Reasoning About a Simulated Printer Case Investigation with Forensic Lucid

Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi

Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada
{mokhov,paquet,debbabi}@encs.concordia.ca

Abstract. In this work we model the ACME (a fictitious company name) "printer case incident" and make its specification in Forensic Lucid, a Lucid- and intensional-logic-based programming language for cyberforensic analysis and event reconstruction specification. The printer case involves a dispute between two parties that was previously solved using the finite-state automata (FSA) approach, and is now re-done in a more usable way in Forensic Lucid. Our approach is based on the said case modeling by encoding concepts like evidence and the related witness accounts as an evidential statement context in a Forensic Lucid "program". The evidential statement is an input to the transition function that models the possible deductions in the case. We then invoke the transition function (actually its reverse) with the evidential statement context to see if the evidence we encoded agrees with one's claims and then attempt to reconstruct the sequence of events that may explain the claim or disprove it.

Keywords: Forensic Lucid, cybercrime investigation modeling, intensional logic and programming, cyberforensics, finite-state automata.

1 Introduction

1.1 Problem Statement

The very first formal approach to cyberforensic analysis and event reconstruction appeared in two papers [1,2] by Gladyshev et al. that relies on the finite-state automata (FSA) and their transformation and operation to model evidence, witnesses, stories told by witnesses, and their possible evaluation for the purposes of claim validation and event reconstruction. One of the examples the papers present is the use-case for the proposed technique – the "ACME Printer Case Investigation". See [1] for the corresponding formalization using the FSA by Gladyshev and the proof-of-concept LISP implementation. We aim at the same case to model and implement it using Forensic Lucid, which paves a way to be more friendly and usable in the actual investigator's work and serve as a basis to further development in the area.

P. Gladyshev and M.K. Rogers (Eds.): ICDF2C 2011, LNICST 88, pp. 282–296, 2012.

1.2 Proposed Solution

We show the intensional approach to the problem is an asset in the field of cyberforensics as it is promising to be more practical and usable than the plain FSA and LISP. Since Lucid was originally designed and used to prove correctness of programming languages [3,4,5,6], and is based on the temporal logic, functional and data-flow languages its implementation to backtracking in proving or disproving the evidential statements and claims in the investigation process as a evaluation of an expression that either evaluates to *true* or *false* given all the facts in the formally specified context.

Intensional Logic

From the logic perspective, it was shown one can model computations (the basic unit in the finite state machines in [1,2]) as logic [7]. When armed with contexts as first-class values and a demand-driven model adopted in the implementation of the Lucid-family of languages [8,9,10] that constrains the scope of evaluation in a given set of dimensions, we come to the intensional logic and the corresponding programming artifact. In the essence, we model our forensic computation unit in the intensional logic and implement it in practice within an intensional programming platform [11]. We project a lot of potential for this work to be successful, beneficial, and usable for cyberforensics investigation and intensional programming communities.

Approach Overview

Based on the parameters and terms defined in the works of Gladyshev [1,2], we have various pieces of evidence and witnesses telling their own "stories" of an incident. The goal is to put them together to make the description of the incident as precise as possible. To show that a certain claim may be true, the investigator has to show that there are some explanations of evidence that agree with the claim. To disprove the claim, the investigator has to show there is no explanation of evidence that agree with the claim [1].

The authors of the FSA approach did a proof-of-concept implementation of the proposed algorithms in CMU Common LISP [1] that we target to improve the usability of by re-writing it in a Lucid dialect, that we call Forensic Lucid (with a near-future possibility to construct a data-flow graph-based [12,13] IDE for the investigator to use and train novice investigators as an expert system).

In this particular work we focus on the specification of the mentioned sample investigation case in Forensic Lucid while illustrating relates fundamental concepts, operators, and application of context-oriented case modeling and evaluation. Common LISP, unlike Lucid, entirely lacks contexts build into its logic, syntax, and semantics, thereby making the implementation of the cases more clumsy and inefficient (i.e. highly sequential). Our system [8,14] offers distributed demand-driven evaluation of Lucid programs in a more efficient way and is more general than LISP's compiler and run-time environment.

