

Representing Systems of Interacting Components in EUCLID (Extended Abstract)

K J Dryllerakis and M J Sergot

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK
{kd,mjs}@doc.ic.ac.uk

Introduction

The general problem we want to address is the modelling of systems made up of interacting components, where existing mathematical models are available for the individual components and where a model of the system as a whole can be obtained by specifying how these components interact.

The examples we use for illustration in the second part of the paper are electrical circuits. Here, the elementary components are the various devices in the circuit, each of which has a simple mathematical formulation of its characteristic behaviour; the interactions between these components are determined by the connections in the circuit and the Kirchoff laws.

In the case of electrical circuits the elementary components correspond to actual physical devices. This need not always be so. A train leaves station *A*. It accelerates—at some fixed rate—until it reaches a pre-determined cruising speed. It travels at this constant speed, and then decelerates—again at some given constant rate—to come to rest at station *B*, a known distance from *A*. The motion of the train over the complete journey may be thought of as constructed from three separate components: the acceleration, the cruising, and the deceleration. Given that one knows how to formulate equations of motion for linear acceleration and motion at constant velocity, modelling the behaviour of the train reduces to specifying how the components of the motion fit together. The problem is conceptually straightforward, but sufficiently complicated that it gives many High School students (and some University students) pause for thought. One might want to go on to construct a more complicated model describing how several trains move up and down a railway track. A train may trigger signals that affect the trains behind, cause another train to brake suddenly to avoid a collision, and so on. The point is that the movement of each individual train is still decomposed into acceleration, cruising, deceleration components. The difficulty in constructing the model lies not in devising mathematical models for the acceleration, cruising, deceleration components, but in specifying how these components interact.

We should like to retain standard mathematical models and their associated problem solvers where they are available, and provide a symbolic AI language—a logic programming dialect—to specify the interactions. Within this general context we present the language EUCLID, an instance of an extended form of logic programming very much in the spirit of the CLP(X) scheme of constraint logic programming languages (Jaffar & Lassez 1987).

We term this extension *domain logic programming*. The basic idea is to introduce multiple mathematical ‘domains’ (real and complex numbers, functions, vectors, and so on), each with some pre-defined operations and its own specialised problem solver. EUCLID can be seen as an attempt to integrate modern algebraic manipulation packages (such as Mathematica, Maple etc) with logic programming languages. It may also be seen as a kind of front-end to Mathematica, though this is not a view we would particularly want to emphasise.

We first give a brief description of the language EUCLID and then illustrate its use. As a source of examples we choose the representation of electrical circuits: they are well understood, the required mathematics is simple, and the rôle of a reasoning component can readily be seen. Electrical circuits were also used in the past to show the advantages of the constraint logic programming language CLP(\mathcal{R}) over Prolog—cf. (Heintze, Michaylov, & Stuckey 1987) and (Heintze *et al.* 1991).

A Brief Description of EUCLID

Generalities

EUCLID is a programming language belonging to the logic programming paradigm. Syntax, query mechanisms, and operational semantics are very similar to those of the CLP(X) family of languages (Jaffar & Lassez 1987; Jaffar & Michaylov 1987). The main difference is that EUCLID contains built-in knowledge for a number of domains (including real numbers, intervals and real functions). Programs can be viewed as logical theories with the meaning of certain terms (the *domain terms*) fixed. Constraints are handled by specialised solvers associated with each domain.

The language of EUCLID contains a mixture of domain and classical terms, constraints and predicates. The difference between the two categories is their meaning. The truth or falsehood of constraints is established by the solver; predicates are treated by logical deduction.

It can be shown that the language performs complete and correct computations as long as the mathematical solvers perform complete and correct calculations. However, given the range and generality of the domains we wish to support, correctness and especially completeness of all the underlying solvers is not a realistic demand. This is not a limitation of the framework itself but rather a feature of the domains we want to provide. EUCLID is currently under a prototype implementation in Prolog and C utilizing Mathematica (Wolfram 1991) and CLP(\mathfrak{R}) as its external solvers.

Syntax

EUCLID follows most of the conventions of standard Prolog and CLP(X) systems: predicates/constraints start with a lowercase letter while variables start with an uppercase one.

A *domain* is a collection of terms and constraints (the domain terms and domain constraints) which are created from a first-order language alphabet. The terms are built up from domain variables, constants and a set of pre-defined “functions” for each domain; to avoid confusion we will use the term “operation” for these since “functions” is the name of one of the EUCLID domains. The domains currently supported in EUCLID will be introduced shortly. In EUCLID variables are tagged; they belong to a specific domain: if X is a real variable it can only be substituted by a real constant (e.g. 5), or a more complex domain term (e.g. $5 + \cos(Y)$ where Y is a real variable).

