Defining and Proving Invariants of Clean Programs*

Máté Tejfel, Zoltán Horváth, Tamás Kozsik

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest
e-mail: matej@inf.elte.hu, hz@inf.elte.hu, kto@elte.hu

Abstract. In a pure functional language like Clean the variables represent constant values; variables do not change in time. Hence it seems that temporality has no meaning in functional programs. However, in certain cases (e.g. in interactive or distributed programs, or in those that use IO) a series of values computed from one another can be considered as different states of the same "abstract object". For this abstract object temporal properties (e.g. invariants) can already be proved. This paper presents a method for defining and proving invariant properties of Clean programs. We utilise Sparkle, a theorem prover designed for Clean. Since Sparkle is not capable of handling temporal logical properties, its original version has been extended to support object abstraction and invariant properties.

1 Introduction

Temporal logic is a language for specifying properties of reactive and distributed systems. It is very useful for proving correctness of (sequential or parallel) imperative programs.

When proving correctness of functional programs, the practicability of temporal operators is not so obvious. In a pure functional programming language a variable is a value, like in mathematics, and not an "object" that can change its value in time, viz. during program execution. Due to referential transparency, reasoning about functional programs can be accomplished with a fairly simple mathematical machinery, using, for example, classical logic and induction (see [19]). This fact is one of the basic advantages of functional programs over imperative ones.

In our opinion, however, in certain cases it is natural to express our knowledge about the behaviour of a functional program (or, we had better say, our knowledge about the values the program computes) in terms of temporal logical operators. Moreover, in the case of parallel or distributed functional programs, temporal properties are exactly as useful as they are in the case of imperative programs. For example, those invariants which are preserved by all components of a distributed or parallel program, are also preserved by the compound program.

From this point of view, certain values computed during the evaluation of a functional program can be regarded as successive values of the same "abstract object". This corresponds directly to the view which certain object-oriented functional languages hold.

* Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr.T037742 and by the Bolyai Research Scholarship of the Hungarian Academy of Sciences.

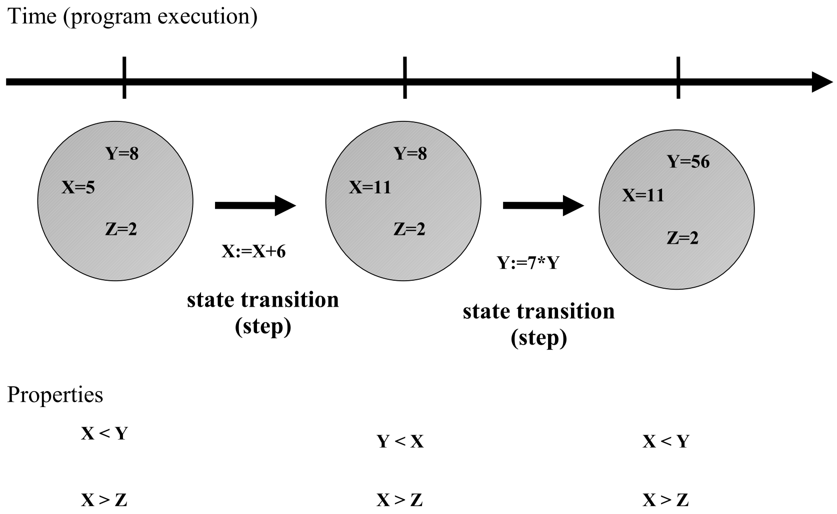


Fig. 1. Temporality (imperative view)

We have chosen Clean [21], a lazy, pure functional language for our research. An important factor in our choice was that a theorem prover, Sparkle [19] is already built in the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly. We extended the basic logic used by Sparkle with temporal operators.

Earlier we have provided correctness proofs about interactive, concurrent (interleaved) Clean programs, namely Object IO processes in [12,13]. However, these proofs were carried out by hand. We argue that the extension of the theorem prover with tools supporting temporal logical operators makes the reasoning about interactive or concurrent Clean programs easier, since temporal logical reasoning can be performed *within* the theorem prover.