2 Background and Related Work

To remain stand-alone and self-sufficient in this work we recite some material in part that we extend from, or, deemed otherwise relevant works, such as previously presented posters, works-in-progress, and conference papers [15,16,17,18,19] and other related cited works for the benefit of the readers. An expanded e-print version of this paper (that is also being updated from time to time) with more background information and references therein can be found at `http://arxiv.org/abs/0906.5181`.

2.1 Intensional Logic and Programming

Intensional programming is based on intensional (or, in other words, multidimensional) logics, which, in turn, are based on Natural Language Understanding (aspects, such as, time, belief, situation, direction, etc.). Intensional programming brings in *dimensions* and *context* to programs (e.g. space and time in physics or chemistry). Intensional logic adds dimensions to logical expressions; thus, a non-intensional logic can be seen as a constant or a snapshot in all possible dimensions. *Intensions are dimensions* at which a certain statement is true or false (or has some other than a Boolean value). *Intensional operators* are operators that allow us to navigate within these dimensions [20].

2.2 Lucid Overview

Lucid [3,4,5,6,21] is a dataflow intensional and functional programming language. In fact, it is a family of languages that are built upon intensional logic (which in turn can be understood as a multidimensional generalization of temporal logic) promoting context-aware demand-driven parallel computation model [18]. A program written in some Lucid dialect is an expression that may have subexpressions that need to be evaluated at certain *context*. Given the set of dimensions $D = \{dim_i\}$ in which an expression varies, and a corresponding set of indexes, or, *tags*, defined as placeholders over each dimension, the context is represented as a set of $< dim_i : tag_i >$ mappings. Each variable in Lucid, called often a *stream*, is evaluated in that defined context that may also evolve using context operators [22,23].

The first generic version of Lucid, the General Intensional Programming Language (GIPL) [20], defines two basic operators @ and # to navigate (switch and query) in the context space $\mathcal{P}$. The GIPL is the first[1] generic programming language of all intensional languages, defined by the means of only two intensional operators `@` and `#`. It has been proven that other intensional programming languages of the Lucid family can be translated into the GIPL [20].

[1] The second is Lucx [23], and third is TransLucid [9].

2.3 Forensic Lucid

This section summarizes concepts and considerations in the design of the Forensic Lucid language, large portions of which were studied in the earlier work [18]. The end goal of the language design is to define its constructs to concisely express cyberforensic evidence as a context of evaluations, which can be the initial state of the case (e.g. initial printer state when purchased from the manufacturer, see Section 3), towards what we have actually observed (as corresponding to the final state in the Gladyshev's FSM) (e.g. when an investigator finds the printer with two queue entries ($B_{deleted}, B_{deleted}$)). One of the evaluation engines (a topic of another work) of the implementing system [11] is designed to backtrace intermediate results to provide the corresponding event reconstruction path if it exists. The result of the expression in its basic form is either *true* or *false*, i.e. "guilty" or "not guilty" given the evidential evaluation context per explanation with the backtrace(s). There can be multiple backtraces, that correspond to the explanation of the evidence (or lack thereof) [18].

Language Characteristics

We use Forensic Lucid to model the evidential statements and other expressions representing the evidence and observations as context. An execution trace of a running Forensic Lucid program is designed to expose the possibility of the proposed claim with the events that lead to a conclusion. Forensic Lucid capitalizes its design by aggregating the features of multiple Lucid dialects needed for these tasks along with its own extensions [18].

The addition of the context calculus from Lucx (stands for "Lucid enriched with context" that promotes contexts as first-class values) for operators on simple contexts and context sets (`union`, `intersection`, etc.) are used to manipulate complex hierarchical context spaces in Forensic Lucid. Additionally, Forensic Lucid inherits many of the properties of Objective Lucid and JOOIP (Java-embedded Object-Oriented Intensional Programming language) for the arrays and structural representation of data for modeling the case data structures such as events, observations, and groupings and correlation of the related data, and so on [18]. Hierarchical contexts in Forensic Lucid also follow the example of MARFL [24] using a dot operator and by overloading both @ and # to accept different types as their arguments.

The syntax and the operational semantics of Forensic Lucid were primarily maintained to be compatible with the basic Lucx and GIPL [18]. This helpful (but not absolutely necessary) when complying with the compiler and and the runtime subsystems within the implementing system, the General Intensional Programming System (GIPSY) [11].

