Proceedings of the 7^{th} International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 155–162.

Proving composed specifications of clean programs in Sparkle-T*

Máté Tejfel

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

Abstract

An important class of software systems uses mobile components: components that are downloaded through the network and integrated into a running application. Clean Dynamics can be used for implementing mobile code in a functional programming language. Our goal is to support the verification of correctness of such applications. The correctness of these applications depend on the properties of the mobile components. A technique called "composing specifications" [1] is applicable in the above case. One can reason about the correctness of a compound system with respect to the properties of some of its components, even if these components themselves are unknown. We have used Sparkle-T [13], an extended version of Sparkle [11], the dedicated theorem prover of Clean [3] for dealing with composed specifications. In this paper a case study is introduced. An invariant property of a compound program is proven with the Sparkle-T system. The proof is based on the invariant properties of its components.

Keywords: Clean, Sparkle, Verification, Composing Specifications

MSC: 68N18, 68N30, 68Q60

1. Introduction

Nowadays a huge amounts of software systems use mobile components. Components that can be downloaded through the network and integrated into a running application. Usually these components are created by a second party, therefore it is extremely important to investigate the correctness of the components, the correctness of the whole running application and a safe manner of the transmission between two remote applications. This safe transmission can be done, for example, with the Certified Proved-Property-Carrying Code (CPPCC) architecture (see [4]).

*Supported by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK and by the "Stiftung AÖU, Wissenschafts- und Erziehungskooperation 66öu2 Programm-Verifikation mit Hilfe algebraischer Methoden."

With the use of such kind of architecture we can guarantee that the mobile components satisfy some required properties. So one can reason about the correctness of a compound system with respect to the properties of some of its components, even if these components themselves are unknown. This technique is called "composing specifications" [1]. It is well applicable in the case of temporal properties. For example, those invariants which are preserved by all components of a program, are also preserved by the compound program.

Temporal properties are very useful for proving the correctness of (sequential or parallel) imperative programs. In the case of the correctness of functional programs, the practicability of temporal operators is not so evident. In a pure functional programming language a variable is a value and not an object that can change its value in time, viz. during program execution.

In our opinion, however, in some cases it is natural to express our knowledge about the computed values of the program in terms of temporal logical operators. Moreover, in the case of parallel or distributed functional programs, and in the case of reactive programs temporal properties are exactly as useful as they are in the case of imperative programs. 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".

For our research Clean [3], a lazy, pure functional language was chosen. One benefit of this choice is that a theorem prover, Sparkle [11] is already built into the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly. Other benefit is that we can apply the Dynamic technique of Clean to compose the mobile component to the running application.

In order to formulate and prove 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. The extended Sparkle system is called *Sparkle-T* [13], where T stands for "temporal". The formal description of how Sparkle-T handles object abstraction and temporal logical properties was presented in [12].

This paper illustrates an extension of Sparkle-T which make it possible to describe composing specification within the theorem prover. The paper introduce a case study which illustrate how can be applied the composing specification technique in the case of Clean programs.

The rest of the paper is structured as follows. First the object abstraction method and the way to express temporal propositions in the Sparkle-T framework is introduced through a simple example (Section 2). Next the technique of composing specifications is illustrated (Section 3). Then a more complex example based on the case study of [10] is presented (Section 4). This section explains how Sparkle-T supports the description of composite specifications. Finally, the conclusion and the discussion of related work is presented (Section 5).

2. Object abstraction

Consider the `sort3` function below which is written in Clean. It takes three integers and returns a tuple containing the same values in increasing order. The definition of `sort3` makes use of another function, `sort2`, which puts two integer values in increasing order.

```
sort3 a_1 b_1 c_1
   # (a_2, b_2) = sort2 a_1 b_1
   # (a_3, c_2) = sort2 a_2 c_1
   # (b_3, c_3) = sort2 b_2 c_2
   = (a_3, b_3, c_3)
```

```
sort2 x y
   | x > y = (y,x)
   | otherwise = (x,y)
```

Note that according to the scoping rules in Clean, this function definition is equivalent to the one, where indexes is not used for `a`, `b` and `c`.

Here the values a_1, a_2 and a_3 may be associated to the same abstract object, e.g. `a_obj`. Similarly, the values b_i and c_i may be associated to `b_obj` and `c_obj`, respectively. So the let-before expressions (denoted by #) will become the state transitions (atomic actions) of this program.

For the demarcation of “abstract objects” Clean has to be extended with two new language constructs. 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 may change their values. State transitions will be regarded as atomic actions with respect to the temporal logical operators, and will be referred to as “steps” in the following sections. The partial description of these language constructs can be found in [12].