For formulating and proving temporal properties of a Clean program, the "abstract objects" have to be determined, that is it has to be specified which functional (mathematical) values correspond to different states of the same abstract object. Therefore, Clean and correspondingly Sparkle have to be extended with some new elements.

There are several kinds of temporal logic, eg. Linear Temporal Logic (LTL), Computation Tree Logic (CTL) etc. We use a UNITY-like special temporal logic [4]. The main operators of this logic are the *invariant*, *unless*, *ensures* and *leads-to* operators. All these operators can be expressed with the help of the "weakest precondition" operator [6,11].

In the rest of the paper first these operators will be shortly described (Section 2). Then the object abstraction method will be introduced through a simplistic example (Section 3). Next it will be explained how temporal propositions can be expressed in

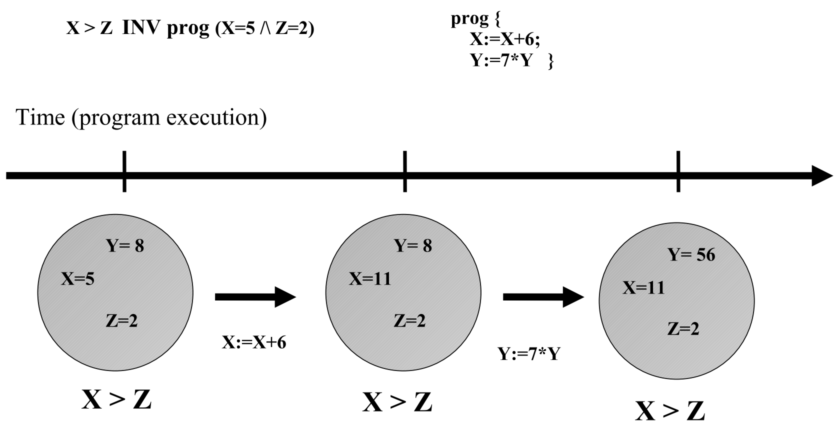


Fig. 2. Invariant property. *P* INV *prog Q*: *P* holds during the execution of program "prog", if *Q* holds initially.

the extended Sparkle framework (Section 4). Finally some proofs will be presented in the extended Sparkle (Section 5).

2 Temporal operators

The temporal logical operators describe how the values of the program variables (the so-called program state) vary in time. To make it more understandable, first an imperative example is considered. This example is shown in Figure 1, where there are three variables (x, y and z) in the state space of the program.

The program is a sequence of two commands, $x := x+6$ and $y := 7*y$ (the order of execution is fixed). The figure shows how the program state varies in time during the execution. In every state it is shown what conditions hold, and what the relationship between them is. For example, the condition $x > z$ holds in every state in the previous example. This will be the base of a very useful temporal operator, "invariant".

A property *P* is an invariant with respect to a program "*prog*" and an initial condition *Q* ("*P* INV *prog Q*") if *P* holds initially (namely it is implied by *Q*) and all the state transitions of the program preserve *P*. As a consequence, *P* holds during the whole program execution if *Q* holds initially. For instance, in the previous example the invariant property "$(x > z)$ INV *prog* $(x = 5 \wedge z = 2)$" holds (see Figure 2). The invariant property is a safety property.

The second temporal operator which is investigated is "unless". Informally the property "*P* UNLESS *prog Q*" expresses that during the execution of the program "*prog*", if once *P* holds, it remains to hold at least until *Q* becomes true. Namely, in any instance of time during program execution, if *P* was once true before and *Q* was never true since then, then *P* holds. Therefore, this property is also a safety property. For instance, in

