Automated Result Verification with AWK

Balkhis Abu Bakar* and Tomasz Janowski
The United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau
{bab,tj}@iist.unu.edu

Abstract

The goal of result-verification is to prove that one execution run of a program satisfies its specification. Compared with implementation-verification, result-verification has a larger scope for applications in practice, gives more opportunities for automation and, based on the execution record not the implementation, is particularly suitable for complex systems. This paper proposes a technical framework to apply this technique in practice. We show how to write formal result-based specifications, how to generate a verifier program to check a given specification and to carry out result-verification according to the generated program. The execution result is written as a text file, the verifier is written in AWK *(special-purpose language for text processing) and verification is done automatically by the* AWK *interpreter, given the verifier and the execution result as inputs.*

Keywords: Formal Specification, Result Verification, Error Detection, Program Generators, Software Components

1. Introduction

The goal of implementation-verification is to prove that a given program behaves correctly under all possible executions, for instance that it produces correct output for all legal inputs. There are several conditions to carry out such verification in practice: (1) The implementation is available in a form suitable for its analysis. (2) The implementation "size" is such that the proof remains feasible. (3) It is written in a language which supports specification and proof. (4) There is enough human expertease to guide the proof. Such conditions are often not satisfied in practice.

Result verification is a possible alternative. It relies not on the implementation of the program but its execution record. This record may be a sequence of inputs and the corresponding outputs from the program, when it computes a certain function, or a sequence of values observed about the changing state of the program, as it interacts with its environment, or perhaps such values augmented with the time-stamps. Whatever the form, the goal is to decide if this record describes a correct execution of the program. The positive answer does not mean that the program is fault-free, only that this execution did not exhibit any errors. On the other hand, finding an error is like constructing a counter-example for claims of correctness about the program. Due to its modest goals, result-verification has several advantages over implementation-verification: (1) It can be carried out off-line, after the execution terminated, or on-line, as the execution unfolds. (2) Its application scope includes both off-the-shelf components (binary files) and remote service providers (distributed objects). (3) Due to the simple structure involved (a sequence), it is easier to automate. (4) Based solely on the execution record, its complexity is largely independent from the "size" of the implementation and the language used, what makes it particularly suitable for complex, heterogeneous systems.

In this paper we propose a technical framework to carry out automated result-verification in practice. The framework is described in Figure 1. Its main features are:

1. The execution result is a simple text file. Many programs produce such (log) files during their normal operations, for administrative purposes. A general technique to record exactly the information needed for verification, is to introduce a program wrapper.

2. The execution result is given as input to the verifier program, which does the actual verification. Given the execution result in a text file, we consider result-verification as the text-processing task. Accordingly, the verifier is written in AWK, which is a special-purpose language for text processing [12], implemented for most computing platforms. Verification is done by the AWK interpreter, given the execution result and the verifier program as inputs.

*On leave of absence from the University of Northern Malaysia.

0-7695-0583-X/00 $10.00 © 2000 IEEE

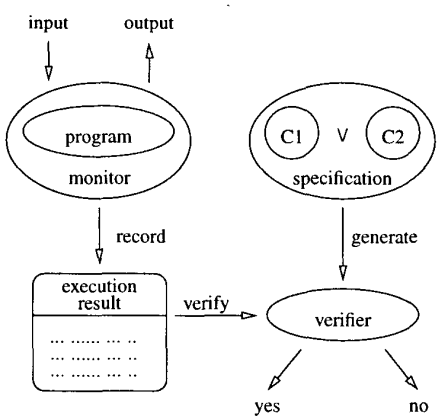

Figure 1. Framework for result verification.

3. The verifier program is not written by hand but specification-generated. The generator takes the property the verifier should check, called result-based specification, as input and produces the code of the verifier program as output. The specification-to-verifier generator is itself written in AWK.

4. A result-based specification is a first-order property built with two kinds of variables (state- and result-variables) using the functions and predicates over their respective types. A result variable refers to the contents of the result file. A state variable allows to calculate the values that are not explicitly recorded in the result file but derived from it. For each state variable we define its type, initial value and how this value changes for different operations invoked. We also allow specifications to be combined.

This paper continues the line of research on result-verification. Its main contribution is demonstrating how result-verification can be carried out using the current computing technology, based on the formally-defined language for result-based specifications. In comparison, [1, 17] studies algorithmic foundations for result-verification, for functional problems and correctness expressed in probabilistic terms. Here we focus more on the systematic design and use of specifications and define correctness in deterministic terms. In [8] a language is proposed which supports on-line automatic result-verification of imperative implementations based on their complete functional specifications. Here, specifications can be partial, they are expressed logically and have their semantics defined formally. Also verification is possible both off-line and on-line (although this is not much studied). In [7] we introduced logic-based regular expressions to carry out on-line result-verification as a kind of patter-matching. Compared with specifications introduced here, patterns are more expressive but also more expensive to verify, both time and memory requirements. It remains to be seen if the framework described here can also support pattern-based result verification.

The rest of the paper is as follows. Section 2 illustrates the concept of a software component, used to model the behaviour of a program. Section 3 describes how to wrap a component to record the relevant observations during its execution. Result-based verification, using a hand-written verifier program, is the subject of Section 4. Section 5 defines a result-based specification, a logical property the verifier is designed to check. Section 6 describes various ways such specifications can be combined. Section 7 describes the generator which produces the code of the verifier program given the specification. Section 8 contains an example and Section 9 presents some conclusions.

