# Software Reuse by Model Reification

F. Luís Neves [*] and José N. Oliveira [†]

INESC Group 2361 and Dep. Informática, Universidade do Minho
Campus de Gualtar, 4700 Braga, Portugal
Tel: 351+53-604470
Fax: 351+53-612954

**Abstract**

This paper builds upon earlier work on systematically calculating implementations from *model-oriented* specifications based on abstract data types. The calculation process is called *reification* and is usually done by "hand", requiring some ingenuity and time.

The adoption of model-oriented software specification encompasses the ability to classify and compare data structures. The *reification* of such data models (specifications of software data structures) also provides a powerful reuse mechanism up to isomorphic and inequational refinements. Following the underlying formal calculus (*Sets*), one can easily demonstrate that, for instance, a certain C++ class can be implemented by the composition of already available ones.

Our aim is to develop an automatic "reificator", based on term-rewriting theory, which is intended to take the specification of a data structure, written in the *Sets* notation, and to refine it until a collection of reusable data models are identified.

We also refer that genetic algorithms have the potential to support inequational term-rewriting, such as is required in animating the *Sets* calculus. This topic is briefly presented.

**Keywords:** Software reuse, model-oriented specification, reification, rewrite, genetic algorithms.

**Workshop Goals:** Learning; presenting reuse experiments.

**Workshop Groups:** Reuse and formal methods, tools and environments.

[*]Email: fln@di.uminho.pt; Http: http://s700.uminho.pt/~fln
[†]Email: jno@di.uminho.pt; Http: http://www.di.uminho.pt/~jno

# 1 Background

The authors were involved in the SOUR [1] project for comparing, classifying and retrieving information about large software systems. SOUR attempts to improve software developers productivity by introducing techniques for reusing pre-existing software components. Related research on software component knowledge elicitation and classification using the *Sets* reification Calculus [Oli92a, Oli92b] is reported in [OC93].

# 2 Position

This paper sketches the idea that software reification calculi such as *Sets* [Oli92a, Oli92b] can be of some help in formalizing a simple yet powerful *software reuse discipline* based on a mathematical notion of a software component. In the approach, a software component is understood as a specification of the component's internal state space (described by set-theoretical constructs) together with a pre-/post-condition pair for each action which can change or observe the component's internal state. A repository of such abstract components has been academically built up [Oli91] which follows a *classify by state structure* partial-order based on the *Sets*-term instantiation ordering [OC93] [2].

The *Sets* Calculus is based on set-theory up to isomorphism. A collection of $\cong$-equalities forms the so-called $\cong$-Subcalculus. When partial data-structure transformations are present, abstraction invariants arise according to the so-called $\trianglelefteq$-Subcalculus. (Informally, while $A \cong B$ means that $A$ and $B$ are interchangeable data structures, $A \trianglelefteq B$ means that only $B$ can replace (implement) $A$.)

The *Sets model-reification* process uses the laws of these two subcalculi to derive new data models from existing ones, with correctness warranty.

Attempts to animate the *Sets* Calculus have been made recently [NRO19]. A "reificator" is expected soon made of two fundamental components: first, a description language for input of relevant data such as axioms, models, type of reification and so on; second, the "engine" of the calculus which incorporates, in the current prototype, the hybridization of two theories: genetic algorithms [Gol89] and rewriting systems [Hue80].

# 3 Instantiation and Reuse

The notion of *instantiation* is closely related with the reuse problem. It exploits the capability of identifying expressions (in our case, *Sets* terms specifying software data structures) as particular cases of others.

In this paper's setting, *software reuse* consists basically of applying rewrite rules obtained directly from the *Sets* laws, in order to rewrite (reify) a software model and decompose it in a collection of sub-models which instantiate pre-existent components (see also [OM95] in this respect).

[1] SOUR is the acronym of EUREKA 379 project "Software Use and Reuse" [SS94].

[2] See [OC93] also for details on ER ("Entity-Relationship") diagram inversion and classification using this approach.

### 3.1 Instantiation

We proceed to a brief illustration of our strategy, whereby an instantiation rule will be induced from the concrete example which follows. Suppose that we want to implement the following C++ class which associates to each OR (Object Reference) its type and a set of attribute values:

```
#define MAX_A 3                // Cardinal of _A
enum _A {OR1,OR2,OR3};
#define MAX_D 1                // Cardinal of _D
enum _D {TYPE0};
#define MAX_B 3                // Cardinal of _B
enum _B {ATT1,ATT2,ATT3};
#define MAXSTR 255             // String lenght
typedef char _C[MAXSTR];


class X
{
    private:
        struct
        {
            _A A;                                // A
            _D D;                                // D
            struct {_B B; _C C;} ffBtoC[MAX_B]; // B -> C
        } ffAtoDxffBtoC[MAX_A];                 // A -> D x (B -> C)
    public:
        // ...
};
```

In *Sets*, data structures of this class can be expressed by instatiating the *Sets*-term [3]

$$A \hookrightarrow D \times (B \hookrightarrow C)$$

Now, consider the following C++ class, which is presumed to be available in the repository [4]:

```
class Y
{
    private:
        _A setA[MAX_A];           // 2^A
        struct
        {
            _A A;                 // A
            _B B;                 // B
            _C C;                 // C
        } ffAxBtoC[MAX_A*MAX_B];  // (A x B) -> C
```

[3] $A \times B$ denotes the cartesian product of $A$ and $B$; $A \hookrightarrow B$ denotes the set of all partial mappings from $A$ to $B$. See [Oli92b] for details.

