cc — A Generic Framework for Domain Specific Languages (Extended Abstract)

Markus P J Fromherz [*] Vineet Gupta[*] Vijay Saraswat [†]

Abstract

cc programming is a general framework for constructing a wide variety of domain-specific languages. In this paper we show how such languages can be easily constructed using cc, and why cc is particularly suitable for the construction of such languages.

1 Introduction

Increasingly, the widely available cheap and powerful computers of today are being applied in extraordinarily diverse settings — from powering photocopiers and telephony systems and other real-time testing, control and diagnosis systems, to automation of inventory and account management at the neighbourhood video rental store or the phone-order catalog company, to suporting net-based publication and electronic communities. This brings computational scientists (and their tools and techniques) in contact with practitioners of other disciplines whose work touches these diverse areas of human activity — engineers, control-theorists, management scientists, social scientists and engineers, etc.

Increasingly, within such contexts the question arises whether one can improve the usability of these systems by designing them to better reflect the teminology, concepts and reasoning techniques native to the use-domain at hand. For instance, one might develop a framework for the control of systems of electromechanical systems by directly representing symbolic models of components using the vocabulary and techniques of practicing engineers in the field, and providing support for inferring the model of a composite system from the model of its components. This can lead to systems of considerable flexibility and richness for simulation, control, diagnosis and testing [FS95].

Must computational frameworks for the representation of such domain-specific knowledge be developed from scratch for each domain? With its analysis of models, formal languages, representation and denotation it seems natural to look towards logic (specifically, first-order logic) as providing a generic conceptual framework for developing such domain-specific theories. To be useful computationally, however, it also seems appropriate to look for subsets of logic that come equipped with a natural computational interpretation, so that all intermediate steps from logical formulation to procedural representation can be bypassed.

In this paper we argue that the recently emerging framework of *constraint programming* [vHS96] is an appropriate generic framework for domain-specific languages. A *constraint* is essentially just a precisely specifiable relation among several unknowns or variables that represents some partial information about the values the variables can take. Different types of constraints can typically be checked for satisfiability and entailment using very different algorithms. Regardless, constraints enjoy certain natural properties that arise from the nature as carriers of partial information: they are typically additive, non-directional and declarative. Constraint Programming (CP) is the study of computational systems based on constraints. It

[*]Xerox PARC, 3333 Coyote Hill Road, Palo Alto CA 94304; {fromherz, vgupta}@parc.xerox.com
[†]AT&T Labs, 600 Mountain Avenue, Murray Hill NJ 07974; vj@research.att.com

represents a harnessing of the centuries old notions of analysis and inference in mathematical structures with several modern concerns: general languages for computational representation, efficiency of analysis and implementation, tolerance for useful (albeit incomplete) algorithms (tied perhaps to "weak" methods such as search) — all in the service of design and implementation of systems for programming, modeling and problem-solving in different domains.

The essence of this framework is the separation of concerns into two levels. The first level is that of very generally defined *constraint systems* — systems of inference with pieces of partial information based on such fundamental operations as constraint propagation, entailment, satisfaction, normalization and optimization. In addition to the traditional constraint systems that have already been investigated over centuries (such as over the real numbers, integers modulo p), CP brings a focus on a wide variety of systems (arising often from application concerns) ranging from "unstructured" finite domains to equations over trees ("term-unification") to temporal intervals. Increasing attention is being paid to discovering efficient techniques for performing these constraint operations across wide classes of such constraint system, and to discovering common exploitable structure across constraint systems.

Operating above this level is the modeling language which allows the user to specify more information about which constraints should be generated, how they should be combined and processed etc. while exploiting logic based control constructs. The two primary frameworks exploiting these ideas are constraint logic programming [JL87] and concurrent constraint programming (CCP), as operationalized in the cc family of languages [Sar93, SR90, SRP91]. (The latter extends the former by also exploiting the operational interpretation of intuitionistic implication as synchronization, as discussed in the next section.) These languages interact with the first level purely via the basic constraint operations. This provides the user with a very expressive framework (parametric in the underlying constraint system) for generating, manipulating and testing constraints, while preserving their declarative character.

As developed in CCP, this central organizational idea has many ramifications for domain-specific languages. First, cc is completely generic with respect to the constraint system. Thus it is possible for the developer in a given domain to implement his or her own constraint system, taking advantage of the fact that the language implementation will work with the system as intended. This saves the effort of reimplementing the language, and getting used to new syntax. Extensibility is important even within the same domain, since many domain-specific languages are also task-specific languages. When built on top of cc, they can be extended easily for different tasks.