2. Modelling a Program

Suppose a program contains an internal state and provides several operations to access this state. The operations are readers, they return a value without modifying the state or writers, they change the current state without returning any value. A reader can be a constant, which result does not depend on the current state, or an observer. Likewise, a writer can be a generator, including the initial state *init*, or a modifier. The state does not change between invocations of generator/modifier operations (it is persistent) and the result of an operation is independent on other operations being executed concurrently (execution is atomic).

Suppose *State* represents the program's state-space and *Type* is any other type. The class of an operation depends on the position of *State* in its signature. In the simplest case:

constant: Type
generator: State
observer: State $\rightarrow$ Type
modifier: State $\rightarrow$ State

In general, we assume that operations can take several arguments and return results of different types. *Type* includes **Bool**, **Nat**, Cartesian product of two types etc.

Type ::= **Bool** | **Nat** | Type $\times$ Type | ...

As a simple illustration consider an editor program for creating and modifying lists of elements. The elements are values of the abstract type *Elem*, can be characters, words, records etc. Editing is done relative to the position of a cursor which points at any element in the list or one position beyond the last element. We have one generator *init* which makes the list empty and sets the cursor position to one. Two observers: *pos* returns the cursor position (natural number) and *list* returns the actual list. Four modifiers:

left and *right* move the cursor one position backward or forward, *insert* inserts a given element at the current position and moves the cursor forward and *delete* removes the element pointed by the cursor. The abstract specification for this editor can be formally described as below:

type	**value**
Elem, State	init: State, pos: State → **Nat**, list: State → Elem*, insert: Elem × State → State, left,right,delete: State $\xrightarrow{\sim}$ State

This notation is part of RSL, the RAISE Specification Language [10]. We chose RSL to present examples, due to its formal basis and support for abstraction. We only use the small subset of the language.

The actual behaviour of the editor is captured by the axioms below. The following axiom constraints the state *init* to return position one and the empty list, and limits the current position to be always, for all states, in the range from one to one plus the length of the list.

$$\text{pos(init)} = 1 \wedge \text{list(init)} = \langle\,\rangle \wedge$$
$$(\forall\ \text{s:State} \bullet \text{pos(s)} > 0 \wedge \text{pos(s)} \leq \text{len(list(s))}+1)$$

As the list has no upper limit, *insert* is a total function which can be applied to any value of its arguments (→). The axiom below describes the effect of *insert*: insert a given element at the current position, move forward the previous occupant and all its successors, increase by one the cursor position and the length of the list, leave the rest unchanged.

$$\begin{array}{l}\forall\ \text{s:State, e:Elem} \bullet \\ \quad \text{pos(insert(e,s))} = \text{pos(s)}+1 \wedge \\ \quad \text{len(list(insert(e,s)))} = \text{len(list(s))}+1 \wedge \\ \quad (\forall\ \text{i:}\mathbf{Nat} \bullet \\ \qquad \text{i} > 0 \wedge \text{i} \leq \text{len(list(insert(e,s)))} \Rightarrow \\ \qquad (\text{i} < \text{pos(s)} \Rightarrow \text{list(insert(e,s))(i)} = \text{list(s)(i)}) \wedge \\ \qquad (\text{i} = \text{pos(s)} \Rightarrow \text{list(insert(e,s))(i)} = \text{e}) \wedge \\ \qquad (\text{i} > \text{pos(s)} \Rightarrow \text{list(insert(e,s))(i)} = \text{list(s)(i}-1)) \\ \quad)\end{array}$$

Other modifiers are partial ($\xrightarrow{\sim}$). For instance, the following axiom describes the effect of *left*: for all states where the cursor position is greater than one, *left* decreases the position and leaves the contents unchanged. It says nothing about *left* when the cursor is already on the first position:

$$\begin{array}{l}\forall\ \text{s:State} \bullet \text{pos(s)} > 1 \Rightarrow \\ \quad \text{pos(left(s))} = \text{pos(s)}-1 \wedge \text{list(left(s))} = \text{list(s)}\end{array}$$

Such axioms formalise the required effects of various operations provided by the program, without saying how such operations should be implemented. They provide the basis for implementation-verification. They can also help formulate properties to carry out result-verification, based on the program's execution result.

3. Program Execution Result

We consider that the main purpose of a program is to make available to its users the operations in its interface, and to carry out such operations according to their intended meaning. In order to judge if the program indeed delivers this promised behaviour, we rely on its execution record. Many programs create such records during their normal run in the form of log or history files. However, intended for administrative rather than verification purposes, such files may contain too little or too much information, from the point of view of the properties we want to check. A general technique to create the execution record, according to different verification requirements, is wrapping. A wrapper does not modify the original program, but rather includes this program to carry out certain activities vis-a-vis its usual behaviour. It takes over all communication between the program and the environment, but itself remains transparent. Here we only use the wrapper to record the interactions between the program and the environment and make relevant observations about the resulting state.

As an illustration consider again the list editor. We introduce the type *Operation* to include all state-changing operations with their parameters, the function *exec* to execute a given operation on the state, according to the functions in Section 2, and *possible* to check the precondition for the corresponding operation.

type	**value**
Operation ==	exec: Operation × State $\xrightarrow{\sim}$ State
left |	exec(op,s) ≡
right |	**if** op=left **then** left(s) **else** ...
delete |	**pre** possible(op,s),
insert(Elem)	possible: Operation × State → **Bool**
	possible(op,s) ≡
	(op=left ⇒ pos(s) > 1) ...

