Infrastructure for Model-based computing

Vijay Saraswat

Danny Bobrow

Johan de Kleer
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca 94304

March 15, 1993

Abstract

The constant marketplace demand of ever greater functionality at ever lower price is forcing the artifacts our industrial society designs to become ever more complex. Before the advent of silicon, this complexity would have been unmanagable. Now, the economics and power of digital computation make it the medium of choice for gluing together and controlling complex systems composed of electro-mechanical and computationally realized elements. However, the construction of the software to implement, monitor, control and diagnose such systems has become a gargantuan, nearly impossible, task. We propose a radically new way of viewing the software of these complex systems.

Designers of these composites usually will not, and should not have to have a model of how these components are constructed. From their point of view, these components should be understandable through models oriented around the customer's domains of discourse. We claim that using models built in a constraint-based composition description language, customers can more easily compose and control complex computational systems to solve the evolving tasks at hand. Specifically:

1. We propose a view of architecture of complex computational systems:

Self-descriptive components: Components come equipped with models of their interaction and/or operation, specified using constraints on interface variables

Customer-specified topology: Customer puts together components, and imposes constraints appropriate to the task at hand (e.g., on output).

System-synthesized control: The system solves constraints for control signals and inputs that can be used to control the components to achieve the desired behavior.

Multi-use models: Appropriately defined models can be used not only for control but also for simulation and diagnosis of composite systems.

2. We claim this view is particularly appropriate when the components themselves are complex (e.g. mega-modules, many parameters, with complex constraints between them), or their temporal interactions are complex (e.g., real-time computationally-controlled electro-mechanical systems).

3. We will build a constraint programming framework that will support this view of architecture, and demonstrate its utility on real industrial problems. To support real systems, the framework will be asynchronous and concurrent; to support formal reasoning about the developed systems, it will have a simple semantic foundation; to be usable it will be efficiently implementable (in space and time) and produce code that is embeddable; to be widely usable it will be extensible with different constraint systems.

Contents

1 Technical Rationale

1.1 The basic paradigm

Motivation and Proposal

We are moving into an era in which the dominant characteristics of designed systems are being transformed from the physical (mass, energy) to the informational (computation, communication). Increasingly, multitudes of customers are confronted with sophisticated computationally controlled and/or realized computational *components* — independently maintained software, hardware, or electromechanical devices that have some computational aspects. For example, the engineer designing paper-paths and marking engines (for use in a photo-copier), the atmospheric scientist trying to predict the behavior of the ozone layer, the travel consultant making flight reservations, the surgeon using a robot-controlled arm for precision surgery, the tax-payer using a spread-sheet to compute her income-tax, ... all are customers using such components to solve the problem at hand.

It is unlikely that such customers will have no need to tailor, customize or assemble (from the components at hand) larger systems to satisfy their needs. It is even more unlikely that customers will suddenly start to come equipped with a deep knowledge of software engineering techniques. How, then, can such customers — embedded in their own context — be empowered to engineer collections of such elements to exhibit the purposive behavior they desire? Indeed, even as the nature and capabilities of the available components and technology change?

Computer scientists have traditionally viewed these questions (if at all) from the stance of the *technologist*, from the implementer's eye-view.[1] At PARC, some of us have been developing an approach to computation grounded in the customer's-eye view; in people, not technology.

The approach is grounded in the following beliefs:

People are not computer-scientists: Customers do not care about the technol-

[1] Indeed the traditional terminology — "user" — betrays the inside-out view of the world that is the characteristic hubris of Computer Science. The term is dismissive, almost pejorative — as if the *primary* characteristic of the person is that she uses the system, as if she is not embedded within her own context of intellectual background, values, world-views, tasks, and goals. It is the tool that must fit into that context, not the other way around. Several other authors (e.g., Terry Winograd, Lucy Suchman) have made the same point. If nothing else, the word *customer* is more respectful (though it is not perfectly adequate either) — it is associated with the notion that the use of the system is interesting only in so far as it contributes value to the user.

ogy used to create the elements they have at hand — the programming language or hardware technology they are implemented in, their conventions for handling parameters and data formats, etc.[2] What they care about is components whose functionality can be explained in terms that make sense in their context of usage, in the domain and language the customers are used to.

People create models of components: But customers are not dummies. Our approach is based on the intuition that people maintain robust, coherent behavior in the face of the tremendous complexity of the designed world by building, using, maintaining, debugging (perhaps coarse, approximate) internal *models* of how things work.

In fact, rarely do people have unique models of how things work — often, people have a variety of partially overlapping models (each capturing some aspect of the functioning of the object involved), and seem to choose a model detailed enough to be used to satisfy the needs for the task at hand. The engineer writing the control code for a copier views a paper-path as a transducer that takes as input a sheet of paper of a certain size, velocity, orientation etc. at a certain location, and produces another with somewhat different features at a different location. A TEX user has a model of the program as a transducer that accepts a file with some embedded commands and produces a device-independent model — this is the model to be used when invoking the TEX command. (A very different, more detailed model of TEX (e.g., in terms of glues, vboxes, hboxes etc.) is needed when preparing the input file for processing by TEX.) A customer of a mega-module responsible for planning surface-transportation in San Diego [WWC92] has a model of the program as an entity that can answer queries about transportation in San Diego. (If the customer happened to be a designer of the module, she would also simultaneously possess very different models of the software, reflecting, for example, its internal organization, or the division of responsibilities between the teams producing the software, etc. But such models are unlikely to be of any use if the task at hand is to *use* the program.) Indeed, it seems clear that even as components tend to become increasingly more sophisticated their ease of use will be tied to their ability

[2] Indeed, they do not even care about the efficiency of the realization (whether the last microsecond or byte of memory was squeezed out) — as long as the cost is “acceptable”.

to be modelled in simple terms in the language of discourse familiar to the customer.

