hold other file attributes as well.
A clock counter that is continuously updated models the progress of time, and we model the modification of a file in the file system by assigning the current clock value to the mtime attribute of the file.

[3] This may not be the case for sub-directories, but we exclude them from consideration.

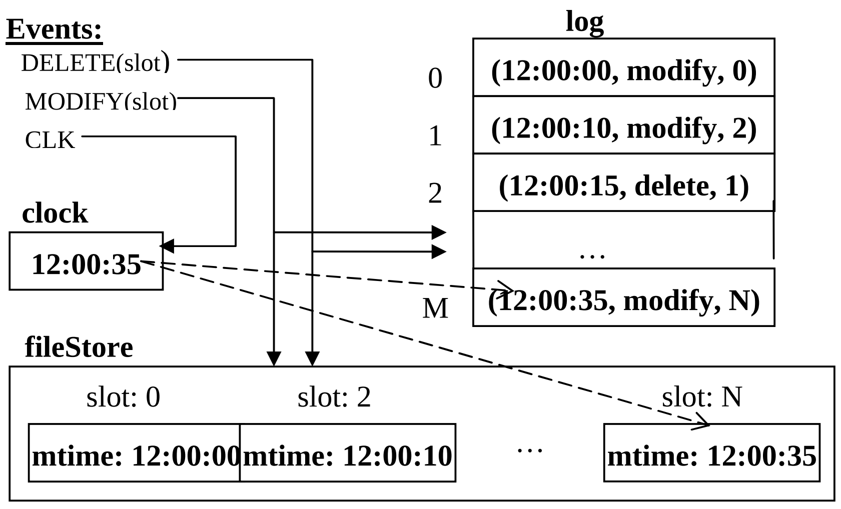


Figure 2 - Main elements of the FTP server model.

The FTP log file is modelled as a list of file modification events, where each log entry is represented as a triple of the form *(TStamp,Event,Slot). TStamp* corresponds to the time the event has occurred. *Event* identifies the type of event, and this can be either *MODIFY* or *DELETE*, corresponding to file uploading and file deletion events, respectively. *Slot* identifies at which position in the file system a file should be modified/deleted in the case of a file modification/deletion event.

Next, we formalised the consistency properties for our model (i.e. the relations between timestamps in the file store and timestamps in the log file). The first of these properties corresponds to the informal consistency property (†) stated at the end of the previous section. This was expressed as a logical formula in terms of our model, whose exact meaning in plain English is as follows:

> *Consistency property 0. If the last MODIFY event (in the log file) for a file X is not followed by a DELETE event, the timestamp of the last MODIFY event in the log, should be equal to the timestamp on the file X.*

During model creation, we realised that a number of other consistency properties should hold in the model. These were also formalised and included into the INVARIANT clause. These informal meaning of these additional properties is as follows:

> *Consistency property 1. The order of timestamps reflects the order of MODIFY events that produced these timestamps. That is, given two timestamps, a timestamps with the greater value corresponds to the MODIFY event which occurred later.*

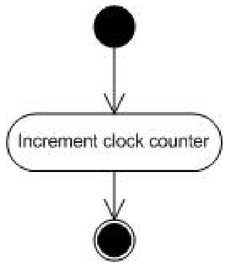
> <u>Consistency property 2.</u> *If the last MODIFY event for a given file has a corresponding DELETE event following it, then the file should not be present in the file store; and vice versa – if there is a file present in the file store, then the last event for that file must not be a DELETE event (it must be a MODIFY event).*

Overview of the formal model

We have used the B Method (Abrial, 1996) to express the abstract *FTPServer* model formally. Using B, we specify the system state by introducing *state variables (*introduced in the VARIABLES clause), and we model state transitions by means of *operations (*introduced in the OPERATIONS clause). The requirements (properties) that we want to verify to hold for our model are specified in the INVARIANT clause. In our case, these properties are the three consistency conditions for modification times presented in the previous subsection. We have used the *Event-B* (Métayer, Abrial & Voisin, 2005) based approach, in which operations are viewed as events triggered by the external environment (e.g., an application program modifying a file). Events are specified as *guarded substitutions* in Event-B. These guarded substitutions are of the form SELECT P THEN S END, or ANY params WHERE P THEN S END in the case of parameterised events. The semantics of the guarded substitutions is that an event is in the waiting (hibernating) state until the guard *P* becomes true. When the guard *P* evaluates to true, the substitutions *S* are performed. The complete B FTPServer model can be found in the appendix of this paper. Next, we describe how the FTPServer events are specified in our formal model. Intuitive graphical representations of the events (using UML activity diagrams) are given next to the event definitions.