3. Composite specifications

Using Clean Dynamics it is possible to exchange a Clean expression between different Clean applications. With this technique components which are created by a second party can be integrate into a running application. For example in the example of the previous section the `sort2` function can be considered as a mobile component which is taken from an other application. This application can write it to a Clean Dynamic with the function `writeDynamic "test" (dynamic sort2) world`. Hereafter this Dynamic can be used for example in the following way.

```
sort3 ::  Dynamic Int Int Int -> (Int,Int,Int)
sort3 (sort2::(Int Int ->(Int,Int))) a_1 b_1 c_1
   # (a_2, b_2) = sort2 a_1 b_1
   # (a_3, c_2) = sort2 a_2 c_1
   # (b_3, c_3) = sort2 b_2 c_2
   = (a_3, b_3, c_3)
```

```
Start world
   # (ok,d,world) = readDynamic "test" world
   | not ok = abort "Could not read"
   | otherwise = (sort3 d 5 2 8, world)
```

The correctness of this kind of application largely depend on its components. But usually the detailed code of the components is not known. However if a safe transmission is applied, for example with the CPPCC architecture ([4]), some properties of the components can be guaranteed. Using composite specification some properties of the compound application can be proved based on the known properties of its components.

Considering our simple example if there is guaranteed the mobile component sorts two integer in increasing order, then the property `sort3` sorts three integer in increasing order can be proved. Namelly for any fun:

$$(\forall v, w, p, q : \; fun \; v \; w \; = \; (p, q) \; \rightarrow \; p \leqslant q)$$
$$\models$$
$$\forall a, b, c, x, y, z : \; (sort3 \; fun \; a \; b \; c \; = \; (x, y, z)) \; \rightarrow \; (x \leqslant y \; \wedge \; y \leqslant z)$$

4. Case study

The investigated method is much more interesting in the case the program has more than one components and temporal properties of the compound program are proved based on temporal properties of the components. In this section a case study is introduced. An invariant property of a compound program is proved based on the invariant properties of its components. The proof is created using the extended Sparkle-T system.

The illustrated case study based on the simple example program introduced in [10]. In this example a transaction is made up of two integer numbers; the first one represents the date when the transaction occurred, and the other one stores 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.

One can develop some basic operations for manipulating the database. A new (empty) database can be created by invoking the function `newDB`. Functions `insertDB`, `removeFirst` and `sortDB` can be regarded as state transition functions, which describe how the state of a database will change. In the FP terminology, these functions compute the “new value” of the database from the “old value”. Function `insertDB` creates a new database from the old one inserting a new transaction to it, `removeFirst` creates a new database from the old one by removing the first transaction, and `sortDB` computes the sorted version of the database. (In this simple example program we might, but not obliged to, assume that the date is a primary key.) The precise specification of the database and the definition of the functions can be found in [10].

Now we can develop a simple “scenario” application, which is built upon the basic operations. One can imagine that this scenario simulates an interactive session between a database management application and an end-user. The input to this scenario is a database and two transactions. First the transactions are inserted into the database, then the resulting database is sorted, finally the first transaction stored in the (sorted) database is removed.

```
scenario db1 t1 t2
   # db2 = insertDB t1 db1
   # db3 = insertDB t2 db2
   # db4 = sortDB db3
   # db5 = removeDB db4
   = db5
```

Here we will introduce a single abstract object, a database, whose consecutive states will be the different `db` values. The three functions `insertDB, sortDB` and `removeDB` can be considered as mobile component created by a second party and integrated into the scenario application in run-time. This can be made using Clean Dynamic for example in the following way, where we have two application. The first one creates a Dynamic from the three functions using the `writeDynamic` function.

```
module WriteDynamic
Start world =
 writeDynamic "test" (dynamic (insertDB, sortDB, removeFirst)) world
```

While the other application use this Dynamic in the scenario.

```
module ReadDynamic
Start world
  # (ok,d,world) = readDynamic "test" world
  | not ok       = abort "could not read"
  | otherwise    = scenario newDB (1,5) (4,7) d

scenario :: !DB !(!Int,!Int) !(!Int,!Int) Dynamic -> DB
scenario db t1 t2 ((insert, sort, remove)
                              :: (!(!Int,!Int) !DB -> !DB,
                                     !DB -> DB, !DB -> !DB))
   # db = insert t1 db
   # db = insert t2 db
   # db = sort db
   # db = remove db
   = db

scenario _ _ _ _ = abort "Wrong type!"
```

Unfortunately the Sparkle system can not handle Clean Dynamic types exactly. Therefore for the proof a simplified version of the above function is used, where the three functions, `insert, sort` and `remove` are simple parameters.

In the case study the database is considered sound if the sum field of the database contains the total sum of the transactions. For the definition of this property two additional function will be used. The function querySum queries the total sum of the transactions from the database. While the function calcSum calculates the total sum of the transactions from the list part of the database.

Applying this functions the soundness of the database can be expressed as an equality between the queried value and the calculated one and can be represented in the following way.

```
soundDB:: !DB -> Bool
soundDB db = (calcSum db == querySum db)
```

We like to proof that the soundness of the database is an invariant in the compound program supposing the components preserve it invariantly. This property can be described more precisely in Sparkle-T in the following way.

```
(soundDB db_o)
    INV(insert t db_o, sort db_o , remove db_o )  (soundDB db_o)
|==
(soundDB db_o)
    INV( scenario newDB t1 t2 (insert, sort, remove) )  TRUE
```

Here the proposition `p INV(f cxs) q` means that proposition p holds invariantly during the evaluation of `f cxs` with respect to the precondition q.