Context of Evaluation

Forensic Lucid provides an ability to encode the "stories" told by the evidence and witnesses. This constitutes the primary context of evaluation. The "return value" of the evaluation is a collection of backtraces (may be empty), which

contain the "paths of truth". If a given trace contains all truths values, it's an explanation of a story. If there is no such a path, i.e. the trace, there is no enough supporting evidence of the entire claim to be true [18].

In its simplest form, the context can be expressed as integers or strings, to which we attribute some meaning or description. The context spaces are finite and can be navigated through in all directions of the along dimension indexes, potentially allowing negative tags in our tag sets of dimensions. Our contexts can also be a finite set of symbolic labels and their values that can be internally enumerated [18]. The symbolic approach is naturally more appropriate for humans and we have a machinery to so in Lucx's implementation in GIPSY [22].

We define streams of observations *os* as our fundamental context units, that can be a simple context or a context set. In fact, in Forensic Lucid we are defining higher-level dimensions and lower-level dimensions. The highest-level one is the *evidential statement es*, which is a finite unordered set of observation sequences *os*. The *observation sequence os* is a finite *ordered* set of observations *o*. The *observation o* is an "eyewitness" of a particular property along with the duration of the observation. As in the Gladyshev's FSA [2,1] that we model after, the basic observations are tuples of (P, min, opt) in their generic form. The observations in this form, specifically, the property P, can be exploded further into Lucx's context set and further into an atomic simple context [23,22]. (Actually P can be any arbitrary expression E). Context switching between different observations is done naturally with the traditional Lucid @ context switching operator [18].

The Gladyshev's concept of a generic observation sequence [1] can be expanded into the context stream using the *min* and *opt* values, where they will translate into index values. Thus, $obs = (A, 3, 0)(B, 2, 0)$ expands the property labels A and B into a finite stream of five indexed elements: $AAABB$. Thus, a Forensic Lucid fragment in Listing 1.1 would return the third A of the $AAABB$ context stream in the observation portion of *o*. Therefore, possible evaluations to check for the properties can be as shown in Figure 1 [18].

The property values of A and B can be anything that context calculus allows or even more generally any arbitrary E allowing to encode all kinds of case knowledge. The `observation sequence` is a finite `ordered` context tag set [22] that allows an integral "duration" of a given tag property. This may seem like we allow duplicate tag values that are unsound in the classical Lucid semantics; however, we find our way around it with the implicit tag index. The semantics of the arrays of computations is not a part of either GIPL or Lucx; however, the arrays are provided by Objective Lucid. We use the notion of the arrays to evaluate multiple computations at the same context. Having an array of computations is conceptually equivalent of running an a Lucid program under the same context for each array element in a separate instance of the evaluation engine and then the results of those expressions are gathered in one ordered storage within the originating program. Arrays in Forensic Lucid are needed to represent a set of results, or *explanations* of evidential statements, as well as denote some properties of observations. (We explore the notion of arrays in Forensic Lucid much greater detail in another work) [18].

```
// Give me observed property at index 2 in the observation sequence
     obs
o @.obs 2
where
  // Higher-level dimension in the form of (P,min,opt)
  observation o;
  // Equivalent to writing = { A, A, A, B, B };
  // Equivalent to writing = A fby A fby A fby B fby B fby eod;
  observation sequence obs = (A,3,0)(B,2,0);
  where
    // Properties A and B are arrays of computations
    // or any Expressions
    A = [c1,c2,c3,c4];
    B = E;
     ...
  end;
end;
```

Listing 1.1. Observation Sequence With Duration

To make equivalence relation with the formal Gladyshev's FSA approach, computations c_i correspond to the states q and event i that enable the transition. For Forensic Lucid, we define c_i as theoretically any Lucid expression $o = E$ [18].

```
Observed property (context): A A A B B
        Sub-dimension index: 0 1 2 3 4

o @.obs 0 = A
o @.obs 1 = A
o @.obs 2 = A
o @.obs 3 = B
o @.obs 4 = B

To get the duration/index position:

o @.obs A = 0 1 2
o @.obs B = 3 4
```

Fig. 1. Handling Duration of an Observed Property in the Context

In Figure 1 a possibility is illustrated to query for the sub-dimension indices by raw property where it is present. This produces a finite stream of valid indices that can be used in subsequent expressions, or, alternatively by supplying the index we can get the corresponding raw property at that index. The latter feature is still under investigation of whether it is safe to expose it to Forensic Lucid programmers or make it implicit at all times at the implementation level. This method of indexing was needed to remedy the "problem" of "duplicate tags": as previously mentioned, observations form the context and allow durations. This means multiple duplicate dimension tags with implied subdimension indexes should be allowed as the semantics of traditional Lucid approaches do not allow duplicate dimension tags. It should be noted however, that the combination of the tag and its index in the stream is still unique and is nicely folded into the traditional Lucid semantics [18].