The *clock* event models the time progress in our system, and it is specified by an assignment to the *clk* variable (incrementing the current value of *clk*), as shown below.

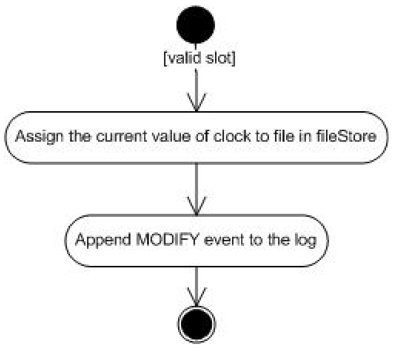
```
clock =
BEGIN
        clk := clk + 1
END;
```



The *clock* event is enabled at all times (its guard is simply *true*), to model the *continuous* progress of time.

The file *modify* event in our model is specified using the ANY guarded substitution for introducing the local parameter *slot*, which identifies the position in the file system (modelled by the state variable *fileStore*) where the file to be modified is located. When the *modify* event is enabled, the action specified by our abstract model is to assign the current value of the clock counter (modelled by the variable *clk*) to position *slot* in the *fileStore*, as shown in the below extract.

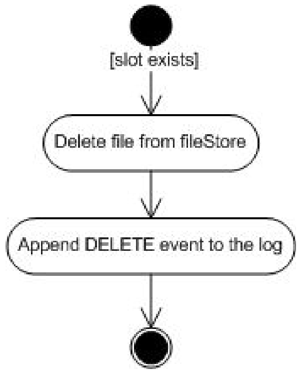
```
ANY slot WHERE $slot \in FILESLOT$
THEN
    $\mathbf{fileStore}(slot) := clk \parallel$
    $log := log \leftarrow (clk, MODIFY, slot)$
END;
```


[valid slot]
Assign the current value of clock to file in fileStore
Append MODIFY event to the log

Simultaneously, we append an entry *(clk,MODIFY,slot)* to the end of the transaction log, to identify which slot has been affected by the modify event. We have modelled the transaction log using a sequence variable *log*.

In addition to modelling modification of files, we also model the deletion of files in our file store by the *delete* event. The body of the *delete* event is shown below.

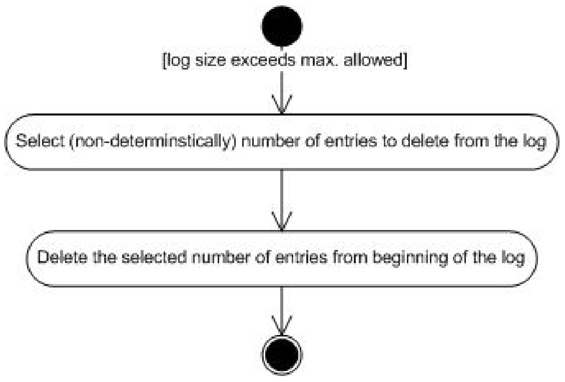
```
ANY slot WHERE $slot \in FILESLOT$
$\wedge\ slot \in \mathbf{dom}(fileStore)$
THEN
    $fileStore := \{slot\} \lhd fileStore \parallel$
    $log := log \leftarrow (clk, DELETE, slot)$
END
```



The *delete* event is enabled when some slot exists in the file store (i.e., there exists some modification time record for the slot). The body of the event is specified using the domain subtraction substitution, which has the effect of removing the modification time record for the slot in question from the file store. Similarly to the *modify* event, an entry of the form *(clk,DELETE,slot)* is appended to the end of the transaction log, to indicate that a deletion event for the specified *slot* occurred at time *clk*.