Models should be articulated: We can take advantage of this intuition in the design of computational systems, by taking models as fundamental. The basis for constructing systems will be a library of *self-descriptive elements*, components that come equipped (even at "run-time") with a description of their capabilities — a declarative specification of the channels of communication between the component and the outside world, and the constraints that they satisfy. The constraints are written down without regard to how the information is to be used. Models for composites can be constructed from models of the components.

People can put things together: What customers can *do* to achieve their desired purposive behavior is *put components together* — only rarely should customers be required to "open up components" and manipulate components inside in order to get their task done with the efficiency that is desired. The customer can put such components together in any configuration she desires to create a system. (In particular, feedback between components is allowed.) She may further constrain various parameters of the system (either input or output) to take on only some values (corresponding to the initial conditions for the computation).

Models yield programs as side-effects: System execution consists of solving the constraints corresponding to the system to obtain a solution for the input and control parameters, and, if there is a solution, running the components with the values of these parameters set according to the solution. (Note that the situation in which constraints are not solvable is quite conceivable — for instance, the customer may have plugged two output wires together, or may have plugged a 12VDC line into a 120 VAC line. The infrastructure for supporting component inter-connections should be designed in a way that such *type-mismatches* can be checked quickly.)

We call the view of design of complex computational systems arising from these beliefs *model-based computing*. It is characterized by (1) specifying components (corresponding to (possibly aggregate) objects in the domain of discourse) with very flexible constraint-based interconnection schemes, (2) letting the customer fit components together as appropriate for the task at hand, (3) using the constraints associated with the components to automatically construct driver programs to solve the problem at hand. It has been made feasible by the emergence of powerful, mathematically simple, declarative, concurrent programming frameworks based on the use of constraints for communication and coordination (see Section 1.4).

We were led to this view of system design through our experience with the Generic Scheduler Factory (see Section 1.2) project at Xerox. In this case, customers were system designers who developed real-time reactive electro-mechanical components such as paper-trays, marking engines, video-sources, collating and stapling finishers etc. The models of interest here specified the timing characteristics ("transfer functions") of the components; what engineers want to be able to do is to plug these components to form copiers, and have programs for controlling the copier (e.g., for scheduling various computationally-controlled components) be automatically generated from the constraints specifying the copier.

We believe that this methodology for the construction of complex systems is widely applicable — particularly as we reach the stage where more and more customers are ignorant of the actual internal construction of the components they use (spread-sheets, anti-lock brakes, simulation models, etc.). It leads to the idea of constraint-based composition description languages for plugging components such as mega-programs together.

Properties of Model-based Computing.

Model-based computing has several obvious advantages. Systems produced in this way are:

understandable — Components are understood at the level of their models, rather than at the level of their implementations. Customers can validate model descriptions. Programs are generated as side-effects of the models. Often, models themselves are much more understandable then the programs generated: local interactions in the model may be translated into non-local interactions in the code.

compositional — Models for composite systems are built up from models of components. Composition is being done at the level of models rather than control programs.

verifiable — Formal techniques can be used to establish properties of models; Reasoning with the model offers a simple way to establish general properties (e.g., safety, liveness) about the system.

radically reconfigurable — alternate elements can be incorporated into a system at customer-use time.

malleable — information embedded in models can in principle be re-used in very different ways, e.g. for schedulers, simulators and diagnosticians. (In practice, this depends on the degree of sophistication of the model.)

Of these, radical reconfigurability and malleability are the most novel, and we now discuss them in more detail.

Radically reconfigurable systems. Self-descriptive elements naturally lead to the notion of *radically reconfigurable* systems. When self-descriptive elements are physically hooked together (at "run-time"), the elements can exchange their internal models to form a model of the whole system. Each element can now figure out the role it must play in the functioning of the entire system. There does not need to be a central "controller", since the constraints are adirectional, symmetric. Master/slave relationships/directionality can be decided by the elements themselves at configuration time, perhaps through the help of intermediate "mediators" [Wie92].

Radically reconfigurable systems empower customers. The customer "programs" the system by plugging together components in the way she wants, and by supplying the context (the environment in which the system is embedded).

The need for radical reconfiguration also shows up in the problem of synthesis of simulators for physical systems. Here the need for self-describing systems is obvious: one would like to have separate descriptions of the components of the physical system, and mechanisms for automatically generating the code corresponding to the entire system, given the configuration of the system.

The many uses of models. But models (in this setting) are not just informal user manuals: they are pieces of information that can be computationally manipulated in powerful, generic ways, regardless of how the system has been put together from elements. As long as the elements are self-descriptive, their models can be put together to form a model for the entire system.

From models to simulators. First, models can be used to describe to the customer how things work. For specified values of inputs and control signals, constraints can be solved (in the "forward direction") to determine the values of the output signals; these can be propagated recursively to obtain values for all the connections in the system, thus effectively simulating the behavior of the system.

From models to controllers. Unlike components, models may be run *backwards*: given information about the outputs required of the component, the constraints associated with a component can be solved to obtain the inputs that if fed into the component would produce the required output. For example, a customer of a photo-copier may specify that she needs five double-sided copies of a seven-page single-sided input document. The models associated with the components making up the copier can be used to deduce the values of signals that must be input to the system (sequence of video images, control signals for various components) in order to make it produce the required output.

This kind of computation usually requires a combinatorial search that falls out automatically from the constraint-solving process. (In the case of a photo-copier, this is the scheduling problem.)

While correctnesss (of schedulers, simulators, diagnosticians) is guaranteed by the soundness and completeness of models, *efficiency* is usually another matter. Usually, efficiency does depend crucially on the efficiency of the constraint system used to represent the system behavior, and on the granularity of modeling used. Our experience has been that a number of *model transformation* techniques can be identified which allow models to be transformed into equivalent (or, sometimes, weaker) models which perform more efficiently with respect to some given inference engine.