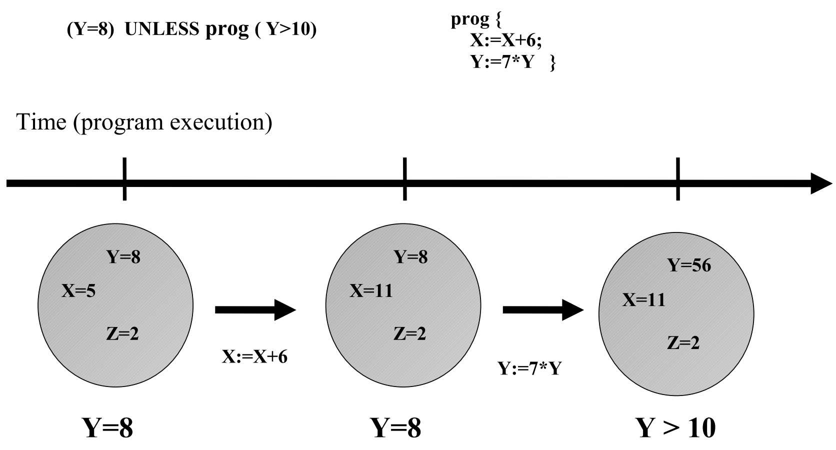


Fig. 3. Unless property. *P* UNLESS *prog Q*: during the execution of program "prog", if *P* holds once, it remains to hold at least until *Q* becomes true.

the previous example the unless property "$(Y = 8)$ UNLESS *prog* $(Y > 10)$" holds (see figure 3).

Figure 4 shows some possible state sequences of a program "*prog*", where property "*P* UNLESS *prog Q*" holds. In the last sequence this property does not hold, because from a state where *P* holds a state is directly reachable where neither *P* nor *Q* holds.

In UNITY progress properties can also be defined using the "ensures" and "leads-to" operators, but in this paper these operators are not investigated.

Until now linear program executions (linear time) have only been investigated, where the order of execution has been known exactly. However, in the case of concurrent execution there many possible execution orders. For example, if the commands $x := x + 6$ and $y := 7 * y$ are executed in a concurrent way, there are two possible execution orders (see Figure 5). (In our model only those kinds of parallel execution are allowed, which have equivalent sequential execution order. This is often referred to as "interleaving" semantics.) Similarly, if three different commands are executed, there are eight different possible executions, and so on. Therefore, it is especially difficult to analyse all possible orders of execution when a temporal property is to be proved. This is one of the reasons why the afore-mentioned temporal properties are expressed with the "weakest precondition" operator. This operator is computable in a more convenient way.

The weakest precondition of a state transition (step) with respect to a postcondition holds for a state "*a*" if and only if the step starting from "*a*" always terminates in a state for which the postcondition holds. The weakest precondition of a step is possible to compute in an automated way: one has to rewrite the postcondition according to the substitution rules defined by the step. For example, Figure 6 shows that if there is a state transition $x := x + 6$ and a state with $x > 10$ is to be reached, then this state transition

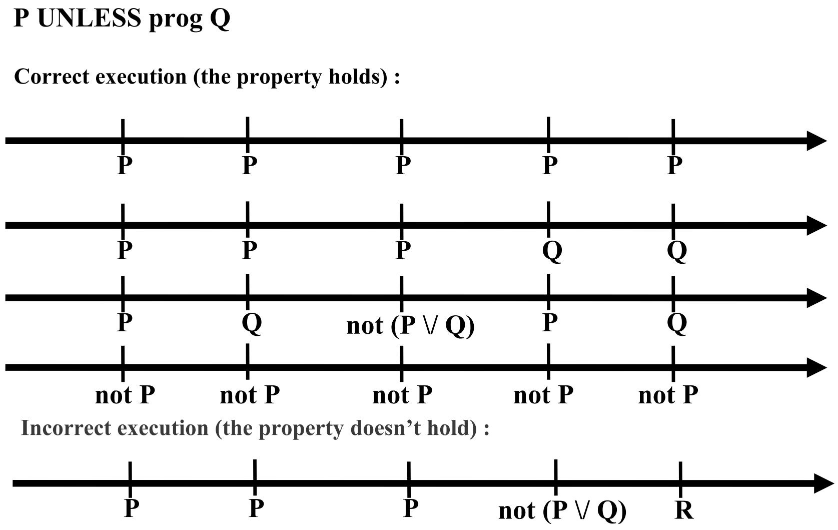


Fig. 4. Unless property

has to be started from a state where $x > 4$ holds. As a consequence, if the state transition is initiated in a step where $x = 5$ holds, it will result in a state with $x > 10$. The details of this rewriting technique can be found in e.g. [14].

