SHA Formalization

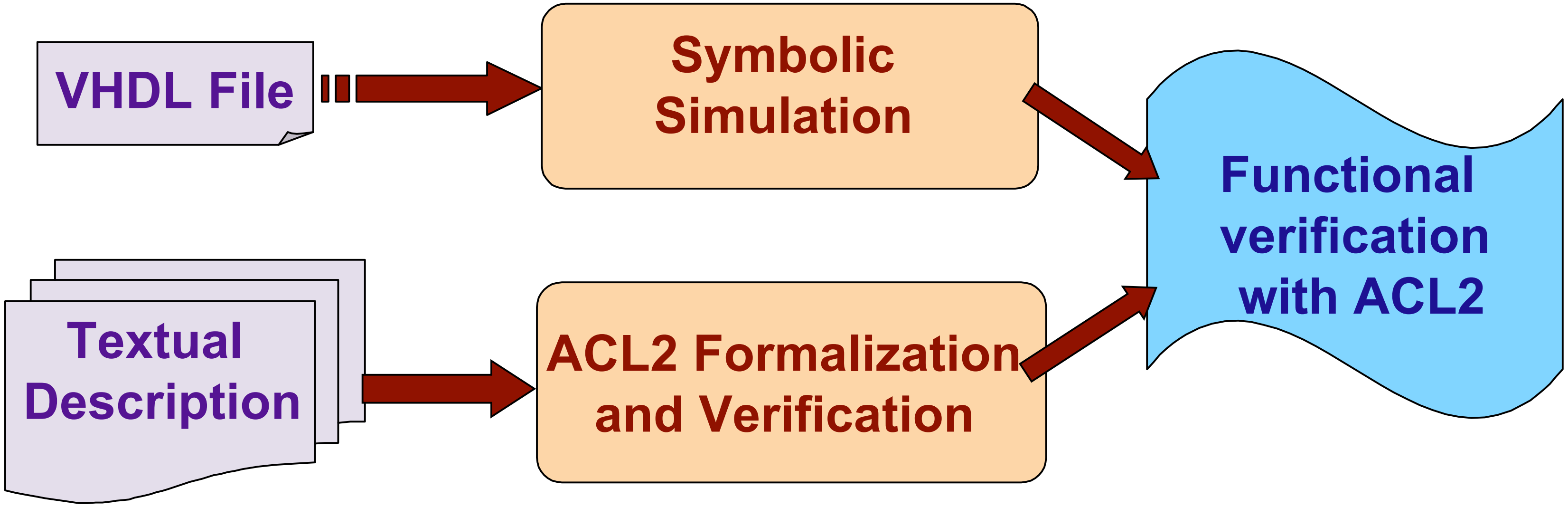
TOMA Diana and BORRIONE Dominique

**TIMA Laboratory - VDS Group,
Grenoble, France**

ACL2 Workshop 2003 Boulder,CO

Project in progress

- Design of a chip for secure transmissions
- Our participation:
  - Validation of the hash block