An environment for model-based computing must, then, make available techniques for evaluating the performance of a certain model with a certain inference engine, and for transforming models into ones with more acceptable computational behavior.

From models to diagnosticians. Third, models can be used to help debug the system — if it doesn't work as advertised. Model-based diagnosis techniques use information about the correct functioning of components to formulate hypotheses about the modes in which the components are operating that are consistent with observations of its actual functioning. This information is then used to guide probes (requests for further observations) to discriminate between the hypotheses, until the misbehaving culprits are identified.

From models to designs for systems. Assume that the customer is given a system specification and is searching for a configuration of available components which will optimize some criterion such as productivity of the system. Assume that it is easier to compute with the models associated with components than to construct actual systems and physically measure the given criterion. In such a case, the models associated with components can be used to evaluate candidate systems by simulating their effect on appropriate inputs and measuring productivity. The resulting search space may then be searched automatically or interactively for an optimal solution.

In general, computational prototyping with constraint languages is a powerful idea. Partial protytpes can be designed and checked for consistency. If a partial prototype does not have even the potential for achieving certain functionality then all members of the family obtained by further instantiating the prototype can be discarded.

From models to programs. Finally, and most importantly, if the component is entirely computational, and the model is described in a constraint programming language, then the model may itself be the realization. There may be no separate program realizing the model.

Because of the development of constraint programming language, this is not as special-purpose a possibility as it sounds. All computation is about some objects of interest, and about establishing some relationships between them — the development of (concurrent) logic and constraint programming over the last decade has shown that many concurrent, indeterminate computations can be cast naturally in this style. The embedded constraint system can be chosen to fit the domain of discourse of the customer. Over and above that, the logical notions offered to the customer are the usual conjunction, implication and quantification. No knowledge of machine-oriented, internal architecture of computational systems is assumed. Thus computational problems can be stated very naturally and at a high-level in a language not unfamiliar to the customer.

Systems built out of such purely declarative components enjoy all the properties of models (invertibility, etc.) discussed above.

1.2 Example: Generic Scheduler Factory

Perhaps the simplest instances of this view arise from our work in generating embedded, real-time machine control systems, simulators, and schedulers (the Generic Scheduling Factory Project).

The problem is as follows: given a description of (1) the (computationally-controlled) components that constitute a digital photocopier (scanner, marker, paper-feeder trays, paper-paths, finishers (staplers, collators, stackers)), (2) their configuration, (3) a description of the job to be output by the system, *determine* the control signals to be sent to each of the components such that the desired job can be produced. Because of the somewhat simple, acyclic nature of a photo-copier, this boils down to determining the sequence of images that must be generated by the video-box — all other decisions can be made deterministically from this decision. The sequence must be such as to minimize some performance criterion associated with the job — usually, the length of the (closed loop) photo-receptor built that is left unused is sought to be minimized. Note that on execution deviations from the prepared schedule may arise. For example, a pitch may need to be skipped (that is, a portion of the photo-receptor belt may have to left without an image) because the video-image took too much time to decompress and process. In such cases, the system must dynamically re-plan the correct sequence of video-signals that must be requested from that point onwards, so that the performance criterion is still minimized.

Hitherto, schedulers have been designed manually at Xerox. Talented engineers identify constraints on timing behavior imposed by different components. (For example, the "tray-less duplex" marker is such that the second side of a duplex print must be placed on the sheet of paper a fixed number of milli-seconds after the first. Most constraints that arise in this domain can be expressed using finite domain variables, or with simple linear arithmetic.) They "solve the constraints in their head" and produce intricate pieces of procedural code that embody a particular way of solving these constraints. Historically, this has been a difficult and error-prone process: these embedded, reactive, real-time pieces of code have been amongst the most complex produced by Xerox engineers, and amongst the most difficult to maintain, extend, analyze and predict the performance of. And important: the productivity of the machine depends directly on the quality of the scheduler. However, this process is now completely inadequate. Modern lines of digital copiers are required to be such that the *customer* can reconfigure the machine — for example, by adding a component that can staple prints after the component that collates sheets. Since these components (which may be entirely new — may not even have been anticipated by the designers of the original copier) may have their own idiosyncratic timing constraints, it is required that new schedulers be generated *automatically* at customer-use time.

The requirement for this kind of *radical substitutability* (plug-and-play by the customer) seems to force us into the notion of model-based computation. It seems what must happen is that some descriptions of the capabilities of components must be associated with each component. These capabilities need to be composed together in the way in which the customer compose components to configure the machine.

Our solution, then, was based on associating each component with constraints that described the timing behavior of the component. For example, the marking engine in a photo-copier is equipped with a model which describes the constraints the component imposes on the input stream of video signals, sheets and control signals and the output stream of single- and double-sided prints. From these constraint programs, it is possible to automatically generate a scheduler for an arbitrary machine produced from the components: the scheduler conjoins the constraints produced from the components (connecting shared variables in exactly the way in which the corresponding components are physically linked together), adds in the constraints corresponding to the output of the whole system (supplied by the customer as part of the job description), and solves these constraints for the control parameters for various components (marker, feeder, finisher, and video-box).[3] The re-scheduling problem can be solved, in principle, by using an incremental constraint-solver.

Note that in this case the model associated with a component is detailed enough as to be useful for simulation as well. In this case, specified inputs are supplied to each of the components (e.g., the control signals to the components). The constraints can be solved to generate the output signal of the entire machine — without having to run the actual machine.

[3] We should point out an intermediate step. Once the machine has been configured, it is possible to collect together the constraints for various components, and compile them into a representation that is more efficient for repeated solving (e.g., a finite state machine).

