Hybrid cc, Hybrid Automata and Program Verification
(Extended Abstract)

Vineet Gupta *  Radha Jagadeesan **  Vijay Saraswat *

1 Introduction

Synchronous programming. Discrete event driven systems [HP85,Ber89,Hal93] are systems that react with their environment at a rate controlled by the environment. Such systems can be quite complex, so for modular development and re-use considerations, a model of a composite system should be built up from models of the components compositionally. From a programming language standpoint, this modularity concern is addressed by the analysis underlying synchronous languages [BB91,Hal93,BG92,HCP91,GBGM91,Har87,CLM91,SJG95], (adapted to dense discrete domains in [BBG93]):

- Logical concurrency/parallelism plays a role in determinate reactive system programming analogous to the role of procedural abstraction in sequential programming — the role of matching program structure to the structure of the solution to the problem at hand.
- Preemption — the ability to stop a process in its tracks — is a fundamental programming tool for such systems [Ber93]. Examples of preemption include process suspension (“cntrl-Z”) and process abortion (“cntrl-C”).
- The language should allow the expression of multiple notions of logical time, i.e. any signal can serve as a notion of time.

The design of synchronous programming languages has two distinct pieces: 1) A notion of defaults analyzed at the level of the basic (untimed) concurrent language. 2) The discrete timed synchronous language obtained by extending the untimed language uniformly over discrete time. Defaults allow the expression of the preemption constructs that rely on instantaneous detection of negative information. Furthermore, there are expressiveness advantages to be gained from building the programming language on top of a logic that allows expression of defaults — recent results [GKPS95] suggest that such programs can be exponentially more succinct than programs that do not admit expression of defaults.

* Xerox PARC, 3333 Coyote Hill Road, Palo Alto Ca 94304; {vgupta,saraswat}@parc.xerox.com
** Dept. of Mathematical Sciences, Loyola University-Lake Shore Campus, Chicago, Il 60626; radha@math.luc.edu
Hybrid cc. In earlier work [GJSB, GJSB95], we extended the analysis underlying synchronous programming to build an executable modeling and programming language for hybrid systems, Hybrid cc. Hybrid cc integrates conceptual frameworks for continuous and discrete change, as exemplified by the theory of differential equations and real analysis on the one hand, and the theory of programming languages on the other. As before, the language is built on top of a notion of defaults analyzed at the level of the basic (untimed) concurrent logic language. The hybrid programming synchronous language is obtained by extending the untimed language uniformly over continuous (real) time.

As in synchronous programming languages, various patterns of temporal activity can be defined in Hybrid cc — for example, the instantaneous preemption combinators of synchronous programming. In [GJSB], we provided precise operational semantics and described an interpreter that implements the operational semantics. In [GJSB95], we showed how to build and execute compositional models for various problems described in the literature. We also demonstrated that the process of building models was facilitated by a denotational semantics that allowed a more abstract view of programs as (temporal) constraints.

This paper. In this paper, we demonstrate the relationship of Hybrid cc to the methodology and tools developed in the research on verification of hybrid systems — for example, see earlier proceedings of this conference for a variety of approaches and tools [GNRR93, hyb95].

We aim to establish Hybrid cc as a high-level programming notation for hybrid automata — much as synchronous programming languages are high level notation for discrete automata. Concretely, we establish the expressiveness of Hybrid cc in two ways:

- For any given hybrid automaton, we describe a Hybrid cc program whose only traces are valid runs of the system.
- For any given safety property expressed in (real-time) temporal logic, we show how to write a Hybrid cc program that "detects" if the property is violated.

Furthermore, we aim to make programs written in Hybrid cc amenable to the tools developed for the verification of hybrid systems. For any Hybrid cc program, we build an automaton whose valid runs are precisely execution traces of the program.

2 Hybrid cc— The underlying computational intuition.

Hybrid cc is a language in the concurrent constraint programming framework, augmented with a notion of continuous time and defaults. In this section, we present an intuitive sketch of Hybrid cc — for a detailed formal development, we refer readers to [GJSB].

The (concurrent) constraint (cc) programming paradigm [Sar89] replaces the traditional notion of a store as a valuation of variables with the notion of a store