Second, cc programs are an executable form of first-order logic. Thus it is possible to build a suite of reasoning tools on top of cc — partial evaluators, program transformers, verification systems and interpreters. Since the language semantics does not depend upon the constraint system, these tools will continue to work with a different constraint system, making the time of development much shorter.

The simple and compositional nature of cc (discussed further below) also facilitates the addition of new combinators, thus allowing customization of the syntax. In fact, a wholly new syntax can be built up over cc to suit the needs of the current application. The user can then program in this syntax without ever being aware of the underlying cc language. However, the reasoning tools still continue to work on the underlying framework and need not be reimplemented. An example of such a language is CDL, discussed below.

Finally, cc has a sound denotational semantics, which makes it easy to extend cc to domains which need different basic models. Examples of such domains are reactive and hybrid computing. We have been able to extend the basic cc computational model to both these domains. We briefly discuss these extensions below, and show how the resulting extensions can subsume some other languages built for these domains.

The rest of this paper is as follows. We discuss cc in some more depth in order to more firmly fix the reader's intuitions. Then we illustrate the use of cc as a generic modeling framework by discussing two distinct ways in which cc has been used to reflect domain-specific features. First we discuss CDL, a domain-specific modeling language that uses a particular constraint system and uses a somewhat restrictive (albeit natural for the domain) syntax which compiles down to cc. Second we discuss how cc ideas have

been extended into the specific domain of timed and hybrid systems via the introduction of new semantic structure (and combinators exploiting this structure). In addition we show how an expressive (and independently developed) language ESTEREL [Ber93] developed specifically for this domain can in fact be directly translated into timed cc. This demonstrates that the generality of the cc computation model does not come in the way of its expressiveness for this domain.

2 The cc programming framework

CCP is a simple and powerful model of concurrent computation obtained by internalizing the notion of computation as deduction over (first-order) systems of partial information. The model is characterized by monotonic accumulation of information in a distributed context: multiple agents work together to produce *constraints* on shared variables. The cc programming paradigm replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. Computation progresses by accumulating constraints in the store, and by checking whether the store entails constraints. Recently, several concrete general-purpose programming languages have been implemented in this paradigm [JH91, SHW94].

A primitive constraint or *token*, over a given finite set of variables, is a finitary specification of possibly partial information about the values the variables can take. A typical example of a token is a first-order formula over some algebraic structure. Tokens come naturally equipped with an entailment relation: $a_1, \ldots, a_n \vdash a$ holds exactly if the presence of tokens $a_1, \ldots, a_n$ implies the presence of the token a. Thus tokens can combine additively, without any prejudice about their source, to produce other tokens. A set of tokens, together with an entailment relation is called a constraint system[Sar92]. Examples of such systems are the system Herbrand (underlying logic programming), FD [HSD92](finite domains), and `Gentzen` [SJG94].

Example 2.1 The Herbrand constraint system. *Let L be a first-order language L with equality. The tokens of the constraint system are the atomic propositions. Entailment is specified by Clark's Equality Theory, which include the usual entailment relations that one expects from equality. Thus, for example, $f(X, Y) = f(A, g(B, C))$ must entail $X = A$ and $Y = g(B, C)$.*

Example 2.2 The FD constraint system. *Variables are assumed to range over finite domains. In addition to tokens representing equality of variables, there are tokens that that restrict the range of a variable to some finite set.*

Example 2.3 The `Gentzen` constraint system. *For real-time computation we have found the simple constraint system $(\mathcal{G})$ to be very useful. The primitive tokens a_i of `Gentzen` are atomic propositions $X, Y, Z \ldots$. These can be thought of as signals in a computing framework. The entailment relation is trivial, i.e. $a_1 \wedge \ \ldots \wedge a_n \vdash_{\mathcal{G}} a$ iff $a = a_i$ for some i.*

An *agent* has access to a finite set of variables — the basic operations it may perform are to constrain some subset of variables it has access to by posting a token (A::=a), to check whether a token is entailed by ones that have already been posted and if so, reduce to another agent (A::=**if** a **then** A), to create new variables (A::=**new** X **in** A), or to reduce to a parallel composition of other agents (A::=A_1, A_2). Note that the language syntax is completely parametric with respect to the constraint system, this is an important feature shared by all languages in the cc framework.