1.3 Example: Constraint-based composition description language for mega-programming

Self-descriptive elements lead to an interesting approach to the design of information systems: independently developed and maintained, self-describing software modules ("mega-programs") are use-time configured by customer to obtain the desired functionality. This approach avoids many of the problems in our current tools for composing large systems from individually created and maintained components.

As the complexity of software components increases, the need to "open up" various aspects of their functioning becomes even more important. One just has to examine the list of flags passed into Unix commands such as `dvips`[4], or the list of variables that the X window system allows the user to set, or the complexity of the interaction mechanisms that Unix programs such as `make` or `rdist` provide.[5] The complexity of such "transformational" components does not arise from their temporal interactions with the outside world (as it did for the electro-mechanical components of the previous section): it arises from the large number of often interlocking parameters that must be made available at the interface. Not all the parameters can be set separately to arbitrary values — there are often interlocking constraints between them that are not represented explicitly at run-time. (For example, `dvips` allows the customer to specify the first and last page numbers to be extracted — obviously it needs to impose the constraint that the first and last page numbers exist in the document, and are in the right order.) This means that it is easy to invoke a component incorrectly — and then be suprised when it aborts unexpectedly.

The Unix environment is a typical example of "mega-programming" languages available today. The primary mechanism for composition is "piping" through filters. This is primitive and inflexible: elements have single input and output "streams", and the behavior of elements must be controlled by passing in concrete values for parameters fixed in advance. Clearly, dramatically more powerful combinations are possible if constraints are used to specify interface requirements for hooking up components. We discuss this with illustrative examples.

---

[4] It takes a "`.dvi`" file (e.g., produced by TEX) and converts it to PostScript.

[5] It is interesting to note that there is a trend towards specifying such interface information declaratively, rather than, say, providing hooks where arbitrary user-code can be run. Our point is that these declarative interfaces can be made significantly more sophisticated and powerful by having a constraint-based composition description language.

Bidirectional Communication. In our current ways of putting components together, it is not possible for "downstream" components to feed information back to "upstream" components, thereby helping them more correctly choose values for their defaults. This lack of bidirectional communication may result in performance that is unacceptable to the customer. For example, `dvips` (on our workstation) defaults to using white-on-black fonts. However, (the single) high-speed printer available at our site is a black-on-white printer — the default font produces an almost illegible copy. `dvips`, of course, does not take as parameter the printer on which the output is to be produced; another program `lpr` does. However, there is no way for `lpr` to inform `dvips` that the printer specified by the customer is black-on-white. The current solution at our site is that a special driver program has been created: `4050dvips` sets flags for `dvips` causing it to use the right font. But now the user has to remember to use the right component ("carry the constraint in her head") when configuring her print service. Clearly, this could have been resolved automatically by conjoining and solving the constraints and preferences for `dvips` and `lpr`: the bidirectionality of constraints is just the right thing here.

Global Coordination through Local Interaction. Another related problem is that often components have only partial information to provide about the choice of some parameters — or perhaps just some preferences. In our current way of configuring components, these components have no mechanism for representing partial information, and for interacting with the rest of the system and dynamically determine a globally optimal solution. Consequently, they must choose solutions separately — and their separate decisions may be mutually inconsistent. The customer may be forced to step in, understand the constraints involved, solve them in her head, and supply the solution. For instance, a text processing system (such as TEX) needs some information about the font to be used in order to make its decisions about breaking lines and pages (among other things). By default, the system makes certain decisions (e.g., TEX chooses 10pt fonts). On the other hand, the display on which the output is to be produced may impose its own constraints on the font size: for example, a high-resolution display may need to use a 14pt font in order to get the same character-size as a 10pt font.[6] From the customer's perspective, the composite system she assembles fails — the 10pt font chosen by TEX is too small to be legible. However, it is easy to see how the right thing could

[6]In fact, if the display is to viewed from a distance (e.g., the text is to be displayed on a flat-panel screen, and the readers are scattered around the room) an even larger font may be preferred.

happen if thes pieces of information (preferences, constraints) were explicitly represented and manipulated at configuration time.

Asynchronous partial information. By its very nature, a constraint-based composition language allows for the asynchronous communication of partial information between components. For example, a mega-module for planning air-travel between San Diego and Washington D.C. (cf. [WWC92]) may not yet have determined the optimal flight pattern between the two cities — but it may yet be able to communicate to other mega-modules that flight time will be at least five hours, and that the plan will stop in Denver (e.g., because other airports are closed). Similarly, not all the elements of a data-structure have to be fully specified before it can be communicated to other mega-modules — logical variables can be left in place to be appropriately constrained later.[7] Inherently asynchronous notion of composition is particularly appropriate for the typically distributed settings of mega-programming.

Dynamic optimization. Consider the "make" system in Unix. It operates with a very simple declarative description of the objects of interest (object and source files), and the relationships between them (primarily, binary dependencies), and commands for creating one form of file from another (e.g., by compiling). Such a system can be made even more powerful, however, with the provision of information specifying the *functionality* provided by each file, with its associated speed and size requirements. At use-time, the make system can then take a specification of the functionality, space and time requirements of the target system (or perhaps a description of the run-time environment in which the resulting code must be embedded), and synthesize a system of components whose associated constraints satisfy the stated requirement (perhaps optimally). Depending on the complexity of the parametrization the system allows for interfaces, it would also have to solve for the code-stubs needed to link up the elements (as specified in CORBA, the Common Object Request Broker Architecture, [DEC91]).

In conclusion then, we claim that constraint-based composition description languages can provide a powerful glue for assembling complex systems.

[7] This is the functionality provided by the SUPPLY and INVOKE operations of the megaprogramming language in [WWC92].