Transition Function

A transition function (derived from the same notion from the works of Gladyshev et al. [1,2]) determines how the context of evaluation changes during computation. It represents in part the case's crime scene modeling. A general issue exists that we have to address is that the transition function ψ is usually problem-specific. In the FSA approach, the transition function is the labeled graph itself [1]. We follow the graph of the case to model our Forensic Lucid equivalent [18].

In general, Lucid has already basic operators to navigate and switch from one context to another, that can be said equivalent to state switching. These operators represent the basic "built-in" transition functions in themselves (the intensional operators such as @, #, `iseod`, `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as their inverse operators [18]. However, a specific problem being modeled requires more specific transition function than just plain intensional operators. In this case the transition function is a Forensic Lucid function where the matching state transition modeled through a sequence of intensional operators [18]. In fact, the forensic operators are just pre-defined functions that rely on the traditional and inverse Lucid operators as well as context switching operators that achieve something similar to the transitions [18].

Generic Observation Sequences

We adopt a way of modeling generic observation sequences as an equivalent to the `box` operator from the Lucx's context calculus [23,22] in the dimensional context that defines the space of all possible evaluations. The generic observation sequence context contains observations whose properties' duration is not fixed to the *min* value as in $(P, min, 0)$ as we studied so far. The third position in the observation tuple, *opt* is not 0 in the generic observation and as a result in the containing observation sequence, e.g. $os = (P_1, 1, 2)(P_2, 1, 1)$. Please refer to [1,2,18,19] for more detailed examples of a generic observation sequence [18].

Primitive Operators

The basic set of the classic intensional operators is extended with the similar operators, but inverted in one of their aspects: either negation of trueness or reverse of direction of navigation. Here we provide a definition of these operators alongside with the classical ones (to remind the reader what they do and enlighten the unaware reader). The reverse operators have a restriction that they must work on the bounded streams at the positive infinity. This is not a stringent limitation as the our contexts of observations and evidence in this work are always finite, so they all have the beginning and the end. What we need is an ability to go back in the stream and, perhaps, negate in it with classical-like operators, but reversed [18].

Following the steps in [20], we further represent the definition of the operators via @ and #. Again, there is a mix of classical operators that were previously

defined in [20], such as `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as the new operators from this work [18].

Forensic Operators

The operators presented here are based on the discussion of the combination [1] function and others that form more-than-primitive operations to support the required implementation. The `comb()` operator is realized in the general manner in Forensic Lucid for combining analogies of multiple partitioned runs (MPRs) [1], which in our case are higher-level contexts, in the new language's dimension types [18].

- `combine` corresponds to the *comb* function described earlier. It is defined in Listing 1.2.

```
/**
 * Append given e to each element of a given
 * stream e under the context of d.
 * @return the resulting combined stream
 */
combine(s, e, d) =
  if iseod s then eod;
  else (first s fby.d e) fby.d combine(next s, e, d);
```

Listing 1.2. The `combine` Operator

- `product` corresponds to the cross-product of contexts, translated from that of the LISP example and added with context. It is defined in Listing 1.3.

```
/**
 * Append elements of s2 to element of s1
 * in all possible combinations.
 */
product(s1, s2, d) =
  if iseod s2 then eod;
  else combine(s1, first s2) fby.d product(s1, next s2)
```

Listing 1.3. The `product` Operator

3 Modeling Printer Case in Forensic Lucid

3.1 ACME Manufacturing Printing Case

This is one of the cases we re-examine from the Gladyshev's FSA approach [1].