Suppose the execution result (type *Result*) is a sequence of observations (type *Observation*), each produced by the function *observe* given an operation and a state on which it is invoked. A wrapper is a function which takes an operation, a state and an execution result as arguments. It executes the operation on the state (*exec*) and creates the record of this execution (via the function *observe*) at the end of the execution result. It assumes that the precondition for the operation is satisfied (*possible*).

type
Observation, Result = Observation*
value
wrapper: Operation × State × Result $\xrightarrow{\sim}$ State × Result
wrapper(op,s,r) ≡
(exec(op,s), r ⌢ ⟨observe(op,s)⟩)
pre possible(op,s),
observe: Operation × State → Observation

So far, *Observation* is defined as an abstract type and the *observe* function is unspecified. Suppose the observation consists of five fields: the operation name (without arguments), its argument (if any), the length, the cursor position, the current element (if any). We introduce the types *Name* for operation names, *Elem'* extending the type *Elem* with the value *none*, and the functions *name* and *argument* to obtain the name and the argument of an operation, respectively.

```
type
  Elem' == none | put(get:Elem),
  Name == left | right | delete | insert,
  Observation = Name × Elem' × Nat × Nat × Elem'
value
  name: Operation → Name,
  argument: Operation → Elem',
  observe: Operation × State → Observation
  observe(op,s) ≡
    (name(op),argument(op),len list(s),pos(s),
     if pos(s) > len list(s)
     then none else put(list(s)(pos(s))) end )
```

Here is a concrete execution of the editor, for editing strings of characters. We create a short string and then locate and correct a character inside. On the left is the execution record; we use '-' for *none*.

name	arg	ln	pos	elem	editor
insert	t	0	1	-	_
insert	e	1	2	-	t_
insert	s	2	3	-	te_
insert	t	3	4	-	tes_
left	-	4	5	-	test_
left	-	4	4	t	tes**t**
delete	-	4	3	s	te**s**t
insert	x	3	3	t	te**x**t
right	-	4	3	t	te**x**t
		4	4	-	tex**t**

In general, we can represent the execution record as a text file with lines representing individual observations. Each line is a sequence of fields separated by spaces, part of the type *Observation*. The format is independent of the language used to write the implementation and is suitable for automatic text-processing.

Without ever looking at the implementation, we can tell from the execution result if the program does not work as expected. Indeed, the record above shows that the invocation of *insert(x)* did not change the cursor position when it should move it forward, as required by the axiom for *insert*. Being able to find such errors is like constructing counter-examples for claims of correctness about the implementation (which produced this particular execution record). On the other hand, verification which failed to discover any errors is inconclusive, similar to testing.

4. Result-Based Verification

In this section we study how to write a verifier program, the program which carries out result-verification. The verifier receives the execution result as input and carries out the computation to establish if this input is correct, which respect to a given property. The property is part of the verifier. Before we discuss in general what is a possible structure for such programs, how to build them to fulfill their intended purpose, we present some examples based on the list editor.

The range of properties we can verify depends on the contents of observations, part of the execution record. Suppose we only record the operation name. Based on this we can check if the editor allows to move the cursor prior to the first position. The verifier is described by the function *pre_first* below. The auxiliary function *pre_first'* is defined recursively on the execution record, given the calculated position value *pos* as its argument: initially one, *pos* is incremented or decremented by different operations. The main property is *pos*>*1*, after the operation *left*.

```
type
  Observation1 = Name,
  Result1 = Observation1*
value
  pre_first: Result1 → Bool
  pre_first(res) ≡
    pre_first'(res,1),
  pre_first': Result1 × Nat → Bool
  pre_first'(res,pos) ≡ res = ⟨⟩ ∨
    hd res = left ⇒ pos > 1 ∧
    case hd res of
      left → pre_first'(tl res,pos−1),
      delete → pre_first'(tl res,pos),
      _ → pre_first'(tl res,pos+1)
    end
```

Similar checking if the editor allows to move the cursor past the last position would require two such variables: one for position (*pos*) and another one for length (*ln*). The main property is *pos*≤*ln*, after the operation *right*:

```
value
  post_last: Result1 → Bool
  post_last(res) ≡
    post_last'(res,0,1),
  post_last': Result1 × Nat × Nat → Bool
  post_last'(res,ln,pos) ≡ res = ⟨⟩ ∨
    hd res = right ⇒ pos≤ln ∧
    case hd res of
      right → post_last'(tl res,ln,pos+1),
      left → post_last'(tl res,ln,pos−1),
      delete → post_last'(tl res,ln−1,pos),
      insert → post_last'(tl res,ln+1,pos+1)
    end
```

Let us extend the execution record with the current length and cursor position. This allow checking if different operations perform their actions correctly with respect to those two values. As we examine the execution record, we calculate the values of *ln* and *pos* based on the operation invoked, as before, and compare for every observation if the values stored and calculated are the same.

```
type
  Observation2= Name × Nat × Nat,
  Result2 = Observation2*
value
  check_len_pos: Result2 → Bool
  check_len_pos(res) ≡
    check_len_pos'(res,0,1),
  check_len_pos': Result2 × Nat × Nat → Bool
  check_len_pos'(res,ln,pos) ≡
    res = ⟨⟩ ∨
    let (op,ln',pos')=hd res in
      ln = ln' ∧ pos = pos' ∧
      case op of
        left → check_len_pos'(tl res,ln,pos−1),
        right → check_len_pos'(tl res,ln,pos+1),
        delete → check_len_pos'(tl res,ln−1,pos),
        insert → check_len_pos'(tl res,ln+1,pos+1)
      end
    end
```

Suppose we want to check if the elements in the list are inserted correctly. To this end we add to the observation the current element, as in Section 3. The idea is to recreate the contents of the list based on the operations recorded in the execution result. First we introduce the auxiliary function *substr* which returns a sub-list of a given list of elements, determined by its first position and length:

```
value
  substr: Elem* × Nat × Nat ~→ Elem*
  substr(s,n,k) ≡ ⟨ s(i) | i in ⟨ n..n+k−1 ⟩ ⟩
    pre n > 0 ∧ n ≤ len s − k + 1
```