1.4 Infrastructure: Concurrent Constraint Programming

The basic paradigm

The declarative models associated with components fit naturally into the emerging framework of concurrent constraint-based programming (CCP), [Sar89,SR90, SRP91]. Based on the *proof-as-computation* principle pioneered by logic programming, CCP is a declarative, high-level paradigm for concurrent computation that is at once computationally powerful, conceptually simple and semantically elegant.

Perhaps the simplest way of introducing this paradigm is to contrast it with imperative programming. Following [Dij76] we may say that, in imperative programming, the state of the system is specified by a *store*, which is a vector V of n variables of interest, and a *valuation* assigning to each variable in V a fully formed (completely known) value in its domain. Thus each store describes a point in the n-dimensional *state space* for the system, obtained by taking the product of the sets of all possible values for the variables X_i. Let us define a *constraint* as a (possibly infinite) subset of the state space, that is, a set of valuations. When designing a (possibly non-deterministic) algorithm, we are concerned with specifying a mechanism such that when it is initiated in a store that satisfies a given constraint (the *precondition*), it terminates in a store that satisfies another given constraint (the *postcondition*). Hence in any formal notation (programming language) used to describe such algorithms, some basic operations are necessary to transform one store to another. In imperative programming languages, the basic operations executed by an agent are *Read*, which obtains the value of some variable in the store, and *Write* (or assignment), which translates the store parallel to some axis by *changing* the value of some variable (possibly by applying some functions to the values obtained from prior reads). The fundamental difference in concurrent constraint programming is that the *store is itself a constraint*, a set of valuations which provides *partial information* about the possible values for the given variables. The basic operations an agent may execute are *Ask* (instead of Read), which checks whether or not the store *entails* a given constraint (that is, whether every valuation permitted by the store is permitted by the constraint) and *Tell* (instead of Write), which adds a given constraint to the store, provided that the resultant store is *consistent*.[8] Thus, a basic step does not change the value of

[8]This terminology is borrowed from Brachman and Levesque [BFL85], who used it in a closely related context — see [Sar89, Ch. 1] for more on the connections.

a variable, but may rule out certain values that were previously possible; we say that the store is *monotonically refined*, since at any step of the computation, the set of possible values for the variable is contained in the set of possible values at any prior step. Thus the entire constraint programming framework is based on extremely general notions of partial information, with its concomitant notions of consistency and entailment.

A fundamental consequence of this radically different view of the store — which one may regard as the essence of logic programming — is that it enables a simple, elegant, and powerful paradigm for concurrent computation. Briefly, one may imagine multiple agents of the kind described above, executing concurrently in a shared store. Synchronization is achieved through a *blocking Ask*: an agent may block if it attempts to ask whether a constraint is entailed by the given store, and it is not yet so; it remains blocked until such time as (if ever) some other concurrently executing agents add constraints to the store such that the resulting store is strong enough to entail either the query or its negation.

Conceptually, this notion of concurrent constraint programming provides a theoretical footing for the intuitive visions about computing with constraints, evidenced in some of the work of such people as Sutherland, Sussman, Steele, Borning and some of their colleagues. Further, coherent development of this notion leads to a synthesis and extension of some of the work on *concurrent logic programming* that has occupied a number of researchers for the better part of the last decade [Sha87,Sar87,Ued86,CG86] and the more recent notion of *constraint logic programming* due to [JL87,Mah87,Col90,DvH$^+$88]. Their union properly generalizes both: the former because its basic concepts of communication and synchronization are now seen to be applicable in a far more general setting, and the latter because it introduces a simple but powerful framework for control where earlier there was none.

Characteristics of this approach

Constraints and Concurrency. First, the system offers a very powerful unification of concurrency and logic — parallelism is unified with conjunction, suspension with entailment — in such a way that the resulting system has simple logical and mathematical foundations.

Another aspect of CCP is that constraints are embedded within a programming formalism that can itself be described in high-level, logical terms. Indeed, determinate and non-determinate `cc` programs can be read as formulas in (intuitionistic) first-order logic and computation can be regarded as deduction [LS91]. A primitive constraint logically follows from a given agent iff the agent can evolve into a state from which the constraint follows immediately (using only the constraint entailment relation). Hence, the notion of execution embedded in the operational semantics of cc programs is sound and complete with respect to constraint entailment.

A distinguishing aspect of synchronization in cc languages is its *extensive* nature: If a piece of information becomes available during some stage of the computation, it is available at all subsequent stages of the computation.[9] The "stable" nature of communications considerably reduces the conceptual complexity of synchronization.

Mathematically, determinate cc agents can be seen as *closure operators* over the lattice generated by the underlying constraint system [SRP91]. (A function f on a lattice L is a closure operator if it is extensive, monotone and idempotent.) Closure operators have certain remarkable properties that gives rise to a simple and powerful algebra of considerable use in program transformation [SRP91]. These ideas are extended in [SRP91] to produce a model for indeterminate cc languages useful for reasoning about safety and liveness properties and in [JSS91] to non-determinate languages.

This mathematical simplicity makes it possible to consider the development of sophisticated semantically-based programming environments for cc languages, including systems for static polymorphic type-checking, program transformation and interactive program synthesis.

Extensible nature. The cc framework is extensible in two ways. First, new combinators can be designed and integrated into the cc framework, as long as they can be thought of semantically as monotone functions on the underlying constraint lattice. Such new combinators can be combined arbitrarily with the other "standard" combinators, such as parallel composition, hiding and ask. Indeed this flexibility is crucial in capturing patterns and idioms of constraint-processing common in various domains. For example, [HS91] introduces a new combinator corresponding to the notion of *indexical constraints*. This combinator can

[9] Of course, implementations can always "garbage-collect" constraints once it can be shown that no agent can notice the difference. This happens, for example, when a thread exits a lexical scope that had introduced local variables. All constraints purely on those local variables can now be deleted.

be implemented quite efficiently and captures in a high-level, implementation-independent way a pattern of usage common in variations-of-interval reasoning. The design of other combinators capturing aspects of information hiding and delayed information propagation is an active area of study [Sar89,JH91,HS91].