VHDL File

Symbolic Simulation

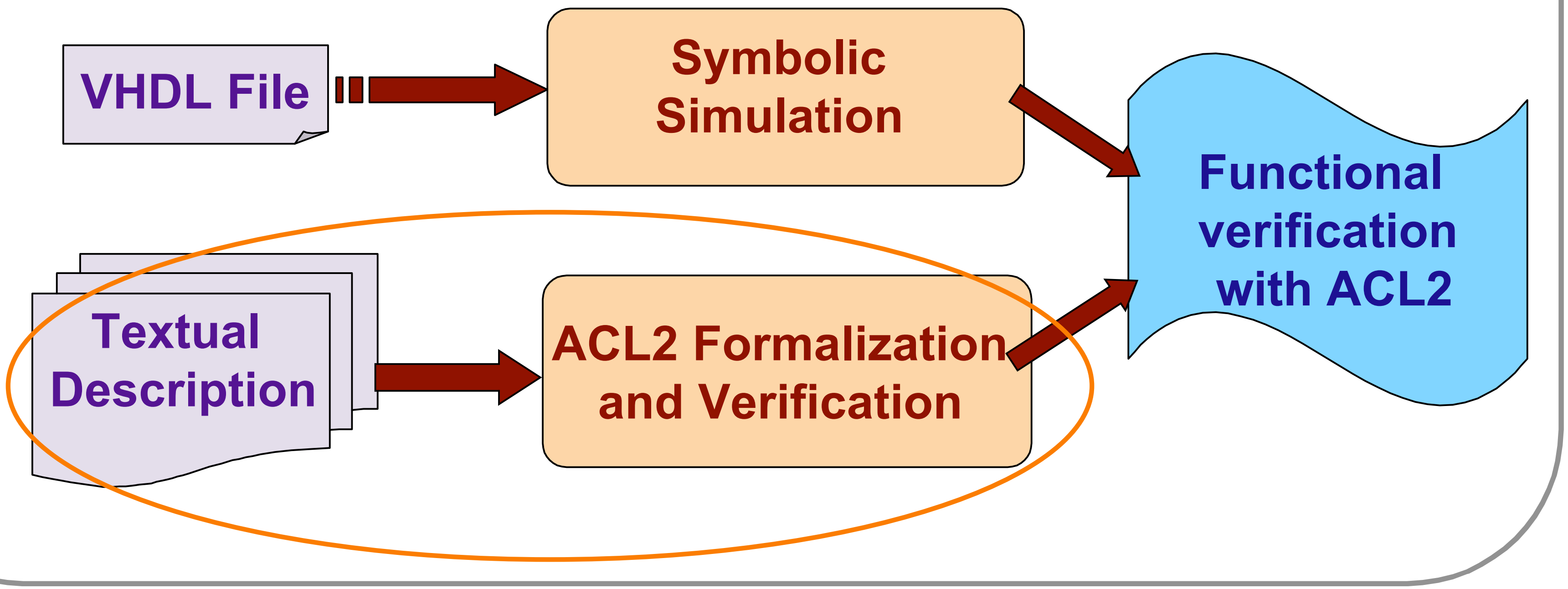
Textual Description

ACL2 Formalization and Verification

Functional verification with ACL2

Project in progress

- Design of a chip for secure transmissions
- Our participation:
  - Validation of the hash block



VHDL File

Symbolic Simulation

Textual Description

ACL2 Formalization and Verification

Functional verification with ACL2

Hash functions

- Among the most widespread cryptographic primitives
- Used in :
  - digital signature schemes together with public key cryptographic systems
  - secret-key Message Authentication Codes (MACs) used in security protocols
  - fast encryption
  - Password storage and verification
  - Computer virus detection

SHA Properties

- Process a message to produce a condensed representation called message digest
- One way hash functions
- Any change to the message will result in a different message digest

Algorithm	Message size	Block size	Word size	Message digest size	Security
SHA-1	$<2^{64}$	512	32	160	2^{80}
SHA-256	$<2^{64}$	512	32	256	2^{128}
SHA-384	$<2^{128}$	1024	64	384	2^{192}
SHA-512	$<2^{128}$	1024	64	512	2^{256}

SHA Properties

- Process a message to produce a condensed representation called message digest
- One way hash functions
- Any change to the message will result in a different message digest

Algorithm	Message size	Block size	Word size	Message digest size	Security
SHA-1	$<2^{64}$	512	32	160	2^{80}
SHA-256	$<2^{64}$	512	32	256	2^{128}
SHA-384	$<2^{128}$	1024	64	384	2^{192}
SHA-512	$<2^{128}$	1024	64	512	2^{256}

SHA Properties

- Process a message to produce a condensed representation called message digest
- One way hash functions
- Any change to the message will result in a different message digest

