# Towards a Theory of Objects in Sequentially Constructive Synchronous Programming

Michael Mendler and Marc Pouzet

[1] University of Bamberg
[2] Ecole Normale Superieure Paris

**Abstract.** The synchronous model of computation reduces the programming of deterministic concurrent systems to the programming of stateful reaction modules that operate in lock-step. At each macro-step, also called synchronous instant, each concurrent program module reads inputs from the environment and executes a step function to change internal memory and produce an output which is consumed by the environment during the same instant. To guarantee overall determinacy, current synchronous programming (SP) languages are heavily restrictive: Modules may only communicate through signals, the modules' step functions must be schedulable so that there is essentially only one write access to a signal and each step function is called at most once within a single instant. Programs which cannot be scheduled to satisfy this are considered non-constructive and rejected. Thus, on the face of it, the synchronous paradigm, as embodied in traditional SP languages, seems to preclude object-style component models, which are common in mainstream imperative programming and natural for modular compilation of synchronous programs.

Previous attempts to add objects to SP have been fairly tentative or remained hidden in the intermediate languages of SP compilers. However, the situation may now be changing. Recent work on a scheduling-centred reconstruction of SP, called the sequentially constructive model of synchronous computation (SCMoC), has introduced a key advance. The SCMoC permits multiple sequential writes to a signal variable under an init-update-read scheduling discipline which relaxes the standard constructiveness analysis for an SP program. Considering that a signal is nothing but a rather special case of a shared object, we show how to enrich earlier tentative synchronous object models by pushing the SCMoC scheduling perspective further. We generalise from simple read/write access functions on signals to module tasks on shared state, and from predefined implicit scheduling disciplines to programmer-defined scheduling policies. In this way, we can encapsulate both memory and synchronous code freely into shared objects just as SCMoC signals can be shared modulo init-update-read protocols. This yields an expressive component model that fills an abstraction gap still prevalent in standard SP languages.