Second, the notion of programming is *parametric* in the constraint system involved: essentially the same set of computational ideas work with *any* constraint system. The notion of a constraint system is extremely general — essentially any system of partial information with the notions of consistency and entailment suffices.[10] But the computational and algorithmic diversity of constraint systems is enormous — on the one hand we can have extremely simple constraint systems (e.g., equality over constants and variables) that can be implemented by simple transitive closure algorithms; on the other hand, we can have constraint systems dealing with systems of equations over booleans, rational trees, concept description languages, feature structures, lambda terms, temporal intervals, or solving integer linear programming problems, etc.: that is computational problems that are difficult or even potentially undecidable.

This separation between the built-in constraint system and the enclosing inference language offers a natural architecture for hybrid reasoning that elegantly satisfies the customer's desire for a domain-specific language and the implementor's desire for a single implementation.

Recent developments in logic (the emergence of linear logic [Gir87]) allow us to extend these computational ideas to a setting in which we can also talk about resource consumption. See, for example, [SL92].

Research themes

Several small projects have explored the practical feasibility of these ideas. We propose here to bring together a team of implementors, theoreticians and modelers to pull together these strands into the design, implementation and use of a powerful mega-modeling environment.

Specifically, we shall focus on the following areas.

Implementation technology. It seems clear how to adapt for concurrent constraint programming the advanced compilation ideas (Warren Abstract Ma-

[10] If it is desired to use existential and universal quantifiers, then the constraint system must also have some minimal first-order structure; see [Sar92].

chine [War83], native code compilation, abstract interpretation) underlying Prolog implementations so as to generate code that is orders of magnitude faster than generated by other AI modeling environments, as well as inter-operable with procedural languages like C. Because of the embedded nature of applications of interest to us, we shall design the system so that the size of the run-time code generated by the compiler is minimized. This will probably necessitate using a threaded native-code generation scheme.

A significant portion of the work in this project will be devoted towards obtaining a practical implementation of these ideas.

Integration of constraint solvers We will develop a detailed architecture for hooking constraint solvers into inference engines underlying concurrent constraint languages. This will enable users of the system to design and implement their own application-specific constraint systems, and integrate them with the implementation. In particular, we shall implement several constraint-systems of interest in the domain of modeling electro-mechanical systems.

Mathematical foundations. With constraint languages being closer to the problem-domain than the implementation domain, it becomes crucial to extract as much information as possible about the run-time behavior of the system at compile-time. The scope of possible behaviors of the system at run-time is so tremendous that otherwise very few commitments to efficient representations can be made. Hence it is important to develop semantically-based static program evaluation techniques such as abstract interpretation.

Our ultimate goal is to work towards a semantically-based program development environment for the conceptualization, creation, construction and maintenance of reliable (possibly real-time) concurrent and distributed software. Such environments have been proposed before for sequential programming systems [SS83], and indeed are to some extent already available [Smi91]. But we believe that they are even more crucial in the substantially more complex world of concurrent systems. In this project, we can take only the first steps towards such a vision. Accordingly, we propose to focus our attention on the development of proof systems for safety and liveness properties of (linear) cc programs, and the development of various (polymorphic) type-inference mechanisms for such languages. Static type checking schemes are necessary so that a minimal amount of logical consistency checking can be done at compile time. Because strong typing rapidly becomes cumbersome it becomes necessary to develop polymorphic typing systems, in the style of Robin Milner's ML.

1.5 Related work

The ideas of model-based computing arise from a background of research in a variety of areas — AI and model-based reasoning and diagnosis, AMORD-like problem-solving languages, object-oriented languages, analysis and design, design and implementation of constraint programming languages and theoretical foundations of programming languages. We briefly describe relationships with major efforts in these areas.

Software architectures

OOAD. The notion of associating with components (objects) a specification of their behavior arises also in object-oriented programming, analysis and design. However, the usual notions of message-passing and interface protocols are too local (in both space and time) and low-level for describing the complex symbolic computations that can arise when linking together self-describing elements. (For example, program verification tasks require constraints to be specified across interfaces and across time; this cannot elegantly be captured in a concrete message-passing style.)

Constraint programming notions offer complementary ideas for the declarative description of complex, interacting system-behavior. For instance, unlike message-based communication, constraints can be used to communicate partial information *through* objects, because of transitivity of entailment. Indeed, the development of an appropriate integration of object-oriented and constraint programming ideas is an exciting related research topic we expect to explore in the coming years. We believe that very powerful combinations of these ideas are possible, particularly in a concurrent setting.

We expect to track emerging industry standards in the design of architectures and protocols for object-interaction. In particular, we intend to provide interfaces to our implementation of constraint languages that conform to the standard ORB (Object Request Broker) architecture.

Module inter-connection languages. The many connections with the work of [WWC92] are obvious. The novel features of constraint-based composition description languages have already been discussed earlier.

Representation and Reasoning.

Model-based reasoning (diagnosis, design). The notions of large-scale modeling of physical systems underly a large body of closely related research in AI, Qualitative Physics, and Model-Based Reasoning over the last two decades. For example, work in Qualitative Physics has examined coarse constraint-based models for describing continuous physical systems. This modeling work is directly useful in the current setting; indeed several constraint systems of interest to us (e.g., for temporal reasoning) are directly inspired by on-going work in this community. Recent work on Compositional Modeling Language (CML) is bringing together work done at Northwestern, Austin and PARC. We expect the constraint-based modeling language to be powerful enough to express the models that can be described in CML; the semantic foundations and implementation work that we propose will serve to complement the work of these other groups.