Algorithm	Message size	Block size	Word size	Message digest size	Security
SHA-1	$<2^{64}$	512	32	160	2^{80}
SHA-256	$<2^{64}$	512	32	256	2^{128}
SHA-384	$<2^{128}$	1024	64	384	2^{192}
SHA-512	$<2^{128}$	1024	64	512	2^{256}

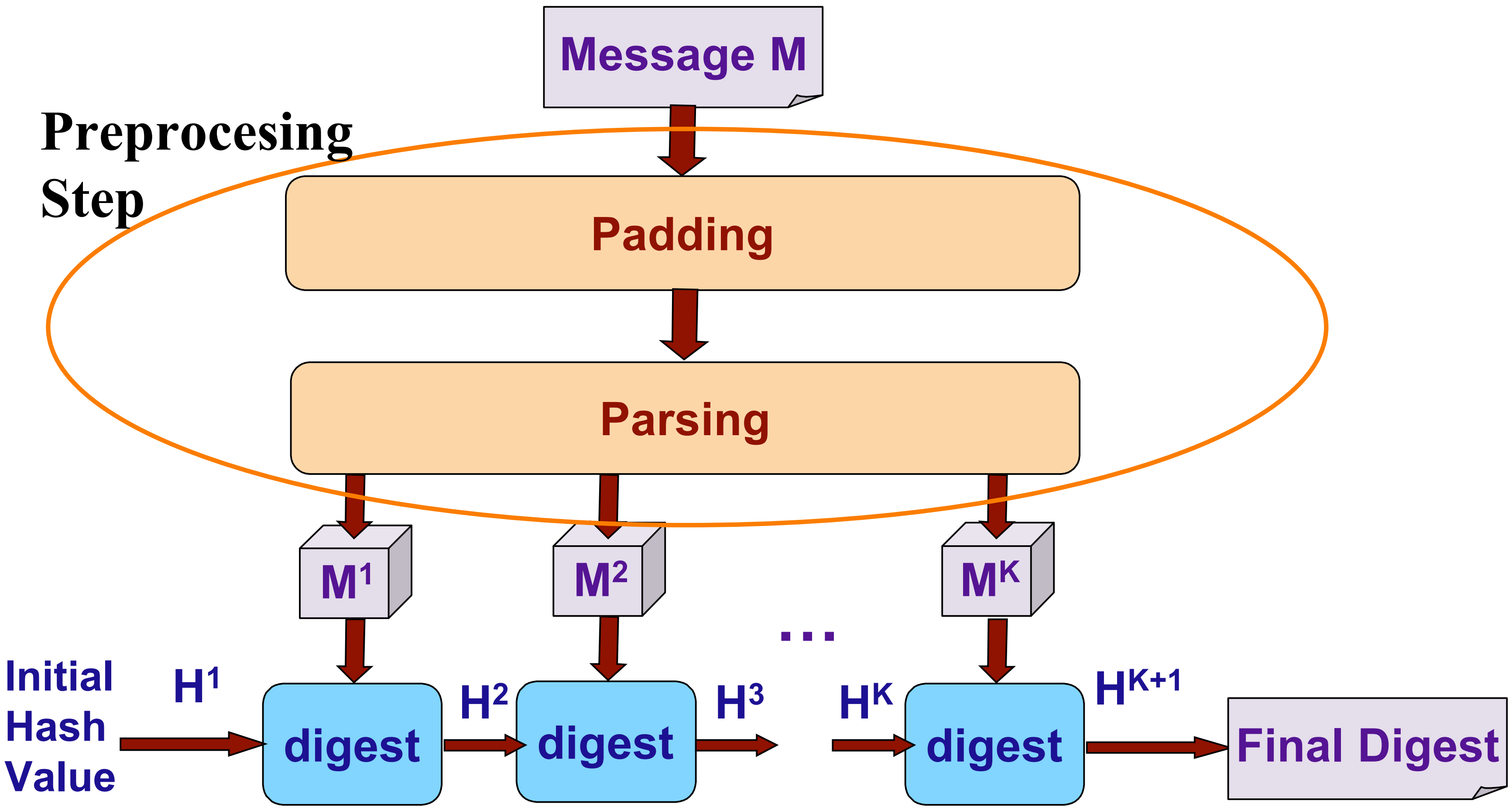
Textual description

- Bit, word (a w-bit string),
- Integer represented as word
- Operations on words:
  - Logical operations: and, or, xor, not
  - Addition modulo 2^w
  - Right shift, circular right shift, circular left shift
- Logical functions on words
- Constants

The big endian convention is used when expressing words.