Function *check_el'* takes the current list of elements as an argument. With each operation it modifies this list accordingly: for *delete* it removes the element at the current position, for *insert* it inserts the new element at the current position, for *left* and *right* it leaves the list unchanged. The main property is: if the record contains *none* as the current element then *pos* is *1+ln* (i.e. the cursor is outside the list), otherwise the current element on record is the same as the current element in the recreated list.

```
value
  check_el: Result → Bool
  check_el(res) ≡
    check_el'(res,⟨⟩),
  check_el': Result × Elem* → Bool
  check_el'(res,list) ≡
    res = ⟨⟩ ∨
    let
      (op,arg,ln,pos,elem) = hd res,
      bp = substr(list,1,pos−1),
      ap = substr(list,pos,ln−pos+1)
    in
      (elem = none ⇒ pos = ln+1 ∧
       elem ≠ none ⇒ elem = put(list(pos))) ∧
      case op of
        delete → check_el'(tl res, bp ⌢ tl ap),
        insert → check_el'(tl res, bp ⌢ ⟨get(arg)⟩ ⌢ ap),
        _ → check_el'(tl res,list)
      end
    end
```

Based on those examples we make some observations about the possible structure of the verifier program. The program takes the execution result as an argument and produces a true/false answer depending if the result contains an error. The program calculates this answer by examining recursively the execution record. Checking terminates on the first observation which contains an error (producing *false*), otherwise it continues until the record becomes empty. The program maintains a set of variables which values are initialised and then modified depending on the operation invoked. Such variables carry the state information from the first to the last observation, helping to decide if individual observations produced an error. The main property is a logical formula *main* which is expressed in terms of those variables and the contents of the current observation. The structure of the verifier program is shown below.

```
value
  check: Result → Bool
  check(res) ≡
    check'(res,e1,...,en),
  check': Result × T1 × ... × Tn → Bool
  check'(res,v1,...,vn) ≡
    res = ⟨⟩ ∨
    main(hd res,v1,...,vn) ∧
    case op(hd res) of
      op1 → check'(tl res,f1(v1),...,fn(vn)),
      ...
      opm → check'(tl res,g1(v1),...,gn(vn))
    end,
  main: Observation × T1 × ... × Tn → Bool
  main(ob,v1,...,vn) ≡ ...
```

We can see that the main parameters to carry out checking are: the formula which is calculated for each observation to detect an error (*main*) and the variables *vi* used in this formula, their types *Ti*, initial values *ei* and how the values change for different operations.

5. Result-Based Specification

The structure suggested for a verifier in Section 4 highlights what is essential for defining such programs: the property the program is supposed to check (logical part), not how it is checked (computational part). Later we show how to reuse the computational part allowing the verifier to be generated from its property, not written by hand. Now we demonstrate how it is possible to define such properties, called result-based specifications.

Consider again the editor example, the function *pre_first* which checks if the cursor can be moved prior to the first position. As position is not part of the record, it has to be calculated by the verifier. This is the purpose of the second argument to *pre_first'*, which is initialised to one and then modified according to the operation performed. For the verifier, this argument plays the role of a state variable. Function *post_last* requires an additional variable *ln*.

Suppose such variables are declared explicitly: their names, types, initial values and how their values change for different operations. The last involves a list of operation names with corresponding expressions to calculate the new value of the variable, perhaps involving its current value. If an operation is not mentioned in the list, the variable remains unchanged. Here is the syntax:

```
pos : Nat = 1          | ln : Nat = 0
  left -> pos-1        |   delete -> ln-1
  right -> pos+1       |   insert -> ln+1
  insert -> pos+1      |
```

Not only can *pos* be shared between *pre_first* and *post_last*, which have the same observation space, but both *pos* and *ln* can be reused in the definition of *check_len_pos*, which has a different observation space. The variables maintain their meaning across different properties about the editor and differed verifiers to carry out checking.

Such variable declarations provide the context for defining result-based specifications. In addition to the declared variables which we call state-based, we also have predefined variables to refer to the current observation. We call them result variables, and write $\$i$ where i is a natural number between zero and NF; NF returns the number of fields in the current observation. \$0 represent the whole observation, as the sequence of values of its fields. \$1 is the first field. \$2 is value of the second field etc. We introduce a convention that \$1 contains the name of the operation invoked and \$2 is the argument of this operation, if any.

Consider the variable *list*, used by *check_el* to recreate the contents of the editor. The variable contains a list of elements, it is initialised to the empty list and modified by the operations *delete* and *insert*. The expressions to recalculate its value contain the calls to the auxiliary function *substr* and refer to the result variables. Here is the declaration, written in a form understandable by the target language (AWK). In particular, `""` is the empty list, `$2` is the same as the singleton list containing `$2` and list concatenation is represented by juxtaposition.

```
list : Elem-list = ""
  delete ->
    substr(list,1,$4-1)
    substr(list,$4+1,$3-$4)
  insert ->
    substr(list,1,$4-1)
    $2 substr(list,$4,$3-$4+1)
```

The specification is a logical formula built from the state and line variables using propositional connectives, relations and functions over the corresponding domains. The formula evaluates to *false* if an error has been found, otherwise to *true*. The following are result-based specifications for all verifiers in Section 4. Again, we write them in a form understandable for AWK. In particular, `==` represents equality, `!=` inequality, `||` disjunction and `&&` conjunction. Also, implication must be written via negation and disjunction.