Some recent work has focused on issues related to constructing and maintaining large-scale models (e.g., [FF91]). Researchers have examined the idea of producing models in small compositional chunks, and of assembling models for physical scenarios from constituent element models, and a specification of the task to be performed, and on finding a model at the right level of granularity for the task at hand.

However, the community does not have a standardized, well-studied modeling language which is simultaneously clean and simple theoretically, natural to specify models in, and efficiently implementable.

The PACT research project in the Bay Area has been investigating the assembly of large-scale models for physical systems by integrating several different models, built in different languages, and focusing on different stages in the design and manufacturing process (conceptual design, functional design, device layout, etc.). The project has proposed to use KIF as an interlingua for linking together these heterogeneous models. We believe this approach is promising and will investigate linking up with these efforts, through developing translators between KIF and the constraint-based modeling language.

The constraint programming modeling methodology we are proposing here is extremely general, and the infra-structure we are proposing can potentially serve as a way of combining research work in a variety of symbolic computation and reasoning areas. In particular, we intend to track key areas of investigation and ensure that the architecture we propose is general enough to capture the constraint systems and techniques used in these areas.

Planning, Scheduling, Control. Specifically, we will track the work being done for the ARPA planning and scheduling initiative, that is focused around the notion of reasoning with temporal constraints. For instance, we share with the CORTES project [FS90] the intuition that knowledge of domain constraints will lead to a fundamental understanding of the structure of combinatorial search problems underlying planning, scheduling and control problems. Our focus is on developing a notion of constraint-based composition of components, and on developing a powerful foundation for constraint programming that is conceptually simple enough to be analyzable mathematically.

We share considerable intellectual territory with the declarative control schemes of Kohn and Nerode (e.g., [Koh91], and several recent papers), such as the reliance on constraint programming. However, we see our architectural approach as equally applicable to the problem of composing purely software components (e.g., mega-modules). Our emphasis is on an architecture for enabling customers (end-users) make powerful use of components whose internals they have no access to.

Kowledge Representation. The notion of model-based computation discussed here is somewhat different from Pat Hayes' Naive Physics Manifesto, and John McCarthy's Advice Taker. Rather it is a view of computation, to be used in developing flexible, customizable, configurable designed systems. Unlike Lenat's CYC work, we are not interested in modeling the world in all the richness of its unfathomable mysteries; rather we are interested in targetted, task-specific modelsfor designed artifacts.

Constraint Programming The practicality of constraint programming has been successfully established by the development of several constraint logic programming languages such as CHIP [DvH$^+$88,Hen89], CLP($\Re$) [JJL87], and Prolog-III [Col90], some of which are now available commercially. These languages indicate the versatility of integrating into a goal-driven computational model sophisticated constraint solvers over linear rational (real) arithmetic (e.g.[JM87, JMSY,Stu90,VHG92,Col90]), Boolean algebra (e.g. [BS87,Col90]) and finite domains of integers (e.g. [VHD86,Hen89]). They have been applied to a variety of industrial problems including CAD (e.g. test generation [Sim89], hardware verification [SND88], circuit analysis [HMS87,GVHPZ90], testing [GKMP90]), combinatorial optimization (e.g. [Hen89,VH89]), and decision-support systems (e.g. [HL88]).

For many applications, these languages reduce development time by an order of magnitude, yet preserve most of the efficiency of procedural languages. In addition, they facilitate modifiability and maintenance of the programs mainly because of referential transparency. Solving complex applications in CLP has also indicated a variety of limitations in existing languages: for example, the need for extensibility, advanced control frameworks, and incrementality. These limitations are all addressed by CCP; however, we expect to take full advantage of the experience gained in the community in the design and implementation of constraint logic programming languages.

Recently, we have started an informal project with several researchers in Europe and the US on various aspects of concurrent constraint programming (the ACCLAIM project)[11]. The European partners of the project (about seven Principal Investigators) are formally funded by ESPRIT. We expect to coordinate our work very closely with that effort, and leverage their work, particularly in the area of abstract interpretation, parallel implementation techniques, and formal techniques derived from connections with graph-grammar work and the use of true concurrency approaches to semantics. In particular our collaboration with these researchers is strengthened by visits of graduate students and research personnel.

[11]The principal researchers involved are Hassan Ait Kaci (DEC Paris Research Laboratory), Philippe Codognet (INRIA, Paris), Saumya Debray (U. of Arizona), Bart Demoen (Katholikie University, Leuven), Seif Haridi (Swedish Institute of Computer Science), Manuel Hermenegildo (Universidad Polytechnic, Madrid), Ugo Montanari (U. of Pisa), Gert Smolka (DFKI, the German AI Center), Evan Tick (U. of Oregon) and Pascal van Hentenryck (Brown Univ.).

2 Statement of Work

The goal of this project is to provide an infrastructure that will support our view of model-based computing. Members of the Scientific and Engineering Reasoning Area at PARC have pursued aspects of this vision for many years. We have built diagnosis engines, qualitative simulators, and programs to reason about design. But there has been no common framework in which to develop and share models and to build specialized inference engines. Other members of the staff have beeen involved in the development of language systems, both the theory and implementation. These include systems with an object-oriented basis, and others with a basis in concurrent constraint programming. We believe the time is ripe to bring together all these strands to provide for the community, and for our industrial sponsors, an architecture, an implementation, and the mathematical foundations for a general constraint-programming infrastrusture

The work in this project is divided into three major areas.

Models and Task-specific Inference Engines. We will develop models for components. Our primary interest is in components for time-varying computational electro-mechanical systems. For such systems, the problem of control is to compute the coordinated signals that will drive the composite model to achieve a specified goal, or to maintain a set of contraints in the face of an uncertain and changing environment. Models of the task domains will also be developed.