- A book on bit vectors represented as lists with operations on w-bit words, was created.

SHA Algorithm



Padding

- Extends **M** to a multiple of 512 bits
- Padded messsage is:
  - Append bit 1 to the end of message **M**
  - Append k bits 0, s.t.

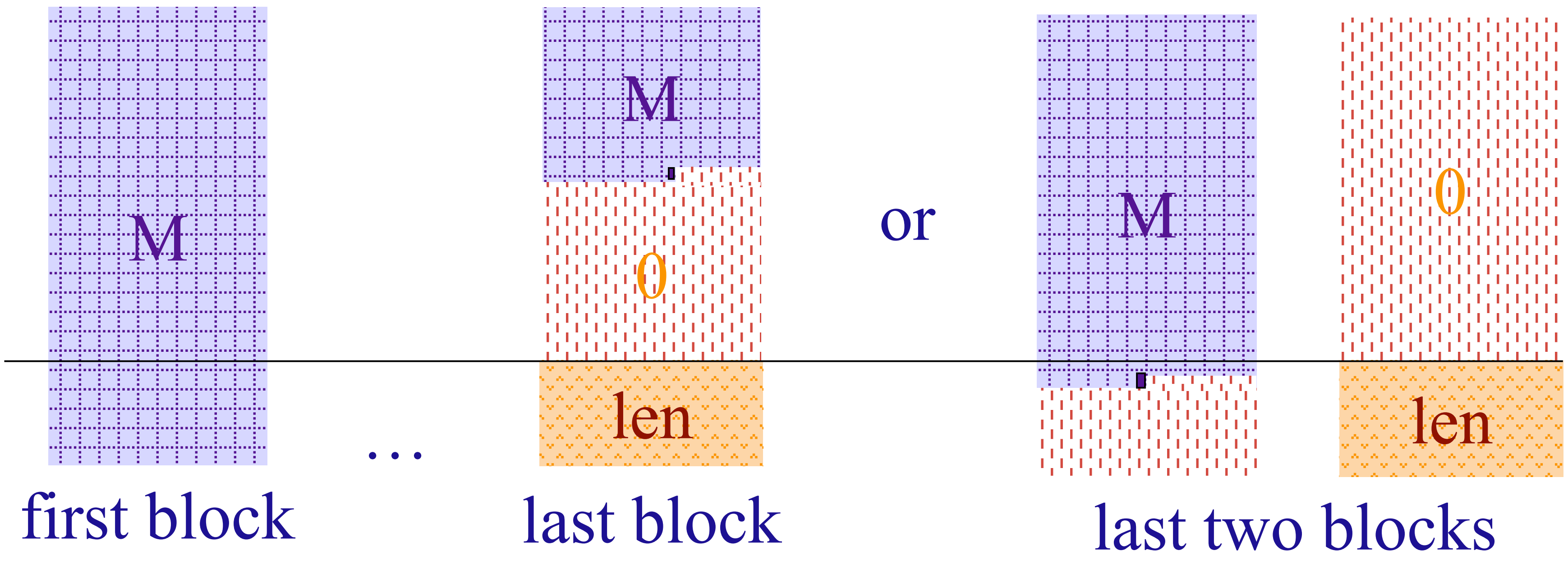
    (*len*+1+k) mod 512 = 448
  - Append the 64-bit binary representation of *len* (*len* is length of **M**)
- Example the 8-bit ASCII message “abc”