1. According to *pre_first*, if the operation is *left* then the cursor position must be greater than one.

   ```
   $1!="left" || pos>1
   ```

2. According to *post_last*, if the operation is *right* then the position value is not greater than the length.

   ```
   $1!="right" || pos<=ln
   ```

3. According to *check_len_pos*, the stored and calculated values for position and length must be the same.

   ```
   $3==ln && $4==pos
   ```

4. According to *check_el*, if the recorded element equals '-' (denotes *none*) then the position value is one plus the length, otherwise the recorded element is the same as the current element in the recreated list. In order to extract the current element from the list we use the function *substr*.

   ```
   ($5!="-" || $4==$3+1) &&
   ($5=="-" || $5==substr(list,$4,1))
   ```

Consider such specifications in general. Being part of the verifier program, the specification must be concrete enough to decide its validity in a given state by calculation (automatically) rather than deduction (interactively). This in turn rules out the use of the full first order logic, in particular quantifications over infinite domains, functions defined by pre/post conditions etc. We define a specification to be a first-order predicate build from propositional connectives, relations over value terms, and a limited kind of quantification over a range of field variables. Terms are build from the state- and result-variables using the functions over their

respective domains. Below, *var* represents a variable, *num* is a natural number and *nvar* is a numerical variable. We also use *fun* to represent a function and *rel* to represent a relation. We define the syntax of specifications as follows:

$$\begin{array}{rcl} term & ::= & var|\$num|\$nvar|fun(term,\ldots,term) \\ spec & ::= & true|!spec|spec\&\&spec|\ldots \\ & & rel(term,\ldots,term)| \\ & & all\ nvar : num.num : spec \end{array}$$

Consider the formal semantics. We interpret *spec* against the execution result, *res*, and the contents of the state variables, *st*. *res* is a sequence of observations and each observation consists of a list of fields, the first being the operation name. We apply the usual head and tail functions to *res* and all its elements, in particular *hd(hd(res))* is the most recent operation. Suppose *ini* represents the initial value of all state variables and *trans(st,op)* returns their new values when the operation was *op*. Then $res \models spec$ means *spec* is true about the execution res. It is defined as follows:

$$\begin{array}{l} res \models spec \text{ iff } res, ini \models spec \\ res, st \models spec \text{ iff} \\ \quad res = <> \vee \\ \quad hd\ res, st \models spec \wedge \\ \quad tl\ res, trans(st, hd\ hd\ res) \models spec \end{array}$$

This definition takes into account the current observation in the execution record and the changes to the state variables caused by the operations. It relies on the evaluation of *spec* based on *st* and the contents of individual observations ($hd\ res, st \models spec$). Let *ob* represent such an observation. Here is the interpretation, defined inductively on the structure of the specification:

$$\begin{array}{rcl} ob, st & \models & true \\ ob, st & \models & !spec \text{ iff not } ob, st \models spec \\ ob, st & \models & spec_1 \&\& spec_2 \text{ iff } ob, st \models spec_i, i = 1, 2 \\ ob, st & \models & rel(t_1, t_2) \text{ iff } rel([\![t_1]\!]_{ob,st}, [\![t_2]\!]_{ob,st}) \\ ob, st & \models & all\ i : n_1.n_2 : spec \text{ iff} \\ & & ob, st \models spec[ob(i)/\$i], i = n_1 \ldots n_2 \end{array}$$

The last rule applies substitution within *spec* of the result variable $\$i$ with $ob(i)$, the ith field of the observation. To evaluate a relation, we have to first calculate the values of its arguments, represented by the terms. Here is how we evaluate the terms, given the observation *ob* and the state *st*. In particular, we evaluate a state variable *v* by taking its value according to *st* (*st(v)*), the result variable *$n* by taking the value of the n-th field in *ob* (*ob(n)*) and *$nv* where *nv* is a numerical variable, by taking the field in *ob* according to the value of *nv* (*ob(st(nv))*).

$$\begin{array}{l} [\![v]\!]_{ob,st} = st(v) \\ [\![\$n]\!]_{ob,st} = ob(n) \\ [\![\$nv]\!]_{ob,st} = ob(st(nv)) \\ [\![fun(t_1, t_2)]\!]_{ob,st} = fun([\![t_1]\!]_{ob,st}, [\![t_2]\!]_{ob,st}) \end{array}$$

6. Result-Based Specification Composition

By specification composition we mean to build a specification from simpler specifications, according to some formation rules and intended semantics. The semantics of composition expresses how to calculate the result of checking a composite specification from the results of checking component specifications (on the same execution record).

Conjunction is a particularly useful composition method, allowing for one of several properties to produce an error. One example is *check_len_pos* which checks if both length and position are recorded correctly. The specification is really build from two specifications, one to check the length (*len_ok*) and another to check the position (*pos_ok*).

```
len_ok is ($3==ln)
pos_ok is ($4==pos)
check_len_pos is len_ok && pos_ok
```

Suppose *pre_first* is carried out on the record which has the position value written down. How useful is then to know that the cursor moved prior to the first position, if this position is wrongly recorded in the first place? We would rather like to check the conjunction below, stopping as soon as the precondition is violated or the actual error detected:

```
pos_ok && pre_first
```

Conjunction is inappropriate if one specification defines relevant observations for another specification. For instance, only if the operation was *left* then *pos* must be greater than one. In such cases we need to use implication.

```
$1!="left" || pos>1
```

We may also like to combine implication with conjunction, to check if the precondition (correctness of the value of position) holds on all observations:

```
pos_ok && ($1!="left" || pos>1)
```

Or if it holds on the relevant observations only:

```
$1!="left" || (pos_ok && pos>1)
```

So far we only referred to different fields of the observation record by their numbers. Sometimes, we may like to check properties which are shared between different fields. For instance the property which must be satisfied for the third and fourth fields of the observation record (length and position) is that their values are non-negative. This is where we can use numerical variables to refer to the range of fields and use quantification over such variables, as below.

```
all i: 3.4: $i>=0
```

The treatment of composition in this paper is limited by the fact that checking produces a yes/no answer (if an error occurred) for the whole record. Instead, suppose verification produces a yes/no answer for every observation in the record (if and where the error occurred). Let 0 represent the presence and 1 the absence of an error.

