Can computational logic provide a paradigm for both the specification and implementation of concurrent systems?¹

David Gilbert, City University, London UK
drg@cs.city.ac.uk

Most computational systems in use today comprise a strong element of concurrency, often within a framework of distribution over networks. It is for this reason that I believe that one of the significant challenges facing the computational logic community is to demonstrate that logic can provide a paradigm for both the specification and efficient implementation of such systems, and to convince system designers and developers of the usefulness of this methodology. In this article I propose that a serious effort is made to accept this challenge and suggest that the concurrent constraint paradigm is now mature enough to be used in this role; moreover one way to achieve this goal is to develop tools for automated reasoning about concurrent systems, based on the semantics of this paradigm. For the purposes of this discussion I shall take the definition of concurrency to be that given by Milner [6].

The parallel logic programming languages [9] which were developed a decade ago lie uncomfortably between the categories of efficient implementation languages and languages used for the formal description of concurrent systems. Historically they evolved by modifying sequential programming languages and were mainly used as experimental system programming languages [4, 8]. Had they been intended to be used for formal descriptions then their design would have been shaped by the need for clear and unambiguous semantics powerful enough to permit both the description and reasoning about the behaviour of systems.

The generalisation of the logic programming paradigm to encompass computing constraints over various domains [10] and the adoption of this paradigm to concurrent constraint logic programming (CCLP) [5, 7] is, I believe, the key to progress. There are now many researchers, too numerous to cite here, who are developing concurrent constraint based languages and associated theories of semantics; their work is evidence that this exciting field is rapidly expanding.

In the CCLP paradigm [3] a simple constraint $c$ is a token which may be added to a store or set of such constraints $\sigma$, but may never be removed from the store — i.e.

---

¹In the June 1996 Special Issue of ACM Computing Surveys, Volume 28, Number 2, ISSN 0360-0300, pp 303-305: Symposium on Models of Programming Languages and Computation. C. Hankin and HR Nielsen, Eds.

ACM Copyright Notice
Copyright @ACM 1996 0360-0300/96/0600-0303 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.
updates are monotonic. This update operation is called “tell” and has no effect on the
store if it already contains the told constraint. The state of the store can be queried
by an “ask” operation which succeeds if the store entails the asked constraint. We can
conceptualise the store as a blackboard (potentially distributed), parts of which can be
made private to designated processes, ask operations as reads and tells as writes on data.
In classical concurrency terminology, a process which queries the state of the store with an
ask is a consumer while one which updates the store with a tell is a producer; producers
and consumers communicate via the store of constraints. The communication engendered
is inherently multi-party, since there is no restriction on the number of concurrent ask
operations that may be made on one constraint, nor on the number of concurrent tells on
that constraint, as long as they are consistent with the store. Moreover the paradigm is
classically asynchronous since an ask operation blocks if the information in the store is
not complete enough to entail the asked constraints, whereas the tell operation is eager
and does not block. CCLP languages have inherited the committed choice “don’t care”
operator from CLP, have a parallel operator with interleaving semantics, and a sequencing
operator over processes.

I am exploring the synchronous paradigm together with my colleagues Luboš Brim and
Mojmir Křetínský of Masaryk University in the Czech Republic and Jean-Marie Jacquet
of the University of Namur in Belgium. The communication primitives act on a given
store in the following way: as usual, given a constraint c, the process ask(c) succeeds if
c is entailed by the store, otherwise it is suspended until it can succeed. In our language
synchronisation is achieved by forcing a tell(c) operation to suspend if the constraint c
is not entailed by the store; it can be resumed synchronously with an ask(d) operation
provided that the conjunction of the store and c entails d, in which case the store is
updated with c. A tell(c) operation is only eager if the constraint c is entailed by the
store; however the behaviour of ask(c) is classical in that it succeeds if c is entailed by
the store, otherwise it suspends until it can succeed. The scheme is generalised to permit
the synchronisation of more than two partners, and also by the introduction of a variant
of the tell primitive which leaves the store unchanged. The latter primitive enables us to
describe the re-use of resources in a succinct manner by avoiding the need to explicitly
prevent the re-use of messages about the state of these resources.

The inherently multi-party form of communication, and the natural way in which
communication and data passing are integrated in one coherent theory makes the CCLP
paradigm an excellent candidate for the formal description of concurrent systems. To
date, most of these languages employ asynchronous producers (“eager tells”) for reasons
of efficient implementation. Very little attention has been paid by designers of concurrent
constraint languages to synchronous communication; however, specifications of the use of
limited resources are often best made using a synchronous model of communication in
order to facilitate the task of reasoning about the descriptions, for example the bakery
algorithm. Other work on synchronous communication within the CCLP paradigm is
based on the concept of extending clauses to permit multiple heads and allowing the
simultaneous unification of concurrent atoms with these heads, for example [7]. In contrast,
our approach permits synchronisation on data rather than on processes, and in this respect
is more akin to algebraic theories of concurrency [6].

Our intention is that this language should be used to specify and reason about concurrent
systems, and that tools based on its semantics can be constructed to permit mechanised reasoning about the behaviour of programs. We are planning the construction
of an integrated workbench which will enable the specifier of concurrent constraint based
systems to reason about the behaviour of programs and explore their potential computations. The user of the workbench should be able to define semantics for his own language and then to reason mechanically about the behaviour of programs that he has written. We intend to extend the functionality of the workbench to be able to model the execution of asynchronous constraint programs, and hope that the designers of other concurrent constraint languages will see the building of such specification or program construction environments as a useful goal.

Perhaps the real challenge, however, lies in the design and construction of concurrent constraint languages which permit efficient execution in a distributed environment, and to couple these implementation platforms with tools for automated (or semi-automated) reasoning about programs. Additionally, program transformation and synthesis tools will be required; ideally these should be designed within the same paradigm of concurrent constraint programming. All of this should help towards the ultimate goal of getting the advantages of declarative programming accepted by the wider community. If we cannot achieve this then the danger is that the concurrent logic programming paradigm will remain a curiosity for the majority of system designers and implementers, and the hopes expressed in the 1980’s that logic programming would be widely taken up by industry for the design and construction of large concurrent systems will finally be dashed.

References