The investigated two temporal properties can be formally expressed with the help of the weakest precondition operator in the following way. For a program "*prog*" and properties P and Q the temporal property "P INV *prog* Q" holds, if P follows from Q, and for all state transitions (steps) of the program "*prog*", the weakest precondition of the step with respect to P follows from P. Moreover, for a program "*prog*" and properties P and Q, the temporal property "P UNLESS *prog* Q" holds, if for all state transitions (steps) of the program "*prog*", the weakest precondition of the step with respect to $P \vee Q$ follows from $P \wedge \neg Q$.

3 Object abstraction

Before formulating and proving temporal properties of a functional program, "abstract objects" have to be defined. To denote them two new language constructs are needed. One of the constructs will be used to define which values (functional variables) correspond to different states of the same abstract object. The other construct will mark the state transitions of the program: in each state transition, one or more objects can change their values. State transitions will be regarded as atomic actions with respect to the temporal logical operators.

The first construct is denoted by "`.|.`". It has two arguments: an object identifier and a value identifier, like in "`.|. object_id value_id`". This means that the value identified by `value_id` is associated to the abstract object identified by `object_id`,

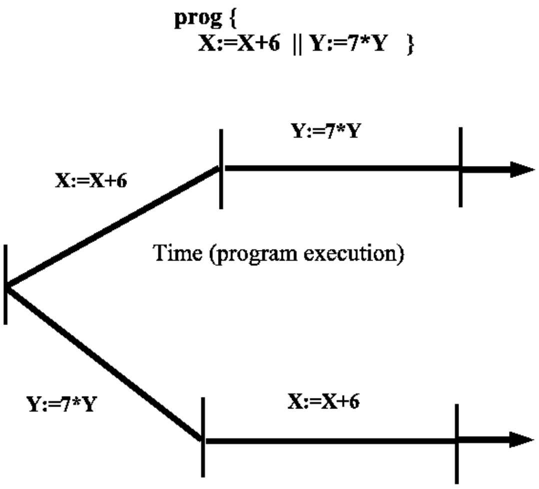


Fig. 5. Parallel execution

that is `value_id` identifies a state of `object_id`. The second construct, used for marking steps, is similar to the let-before (`#`) construct of Clean, hence a similar syntax has been chosen for it: "`.#.`".

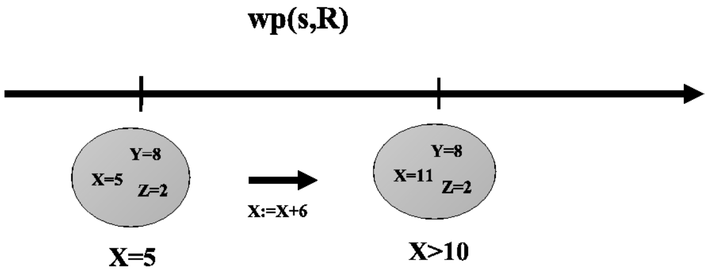
A function definition may introduce objects, only if the body of the function is made up of let-before constructs. The objects are local to the function definition, and the same object name refers to the same object in this scope. (Defining multiple objects with the same name within the same function definition is disallowed.) Objects can only be used in "steps". In every binding within a step the same object can appear at most once on the left, and at most once on the right-hand side. Obviously, the variables constituting the states of an abstract object must be of the same type. Finally, we have to note that currently we only support objects within a single function definition: the variables that make up the states of the object must be defined in the same function.