A EUCLID *program* is a set of constraint clauses, representing a logical theory of the problem in hand. Each constraint clause takes the form

Head : $-$ [**Typing**], [**Constraints**], [**Body**]

where **Head** is an atomic formula (*not* a constraint), **Typing** is a possibly empty set of typing statements, **Constraints** is a (possibly empty) conjunction of constraints from the domains supported by the logic and **Body** is a (possibly empty) conjunction of atomic formulae.

EUCLID Domains

Reals The most commonly used domain and the basis for the others is the domain **Reals** of real numbers. Its usefulness and importance has been proven by languages like CLP(\mathfrak{R}) and the applications developed in them. The domain supports real constants (numbers with or without a decimal part), the operations $+$, $-$, $*$, $/$, \wedge , \sin , \cos , \tan , \arcsin , \arccos , \arctan , \log , $\sqrt{\quad}$ and binary constraints between real terms $=$, $<$, $>$, $<=$, $>=$, $\backslash=$. In contrast

with CLP(\mathfrak{R}) non-linear constraints are not delayed but are simplified directly on each step.

Real formulae The domain of real formulae is used to deal with the syntactic form of real functions of one real argument. The term *function* itself is reserved in EUCLID to denote a more complex object (described below). It should be stressed that real formulae are *expressions*: the formula $f(t) = 3 * \cos \omega t$ where ω is a real variable can be represented in EUCLID as $F=(3*\cos(W*T))//T$. The domain of real formulae comes equipped with the operations $+$, \wedge , $-$, $*$, $/$, **limit**, **o**, **integral**, **derivative**. The operations **limit** and \wedge (“power of”) are cross-domain functions of type $F \times \mathcal{R} \rightarrow F$. The symbol “o” denotes composition of functions. **integral** and **derivative** give the standard integral and total derivative respectively of the formula at hand. The value of a formula for a given argument value is obtained by means of an operation **@** of type $F \times \mathcal{R} \rightarrow \mathcal{R}$: for the example above $f(2)$ would be represented by the expression **F@2**.

Intervals Open and closed intervals over the real numbers are supported as a separate domain. Intervals are represented as **cc(X,Y)** for $[x,y]$, **oc(X,Y)** for $(x,y]$, and likewise for the other two cases. Supported relations between intervals include $<$ (before), $>$ (after), **meets**, **in** and so forth.

Functions Intervals are mostly used for building up functions. A *function* is an ordered collection of (interval, formula) pairs such that the interval of each pair is **before** the interval of the next pair. If such a pair of intervals meets at a point both formulae must have the same value for that point. This type of function is sometimes referred to as a multi-branch function. The representation of the function

$$f(x) = \begin{cases} 1 + 3 * x^2 & \text{if } x \leq 0 \\ x - 2 * \omega & \text{if } x > 0 \\ e^x & \text{if } x = 0 \end{cases}$$

for example would be

```
F={{[
cc(-infinity,0): (1+3*L^2)//L,
cc(0,0): (e^L)//L,
oc(0,infinity): (L-2*W)//L ]}}
```

where W is a real variable. The operations available for the domain of functions are $*$ (of type $\mathcal{R} \times F \rightarrow F$), **@** (as for real formulae, of type $F \times \mathcal{R} \rightarrow \mathcal{R}$, used for getting values out of functions), **integral** and **derivative** (the standard integral and total derivatives). **#** (of type $F \times F \rightarrow F$) is a function construction operator used for the building of multi-branch functions. The domain also supports the constraint **continuous** which is true for real continuous functions. The domain of functions in EUCLID provides a number of other powerful features. We do not list these since functions are not used in the examples presented in this paper.

Queries The mechanism for obtaining information from a EUCLID program is the *query*. A query has the form of a clause without a head. If the query cannot be proven using the program (theory) the answer is simply **No**. Should the query be proven, the answer is **Yes** qualified by bindings to non-domain variables and a set of simplified domain constraints over the domain variables of the query.