It is possible that a log might be truncated (or completely cleared) when a certain limit has been reached. This happens, for example, when the server administrator erases old log files. Our specification takes this possibility into account by introduction of the *trunc_log* event. The *trunc_log* event becomes enabled when the current size of the transaction log exceeds the maximum size allowed (modelled by the constant *max_log_size*). The body of *trunc_log* is specified as follows:

```
ANY $tval$ WHERE $tval \in \mathbb{N}_1$
        $\wedge\ tval \leq max\_log\_size$
THEN
    $log := log \downarrow tval$
END
```


[log size exceeds max. allowed]
Select (non-determinstically) number of entries to delete from the log
Delete the selected number of entries from beginning of the log

It non-deterministically selects a positive natural number *tval* less than the maximum allowed log size (*max_log_size)*. The substitution *log* := *log*□*tval* then drops the first *tval* elements from the beginning of the *log* sequence. Removal of elements from the log results in the loss of information about the past operation of the system. After truncation there may be files in the file store that no longer have associated entries in the log. The formal model takes this into account by requiring the formulated invariant properties to hold *only* for those slots that are (1) present in the fileStore and (2) have associated entries in the log.

The consistency properties that should hold in our model are then expressed as part of the state invariant. As an example, the first part of consistency property 1 (see previous section) is introduced into the invariant in the following way:

$$\forall(slotX,slotY).(slotX \in \mathbf{dom}(fileStore) \wedge slotY \in \mathbf{dom}(fileStore) \wedge (slotX \neq slotY) \wedge$$
$$LAST\_INDEX(fileStore(slotX),MODIFY,slotX) < LAST\_INDEX(fileStore(slotY),MODIFY,slotY)$$
$$\Rightarrow \quad fileStore(slotX) \leq fileStore(slotY))$$

Universal quantification (is used to state that the property should hold for *all* distinct values *slotX* and *slotY* in the *fileStore*. For such slots, in case the last occurrence of a modify event for *slotX* is preceding the last occurrence of a modify event for *slotY*, it needs to be the case (logical implication) that the modification timestamp of *slotX* in the *fileStore* is less than or equal to the modification timestamp of *slotY*. The *LAST_INDEX* definition returns the index position for the last occurrence of an event (here, modify) for a given slot in the log. The other consistency properties for FTP are formulated in a similar way.

We have used the ProB (Leuschel & Butler, 2003) tool (the animator and the temporal model checker) for the initial validation of the B formal model, and the automatic and interactive provers of Atelier-B (Steria, 2001) have been used to discharge the associated consistency proof obligations.

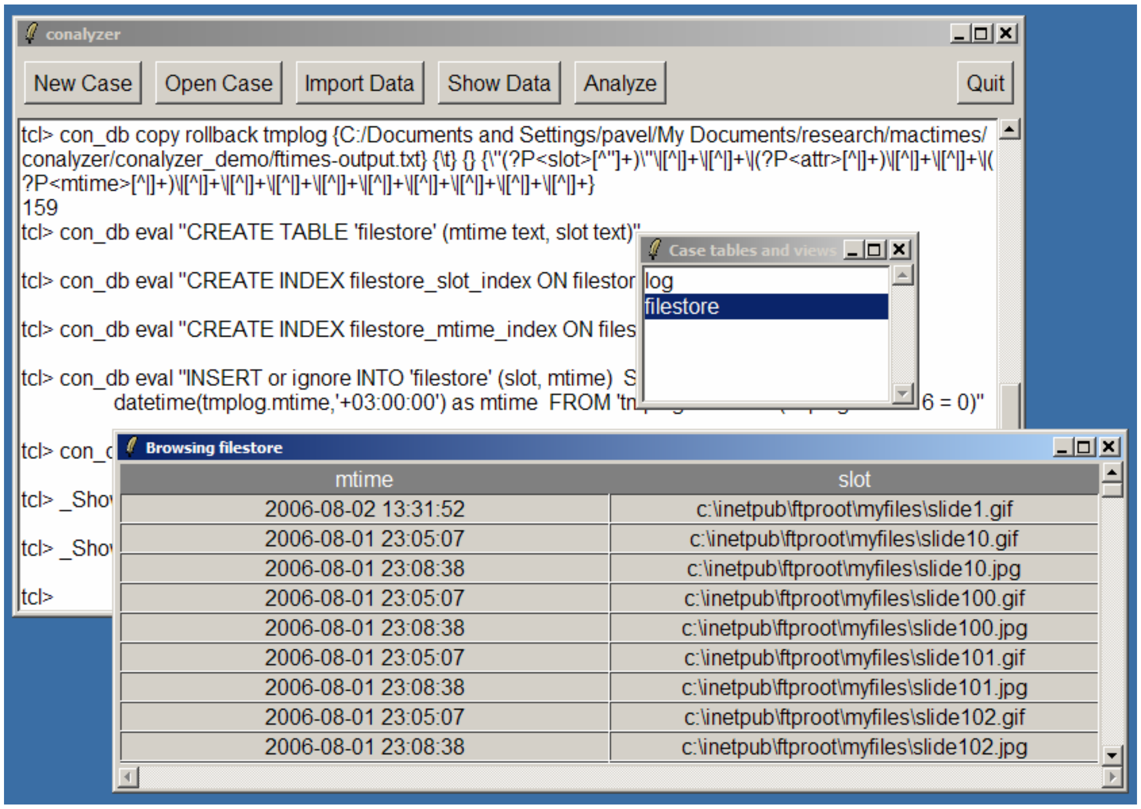
ConAlyzer: an inconsistency checker for FTP server logs

