

# Using Coq and Pvs for certifying properties on the Cache Subsystem of a functional Video-on-Demand Server <sup>\*</sup>

J. Santiago Jorge, Víctor M. Gulías, and Laura M. Castro

MADS group, Department of Computer Science, Universidade da Coruña  
Campus de Elviña, s/n, 15071 A Coruña (Spain)

{sjorge, gulias, lcastro}@udc.es

www.madsgroup.org

## Extended Abstract

In this work, we explore the use of theorem provers to certify particular properties of software. Two different proof-assistants are used to illustrate the method: COQ [1] and PVS [2]. By comparing two theorem provers conclusions about their suitability can be stated. The selected scenario is *part* of a real-world application: a distributed *Video-on-Demand* server [3], implemented with the concurrent functional programming language ERLANG. Large systems are difficult to study as a whole and therefore, they must be decomposed into small pieces in order to study each one separately; later, when we combine other partial results, conclusions on the whole system could be drawn. The development consists on two steps: first, the definition in the COQ and PVS proof assistants of the studied algorithm and some relevant properties to be proved; second, the coding of the theorems. The disadvantage of this approach is that it does not guarantee the correctness of the actual code, but some abstract *model* of it; however this helps to increase the reliability on the system [4].

The piece of software to be verified is (a simplification of) part of the cache subsystem of a *Video-on-Demand* server: the block allocation algorithm. If a media object must be fetched to cache (because of a cache miss), enough space must be booked to load the object from the storage. In order to release some blocks, we have to assure that these blocks are not in use by other pending tasks. As locality in cache is important for performance, blocks of the same media object are in close proximity and we can speak of *block intervals*. A block interval is modeled as a triple  $(a, b, x)$ : the interval between blocks  $a$  and  $b$  (inclusive) has  $x$  pending tasks. A list is used to store the whole sequence of block intervals. Next, (a part of) the Coq model is presented in the first column and the same is translated into PVS in the second one.

```
Inductive interval: Set:=          interval: TYPE = [nat, nat, nat]
  tuple: nat->nat->nat->interval.   seq: TYPE = list[interval]
Inductive seq: Set:= Nil: seq
  | Cons: interval->seq->seq.
```

The core of the studied code is the function `add` which sums up a new block request over an interval sequence, computing the new interval sequence in addition to all the

---

<sup>\*</sup> Work supported by MEC TIN2005-08986 and Xunta de Galicia PGIDIT05TIC10503PR

new blocks allocated. Function `add` must satisfy that, in the resulting interval sequence, the number of requests over the requested block is incremented by one, while all the other blocks remain unchanged. This property is splitted into two theorems provided that `i` equals `n` or not. The auxiliary function `nth` returns the number of pending requests on the `nth` block.

```

Theorem nth_add_1:(l:seq;n,i:nat)   nth_add_1: THEOREM
i=n->                                FORALL (m,n:nat), (l:seq): m = n =>
  (nth n (add i l))=(S (nth n l)).  nth(n, add(m, l)) = 1 + nth(n, l)

Theorem nth_add_2:(l:seq;n,i:nat)   nth_add_2: THEOREM
~i=n->                                FORALL (m,n:nat), (l:seq):
  (nth n (add i l))=(nth n l).      NOT (m = n) =>
                                      nth(n, add(m, l)) = nth(n, l)

```

The representations of the interval sequences should be canonical (in the sense that two different representations always correspond to two different objects) to achieve better space and time behaviour. Intervals  $[(a_1, b_1, x_1), \dots, (a_n, b_n, x_n)]$  are kept sorted  $\forall i, a_i \leq b_i \wedge b_i < a_{i+1}$ . Moreover, space is saved if the sequence is kept compact, i.e.  $\forall i, x_i \neq 0$ , and  $(x_i \neq x_{i+1}) \vee ((x_i = x_{i+1}) \wedge (b_i + 1 < a_{i+1}))$ . And every block in the sequence must have at least one request  $\forall i, x_i \neq 0$ .

Some definitions are introduced (e.g. `pack`, `canonical`) in order to state this canonical representation. We should demonstrate that the application of `add` to an interval sequence in canonical form always delivers a new canonical sequence.

```

Theorem canonical_pack_add:          canonical_pack_add: THEOREM
(l:seq;n:nat) (canonical l) ->     FORALL (l: seq) (n: nat):
  (canonical (pack (add n l))).      canonical(l) =>
                                      canonical(pack(add(n, l)))

```

Theorem provers help to achieve software verification assuring that some relevant properties hold in a program. COQ and PVS have different philosophies: COQ formalizes higher-order logic with inductive types, which gives a rigorous notion of proof and the possibility of extracting functional programs from the algorithmic content of the proof. PVS does not have such a rigorous notion of proof but it incorporates powerful automatic theorem-proving capabilities that reduces the amount of work for the developer.

## References

1. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag (2004)
2. Owre, S., Shankar, N., Rushby, J.M.: The PVS Specification Language. Computer Science Laboratory, SRI International (1995)
3. Gulías, V.M., Barreiro, M., Freire, J.L.: VODKA: Developing a video-on-demand server using distributed functional programming. *Journal of Functional Programming* **15**(4) (2005) 403–430
4. Peled, D.A.: Software Reliability Methods. Springer-Verlag (2001)