[4] As a software component implementation, the code of this class should include all the intrinsic functionality, which is omitted here for economy of space.

```
    public:
        // ...
};
```

The corresponding *Sets* specification is the following expression: $2^A \times ((A \times B) \hookrightarrow C)$. From [Oli92b] we know that

$$A \hookrightarrow (B \hookrightarrow C) \quad \trianglelefteq \quad 2^A \times ((A \times B) \hookrightarrow C) \tag{1}$$

holds, as a particular case of instantiating $D = 1$ in

$$A \hookrightarrow D \times (B \hookrightarrow C) \quad \trianglelefteq \quad (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{2}$$

that is to say [5]:

$$\begin{array}{rcll} A \hookrightarrow 1 \times (B \hookrightarrow C) & \trianglelefteq & (A \hookrightarrow 1) \times ((A \times B) \hookrightarrow C) & \text{by (2)} \\ & \trianglelefteq & 2^{(A \times 1)} \times ((A \times B) \hookrightarrow C) & \text{by law } A \hookrightarrow B \trianglelefteq 2^{(A \times B)} \\ & \cong & 2^A \times ((A \times B) \hookrightarrow C) & \text{by law } A \times 1 \cong A \end{array}$$

which shows that Class Y can be reused in order to implement Class X.

### 3.2 Reuse

The reuse concept may be compared to the resolution of a mathematic equation (or inequation). Suppose we have the following equation

$$x + 2y - z = 0 \tag{3}$$

in which, given the values of $y$ and $z$, we want to know the values of $x$ which satisfy the equation. Using some basic mathematical rules, we can write $x$ *in function of* $y$ and $z$, i.e., by rewriting (3) in the form $x = f(y, z)$. In the above equation, this is equivalent to having

$$x = -2y + z \tag{4}$$

From this point on, we just have to use the well known arithmetical operations to finally obtain the $x$ value.

This is the approach we propose for *reuse*, in particular for data models reuse, i.e., the notion of writing something *in function of*. From the "reificator" point of view, we will have that a set of *Sets* models may be kept in a **repository**, which is basically a store of reusable material [Oli19]. This is much in the tradition of the classical handbooks in Mathematics, for example in helping to solve integral/differential equations. In the above example, reasoning will proceed by instantiating variables $y$ and $z$ in terms of available information in the repository. The striking difference is that we are dealing with discrete mathematics, a topic rarely found in traditional engineering curricula, which are too much concerned with the mathematics of continuity to leave room for the mathematics of discreteness [Cua94]. This may explain why many "conventional" engineers are so bad software designers.

[5] Note that MAX_D = 1, which in *Sets* is equivalent to saying that $\_D \cong 1$.

## References


[Cua94] J. Cuadrado. Teach Formal Methods. *Byte*, page 292, December 1994.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[Hue80] Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM, 27(4):797-821*, October 1980.

[NRO19] F. L. Neves, J. V. Ranito, and J. N. Oliveira. Implementation Studies about Automatic Model Reification based on the SETS Calculus. 19?? (In preparation).

[OC93] J. N. Oliveira and A. M. Cruz. Formal Calculi Applied to Software Component Knowledge Elicitation. Technical Report C19-WP2D, DI/INESC, December 1993. IMI Project *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete.*

[Oli19] J. N. Oliveira. *A Reification Handbook.* 19?? (In preparation).

[Oli91] J. N. Oliveira. *Especificação Formal de Programas.* Univ of Minho, $1^{st}$ edition, 1991. Lecture Notes for the UM M.Sc. Course in Computing (in Portuguese; incl. extended abstract in English).

[Oli92a] J. N. Oliveira. A reification Calculus for Model-Oriented Software Specification. *Formal Aspects of Computing, Vol.2, 1-23*, 1992.

[Oli92b] J. N. Oliveira. Software reification using the sets calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992.

[OM95] J. N. Oliveira and F. S. Moura. Can Distribution Be (Statically) Calculated? Technical report, DI/INESC, 1995. (In preparation).

[SS94] Systena and Syntax Sistemi Software. Integrated SOUR Software System—Demo Session Manual. Technical report, SOUR Project, 1994. Ver.1.2, © Systena & SSS, Via Zanardelli 34, Rome & Via Fanelli 206-16, Bari, Italy.


## 4 Biography

**F. Luís Neves** holds a degree in *Mathematics and Computer Science* from Minho University (1992). He is currently a research assistant at INESC group 2361 (*Programming and Formal Methods*), sited at the University of Minho Campus, Braga, Portugal. For the last two years, he was fully engaged in the INESC participation in the SOUR Project (EU 379). His main interests are graphical environments, formal methods and genetic algorithms.

**José N. Oliveira** is a senior lecturer at the Computer Science Department of Minho University at Braga, Portugal. He obtained his MSC degree (1980) and PHD (1984) in Computer Science from Manchester University, UK. His main research areas are formal specification of software and program calculi.

He is also a senior member of INESC and leader of INESC group 2361. He was responsible for the INESC partnership in the SOUR project.