Apart from rigorous verification of consistency properties, another major benefit of using the B Method turned out to be the ease of implementing the formalised inconsistency checker in software. Having formally verified consistency properties of

FTP servers, we found that these properties could be easily converted into a handful of SQL statements searching evidential data for violations of these properties.

Initially, we considered the use of the Microsoft LogParser toolkit (Microsoft, 2005) as the basis of our inconsistency checker. The LogParser toolkit is an incident analysis tool that can extract information from many types of operating system objects (logs, file systems, etc.) and filter this information using SQL queries. Unfortunately, the subset of SQL supported by the current implementation of the LogParser toolkit is rather limited. In particular, it cannot perform SQL queries that relate (join) several different data sources. This was perceived as a major limitation from consistency checking point of view, because much of unobvious inconsistencies are found by comparing information contained in different sources of evidence.

In the end it was decided to use an open sourced, embeddable database engine SQLite[4] as the basis for our FTP server inconsistency checking tool, which has been called “ConAlyzer” (Consistency Analyzer). The default data import capabilities of SQLite have been expanded with data carving functionality (based on regular expressions), and the ability to execute third party utilities. ConAlyzer GUI is written in Tcl/Tk. A screenshot of the GUI is shown in Figure 3.



```
tcl> con_db copy rollback tmplog {C:/Documents and Settings/pavel/My Documents/research/mactimes/
conalyzer/conalyzer_demo/ftimes-output.txt} {\t} {} {\"(?P<slot>[^"]+)\"\|[^|]+\|[^|]+\|(?P<attr>[^|]+)\|[^|]+\|[^|]+\|(
?P<mtime>[^|]+)\|[^|]+\|[^|]+\|[^|]+\|[^|]+\|[^|]+\|[^|]+\|[^|]+\|[^|]+\|[^|]+}
159
tcl> con_db eval "CREATE TABLE 'filestore' (mtime text, slot text)"

tcl> con_db eval "CREATE INDEX filestore_slot_index ON filestor

tcl> con_db eval "CREATE INDEX filestore_mtime_index ON files

tcl> con_db eval "INSERT or ignore INTO 'filestore' (slot, mtime)  S
          datetime(tmplog.mtime,'+03:00:00') as mtime  FROM 'tn                          6 = 0)"

tcl> con_d

tcl> _Sho

tcl> _Sho

tcl>
```

mtime	slot
2006-08-02 13:31:52	c:\inetpub\ftproot\myfiles\slide1.gif
2006-08-01 23:05:07	c:\inetpub\ftproot\myfiles\slide10.gif
2006-08-01 23:08:38	c:\inetpub\ftproot\myfiles\slide10.jpg
2006-08-01 23:05:07	c:\inetpub\ftproot\myfiles\slide100.gif
2006-08-01 23:08:38	c:\inetpub\ftproot\myfiles\slide100.jpg
2006-08-01 23:05:07	c:\inetpub\ftproot\myfiles\slide101.gif
2006-08-01 23:08:38	c:\inetpub\ftproot\myfiles\slide101.jpg
2006-08-01 23:05:07	c:\inetpub\ftproot\myfiles\slide102.gif
2006-08-01 23:08:38	c:\inetpub\ftproot\myfiles\slide102.jpg

Figure 3 – ConAlyzer GUI.

[4] http://www.sqlite.org

ConAlyzer input consists of two files: the FTP server log file, and the file with a list of file timestamps. ConAlyzer creates two tables (fileStore and log) and imports the content of both files into these tables. After that, ConAlyzer performs consistency checks on the information contained in the tables.