For handling this kind of properties two new proof tactics was integrated into Sparkle-T. One to unfold the complex invariant hypothesis to separate invariant hypotheses refer to the different parameters. In the above case using this tactic three new invariant hypotheses can be created refer to insert, sort and remove.

The other integrated tactic consider the parameter functions as atomic steps and rewrite the above hypotheses into classical logical implications according to the invariant definition based on the weakest precondition operator wp [5]. For the application of this rewrite rule information is needed which part of the inputs and outputs of the parameter functions are considered as object values. This information has to be integrated also into the used Clean Dynamics. After the application of the two new tactic the proof can be accomplished using the invariant tactic introduced in [12] and the original tactics of Sparkle.

5. Conclusions

This paper introduced a method that allows the usage of the proof technique "composing specification" in the case of programs written in an extended version of the pure functional language Clean, which support the concept of object abstraction. This technique makes it possible to reason about the correctness of a compound system with respect to the properties of some of its components, even if these components themselves are unknown. Using Clean Dynamics it is possible to implement applications uses mobile components. In the case of these programs the illustrated technique is essential.

Proofs based on composite specifications are processed by a theorem prover framework, Sparkle-T. This framework was obtained by enabling Sparkle, the theorem prover designed for Clean, to manage object abstraction, temporal propositions, and composed specifications. This paper focuses on composed specification, and presents a case study illustrating the usage of this technique.

Considering related work, a framework for reasoning about file I/O in Clean and Haskell programs is described in [6, 7]. The semantics of file operations is defined in an explicit model of external world-state. Proofs are performed regarding the observable effect of real-world functional I/O. The Sparkle proof-assistant was used for machine-verify these proofs. Sparkle does not support Clean programs with I/O, so the proofs are meta-proofs and the properties of the I/O operations as state-transformers are not formulated in temporal logic. The state of the external world, including the file system can be regarded as an abstract object [8], of which temporal properties may be proved based on the lemmas about file-system operations presented in [6, 7].

The initial results of our research on this topic were also related to functional (Clean) programs using I/O [8]. Then temporal properties of mobile code were addressed in [9]. In [10] the concept of object abstraction was introduced, and invariants of Clean programs were discussed. Then [12] provided the semantical basis for Sparkle-T by extending Sparkle with a formal definition of *invariants* and *unless* properties and the corresponding tactics. Furthermore, the first implementation of the introduced concepts were described and used in some examples. The paper [13] enhances Sparkle-T, the concept of theorems is generalized to allow (classical logical) hypotheses. Hypotheses provide requirements addressing the parameters of programs. Moreover, support for programs containing case-expressions, guards and pattern matching was added to Sparkle-T.

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 safe manner. This transmission can be done, for example, with the Certified Proved-Property-Carrying Code architecture [4].

In the future the Sparkle-T framework will be made capable of handling additional temporal propositions, namely progress propositions (such as "ensures" and "leads-to" [2]). The implementation of additional tactics for the handling of these propositions is also planned.

References

[1] Abadi, M., Lamport, L., Composing specifications. *ACM Trans. Program. Lang. Syst.* 15, 1 (Jan. 1993), 73–132.

[2] Chandy, K. M., Misra, J., Parallel Program Design: a Foundation, *Addison-Wesley*, (1989).

[3] Clean homepage: `http://www.cs.ru.nl/~clean/`

[4] 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), 301–310.

[5] Dijkstra, E. W., A Discipline of Programming. *Prentice-Hall Inc.*, Englewood Cliffs (N.Y.), (1976).

[6] 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*, Lübeck, (September 8–10, 2004) 469–480.

[7] Dowse, M., Butterfield, A., van Eekelen, M., Reasoning About Deterministic Concurrent Functional I/O, *Implementation and Application of Functional Languages: 16th International Workshop, IFL 2004*, Lübeck, Germany, September 8-10, 2004 Revised Selected Papers Springer, LNCS, Volume 3474 (2005), 177–194.

[8] 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), 113–125.

[9] 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), 203–218.

[10] 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).

[11] de Mol, M., van Eekelen, M., Plasmeijer, R., Theorem Proving for Functional Programmers – SPARKLE: A Functional Theorem Prover, In: *Arts, Th., Mohnen M.*, eds.: *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001*, Selected Papers, Älvsjö, Sweden, Springer-Verlag, LNCS 2312, (September 24–26, 2001), 55–71.

[12] Tejfel, M., Horváth, Z., Kozsik, T., Extending the Sparkle Core language with object abstraction, *CSCS 2004, The Fourth Conference of PhD Students in Computer Science Szeged*, Hungary, July 1–4, 2004. *Acta Cybernetica*, Vol. 17 (2005), 419–445.

[13] Tejfel, M., Horváth, Z., Kozsik, T., Temporal Properties of Clean Programs Proven in Sparkle-T, *Proceedings of Central-European Functional Programming School, CEFP 2005*, Budapest, Hungary, July 4-16, 2005, Springer Verlag, LNCS 4164 (2006), 168–190.