VIII
Related work

Figure VIII.1 presents an overview of work related to Céu, pointing out supported features which are grouped by those that reduce complexity and those that increase safety. The line *Preemptive* represents asynchronous languages with preemptive scheduling [9, 29], which are summarized further. The remaining lines enumerate languages with goals similar to those of Céu that follow a synchronous execution semantics.

Many related approaches allow events to be handled in sequence through a blocking primitive, overcoming the main limitation of event-driven systems (column 1 [14, 5, 33, 4, 27]). As a natural extension, most of them also keep the state of local variables between reactions to the environment (column 2). In addition, Céu introduces a reliable mechanism to interface local pointers with the system through finalization blocks (column 8). Given that these approaches use cooperative scheduling, they can provide deterministic and reproducible execution (column 5). However, as far as we know, Céu is the first system to extend this guarantee for timers in parallel.

Synchronous languages first appeared in the context of WSNs through OSM [28] and Sol [27], which provide parallel compositions (column 3) and distinguish themselves from multi-threaded languages by handling thread destruction seamlessly [35, 7]. Compositions are fundamental for the simpler reasoning about control that made possible the safety analysis of Céu. Sol detects infinite loops at compile time to ensure that programs are responsive (column 6). Céu adopts the same policy, which first appeared in Esterel. Internal events (column 4) can be used as a reactive alternative to shared-memory communication in synchronous languages, as supported in OSM [28]. Céu introduces a stack-based execution that also provides a restricted but safer form of subroutines.

nesC provides a data-race detector for interrupt handlers (column 7), ensuring that *"if a variable x is accessed by asynchronous code, then any access of x outside of an atomic statement is a compile-time error"* [19]. The analysis of Céu is, instead, targeted at synchronous code and points more precisely when accesses can be concurrent, which is only possible because of its restricted

Figure VIII.1: Table of features found in work related to Céu.
The languages are sorted by the date they first appeared in a publication. A gray background indicates where the feature first appeared (or a contribution if it appears in a Céu cell).

semantics. Furthermore, Céu extends the analysis for system calls (*commands* in *nesC*) and control conflicts in trail termination. Although *nesC* does not enforce bounded reactions, it promotes a cooperative style among tasks, and provides asynchronous events that can preempt tasks (column 6), something that cannot be done in Céu.

On the opposite side of concurrency models, languages with preemptive scheduling assume time independence among processes and are more appropriate for applications involving algorithmic-intensive problems. Preemptive scheduling is also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [9, 16, 17, 29]. The choice between the two models should take into account the nature of the application and consider the trade-off between safe synchronization and predictable responsiveness.