3 Defining a DSL on top of CC: the Component Description Language

The Component Description Language (CDL) is a language for modeling electro-mechanical machines for simulation, productivity analysis, and scheduling. In particular, CDL has been applied for the configuration of scheduling software in reprographic machines (printers, copiers, etc.), where it allows engineers to describe a machine's capabilities, together with the constraints that have to be satisfied when executing them [FS95].

A sheet inverter, for example, has two capabilities: to invert a sheet of paper, or to pass it through unchanged. Typical constraints are input/output constraints and timing constraints. An inverter may impose input/output constraints on, say, the size of a sheet, and change the sheet orientation on inversion. Timing constraints may include the time to perform the capability and the resources that have to be allocated. Further typical components are feed and output trays, markers, belts, gates, staplers, binders, etc.

It is practical to model reprographic machines as concurrent processes that take streams of inputs and transform them to outputs. This makes a cc language a good basis for this purpose. CDL, constructed on top of cc, simplifies the specification of structural elements, while input/output and timing constraints are defined as in the cc base language. CDL models are compiled easily to cc programs. Informally, the CDL model

```
Component Inverter(int speed) {
   declarations
   Capability Invert() {
      input(in, s_in, t_in); output(out, s_out, t_out);
      s_out == s_in except orientation==Flip(s_in.orientation);
      t_in.start + s_in.length/speed == t_out.start;
   }
}
```

is translated to the cc process

```
inverter(Control, In, Out, Speed) {
   if (Invert:Control, event(S,T_in):In) {
      s_out == s_in except orientation==Flip(s_in.orientation);
      t_in.start + s_in.length/Speed == t_out.start;
      event(S,T_out):Out;
      inverter(Control, In, Out, Speed);
   }
}
```

The primary motivation for CDL was to provide a domain-specific modeling language. Engineers feel more comfortable with this language than with the general-purpose cc language underneath. Constructing CDL in this way then had several advantages. First, by translating CDL to cc programs, we could make use of our expertise and algorithms from reasoning tools for constraint and logic programming languages. Typical reasoning tasks are simulation, automatic composition (e.g., component models to machine models), partial evaluation (e.g., to reduce the model size), and abduction (e.g., to derive all possible capabilities of a composite machine).

Second, many of these reasoning tasks are difficult to automate or specify for general-purpose cc programs. For example, the objective for partial evaluation may be to unfold all input/output constraints in capability models, but not the timing constraints. Traditional reasoning tools often resort to asking the user for help. In contrast, this kind of structural knowledge is available in or can be inferred from a CDL model, which also provides the right level of abstraction for users. Our approach is therefore for the compiler to add suitable annotations to the cc programs, and to modify reasoners such that they take advantage of these

annotations. Reasoning tools that provide hooks for user interaction (e.g., partial evaluators and declarative debuggers) can be readily adapted in this way.

Third, while cc provides a sound theoretical basis for CDL, certain simplified CDL models can also be translated directly to a procedural language if required. This is important when embedding models into a machine's control code. A typical development process in the scheduling context is to build component models in CDL, to compile them to the underlying cc language and partially evaluate and simplify them to a model of the composite machine, to decompile the result to a CDL machine model, and then to translate this model to a representation in the final procedural language.

CDL models are used for scheduling machines in the following way. Desired outputs are matched against outputs produced by the model (e.g., planning by abduction). This results in a trace of component capabilities and their associated timing constraints. The timing constraints are then solved in the context of constraints for other outputs produced concurrently in order to generate a correct schedule.

Models for scheduling can be extended easily to models for design optimization by adding design constraints and a cost model (cf. [KF97]), which requires only minor changes to the CDL compiler and none in the generic tools. This is an example for how the language can be targeted to a different task with different application requirements without changing the underlying language framework.

4 Defining a DSL as an extension to CC: Esterel

The denotational model of cc makes it easy to extend it to some other domains. In the past three years, we have extended it to the real-time and hybrid computing domain [SJG, GJS] as Timed Default cc and Hybrid cc. Extending cc gives these languages the same advantages that cc enjoys — parametricity over constraint systems, declarative behavior, compositionality, reasoning tools etc.