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ 1\ \overbrace{00\ldots00}^{423}\ \overbrace{00\ldots011000}^{64}$$

Padding

Two cases:

- on one block : previous example
- on several blocks



Padding Formalization

```
(defun padding (M)
    (if (and (bvp M) (< (len M) (expt 2 64)))
         (if (<= (mod (1+ (len M)) 512) 448)
             (append M (list 1)
                 (make-list (- 448 (mod (1+ (len M)) 512))
                       :initial-element 0)
                 (bv-to-n (int-bv-big-endian (len M)) 64))
             (append M (list 1)
                 (make-list (- 960 (mod (1+ (len M)) 512))
                       :initial-element 0)
                 (bv-to-n (int-bv-big-endian (len M)) 64)))
          nil))
```

- *bvp (m)* recognizes a bit vector
- *bv-to-n (m,i)* forces the bit-vector *m* to the length *i*
- *int-bv-big-endian (i)* transforms the integer *i* into the corresponding bit vector, with the most significant bit on the leftmost-bit position

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))

(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The padded message
is a bit vector

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))

(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))

(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The length of
the padded message
is a multiple of 512

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))
(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))

(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))

(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The length of
the padded message
is greater or equal to 512

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))
(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))

(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64)
                                   (padding M)))
            (len M))))

(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The last 64 bits of the padded message represent the length of **M**

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))
(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))

(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))

(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))
(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The first ***len (M)*** bits of the padded message represent the initial message

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))
(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))

(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))

(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64) (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The next bit after **M** in the padded message marks the end of the messsage

Padding Verification

```
(defthm bvp-padding (bvp (padding m)))
(defthm len-padding-mod
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (mod (len (padding M)) 512) 0)))
(defthm len-padding>=512
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (<= 512 (len (padding M)))))
(defthm last64-padding=len
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (bv-int-big-endian (nthcdr (- (len (padding M)) 64) (padding M)))
            (len M))))
(defthm first-padding=message
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (firstn (len M) (padding M)) M)))
(defthm end-message-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (nth (len M) (padding M)) 1)))

(defthm 0-fill-padding
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (equal (segment (1+ (len M)) (- (len (padding M)) 64)
                                   (padding M))
                   (make-list (- (len (padding m)) (+ 65 (len M)))
                              :initial-element 0))))
```

The bits between the end-of-the-message bit and the last 64 bits are all 0

Parsing

- Splits the padded message into N-bit blocks
  (512 for SHA-1 and SHA-256; 1024 for the others)

```
 (defun parsing (l n)
     (if (and (integerp n) (<= 0 n) (true-listp l))
         (if (or (endp l) (zp n)) nil
             (cons (firstn n l) (parsing (nthcdr n l) n))))
     nil))
```

```
ACL2!>(parsing `(1 2 3 4 5 6 7 8 9) 4)
((1 2 3 4) (5 6 7 8) (9))
```

```
(defun el-of-eq-len (l)
   (if (true-listp l)
       (if (or (endp l) (endp (cdr l))
           (and (equal (len (car l)) (len (cadr l)))
                (el-of-eq-len (cdr l))))
       nil))
```

Parsing Verification

```
(defthm parsing-right-len
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (el-of-eq-len (parsing l n))))

(defthm len-parsing
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (equal (len (parsing l n)) (/ (len l) n))))
(defthm wvp-parsing
    (implies (and (bvp m) (integerp n) (< 0 n)
                  (equal (mod (len m) n) 0))
             (wvp (parsing m n) n)))
(defthm wvp-parsing-padding
    (implies (and (bvp M) (< (len M) (expt 2 64)))
             (wvp (parsing (padding M) 512) 512)))
```