First a Clean program, which is similar to the previous imperative one, is shown.

```
exec1 x y z
  # x = x + 6
  # y = 7*y
  = (z, x, y)
```

According to the Clean scoping rules this program is equivalent to the following one.

```
exec1 x1 y1 z1
  # x2 = x1 + 6
  # y2 = 7*y1
```



$$wp(X{:=}X{+}6, X{>}10) = (X^{\leftarrow x+6} > 10) = (X+6 > 10) = X > 4$$

$$X = 5 \Rightarrow X > 4$$

Fig. 6. The weakest precondition

```
  = (z1, x2, y2)
```

Here the different x values are considered as different values of the same abstract object (for example obj_x) and similarly, the different y values as the values of object obj_y and finally the value z as the only value of the object obj_z. Also, we consider every let-before expression as a state transition.

Using the two constructs introduced above the description can be rewritten in the following way.

```
exec1 (.|. obj_x x) (.|. obj_y y) (.|. obj_z z)
  .#. (.|. obj_x x) = (.|. obj_x x) + 6
  .#. (.|. obj_y y) = 7*(.|. obj_y y)
  = ((.|. obj_z z), (.|. obj_x x), (.|. obj_y y))
```

For these abstract objects temporal properties can be defined, such as the invariant property, which has been investigated previously for the imperative case.

```
   ∀x∀y∀z (obj_x > obj_z) INV (exec1 x y z)  (z = 2 /\ x = 5)
```

The proof of this property contains three main parts. Firstly, it has to be proved that (obj_x > obj_z) holds initially. Formally (in Sparkle) this will be written as follows.

```
(z = 2 /\ x = 5)
   -> (obj_x_var_0 = x /\ obj_y_var_0 = y /\ obj_z_var_0 = z)
        -> (obj_x_var_0 > obj_z_var_0)
```

Here the objects (obj_x, obj_y and obj_z) were replaced with variables (obj_x_var0, obj_y_var0 and obj_z_var0), so this part is already provable with the original version of Sparkle.

The second part of the proof has to show that the first state transition preserves the invariant property. Formally,

```
(obj_x_var1_old > obj_z_var1_old)
   -> (obj_x_var1 = obj_x_var1_old + 6)
     -> (obj_y_var1 = obj_y_var1_old)
       -> (obj_z_var1 = obj_z_var1_old)
         -> (obj_x_var1 > obj_z_var1)
```

In this proof step two values of every abstract object appear; one before, and one after the state transition. The first values of `obj_x`, `obj_y` and `obj_z` were marked with `obj_x_var1_old`, `obj_y_var1_old` and `obj_z_var1_old`, and their second values with `obj_x_var1`, `obj_y_var1` and `obj_z_var1`, respectively. Furthermore, the relationship between the two values were described, namely the weakest precondition of the state transition was calculated.

The final step is to prove that the second state transition preserves the invariant property. The formal description is similar to the previous one.

```
(obj_x_var2_old > obj_z_var2_old)
   -> (obj_y_var2 = 7 * obj_y_var2_old)
     -> (obj_x_var2 = obj_x_var2_old)
       -> (obj_z_var2 = obj_z_var2_old)
         -> (obj_x_var2 > obj_z_var2)
```

The whole proof contains about twenty proof steps in the extended version of Sparkle. (In the three formal descriptions above a handy feature of Sparkle was applied. Namely, if "free" variables are used in the propositions, these variables will be universally quantified in an automatic way.)

4 A complex example

Let us consider a more complex example. In this example a database of financial transactions is modelled. A transaction is made up of a timestamp, describing when the transaction occurred, and an integer number describing the amount of money transferred in the transaction. The database contains a list of transactions and the overall sum of the amounts transferred in the transactions. The forthcoming definitions are written in Clean.