Consider some examples of unary operators for specification-composition. *stop(spec)* returns the same results as checking *spec* until and including the first error, but only errors afterwards. *second(spec)* returns the same results as *spec* except it does not report the first error. *twice(spec)* does not report on the isolated errors when checking *spec* but only on two occurring consecutively. *quarter(spec)* reports an error when *spec* produces more than 25% of errors compared with the total number of observations so far. Different methods can be combined, for instance *stop(second(spec))* returns the same results as checking *spec* except it does not report on the first error and after the second error it only returns errors. See Table 1.

spec	stop (spec)	second (spec)	twice (spec)	quarter (spec)	stop (second (spec))
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	1	0	1
1	0	1	1	1	1
1	0	1	1	1	1
1	0	1	1	1	1
1	0	1	1	1	1
0	0	0	1	1	0
1	0	1	1	1	0
0	0	0	1	0	0
0	0	0	0	0	0

Table 1. Unary composition methods.

Consider some examples of binary methods, combining specifications *spec1* and *spec2*. *spec1*∧*spec2* reports an error when at least one of *spec1* or *spec2* does. *spec1*→*spec2* checks *spec1* until it detects an error, then from the next observation it starts to check *spec2*. *spec1*↔*spec2* checks *spec1* until it produces an error, then it starts checking *spec2*, however when *spec2* itself detects an error then we resume checking *spec1*, and so on. So checking switches back and forth between *spec1* and *spec2*. One example of composition between unary and binary methods is *quarter(spec1)*→*spec2*: we start checking *spec2* when *spec1* reports more than 25% of errors. See Table 2.

All composition methods were presented based on their semantics: how to calculate the results of checking the specification based on the outcome of checking its components. We should be also able to show how such effects can be obtained by symbolic manipulation on component specifications and their declaration context (variables). This topic will be the subject of a companion paper.

spec1	spec2	spec1 ∧ spec2	spec1 → spec2	spec1 ↔ spec2
1	1	1	1	1
1	1	1	1	1
0	1	0	0	0
1	1	1	1	1
0	0	0	0	0
1	0	0	0	1
1	1	1	1	1
1	1	1	1	1
0	1	0	1	0
1	0	0	0	0
1	1	1	1	1

Table 2. Binary composition methods.

7. Specification-to-Verifier Generator

Instead of writing the verifier program by hand we decide to generate it from the specification it should check. Here we discuss the working of the generator by presenting its inputs and corresponding outputs (edited for presentation purposes) for three examples based on the list editor. The generated verifier programs are written in AWK, which is a special-purpose language for text processing. The language has been chosen for two main reasons: it allows to write such programs conveniently, by addressing directly their particular purpose, and it provides the execution environment to carry out verification (the AWK interpreter).

The input to the generator program is the name of the declaration file and the specification the verifier is supposed to check. Below, we can see the generator program *gen* invoked from the command like. Its first argument is *editor*, the name of the file that contains declarations of all variables relevant for the editor example, like *pos*, *ln* and *list*. The second argument is `$1!="left" || pos>1`, the specification the verifier is supposed to check (cursor cannot be moved prior to the first position), in quotes to prevent its interpretation by the command processor.

```
$ gen editor '$1!="left" || pos>1'
```

The output from the generator is the verifier program written in AWK, to carry out checking of the specification. The program is displayed on the console, as below:

```
BEGIN          {pos=1}
!($1!="left" || pos>1) {exit(FNR)}
$1=="right"    {pos=pos+1}
$1=="insert"   {pos=pos+1}
$1=="left"     {pos=pos-1}
$
```

As all AWK programs, it consists of several (here five) condition/action rules. Processing is done for every line of input: if the line satisfies a condition then the corresponding

action gets executed, if the line satisfies several conditions then all corresponding actions get executed, in the order in which they occur in the program. One special condition is *BEGIN*, it is only true before the first line of input. Here, the *BEGIN* condition causes the variable *pos* to be assigned value one. The second condition is true on all lines which have *left* as the first field and the value of *pos* less than or equal one. It is exactly the negation of the specification, describing when the line contains an error. If so, the action *exit(FNR)* causes the program to exit abnormally with the value *FNR*. *FNR* contains the current line number. There are three more actions, depending on the value of the first field: if *right* or *insert* then *pos* is incremented, if *left* then *pos* is decremented. The absence of a condition involving *delete* means no actions is executed in this case.