If ***len (l)*** is a multiple of **n**,
the result is a list **L**
of blocks of equal length …

wvp (m,n) recognizes a vector of words, each of length n

Parsing Verification

```
(defthm parsing-right-len
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (el-of-eq-len (parsing l n))))

(defthm len-parsing
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (equal (len (parsing l n)) (/ (len l) n))))

(defthm wvp-parsing
    (implies (and (bvp m) (integerp n) (< 0 n)
                  (equal (mod (len m) n) 0))
             (wvp (parsing m n) n)))
(defthm wvp-parsing-padding
    (implies (and (bvp M) (< (len M) (expt 2 64)))
             (wvp (parsing (padding M) 512) 512)))
```

… where the length of **L** is the result of dividing ***len (l)*** to **n**

wvp (m,n) recognizes a vector of words, each of length n

Parsing Verification

```
(defthm parsing-right-len
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (el-of-eq-len (parsing l n))))
(defthm len-parsing
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (equal (len (parsing l n)) (/ (len l) n))))

(defthm wvp-parsing
    (implies (and (bvp m) (integerp n) (< 0 n)
                  (equal (mod (len m) n) 0))
             (wvp (parsing m n) n)))

(defthm wvp-parsing-padding
    (implies (and (bvp M) (< (len M) (expt 2 64)))
             (wvp (parsing (padding M) 512) 512)))
```

If a bit vector **m** is parsed, the result is a vector of words, each of length **n**

wvp (m,n) recognizes a vector of words, each of length n

Parsing Verification

```
(defthm parsing-right-len
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (el-of-eq-len (parsing l n))))
(defthm len-parsing
    (implies (and (true-listp l) (integerp n) (< 0 n)
                  (equal (mod (len l) n) 0))
             (equal (len (parsing l n)) (/ (len l) n))))
(defthm wvp-parsing
    (implies (and (bvp m) (integerp n) (< 0 n)
                  (equal (mod (len m) n) 0))
             (wvp (parsing m n) n)))
```

```
(defthm wvp-parsing-padding
    (implies (and (bvp M) (< (len M) (expt 2 64)))
             (wvp (parsing (padding M) 512) 512)))
```

After parsing the padded message, the result is a vector of words, each of 512 bits.

wvp (m,n) recognizes a vector of words, each of length n

Message digest

- For each block M^i of 512 bits

1. Parse M^i in 16 words W_i^0, W_i^1,…, W_i^{15}, each of 32 bits and compute

   $W_i^j = ROTL^1(W_i^{j-3} \oplus W_i^{j-8} \oplus W_i^{j-14} \oplus W_i^{j-16})$, $16 <= j < 80$

```
(defun prepare (M-i)
  (if (wordp M-i 512)
      (prepare-ac 16 (parsing M-i 32))
      nil))

(defun prepare-ac (j M-i)
(declare (xargs :measure (acl2-count (- 80 j))))
  (if (and  (integerp j) (<= 16 j) (wvp  M-i 32))
      (cond ((<= 80 j)  M-i)
            ((<= j 79)
              (prepare-ac (1+ j) (append  M-i
                  (list (rotl 1 (bv-xor (nth (- j 3)  M-i)
                                        (nth (- j 8)  M-i)
                                        (nth (- j 14) M-i)
                                        (nth (- j 16) M-i)) 32))))))
      nil))
```

2. Initialize the working variables with intermediate hash value (for M^1 - initial hash value)

Message digest

- For each block M^i of 512 bits

3. Apply eighty times the digest step

```
(defun digest-one-block (hash-values M-i-ext)
   (if (and  (wvp hash-values 32) (equal (len hash-values) 5)
             (wvp M-i-ext 32) (equal (len M-i-ext) 80))
       (digest-one-block-ac 0 hash-values  M-i-ext)
       nil))

(defun digest-one-block-ac (j working-variables M-i-ext)
(declare (xargs :measure (acl2-count (- 80 j))))
    (if (and (wvp working-variables 32) (equal (len working-variables ) 5)
             (integerp j) (<= 0 j) (wvp M-i-ext 32) (equal (len M-i-ext) 80))
        (if (<= 80 j) working-variables
            (digest-one-block-ac (+ 1 j)
               (list  (temp j  working-variables M-i-ext)
                      (nth 0 working-variables)
                      (rotl 30 (nth 1 working-variables) 32)
                      (nth 2 working-variables) (nth 3 working-variables))
               M-i-ext))
        nil))
```

4. Compute the intermediate hash values

Message digest

- The intermediate hash value of the block M^i is the input hash value of the block M^{i+1}
- The result of applying step one to four to all K message blocks represents the message digest of message M.