The representation of the type `Timestamp` is irrelevant in this example, hence this type is defined as abstract. Two operations are needed on `Timestamp`, namely "<" and "`eval`". The type class "<" (from the standard library of Clean) denotes an ordering, while the type class "`eval`" (from the standard library of Sparkle) is used to rule out (partially) undefined expressions.

```
:: Timestamp
instance < Timestamp
instance eval Timestamp
```

The operations have to satisfy the following lemma. If the timestamps t_1 and t_2 are not partially undefined, then $t_1 < t_2$ is not partially undefined, either.

$$\forall t_1, t_2 \in \mathrm{T}imestamp:\ \mathrm{e}val(t_1) \wedge \mathrm{e}val(t_2) \Rightarrow \mathrm{e}val(t_1 < t_2)$$

Type `Transaction` also has “<” and “`eval`” operations. Furthermore, two getter operations have been provided as well. Transactions are ordered according to their timestamps, and a transaction is not partially undefined if (and only if) neither of its two components are.

```
:: Transaction = Tx Timestamp Int
timestamp (Tx timestamp amount) = timestamp
amount    (Tx timestamp amount) = amount
instance < Transaction where (<) (Tx t1 a1) (Tx t2 a2) = t1 < t2
instance eval Transaction
  where eval (Tx timestamp amount) = eval timestamp && eval amount
```

The database—given by the synonym type `DB`—is also an instance of the type class “`eval`”. (In this simple example program we might, but not obliged to, assume that the timestamp of the transactions is a primary key.)

```
:: DB :== (Int,[Transaction])
instance eval DB where eval (sum,list) = eval sum && eval list
```

Our example is based on the following database operations. The first one creates an empty database, the second one adds a transaction to the beginning of the transaction list, the third one removes the first transaction from the list and, finally, the fourth one sorts the transactions according to their timestamps. These operations describe some of the basic state transitions of the databases.

```
newDB :: DB
newDB = (0,[])

insertFirst :: Transaction DB -> DB
insertFirst tx=:(Tx tstamp amount) (sum,txs)
                            = (sum+amount,[tx:txs])

removeFirst :: DB -> DB
removeFirst db=:(_,[]) = db
removeFirst (sum,[(Tx tstamp amount):txs]) = (sum-amount,txs)

sortDB (sum,txs) = (sum, isort txs)
```

For sorting, a simple insertion sort function has been applied, also used by [17].

```
ins :: a [a] -> [a] | < a
ins e [] = [e]
ins e [x:xs] = if (x<e) [x:ins e xs] [e:x:xs]

isort :: [a] -> [a]  | < a
isort [] = []
isort [x:xs] = ins x (isort xs)
```

Now a simple "scenario" application can be developed, built upon the basic operations, which simulates an interactive session between a database management application and an end-user. The input to this scenario is a database and a transaction. First the transaction is inserted into the database, then the resulting database is sorted, finally, the first transaction stored in the (sorted) database is removed.

```
scenario :: DB Transaction -> DB
scenario db tx
   # db = insertFirst tx db
   # db = sortDB db
   # db = removeFirst db
   = db
```

This scenario can be now rewritten in extended Clean, making use of the object abstraction technique.

```
scenario (.|. obj db) tx
   .#. (.|. obj db) = insertFirst tx (.|. obj db)
   .#. (.|. obj db) = sortDB         (.|. obj db)
   .#. (.|. obj db) = removeFirst    (.|. obj db)
    =  (.|. obj db)
```

As it is mentioned earlier, the invariant property of databases is that the first component equals to the total sum of the money transferred by the transactions stored in the second component. In Sparkle, the definitions (functions and predicates) required to formulate a theorem should be given in Clean. We will make use of a function that sums up the amounts of money appearing in a list of transactions.

```
sumUp :: [Transaction] -> Int
sumUp [] = 0
sumUp [tx:txs] = amount tx + sumUp txs
```

Now the invariant property can be formalised:

$$\begin{array}{l} \forall db\, \forall tx : \\ \big(eval\ obj\ \wedge\ fst\ obj = sumUp\ (snd\ obj)\big) \quad \text{inv} \quad \big(scenario\ db\ tx\big) \\ \qquad\qquad\qquad\qquad \big(eval\ db\ \wedge\ eval\ tx\ \wedge\ fst\ db = sumUp\ (snd\ db)\big) \end{array}$$

Let us introduce the following abbreviation:

$$\mathrm{Q}\ =\ \big(eval\ db\ \wedge\ eval\ tx\ \wedge\ fst\ db = sumUp\ (snd\ db)\big)$$

$$\mathrm{I}(x)\ =\ \big(eval\ x\ \wedge\ fst\ x = sumUp\ (snd\ x)\big)$$

The "Invariant" tactic of extended Sparkle will introduce the universally quantified variables *db* and *tx* among the declared variables, and then rewrite the above invariant into the following subgoals.