> *The local area network at some company called ACME Manufacturing consists of two personal computers and a networked printer. The cost of running the network is shared by its two users Alice (A) and Bob (B). Alice, however, claims that she never uses the printer and should not be paying for the printer consumables. Bob disagrees, he says that he saw Alice collecting printouts. According to the manufacturer, the printer works as follows:*

1. *When a print job is received from the user, it is stored in the first unallocated directory entry of the print job directory.*
2. *The printing mechanism scans the print job directory from the beginning and picks the first active job.*
3. *After the job is printed, the corresponding directory entry is marked as "deleted", but the name of the job owner is preserved.*
4. *The printer can accept only one print job from each user at a time.*
5. *Initially, all directory entries are empty.*

The investigator finds the current state of the printer's buffer as:

1. *Job From B Deleted*
2. *Job From B Deleted*
3. *Empty*
4. *Empty*
5. *...*

Investigative Analysis

If Alice never printed anything, only one directory entry must have been used, because the printer accepts only one print job from each user at a time [1]. However, two directory entries have been used and there are no other users except Alice and Bob. Therefore, it must be the case that both Alice and Bob submitted their print jobs in the same time frame. The trace of Alice's print job was overwritten by Bob's subsequent print jobs. As a result, a finite state machine is constructed to model the situations as in the FSA [1] to indicate the initial state and other possible states and how to arrive to them when Alice or Bob would have submitted a job and a job would be deleted [1]. The FSM presented in [1] covers the entire case with all possible events and transitions resulted due to those events. It is modeled based on the properties of the investigation, in this case the printer queue's state according to the manufacturer specifications and the two potential users. The modeling is assumed to be done by the investigator in the case in order to perform a thorough analysis. It also doesn't really matter how actually it so happened that the Alice's print job was overwritten by Bob's subsequent jobs as is not a concern for this case any further. Assume, this behavior is derived from the manufacturer's specification and the evidence found. The investigator will have to make similar assumptions in the real case [1].

The authors of [1] provided a proof-of-concept implementation of this case in Common LISP (not recited in here) which takes about 6-12 pages of printout depending on the printing options set and column format. Using our proposed solution, we rewrite the example in Forensic Lucid and show the advantages of a much finer conciseness and added benefit of the implicit context-driven expression and evaluation, and parallel evaluation that the LISP implementation lacks entirely.

3.2 Sample Forensic Lucid Specification

The simulated printer case is specified in Forensic Lucid as follows. ψ is implemented in Listing 1.5. We then provide the implementation of Ψ^{-1} in [18] in

Listing 1.6. Finally, the "main program" is modeled in Listing 1.4 that sets up the context hierarchy and the invokes Ψ^{-1}. This specification is the translation of the LISP implementation by Gladyshev described earlier [1] and described in this section in semi-structured English.

The "Main Program"

In Listing 1.4 where the computation begins in our Forensic Lucid example. This is an equivalent of `main()` or program entry point in other mainstream languages. The goal of this fragment is to setup the context of evaluation which is core to the case – the evidential statement `es`. This is the highest level dimension in Lucid terms, and it is hierarchical. This is an unordered list (set) of stories and witness accounts of the incident (themselves known as observation sequences); ordering in the program of them is arbitrary and has an array-like structure. The relevant stories to the incident are that of Alice, the evidence of the printer's final state as found by the investigator, and the "expert testimony" by the manufacturer of how the printer works. These observation sequences are in turn defined as ordered collections of observations nesting one lever deeper into the context. The printer's final state dimension `F` is the only observation for the printer found by the investigator, which is an observation of the property of the printer's queue "Bob's job deleted last" syntactically written as "B_deleted" as inherited from Gladyshev's notation. Its duration is nothing special, that it was simply present. The `manuf` observation sequence dictated by the manufacturer's specification that the printer's queue state was empty initially for an undetermined period `$` of time when the printer was delivered. These are two observations, followed in time/ Alice's line (also tow observations) is that from the beginning Alice did not not any actions signified by the properties P such as "add_B" or "take" (implying the computation "add_A" has never happened (0 duration for the "infinity" i.e. till the investigator examined the printer); which is Alice's claim. `alice_claim` is a collection of Boolean results for possible explanations or lack thereof for Alice's claim in this case at the context of all this evidence and as evaluated by `invpsiacme` Ψ^{-1}. If Alice's claim were to check out; the results would be "true"; "false" otherwise.