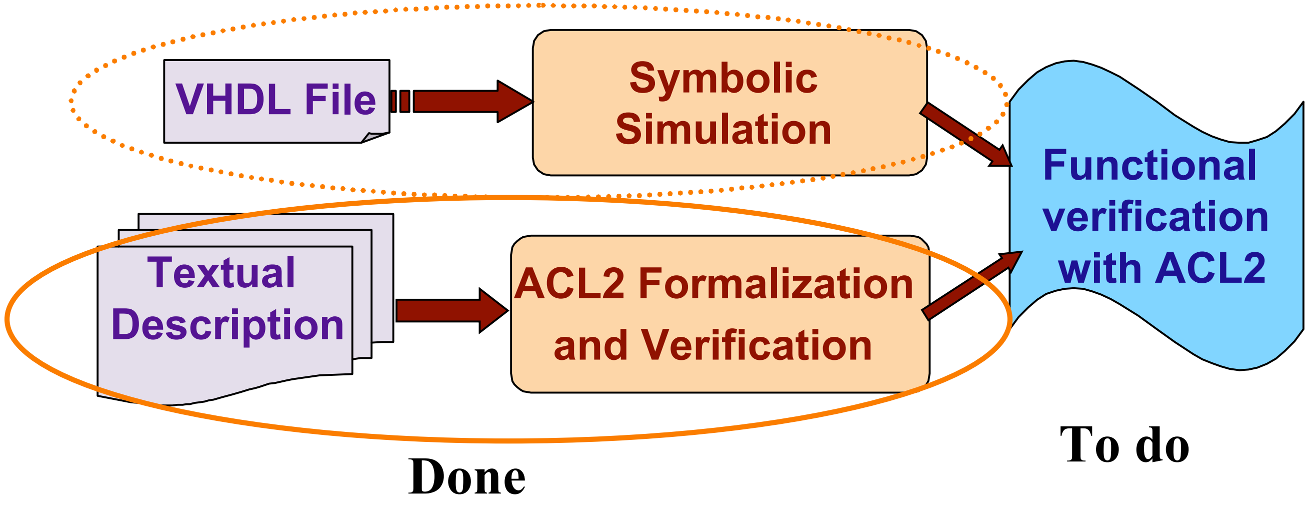
```
(defun sha-1 (M)
  (if (and (bvp M) (< (len M) (expt 2 64)))
      (digest (parsing (padding M) 512) (h-1))
      nil))

(defun digest (M hash-values)
  (if (and (wvp M 512) (wvp hash-values 32) (equal (len hash-values) 5))
      (if (endp M)  hash-values
          (digest (cdr M)
               (intermediate-hash hash-values
                     (digest-one-block hash-values (prepare (car M))))))
       nil))

(defthm wvp-sha-1
   (implies (and (bvp M) (< (len M) (expt 2 64)))
            (and (wvp (sha-1 M) 32) (equal (len (sha-1 M)) 5))))
```

Conclusion

- Formalization of SHA algorithms and verification of safety theorems
- Numeric execution on the tests provided in the standard document
- Development of a book for bit vectors represented as lists with high order bits on the left, closer to the VHDL bit vectors representation.

Future Work

- Prove equivalence between list representation and IHS book
- Prove correctness of SHA implementation


VHDL File
Symbolic Simulation
Textual Description
ACL2 Formalization and Verification
Functional verification with ACL2
To do
Done

Thank you