Consider the second property: cursor cannot be moved past the last position (*post_last*). We initialise *pos* to one and *ln* to zero. The error condition checks if the operation is *right* and *pos* is greater than *ln*. Otherwise, *right* increments and *left* decrements *pos*, *delete* decrements *ln* and *insert* increments both *pos* and *ln*. Here is how the program is generated:

```
$ gen editor '$1!="right" || pos<=ln'
BEGIN          {ln=0; pos=1}
!($1!="right" || pos<=ln) {exit(FNR)}
$1=="right"    {pos=pos+1}
$1=="left"     {pos=pos-1}
$1=="delete"   {ln=ln-1}
$1=="insert"   {pos=pos+1;ln=ln+1}
```

Consider checking if the current element is recorded correctly, *check_el*. The following program checks this specification using the variable *list*. With this variable we recreate the contents of the editor based solely on the invoked operations. *list* is initialised to the empty string "" and then modified according to the operation invoked: *delete* removes the element on the current position, *insert* inserts its argument (value of the second field) at the current position. Operations *left* and *right* do not change the variable. Here is how the verifier program is generated:

```
$ gen editor '
($5!="-" || $4==$3+1) &&
($5=="-" || $5==substr(list,$4,1))'
BEGIN {list=""}
!(($5!="-" || $4==$3+1) &&
  ($5=="-" || $5==substr(list,$4,1)))
 ) {exit(FNR)}
$1=="delete" { list =
   substr(list,1,$4-1)
   substr(list,$4+1,$3-$4)}
$1=="insert" { list =
   substr(list,1,$4-1)
   $2 substr(list,$4,$3-$4+1)}
```

To carry out verification we give the verifier program as input to the AWK interpreter, together with the file containing the execution result. Below we execute the verifier generated above (file *check_el*) on the execution record for the list editor (file *record*), as in Section 3. We use the Bash shell to execute two operations. The first operation is *gawk*, the invocation of the GNU interpreter for AWK; the -f option informs the interpreter that *check_el* is the AWK program. The second operation displays on the screen the contents of the environment variable ?, which holds the exit code of the last invoked operation (according to the Bash shell). Here it displays the value 9, which indicates that line 9 of the execution result contains an error:

```
bash-2.01$ gawk -f check_el record
bash-2.01$ echo $?
9
```

The generator is itself written in AWK. When invoked, it first looks into the specification, given as the second argument in the command line. From this specification it creates the array of all variables used. Second, it examines the contents of the declaration file, given as the first argument in the input line, to find the declarations for all variables in the array. Third, it prints on the screen the *BEGIN* condition with the semicolon-separated list of initialisations of all variables in the array. Four, it prints the negated contents of the specification with the exit action immediately following. Five, for each operation modified by at least one variable in the array it prints the condition `$1=="operation"` and then the semicolon-separated list of assignment expressions, one for each variables modified by the operation, according to its declared effect on this variable.

8. Application: WWW access log

Consider an example from real-life computing: WWW access log, created and maintained by the WWW server. According to the HTTP protocol [2] the file contains a separate line for each request, composed of several tokens separated by spaces: the client's IP address, the client's identity and password information, date and time of the request, the request itself, the response returned to the client and the message-body. Figure 2 shows an example. Programs like Analog [16] read this file and give summaries on its contents, e.g. how many times the server was accessed each day, which files were requested and so on. Our approach is similar, but we look into the file to check if the applications which create and modify this file behaved correctly.

The HTTP protocol presents many properties a well-formed access log should possess, all described informally. For example: "All responses to the HEAD request and all $1xx$, 204 and 304 responses must not include a message-body. All other responses do, although it may be of zero

```
195.92.198.83 - - [07/Mar/2000:00:25:35 +0800] "GET /~rm/goss/models/c/cornish_pasty.html HTTP/1.1" 200 1407
208.219.77.29 - - [07/Mar/2000:00:33:50 +0800] "HEAD /~rm/goss/models/l/lyme.html HTTP/1.1" 200 0
194.128.161.132 - - [07/Mar/2000:01:08:25 +0800] "GET /home/Unuiist/newrh/II/1/3/1/favicon.ico HTTP/1.1" 404 317
216.35.116.96 - - [07/Mar/2000:01:12:19 +0800] "GET /robots.txt HTTP/1.0" 404 276
199.172.149.143 - - [07/Mar/2000:11:26:21 +0800] "GET /raise HTTP/1.0" 301 306
131.114.9.246 - - [07/Mar/2000:21:28:15 +0800] "GET /~tj/pictures/tj1.gif HTTP/1.1" 304 -
192.203.232.241 - - [07/Mar/2000:23:59:22 +0800] "GET /www/image/backgrd-light-2.jpg HTTP/1.0" 206 0
```

Figure 2. Example execution record: fragment of the WWW access log file.

length." [2] (Section 4.3). Such a property can serve as a specification to check about the log file. The specification is presented below. The field number six ($6) contains the name of a method, $NF-1 contains the status code and the last field ($NF) contains the length of a message. Hyphen (-) means the absence of a value.

```
!($6=="\"HEAD || $NF-1<200 ||
 $NF-1==204 || $NF-1==304) || $NF=='-'
```

The Awk program to check this property is below. The program examines every line of the log file and exits with the current line number as soon as the property is violated:

```
!(
 !($6=="\"HEAD || $NF-1<200 ||
  $NF-1==204 || $NF-1==304) || $NF=='-'
) {exit(FNR)}
```

Another property: "Response status codes beginning with the digit '5' indicate cases in which the server is aware that it has erred or is incapable of performing the request." [2] (Section 10.5). Here we may like to check if a server-side error occurs more than a certain number of times within a given time interval. Such a check could help to monitor the performance of the server. The verifier needs four variables: *count* to calculate the number of error occurrences, *from* and *to* to represent the bounds of the time interval, and *max* for the maximum number of errors allowed. *count* is initialised to zero and is incremented with every error, i.e. on all lines where the status code is at least 500 and which time-stamp is between the bounds. This, however, requires to extend the use of the plain operation names in variable declarations, into logical properties. The specification is simply `count<=max`. The (hand-written) verifier program requires the function *before* to decide about chronological ordering between two time-stamps. The program is below.

```
BEGIN {count=0}
!(count <= max) {exit(FNR)}
(before(from,$4) && before($4,to) &&
 $(NF-1)>500) {count=count+1}
```