In our experiments we copied the FTP log file from the live system and used the grave-robber tool[5] to obtain file timestamp information about the files in the FTP file repository. Although we performed a "live" analysis of the system, there same kind of information can be obtained from disk images using a variety of digital forensics tools – both commercial (EnCase, FTK) and open source (The Sleuth Kit's icat, fls/ils).

ConAlyzer is not yet publicly released, but if you are interested in experimenting with it, please contact the authors directly.

Section 4: Conclusions

In this paper, the application of the B Method for the analysis of correctness of digital evidence inconsistency checks has been explored. The following positive aspects have been identified:

1. The use of formal methods facilitates rigorous analysis of inconsistency checks.
2. The need to develop a formal model helps the investigator to better understand the system.
3. Formal inconsistency checks can be converted into succinct SQL statements.
4. An SQL-based tool supporting the automatic analysis of inconsistency checks has been developed.

On the negative side, formal analysis requires considerable time to develop a formal model and to formalise properties. The not-so-mathematical syntax of B helps in this process. Formal development would be justified, when correctness of analysis is crucial. This could be forensic analysis in high-profile cases, and development of commercial digital forensics tools.

The application of formal methods to the analysis of digital evidence is relatively new, and therefore it does not exist much related work in the area. Stephenson (2003) presents an approach to the formal modelling of root cause analysis, for which he employs coloured Petri nets. The idea of the root cause analysis is to trace the origin (root cause) of an incident in order to be able to prevent similar incidents from occurring in the future. The use of formal modelling helps to bring more confidence into such analysis. Leigland and Krings (2004) explore a formalisation of digital forensics.

[5] http://www.fish2.com/tct/

Our future research will concentrate on extending the formal analysis to other areas of digital forensics, and on the application of the formal approach to the analysis of other types of forensic artefacts.

© Copyright 2007 International Journal of Digital Evidence

About the Authors

Dr. Pavel Gladyshev is a lecturer in the School of Computer Science and Informatics at University College Dublin, where he manages the MSc programme in Forensic Computing and Cybercrime Investigation. His research interests are in the area of information systems security and digital forensics. Before joining UCD as a lecturer, Dr. Gladyshev worked as a senior consultant in information systems. Dr. Gladyshev holds PhD and MSc degrees in Computer Science and a primary degree in computer engineering. pavel.gladychev@ucd.ie

Andreas Enbacka is conducting research in the area of theoretical digital forensics. His MSc thesis work in Computer Science was on formal methods based approaches to digital forensics. Mr. Enbacka is a member of Formal Methods Europe (FME), and he also works as a software engineer. aenbacka@abo.fi

References

Abrial J.-R. (1996) *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Australian Computer Emergency Response Team (AusCERT) (2002) Alert AA-2002.03 *File-Sharing Activity Part 2 of 2 - Increased intruder attacks against servers to expand illegal file sharing networks*. Retrieved March 19, 2007, from http://www.auscert.org.au/render.html?it=2229

Behm P., et al. (1999) *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. Springer Verlag.

Leigland R., Krings A. W. (2004) *A Formalization of Digital Forensics*. *International Journal of Digital Evidence*, 3(2).

Leuschel M., Butler M. (2003) *ProB: A Model Checker for B*. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli (Eds.), *Proceedings of FME 2003*, Pisa, Italy, pp. 855–874. LNCS 2805, Springer Verlag.

Métayer C., Abrial J.-R., Voisin L. (2005) *Event-B Language*, RODIN (Rigorous Open Development Environment for Complex Systems) Project IST-511599 Deliverable 3.2. Retrieved March 19, 2007, from http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf

Microsoft (2005) *The LogParser toolkit.* Retrieved March 19, 2007, from http://www.microsoft.com/technet/scriptcenter/tools/logparser/

Morgan C. (1994) Programming from Specifications (2nd Ed.). Prentice Hall.

Schneider S. (2001) *The B-Method: An Introduction*. Palgrave.

Stephenson P. (2003) *Formal Modeling of Post-Incident Root Cause Analysis. International Journal of Digital Evidence*, 2(2).