Our first extension of cc was to the real-time domain. We extended cc to allow non-monotonic information detection, by the addition of **else** statements — **if** a **else** A reduces to the agent A if a is not true, and will not be true during the entire computation. Then we extended the model to allow for timed interaction — input now comes as a sequence of constraints, and at each input the program produces an output and is ready for the next input. The only additional operator needed was **hence** A, this starts a new copy of A in each instant after the current one. Thus if a program is viewed as placing constraints upon the evolution of a system, the constraints placed can be understood compositionally as follows:

$\mathtt{A}gents$	$\mathtt{P}ropositions$
a	a holds now
if a **then** A	if a holds now, then A holds now
if a **else** A	if a does not hold now, then A holds now
new X **in** A	there is an instance $A[t/X]$ that holds now
A, B	both A and B hold now
hence A	A holds at every instant *after* now

The combination of non-monotonicity and real-time allows us to define all the combinators defined in ESTEREL, along with several others. The kernel syntax of the pure ESTEREL language [Ber93] is as follows:

```
nothing
emit S
pause
present S then P else Q
suspend P when S
P; Q
loop P end
P || Q
trap S in P end
exit S
signal S in P end
```

For the operational semantics of the combinators, we refer the reader to [Ber93]. Intuitively, `nothing` does nothing; `emit S` causes the signal `S` to be present at the current time instant; `pause` delays for one time tick; the `present` combinator checks for the presence of a signal at the current time tick, triggering one of two possibilities in the two cases; `suspend` freezes its agent `P` whenever the signal `S` is present; ";" sequences its two agents; `loop` restarts its agent whenever it terminates; "`||`" runs its agents in parallel; `trap` immediately aborts the computation of its agent P when it detects T; `exit` signals T; `signal` encapsulates `S` in `P`.

The translation to Timed Default cc is as follows. We first define a Timed Default cc termination process $T(P, d)$ — this takes an ESTEREL process P as argument, and produces a signal d when the processes is completed. Using this we will be able to translate ESTEREL programs into Timed Default cc. For simplicity, the translations below use the following combinators which can be defined in Timed Default cc:

- **next** A reduces to A at the next input instant.
- **first** a **then** A reduces to A at the first instant at which a becomes true.
- **time** A **on** a runs A only at those instants at which a is true.
- $\mathbf{do}\ A\ \mathtt{trap}\ a$ runs A but aborts it, and all its derived agents after the first instant at which A becomes true.

T(`nothing`, d)	=	d
T(`emit S`, d)	=	d
T(`pause`, d)	=	**next** d
T(`present S then P else Q`, d)	=	**if** S **then** T(P, d), **if** S **else** T(Q, d)
T(`suspend P when S`, d)	=	**new** g **in** [**time** T(P, d) **on** g, g, **hence if** S **else** g]
T(`P; Q`, d)	=	**new** d' **in** [T(P, d'), **first** d' **then** T(Q,d)]
T(`loop P end`, d)	=	`true`
T(`P || Q`, d)	=	**new** d1, d2 **in** [T(P, d1), T(Q, d2), **first** d1 then first d2 **then** d, **first** d2 then first d1 **then** d]
T(`trap S in P end`, d)	=	**first** S **then** d, T(P, d)
T(`exit S`, d)	=	d
T(`signal S in P end`, d)	=	T(P, d)

The actual translation of an ESTEREL program P into a Timed Default cc program $\mathcal{T}$(P) is as follows.

$$
\begin{aligned}
\mathcal{T}(\texttt{nothing}) &= \texttt{true} \\
\mathcal{T}(\texttt{emit S}) &= \texttt{S} \\
\mathcal{T}(\texttt{pause}) &= \texttt{true} \\
\mathcal{T}(\texttt{present S then P else Q}) &= \textbf{if } \texttt{S} \textbf{ then } \mathcal{T}(\texttt{P}), \textbf{ if } \texttt{S} \textbf{ else } \mathcal{T}(\texttt{Q}) \\
\mathcal{T}(\texttt{suspend P when S}) &= \textbf{new } \texttt{g} \textbf{ in } [\textbf{time } \mathcal{T}(\texttt{P}) \textbf{ on } \texttt{g}, \texttt{g}, \textbf{hence if } \texttt{S} \textbf{ else } \texttt{g}] \\
\mathcal{T}(\texttt{P; Q}) &= \textbf{new } \texttt{d} \textbf{ in } [\mathcal{T}(\texttt{P}), \texttt{T(P, d)}, \textbf{first } \texttt{d} \textbf{ then } \mathcal{T}(\texttt{Q})] \\
\mathcal{T}(\texttt{loop P end}) &= \textbf{new } \texttt{d} \textbf{ in } [\texttt{d}, \textbf{always if } \texttt{d} \textbf{ then } [\mathcal{T}(\texttt{P}), \texttt{T(P,d)}]] \\
\mathcal{T}(\texttt{trap S in P end}) &= \textbf{new } \texttt{S} \textbf{ in } [\textbf{do } \mathcal{T}(\texttt{P}) \textbf{ trap } \texttt{S}] \\
\mathcal{T}(\texttt{exit S}) &= \texttt{S} \\
\mathcal{T}(\texttt{signal S in P end}) &= \textbf{new } \texttt{S} \textbf{ in } \mathcal{T}(\texttt{P})
\end{aligned}
$$