Modeling Forward Transition Function ψ

In Listing 1.5 ψ illustrating the normal flow of operations to model the scene. Which is also a translation from LISP from Gladyshev [1] using Forensic Lucid syntax and operators described in [18]. The function is modeled per manufacturer specification and focuses on the queue of the printer. "A" corresponds to "Alice" and "B" to "Bob" along with their prompted queue actions to add deleted print jobs. The code is a rather straightforward translation of the FSM/LISP code in [1]. `S` is a collection of state properties observed. `c` is a "computation" action to add or take print jobs by the printer's spooler. `d` is a classical Lucid dimension type along which the computation is happening (there can be multiple dimensions and evaluations going on).

```
alice_claim @ es
where
  evidential statement es = [ printer, manuf, alice ];

  observation sequence printer = F;
  observation sequence manuf = [Oempty, $];
  observation sequence alice = [Oalice, F];

  observation F = (``B_deleted'', 1, 0);
  observation Oalice = (P_alice, 0, +inf);
  observation Oempty = (``empty'', 1, 0);

  // No ``add_A''
  P_alice = unordered {``add_B'', ``take''};

  alice_claim = invpsiacme(F, es);
end;
```

Listing 1.4. Developing the Pinter Case: "main"

```
acmepsi(c, s, d) =
  // Add a print job from Alice
  if c == ``add_A'' then
    if d1 == ``A'' || d2 == ``A'' then s;
    else
      if d1 in S then ``A'' fby.d d2;
      else
        if d2 in S then d1 fby.d ``A'';
        else s;
  // Add a print job from Bob
  else if c == ``add_B'' then
    if d1 == ``B'' || d2 == ``B'' then s;
    else
      if d1 in S then ``B'' fby.d d2;
      else
        if d2 in S then d1 fby.d ``B'';
        else s;
  // Printer takes the job per manufacturer specification
  else if c == ``take''
    if d1 == ``A'' then ``A_deleted'' fby.d d2;
    else
      if d1 == ``B'' then ``B'' fby.d d2;
      else
        if d2 == ``A'' then d1 fby.d ``A_deleted'';
        else
          if d2 == ``B'' then d1 fby.d ``B_deleted'';
          else s;
  // Done
  else s fby.d eod;

  where
    dimension d;
    S = [``empty'', ``A_deleted'', ``B_deleted''];
    d1 = first.d s;
    d2 = next.d d1;
  end;
```

Listing 1.5. "Transition Function" ψ in Forensic Lucid for the ACME Printing Case

Modeling Inverse Transition Function Ψ^{-1}

In Listing 1.6 is the inverse Ψ^{-1} backtracking implementation with the purpose of event reconstruction, also translated from LISP to Forensic Lucid like the preceding fragments using the Forensic Lucid operators. It is naturally more

```
invpsiacme(s, d) = backtraces
where
  backtraces = [A, B, C, D, E, F, G, H, I, J, K, L, M];
  where
    A = if d1 == ``A_deleted''
        then d2 pby.d ``A'' pby.d ``take'' else eod;

    B = if d1 == ``B_deleted''
        then d2 pby.d ``B'' pby.d ``take'' else eod;

    C = if d2 == ``A_deleted'' && d1 != ``A'' && d2 != ``B''
        then d1 pby.d ``A'' pby.d ``take'' else eod;

    D = if d2 == ``B_deleted'' && d1 != ``A'' && d2 != ``B''
        then d1 pby.d ``B'' pby.d ``take'' else eod;

    E = if d1 in S && d2 in S
        then s pby.d ``take'' else eod;

    F = if d1 == ``A'' && d2 != ``A''
        then
          [ d2 pby.d ``empty'' pby.d ``add_A'',
            d2 pby.d ``A_deleted'' pby.d ``add_A'',
            d2 pby.d ``B_deleted'' pby.d ``add_A'' ]
        else eod;

    G = if d1 == ``B'' && d2 != ``B''
        then
          [ d2 pby.d ``empty'' pby.d ``add_B'',
            d2 pby.d ``A_deleted'' pby.d ``add_B'',
            d2 pby.d ``B_deleted'' pby.d ``add_B'' ]
        else eod;

    H = if d1 == ``B'' && d2 == ``A''
        then
          [ d1 pby.d ``empty'' pby.d ``add_A'',
            d1 pby.d ``A_deleted'' pby.d ``add_A'',
            d1 pby.d ``B_deleted'' pby.d ``add_A'' ]
        else eod;

    I = if d1 == ``A'' && d2 == ``B''
        then
          [ d1 pby.d ``empty'' pby.d ``add_B'',
            d1 pby.d ``A_deleted'' pby.d ``add_B'',
            d1 pby.d ``B_deleted'' pby.d ``add_B'' ]
        else eod;

    J = if d1 == ``A'' || d2 == ``A''
        then s pby.d ``add_A'' else eod;

    K = if d1 == ``A'' && d2 == ``A''
        then s pby.d ``add_B'' else eod;

    L = if d1 == ``B'' && d2 == ``A''
        then s pby.d ``add_A'' else eod;

    M = if d1 == ``B'' || d2 == ``B''
        then s pby.d ``add_B'' else eod;

    where
      dimension d;
      S = [``empty'', ``A_deleted'', ``B_deleted''];
      d1 = first.d s;
      d2 = next.d d1;
    end;
```