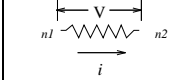
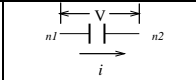
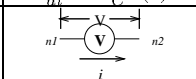
Computations A EUCLID computation is an extension of the resolution-based Prolog mechanism, essentially as in CLP(X) languages. The key step in resolution is *unification*, which is a form of syntactic equality of terms. Since in EUCLID domain terms have a fixed interpretation, for them unification must be extended to include semantic information. The generalisation is straightforward: if the two terms compared are non-domain terms standard unification applies; if the terms are domain terms equality is handled by the appropriate solver. In that sense unifying the terms $5+3$ and 8 succeeds, as does $X+3$ and 8 by generating the simplified constraint $X=5$. A EUCLID *computation* is a derivation from the original query, each step of which is either a resolution of the current goal with a clause in the program, as in Prolog, or a constraint satisfaction/simplification step performed by the appropriate external solver. A stack of bindings for the non-domain variables and a stack of constraints for each solver is maintained throughout the computation. Goal selection and search strategies follow Prolog and CLP(\mathcal{R}). (There are some detailed points of difference with CLP(\mathcal{R}) to be summarised in the full paper.)

Implementation The current implementation of EUCLID has been undertaken primarily as a feasibility study: the aim is to construct a tool that can produce the correct answers in a reasonable time. The implementation use a combination of Prolog and C code. Constraint solving is based on a Prolog guided interface to Mathematica (Wolfram 1991) and CLP(\mathcal{R}) (Heintze *et al.* 1991), the former being used for algebraic and differential equation solving and the latter as a simplex equality-inequality solver. The interface with Mathematica consists of a module for transforming internal constraints to Mathematica comprehensible terms (and vice-versa) and a communication module through `mathlink` (the standard interface to the Mathematica kernel). The higher-level interface is based on a number of predicates close to the Mathematica functionality (`solve_system_of_equations` etc).

The EUCLID interpreter runs on a Sun Sparc workstation, and does produce results in reasonable time. The domains described in the previous sections represent the working part of EUCLID. Future implementations are intended to extend the existing domains with a more comprehensive range of standard operations, as well as providing other mathematical domains used in problem modeling (vectors, complex, tensors etc).

We now illustrate the uses of the EUCLID language by showing how it may be applied to the representation of electrical circuits. Each component in a circuit is a self contained unit with a well known mathematical statement of its characteristic behaviour. Interactions between components are determined by the way they are connected in the circuit, and further constrained by both local and global laws (the two Kirchoff Laws) governing electrical circuits. The choice of electrical circuits as illustrative examples has some additional interest in that they have been used to demonstrate the application of the CLP(\mathcal{R}) language –see (Heintze, Michaylov, & Stuckey 1987) and (Heintze *et al.* 1991, pp. 27–28). Our presentation is intended to show that EUCLID has the ability to deal with time changing quantities (represented by one-variable real formulæ) as well as time independent quantities (represented by real numbers). Examples with capacitors (R–C circuits) are used to demonstrate EUCLID’s handling of differential equations. Background material on electrical circuits can be found in most physics textbooks –e.g. (Halliday, Resnick, & Krane 1992, Chapt. 33,39).

Electrical Components Electrical circuits are constructed from a number of basic components. For present purposes we can take it that every component has exactly two connecting nodes and its behaviour is completely defined by the characteristic relation between the potential drop and the current flowing through it. Since both of these quantities can change with time, the time dependence is represented by means of *real formulæ*. The relevant properties of some component types are summarized as follows:

 <p>Resistance R $V(t) = Ri(t)$</p>	<pre>component(resistor,[R],Vt,It):- real(R),formulæ([Vt,It]), Vt=R*It</pre>
 <p>Capacitance C $\frac{dV}{dt} = \frac{1}{C}i(t)$</p>	<pre>component(capacitor,[C],Vt,It):- real(C),formulæ([Vt,It]), It=C*derivative(Vt).</pre>
	<pre>component(volt_meter,[V],Vt,It):- formulæ([V,Vt,It]), V=Vt, It=0//T</pre>

Governing Laws The laws governing electrical circuits are few and simple. A local law postulates that the sum of all currents reaching a node must be zero (Kirchoff’s first law) while a global one postulates that the algebraic sum of the changes in potential encountered in a complete traversal of any closed cycle within the circuit is zero (Kirchoff’s second law).

The following program applies Kirchoff’s first law by recursively adding a constraint (`TotalCurrent=0//X`)

for each of a given list of nodes:

```
kirchoff_law1([]).
kirchoff_law1([n(Node,Currents)|Nodes]):-
  add_currents(Currents>TotalCurrent),
  TotalCurrent=0//X,
  kirchoff_law1(Nodes).
```

The following piece of code adds the appropriate constraints for each of a list of closed cycles:

```
kirchoff_law2(Voltages,[]).
kirchoff_law2(Voltages,[Loop|Loops]):-
  add_voltages(Loop,Voltages,0//X),
  kirchoff_law2(Voltages,Loops).
```