Steria (2001) *Atelier B. User and Reference Manuals*, Aix-en-Provence, France. Retrieved March 19, 2007, from http://www.atelierb.societe.com/index_uk.html

Appendix: The formal B FTPServer specification

MACHINE *FTPServer*

DEFINITIONS
$MODIFY == 1;$
$DELETE == 2;$
$FILESLOT == \mathbb{N};$
$TIMESTAMP == \mathbb{N};$
$EVENT == \mathbb{N};$
$LAST\_INDEX(X,Y,Z) == \mathbf{max}(\mathbf{dom}(log \rhd \{(X \alpha Y \alpha Z)\}))$

CONSTANTS *max_log_size*

PROPERTIES $max\_log\_size \in \mathbb{N}_1$

VARIABLES *fileStore, clk, log*

INVARIANT

$fileStore \in FILESLOT \mapsto TIMESTAMP \wedge$
$clk \in \mathbb{N} \wedge$
$log \in \mathbf{seq}(TIMESTAMP \times EVENT \times FILESLOT) \wedge$

/ Property 0 */*

$(\forall(slot,mtime,dtime).(slot \in TIMESTAMP \wedge mtime \in TIMESTAMP \wedge dtime \in TIMESTAMP \wedge (mtime,MODIFY,slot) \in \mathbf{ran}(log) \wedge$
$\forall otime.(otime \in TIMESTAMP \wedge (otime,MODIFY,slot) \in \mathbf{ran}(log) \Rightarrow mtime \geq otime) \wedge$
$dtime > mtime \wedge (dtime,DELETE,slot) \notin \mathbf{ran}(log)$
$\Rightarrow fileStore(slot) = mtime)) \wedge$

/ Property 1 */*

$(\forall(slotX,slotY).((slotX \in \mathbf{dom}(fileStore) \wedge (fileStore(slotX),MODIFY,slotX) \in \mathbf{ran}(log) \wedge slotY \in \mathbf{dom}(fileStore) \wedge (fileStore(slotY),MODIFY,slotY) \in \mathbf{ran}(log) \wedge (slotX \neq slotY)) \Rightarrow$
$(LAST\_INDEX(fileStore(slotX),MODIFY,slotX) < LAST\_INDEX(fileStore(slotY),MODIFY,slotY)$
$\Rightarrow fileStore(slotX) \leq fileStore(slotY)))) \wedge$

$(\forall(slotX,slotY).((slotX \in \mathbf{dom}(fileStore) \wedge (fileStore(slotX),MODIFY,slotX) \in \mathbf{ran}(log) \wedge slotY \in \mathbf{dom}(fileStore) \wedge (fileStore(slotY),MODIFY,slotY) \in \mathbf{ran}(log) \wedge (slotX \neq slotY)) \Rightarrow$
$(fileStore(slotX) < fileStore(slotY)$
$\Rightarrow (LAST\_INDEX(fileStore(slotX),MODIFY,slotX) <$
$LAST\_INDEX(fileStore(slotY),MODIFY,slotY))))) \wedge$

/ Property 2 */*

$\forall(slot,mtime,dtime).(slot \in TIMESTAMP \wedge mtime \in TIMESTAMP \wedge dtime \in TIMESTAMP \wedge (mtime,MODIFY,slot) \in \mathbf{ran}(log) \wedge (dtime,DELETE,slot) \in \mathbf{ran}(log) \wedge dtime > mtime \wedge$
$LAST\_INDEX(mtime,MODIFY,slot) < LAST\_INDEX(dtime,DELETE,slot) \wedge \forall otime.(otime \in TIMESTAMP \wedge otime \geq dtime \Rightarrow (otime,MODIFY,slot) \notin \mathbf{ran}(log))$
$\Rightarrow slot \notin \mathbf{dom}(fileStore))$

```
INITIALISATION fileStore := ∅ || clk := 0 || log := [ ]  /* initial state of the model */

OPERATIONS  /* events of the model */

clock =
    BEGIN
       clk := clk + 1
    END;

trunc_log =
 SELECT size(log) ≥ max_log_size
 THEN
   ANY tval WHERE tval ∈  _4_s20 N1  ∧ tval ≤ max_log_size
   THEN
      log := log↓tval
   END
 END;


modify =
   ANY slot WHERE slot ∈ FILESLOT
   THEN
     fileStore(slot) := clk ||
     log := log ← (clkαMODIFYαslot)
   END;

delete =
  ANY slot WHERE slot ∈ FILESLOT ∧ slot ∈ dom(fileStore)
  THEN
     fileStore := {slot}⩤fileStore ||
     log := log ← (clkαDELETEαslot)
  END

END
```