Listing 1.6. "Inverse Transition Function" Ψ^{-1} in Forensic Lucid for the ACME Printing Case

complex than ψ due to a possibility of choices (non-determinism) when going back in time so all of them have to be explored. This backtracking, if successful, for any claim, would provide the Gladyshev's "explanation" of that claim, i.e. the claim attains its meaning and is validated within the provided evidential statement. Ψ^{-1} is based on the traversal from `F` to the initial observation of the printer's queue as defined in "main". If such path were to exist, then Alice's claim would have had an explanation. `pby` (*preceeded by*) is the Forensic Lucid inverse operator of classical Lucid's `fby` (*followed by*). `backtraces` is an array of event backtracing computations identified with variables; their number and definitions depend on the crime scene and are derived from the state machine of Gladyshev.

4 Conclusion

We presented the basic overview of Forensic Lucid, its concepts, ideas, and dedicated purpose – to model, specify, and evaluation digital forensics cases. The process of doing so is significantly simpler and more manageable than the previously proposed FSM model and its common LISP realization. At the same time, the language is founded in more than 30 years research on correctness and soundness of programs and the corresponding mathematical foundations of the Lucid language, which is a significant factor should a Forensic Lucid-based analysis be presented in court. We re-wrote in Forensic Lucid one of the sample cases initial modeled by Gladyshev in the FSM and Common LISP to show the specification is indeed more manageable and comprehensible than the original and fits in two pages in this paper.

We also still realize by looking at the examples the usability aspect is still desired to be improved further for the investigators, especially when modeling ψ and Ψ^{-1}, as a potential limitation, prompting one of the future work items to address it further.

In general, the proposed practical approach in the cyberforensics field can also be used to model and evaluate normal investigation process involving crimes not necessarily associated with information technology. Combined with an expert system (e.g. implemented in CLIPS [25]), it can also be used in training new staff in investigation techniques. The notion of hierarchical contexts as first-class values brings more understanding of the process to the investigators in cybercrime case management tools.

5 Future Work

- Formally prove equivalence to the FSA approach.
- Adapt/re-implement a graphical UI based on the data-flow graph tool [12] to simplify Forensic Lucid programming further for not very tech-savvy investigators by making it visual. The listings provided are not very difficult to read and quite manageable to comprehend, but any visual aid is always an improvement.

- Refine the semantics of Lucx's context sets and their operators to be more sound, including Box and Range.
- Explore and exploit the notion of credibility factors of the evidence and witnesses fully.
- Release a full standard Forensic Lucid specification.

Acknowledgments. This research work was funded by NSERC and the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada. We would also like to acknowledge the reviewers who took time to do a constructive quality review of this work.

