Compositionality Criteria for Defining Mixed-Styles Synchronous Languages*

Florence Maraninchi and Yann Rémont

VERIMAG** - Centre Equation, 2 Av. de Vignate - F38610 GIERES
http://www.imag.fr/VERIMAG/PEOPLE/Florence.Maraninchi

Abstract. This is not a paper about compositionality in itself, nor a general paper about mixing synchronous languages. We first recall that compositionality appears in three places in the definition of synchronous languages: 1) the synchrony hypothesis guarantees that the formal semantics of the language is compositional (in the sense that there exists an appropriate congruence); 2) programming environments offer separate compilation, at various levels; 3) the idea of using synchronous observers for describing the properties of a program provides a kind of assume/guarantee scheme, thus enabling compositional proofs. Then we take an example in order to illustrate these good properties of synchronous languages: the idea is to extend a dataflow language like Lustre with a construct that supports the description of running modes. We show how to use compositionality arguments when choosing the semantics of a such a mixed-style language. The technical part is taken from [MR98].

1 Introduction

1.1 About compositionality

The call for participation contains a definition of compositionality. It says: “Any method by which the properties of a system can be inferred from the properties of its constituents, without additional information about the internal structure of the constituents”.

This implies the existence of two domains: the constituents of systems, and their properties. The relationship between these two domains is a function that associates properties to constituents. Constituents may be composed in order to form bigger systems; properties may be combined so that another property is inferred from a set of already given ones.

Moreover, “without additional information about the internal structure of the constituents” clearly states that the function that associates the properties to a constituent is not injective. Hence there exist two distinct systems having the

* This work has been partially supported by Esprit Long Term Research Project SYRF 22703
** Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

Springer-Verlag Berlin Heidelberg 1998
same “properties”. Having the same properties defines an equivalence of systems. And in any context where objects may be composed and declared equivalent, it is of a particular interest that the equivalence be a congruence for the available compositions. This scheme may be instantiated in a number of ways.

1.2 Compositionality and logical properties

One instance is given by associating logical properties to systems. In this case, two problems are of interest:

- Given a program \( P \), which is the composition of two “constituents” \( P_1 \) and \( P_2 \) — i.e. \( P = op(P_1, P_2) \) —, and a property \( \Phi \) we want to prove for \( P \), how to find a property \( \Phi_1 \) holding for \( P_1 \), and a property \( \Phi_2 \) holding for \( P_2 \), such that we can logically “infer” \( \Phi \) from \( \Phi_1 \) and \( \Phi_2 \)?
- Given properties of the program constituents, how can we combine them into a property of the global program (depending on the way constituents are composed)?

1.3 Compositionality, separate compilation and linkers

Another instance of the general scheme is given by separate compilation and linking.

If the system we consider is a program, written in a classical sequential language, and we call “properties” the object code generated by a compiler, then the general scheme for compositionality describes separate compilation and linking. Everybody knows that the executable code of a program written as a set of C source files (or modules) can be built from the set of separately compiled object files, without additional information about the internal structure of the C files. The intermediate form contains all information needed in order to merge to objects codes produced by compiling two source programs. In sequential languages like C, merging consists in putting together the definition of a function (or procedure) and all the calls to this function, which may appear in different source files. Merging is performed according to the names of the objects. Hence the intermediate object code must contain information about names. This symbol table is clearly something that is not needed any more in the executable code one wants to obtain at the end.

In this case “Composition” is the concatenation of source files (provided they do not define the same global objects); “infer” is the linking process.

This instance of the general scheme has the congruence property: there is no context that would allow to distinguish between two source files having the same object code (this seems to be a very strong property, but one can define notions of equivalences for object code, that loosen the strict syntactic identity).

1.4 Compositionality and Synchronous Languages

In the whole process of designing, implementing and using a synchronous language [BB91], compositionality appears in several places: definition of the formal