Solution Mechanism Electrical circuit problems can now be solved by means of a EUCLID program whose top level clause is as follows:

```
solve_circuit(CircuitDescr):-
  prepare_circuit(CircuitDescr,Circuit),
  calculate_loops(Circuit,Loops),
  Circuit=[Currents,Voltages],
  kirchoff_law1(Currents),
  kirchoff_law2(Voltages,Loops).
```

The first condition `prepare_circuit` transforms the input circuit description to an internal representation which makes use of the component information, and also determines which currents flow into and out of each node in the circuit. This is the list of nodes required as input for `kirchoff_law1`. The second condition `calculate_loops` determines which closed cycles are present in the circuit. This is done by a Prolog encoding of a standard algorithm. `prepare_circuit` and `calculate_loops` use only the Prolog subset of EUCLID and do not call any of the mathematical problem solvers.

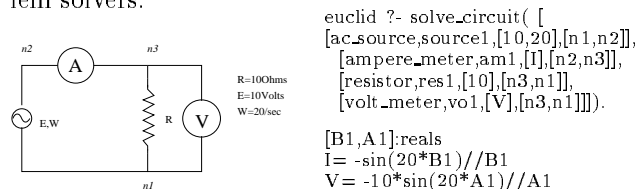


Figure 1: A simple E-R Circuit

A simple resistor circuit with AC source Consider the circuit of figure 1, a simple circuit with one resistor. To give the problem a slightly unusual twist we consider an AC source instead of a DC source and calculate the current flowing through the resistor as a function of time.

An R-C circuit with DC source The circuit shown in figure 2 contains a capacitor, and shows EUCLID's ability to deal with derivatives of real functions.

Wheatstone Bridge The circuit shown in figure 3 is taken from an exercise in a well known (Greek) text in Physics (Kougioumtzopoulos 1984) (p.155, Ex. 171). The problem is to calculate the value of resistor R4 when the voltmeter connected to n_1, n_4 shows no drop in voltage (no current flowing).

Conclusions

We have provided a sketch of the EUCLID language and shown how it may be used to represent some standard

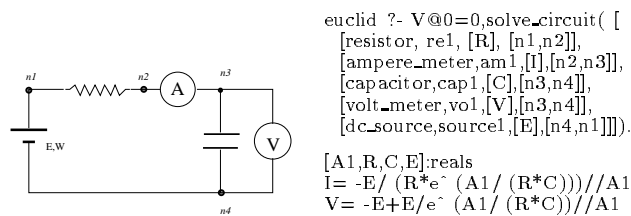


Figure 2: A sample DC R-C Circuit

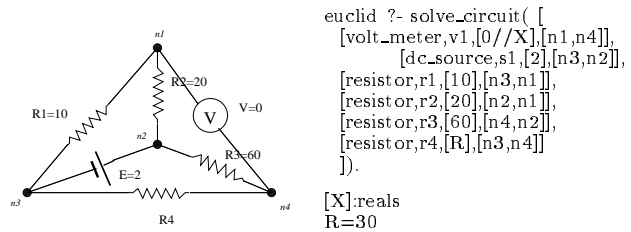


Figure 3: Wheatstone Bridge

problems from Physics in a fairly natural way. Computed answers can be numeric or abstract-symbolic depending on the nature of the constraints imposed by the problem description. This flexibility is important because it allows EUCLID to be used for different kinds of applications: as mathematical calculator, as programming language, or as a tuition aid for mathematics.

The electrical circuits used as examples in this paper do not illustrate the full capabilities of EUCLID. A number of different and more complex examples have been constructed (most though not all taken from Physics). References to these examples will be given; if space permits, one of them will be presented, perhaps in summary form, in the full paper.

References

Halliday, D.; Resnick, R.; and Krane, K. 1992. *Physics*. John Wiley and Sons.

Heintze, N.; Jaffar, J.; Michaylov, S.; and an d R Yap, P. S. 1991. *The CLP(R) Programmer's Manual*, 1.1 edition.

Heintze, N. C.; Michaylov, S.; and Stuckey, P. J. 1987. CLP(R) and some electrical engineering problems. In Lassez, J.-L., ed., *Logic Programming, Fourth International Conference*, volume 2, 675-703. MIT Press.

Jaffar, J., and Lassez, J.-L. 1987. Constraint logic programming. In *Proc 14th ACM POPL Conf, Munich*.

Jaffar, J., and Michaylov, S. 1987. Methodology and implementation of a CLP system. In Lassez, J.-L., ed., *Logic Programming, Proc. of the Fourth International Conference*, volume 1, 196-218. MIT Press.

Kougioumtzopoulos, H. 1984. *Physics: Electricity*.

Wolfram, S. 1991. *Mathematica : a system for doing mathematics by computer*. Addison-Wesley.