This illustrates the idea that starting from cc, a general purpose programming language, does not lose us any expressive power when we operate in specialized domains. The advantages we get by viewing ESTEREL as built over cc are a denotational semantics and reuse of the reasoning tools built for cc and Timed Default cc. Another advantage is that we can use ESTEREL for modeling tasks (currently it is used for controllers only). Finally, the extension of Timed Default cc to a hybrid language (described next) would immediately give us a hybrid version of ESTEREL.

The extension to hybrid systems — systems that involve both discrete and continuous change — was similar in flavor, though this time we had to place more requirements on the constraint systems themselves. However, it is possible to extend a constraint system for use with Hybrid cc, this is called a continuous constraint system. The view of program execution remains the same — intuitively at every instant we execute a cc program to determine the output. The fact that physical systems are piecewise continuous allows us to make this intuition computationally feasible — we execute a cc program at time $t = 0$, then another cc program in the open interval following 0. This tells us the output in the interval following 0, as a continuous function, and also tells us the length of the interval. At the end of the interval we again execute a cc program, and then one in the following open interval and so on.

We have used this language to model several physical systems, including parts of the paper-path of a photocopier [GJSB95, GSS95]. We further plan to explore the use of Hybrid cc in graphics animation, to get a language similar in functionality to TBag [ESYAE94].

References

[Ber93] G. Berry. Preemption in concurrent systems. In *Proc. of FSTTCS*. Springer-Verlag, 1993. LNCS 781.

[ESYAE94] C. Elliott, G. Schechter, R. Young, and S. Abi-Ezzi. Tbag: A high level framework for interactive, animated 3d graphics application. In *Proceedings of the ACM SIGGRAPH conference*, 1994.

[FS95] M. P.J. Fromherz and V. A. Saraswat. Model-based computing: Using concurrent constraint programming for modeling and model compilation. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP'95*, pages 629–635. Springer-Verlag, LNCS 97, Sept. 1995.

[GJS] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*. To appear.

[GJSB95] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. Bobrow. Programming in hybrid constraint languages. In Antsaklis, Kohn, Nerode, and Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture notes in computer science*. Springer Verlag, November 1995.

[GSS95] V. Gupta, V. A. Saraswat, and P. Struss. A model of a photocopier paper path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the of the ACM Symposium on Principles of Programming Languages*. ACM, 1987.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[KF97] Ravi Kapadia and Markus P.J. Fromherz. Design optimization with uncertain application knowledge. Submitted to *IEA-AIE'97*. Also published in the Proc. of the AID'96 Workshop on Logic-based Approaches to AI in Design. 1997.

[Sar92] V. A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

[Sar93] V. A. Saraswat. *Concurrent constraint programming*. Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1993.

[SHW94] G. Smolka, Henz, and J. Werz. *Constraint Programming: The Newport Papers*, chapter Object-oriented programming in Oz. MIT Press, 1994.

[SJG] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*. To appear. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[SJG94] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in timed concurrent constraint languages. In B.Mayoh, E.Tougu, and J.Penjam, editors, *Constraint Programming*, volume 131 of *NATO Advanced Science Institute Series F: Computer and System Sciences*, pages 367–413. Springer-Verlag, 1994.

[SR90] V. A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[HSD92] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.

[vHS96] P. van Hentenryck and V. A. Saraswat, eds. Constraint Programming In *ACM Computing Surveys*, Special Issue on Strategic Directions in Computing, in commemoration of the ACM Fiftieth Anniversary, to appear 1997.