1. $\forall db_1$: $Q \wedge db_1 = db \;\Rightarrow\; I(db_1)$
2. $\forall db_1\ \forall db_2$: $Q \wedge I(db_1) \wedge db_2 = insertFirst\ db_1\ tx \;\Rightarrow\; I(db_2)$
3. $\forall db_1\ \forall db_2$: $Q \wedge I(db_1) \wedge db_2 = sortDB\ db_1 \;\Rightarrow\; I(db_2)$
4. $\forall db_1\ \forall db_2$: $Q \wedge I(db_1) \wedge db_2 = removeFirst\ db_1 \;\Rightarrow\; I(db_2)$

Since the precondition “Q” may only refer to variables, and not to objects, it can be used as a hypothesis in each generated proposition. These propositions can then be proved in Sparkle—the proofs require about 300 steps.

5 Summary

This paper introduced a method that allows the formulation and proof of invariant properties in the pure functional language Clean. The concept of object abstraction has been presented, which is based on contracting functional variables into objects with dynamic (temporal) behaviour. Language constructs describing object abstraction have been introduced into Clean. We consider this extended Clean language as an intermediate language: programs written in this language are intended to be generated by an appropriate integrated development environment. Support for the new language constructs will thus be provided by a graphical user interface.

Programs written in the extended Clean language are processed by a theorem prover framework. This framework was obtained by enabling Sparkle, the theorem prover designed for Clean, to manage object abstraction and temporal propositions. This paper focuses on invariant propositions.

The advantage of our method is that the constructed proof is represented in a completely machine processable form. As a consequence, not only the program, but also its proved temporal properties and the proofs themselves can be stored, transmitted or checked by a computer. This allows the transmission of the code between two remote applications in a secure way. This transmission can be done, for example, with the Certified Proved-Property-Carrying Code (CPPCC) architecture, which contains the following three main components.

1. The producer of the mobile code adds properties of the code and their proofs.
2. The code receiver will execute the code only after all the checks have been done.
3. The certifying authority reduces the work-load of the receiver.

Figure 7 illustrates the CPPCC architecture. (The detailed presentation of this proof-carrying code technique can be found in [7].)

In the future we will make the Sparkle framework capable of handling additional temporal propositions, namely progress propositions (such as “ensures” and “leads-to” [4]). Furthermore, we plan to implement additional tactics in Sparkle for handling these propositions.

References

1. Achten, P., Plasmeijer, R.: Interactive Objects in Clean. *Proceedings of Implementation of Functional Languages, 9th International Workshop, IFL’97* (K. Hammond et al (eds)), St. Andrews, Scotland, UK, September 1997, LNCS 1467, pp. 304–321.