The final property, although not enforced by the HTTP protocol, is that successive entries in the log file have strictly increasing time entries: *before(time,$4)*. *time* is a time variable which holds the time value of the previous access, initially the origin of time. Here is the program:

```
BEGIN            {time = origin}
before($4,time)  {exit(FNR)}
true             {time = $4}
```

9. Conclusions

We discussed result-verification as an alternative to implementation-verification, its limitations, advantages and the technical context to make this technique usable in practice. The main limitation is the fact that result-verification reasons about a single execution run of a program, whereas implementation-verification considers all possible runs. Advantages are: result-verification has fewer preconditions for its applications in practice, gives more opportunities for automation and, based on the execution record not the implementation details, is particularly suitable for complex systems. The technical context includes the following issues: (1) How to obtain the execution record? The record is a text file containing observations made at different points of the execution, line after line: operation invoked and the values observed about the state. It is either obtained as the usual log file produced by a program, or by specially designed monitoring software. (2) How to specify the properties we want to check about the record? The specification is a first-order predicate build from the variables, functions, relations and limited kind of quantification. The variables allow referring to individual fields within the lines of the record, also how their values change for different invoked operations. We introduced the syntax and formal semantics for such specifications. (3) How to build such properties from the existing properties? We put forward several methods for specification composition, depending if checking produces a yes/no answer for the whole record or a yes/no answer for every observation within the record. (4) How to carry out verification for a given record and specification? Verification is carried out automatically and the verifier program is generated from the specification (not written by hand). The generated program is written in AWK [12]. The AWK interpreter is both applied to generate this program and carry out the actual verification. We

showed two examples: the list editor (created by us) and the WWW access log [2] (created by the WWW server).

Result verification receives a growing interest in the technical literature. The concept of a program correctness checker, an algorithm for checking the output of a computation was introduced in [1] and studied for a range of numerical, sorting and matrix problems. Efficient checkers and correctors were introduced in [17]. Both papers concern the algorithmic foundations, for the case of non-reactive software. In the applied stream, [3] demonstrates how program-checking can reduce the amount of verification work using compiler back-ends (generated by unverified construction tools) and [15] presents an industrial application of mechanised result-verification for the automatic train protection system. The ideas in this paper evolved from the earlier paper [7] which introduced logic-based regular expressions to specify the behaviour of software components, suitable for their checking (as a kind of pattern-matching) at run-time. The fail-stop property in that paper linked on-line result-verification with fault-tolerance [13] and its formal verification (e.g. [6]). Result-verification can be used as a tool to carry out specification-based testing [11], as well as during the normal operation. The background for formalisation of software components in this paper is the coalgebraic approach to semantics [5] and its application to object-oriented classes [4]. The idea of software monitoring as a complement to formal techniques to increase application dependability is reviewed in [14]. `Awk` is one in the growing family of the scripting languages for rapid application development [9]. The language presented here to describe result-based specifications is an example of a domain-specific language [18].

There are several directions we would like to continue this work. First, we would like to study the expressive power of result-based specifications, compared in particular with the regular expressions of [7]. Second, we plan to further study specification composition when the verifier produces a yes/no answer for every execution step (if and where an error occurred) or even a number for every step (if, where and why an error occurred). Third, we would like to study theoretical foundations for result-based specifications and their composition, how to formally reason about them, how to refine one specification into another and so on, all related to the results of checking such specifications about particular executions. Fourth, we would like to show how to reason formally about the specification-to-verifier generator, to place more trust on its outputs and to optimise its result. Fifth, we plan to apply the ideas in this paper to on-line checking, when the execution result is actually changing during verification. One of the issues in this case is verification performance. Finally, we intend to provide more examples of result-based verification involving real world computing, especially in a distributed environment.

References

[1] M. Blum and S. Kannan. Designing Programs that Check their Work. *Journal of the ACM*, 42(1):269–291, 1995.

[2] R. Fielding, J. Gettys, et al. Hypertext Transfer Protocol 1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999.

[3] T. Gaul et al. Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. In *Workshop on Run-Time Result Verification, Federated Logic Conference, Trento, Italy*, 1999.

[4] U. Hensel et al. Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools. In *European Symposium on Programming*, LNCS. Springer, 1998.

[5] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[6] T. Janowski. On bisimulation, fault-monotonicity and provable fault-tolerance. In *Algebraic Methodology and Software Technology*, volume 1349 of *LNCS*, 1997.

[7] T. Janowski and W. Mostowski. Fail-Stop Components by Pattern-Matching. In *Formal Methods for Open Object-Based Distributed Systems*. Kluwer, 2000.

[8] E. Lederer and R. Dumitrescu. Automatic Result Verification by Complete Run-Time Checking of Computations. www.ifi.unibas.ch, 2000.

[9] J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *IEEE*, pages 23–30, March 1998.

[10] RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[11] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *ACM Symposium on Software Testing, Analysis and Verification*, 1989.

[12] A. D. Robbins. *Effective AWK Programming*. Specialized Systems Consultants, Inc (SSC) and Free Software Foundation, 1.0.3 edition, Feb 1997.

[13] R. Schlichting and F. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[14] B. Schroeder. On-Line Monitoring: A Tutorial. *IEEE Computer*, pages 72–78, June 1995.

[15] P. Traverso and P. Bertoli. Mechanized Result Verification: an Industrial Application. In *Workshop on Run-Time Result Verification, Federated Logic Conference, Trento*, 1999.

[16] S. Turner. Analog. www.statslab.cam.ac.uk, Nov. 1999.

[17] H. Wasserman and M. Blum. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[18] D. Wile and C. Ramming. Domain-Specific Languages, Special Issue. *IEEE TSE*, 25(3), 1999.