References

1. Gladyshev, P., Patel, A.: Finite state machine approach to digital event reconstruction. Digital Investigation Journal 2(1) (2004)
2. Gladyshev, P.: Finite state machine analysis of a blackmail investigation. International Journal of Digital Evidence 4(1) (2005)
3. Ashcroft, E.A., Wadge, W.W.: Lucid – a formal system for writing and proving programs. SIAM J. Comput. 5(3) (1976)
4. Ashcroft, E.A., Wadge, W.W.: Erratum: Lucid – a formal system for writing and proving programs. SIAM J. Comput. 6(1), 200 (1977)
5. Wadge, W.W., Ashcroft, E.A.: Lucid, the Dataflow Programming Language. Academic Press, London (1985)
6. Ashcroft, E.A., Faustini, A.A., Jagannathan, R., Wadge, W.W.: Multidimensional Programming. Oxford University Press, London (1995) ISBN: 978-0195075977
7. Lalement, R.: Computation as Logic. Prentice Hall (1993); C.A.R. Hoare Series Editor. English translation from French by John Plaice
8. Paquet, J.: Distributed eductive execution of hybrid intensional programs. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), pp. 218–224. IEEE Computer Society, Seattle (2009)
9. Plaice, J., Mancilla, B., Ditu, G., Wadge, W.W.: Sequential demand-driven evaluation of eager TransLucid. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 1266–1271. IEEE Computer Society, Turku (2008)
10. Rahilly, T., Plaice, J.: A multithreaded implementation for TransLucid. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 1272–1277. IEEE Computer Society, Turku (2008)
11. The GIPSY Research and Development Group: The General Intensional Programming System (GIPSY) project. Department of Computer Science and Software Engineering.Concordia University, Montreal (2002–2012), `http://newton.cs.concordia.ca/~gipsy/` (last viewed February 2010)
12. Ding, Y.: Automated translation between graphical and textual representations of intensional programs in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering. Concordia University, Montreal, Canada (June 2004), `http://newton.cs.concordia.ca/~paquet/filetransfer/publications/theses/DingYiminMSc2004.pdf`

13. Mokhov, S.A., Paquet, J., Debbabi, M.: On the need for data flow graph visualization of Forensic Lucid programs and forensic evidence, and their evaluation by GIPSY. In: Proceedings of the Ninth Annual International Conference on Privacy, Security and Trust (PST), pp. 120–123. IEEE Computer Society (July 2011) Short paper; full version online at http://arxiv.org/abs/1009.5423
14. Ji, Y.: Scalability evaluation of the GIPSY runtime system. Master's thesis, Department of Computer Science and Software Engineering. Concordia University, Montreal, Canada (March 2011)
15. Mokhov, S.A.: Enhancing the formal cyberforensic approach with observation modeling with credibility factors and mathematical theory of evidence. Login 34(6), 101 (2009); Presented at WIPS at USENIX Security 2009, http://www.usenix.org/events/sec09/wips.html
16. Mokhov, S.A., Paquet, J., Debbabi, M.: Towards automated deduction in blackmail case analysis with Forensic Lucid. In: Gauthier, J.S. (ed.) Proceedings of the Huntsville Simulation Conference (HSC 2009). SCS, pp. 326–333 (October 2009), http://arxiv.org/abs/0906.0049
17. Mokhov, S.A.: The role of self-forensics modeling for vehicle crash investigations and event reconstruction simulation. In: Gauthier, J.S. (ed.) Proceedings of the Huntsville Simulation Conference (HSC 2009). SCS, pp. 342–349 (October 2009), http://arxiv.org/abs/0905.2449
18. Mokhov, S.A., Paquet, J., Debbabi, M.: Formally specifying operational semantics and language constructs of Forensic Lucid. In: Göbel, O., Frings, S., Günther, D., Nedon, J., Schadt, D. (eds.) Proceedings of the IT Incident Management and IT Forensics (IMF 2008). LNI, vol. 140, pp. 197–216. GI (September 2008)
19. Mokhov, S.A., Paquet, J.: Formally specifying and proving operational aspects of Forensic Lucid in Isabelle. Technical Report 2008-1-Ait Mohamed, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada. In: Theorem Proving in Higher Order Logics (TPHOLs 2008): Emerging Trends Proceedings (August 2008)
20. Paquet, J.: Scientific Intensional Programming. PhD thesis, Department of Computer Science. Laval University, Sainte-Foy, Canada (1999)
21. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM 20(7), 519–526 (1977)
22. Tong, X.: Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering. Concordia University, Montreal, Canada (April 2008)
23. Wan, K.: Lucx: Lucid Enriched with Context. PhD thesis, Department of Computer Science and Software Engineering.Concordia University, Montreal, Canada (2006)
24. Mokhov, S.A.: Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 1288–1294. IEEE Computer Society, Turku (2008)
25. Riley, G.: CLIPS: A tool for building expert systems (2007-2009), http://clipsrules.sourceforge.net/ (last viewed: October 2009)