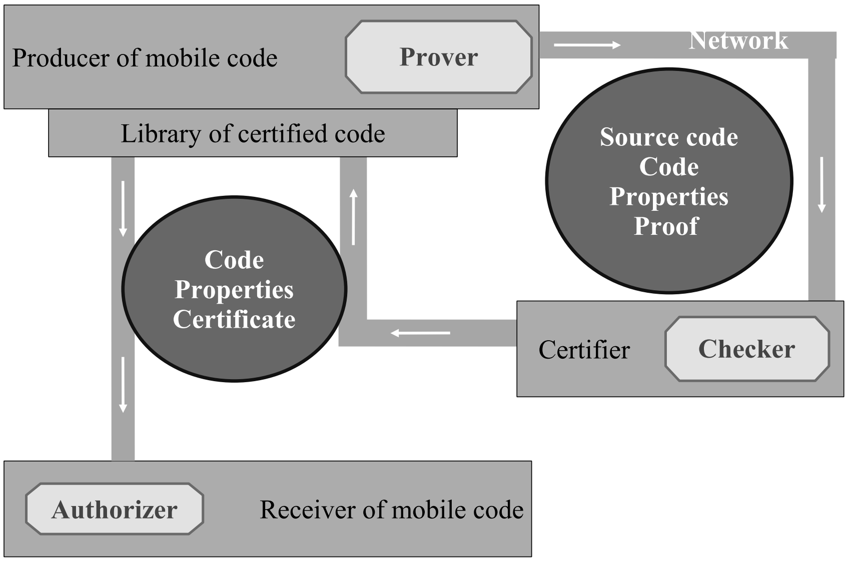


Fig. 7. CPPCC architecture

2. Butterfield, A., Dowse, M., Strong, G.: Proving Make Correct: IO Proofs in Haskell and Clean. *Proceedings of Implementation of Functional Programming Languages*, Madrid, 2002. pp. 330–339.
3. Butterfield, Andrew: Reasoning about I/O and Exceptions. *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 33-48.
4. Chandy, K. M., Misra, J.: *Parallel program design: a foundation*. Addison-Wesley, 1989.
5. Dam, M., Fredlund, L., Gurov, D.: Toward Parametric Verification of Open Distributed Systems. *Compositionality: The Significant Difference* (H. Langmaack, A. Pnueli, W.-P. De Roever (eds)), Springer-Verlag 1998.
6. Dijkstra, E. W.: *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs (N.Y.), 1976.
7. Daxkobler K., Horváth Z., Kozsik T.: A Prototype of CPPCC - Safe Functional Mobile Code in Clean. *Proceedings of Implementation of Functional Languages'02*, Madrid, Spain, Sept. 15-19, 2002. pp. 301-310.
8. Dowse, M., Butterfield, A., van Eekelen, M., de Mol, M., Plasmeijer, R.: Towards Machine-Verified Proofs for I/O *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 469-480.
9. Dowse, M., Butterfield, A.: A Language for Reasoning about Concurrent Functional I/O (Draft) *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 129-141.
10. Home of Clean. http://www.cs.kun.nl/˜clean/
11. Horváth Z.: The Formal Specification of a Problem Solved by a Parallel Program—a Relational Model. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Tomus XVII. (1998) pp. 173–191.

12. Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Proving the Temporal Properties of the Unique World. *Proceedings of the Sixth Symposium on Programming Languages and Software Tools*, Tallin, Estonia, August 1999. pp. 113–125.
13. Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Verification of the Temporal Properties of Dynamic Clean Processes. *Proceedings of Implementation of Functional Languages, IFL'99*, Lochem, The Netherlands, Sept. 7–10, 1999. pp. 203–218.
14. Horváth Z. - Kozsik T. - Tejfel M.: Proving Invariants of Functional Programs. *Proceedings of Eighth Symposium on Programming Languages and Software Tools*, Kuopio, Finland, June 17-18, 2003., pp. 115-126
15. Horváth Z. - Kozsik T. - Tejfel M.: Verifying invariants of abstract functional objects - a case study. *6th International Conference on Applied Informatics*, Eger, Hungary January 27-31 2004.
16. Kozsik T., van Arkel, D., Plasmeijer, R.: Subtyping with Strengthening Type Invariants. *Proceedings of the 12th International Workshop on Implementation of Functional Languages* (M. Mohnen, P. Koopman (eds)), Aachener Informatik-Berichte, Aachen, Germany, September 2000. pp. 315–330.
17. Kozsik T.: Reasoning with Sparkle: a case study. *Technical Report*, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary.
18. de Mol, Maarten. PhD thesis (in preparation), Radboud University Nijmegen.
19. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem Proving for Functional Programmers, Sparkle: A Functional Theorem Prover, Springer Verlag, LNCS 2312, p. 55 ff., 2001.
20. Peyton Jones, S., Hughes, J., et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.
21. Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001. http://www.cs.kun.nl/~clean/Manuals/manuals.html