In addition, we will investigate the suitability of constraint-based descriptions for mega-modules as software components. The key problem here is to identify suitable values for control parameters to achieve the desired output. For this task we need models of the software modules, and descriptions of the parameters associated with elements of the system, and models of the system in which the modules are embedded.

We will incorporate and extend generic inference systems for controlling, diagnosing and simulating systems specified using such models. This work will be coordinated with the larger ARPA Knowledge Sharing effort. In particular, we will work with other researchers to investigate the relative expressivity and intertranslatability of our models and those written in languages such as KIF, Ontolingua and CML.

Constraint-based composition environment. We will design a concrete constraint programming language. The basic language will support reasoning over trees, finite domains and inequalities. This basic language will be embedded in an architecture that will allow plugging in different constraint-solvers with different computational tradeoffs, and/or different domains. We will develop a detailed specification of an open architecture that supports hooking constraint solvers into language compilers. We will also specify and implement constraint solvers for domains of interest in scientific and engineering reasoning, such as temporal intervals, and implement prototype compilers (for constraint programming languages) that generate efficient, embeddable code.

Mathematical foundations. We will develop higher-order constraint languages in which models can be treated as first-class objects (abstracted, applied and transmitted in constraints), ML-style polymorphic types for compile-time consistency checking of models, sound and complete proof-systems for abstract behavioral equivalences of models expressed as constraint programs, and techniques for model transformation, partial evaluation and static analysis based on these equivalences. We will explore environments for modeling resource-conscious systems based on a computational interpretation of Girard's linear logic.

Our research group is working on realizing the vision of model-based computing outlined above, and on related topics such as non-linear optimization. Our project builds on past work and tools developed in the area of model-based reasoning and diagnosis. We are collaborating with other research groups elsewhere (in USA and Europe) on various aspects of the constraint programming paradigm. We are drawing on the considerable expertise within Xerox in the design and implementation of complex real-time control systems. We will build on our continuing interactions with researchers in the ARPA Knowledge Sharing Effort. The funds we seek from ARPA to establish common infrastructure for the basic technologies involved will be considerably amplified by these interactions.

The work we propose is a bringing together of ideas from many disciplines: techniques from areas such as foundations of programming languages, model-based diagnosis and reasoning, logic and proof theory, semantics of concurrency, programming language design, logic programming, analysis and design of algorithms, and constraint-solving will be brought together in the design, implementation and use of a concrete constraint-based environment for building systems composed of self-descriptive components.

3 Description of Expected Results

The primary result of this project is promotion of the idea of *model-based computing*. To make this tenable requires an infrastructure based on concurrent constraint programming. We expect to build that infrastructure, and demonstrate the value of the ideas through new research results, and increased productivity in the production of software for systems.

Infrastructure for model-based computing. In addition to building the task-based models that support model-based computing, we expect research associated with this project to extend our fundamental knowledge of constraint programming.

Specifically, we will develop a compiler-based implementation of constraint languages that is open to the integration of arbitrary constraint solvers. The compiler will focus on producing efficient, compact run-time code so that programs written in this notation can be embedded within other procedural applications. This is crucial for the use of the notation as a glue for components. We will release specifications of the language, abstract machine and prototype implementations to the research community, and encourage the participation of researchers at other institutes in developing the many other tools that go towards building a truly productive computational environment. In particular, we expect our work to be a significant step forward in creating a robust, efficient, usable constraint-based modeling environment for use by researchers in the area of model-based reasoning.

We expect the theoretical, practical and application experience gained in this project to be reported in papers at conferences and journals, in the usual academic tradition. This is particularly important because we expect the work in this project to span a number of traditional academic areas — model-based reasoning, knowledge representation, object-oriented programming, constraint programming, programming language foundations etc.

We will work with vendors of object-oriented languages and tools to understand how to integrate our model-based computing ideas.

We expect to test these claims in the context of real tasks from Xerox product organizations. Our group has an ongoing collaboration with an organization that is producing the software for a new Xerox product family. This work (not supported by ARPA) will be a demonstration of the radically increased productivity we expect from the use of these tools. Our experience in applying these ideas will be reported and will inform our continuing research.

4 Bibliography

References

[BFL85] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. *Readings in Knowledge Represenation*, chapter KRYPTON: A functional approach to Knowledge Representation, pages 411 – 429. Morgan Kaufmann Publishers, 1985.

[BS87] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[CG86] K. L. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *TOPLAS*, 8(1):1–49, January 1986.

[Col90] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[DEC91] et al. Digital Equipment Corporation. The common object request broker: Architecture and specification. Technical report, OMG, 1991. OMG Document Number 91.12.1, Revision 1.1.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DvH+88] M. Dincbas, P. van Hentenryck, H. Simonis F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the FGCS Conference*, November 1988.

[FF91] B. Falkenhainer and K. D. Forbus. Compositional modeling: Finding the right model for the job. *Artifical Intelligence Journal*, 51(1-3):95–143, 1991.

[FS90] Mark S. Fox and Katia P. Sycara. The CORTES project: A unified framework for planning, scheduling and control. In Katia P. Sycara, editor, *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GKMP90] M.M. Gorlick, C.F. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a Logical Methodology for Testing. *Journal of Logic Programming*, 8(1-2):95–119, 1990.

[GVHPZ90] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. *Computers and Mathematics with Applications*, 20(9/10):45–56, 1990.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, 1989.

[HL88] T. Huynh and C. Lassez. A CLP($\Re$) Option Trading Analysis System. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[HMS87] N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and Some Electrical Engineering Problems. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[HS91] P. Van Hentenryck and Vijay A. Saraswat. cc(FD): Constraint Programming over Finite Domains. Technical report, Computer Science Department, Brown University, forthcoming 1991.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*. MIT Press, 1991. (Revised version of SICS Research Report R91:08, to be published).

[JJL87] J. Jaffar and J.-L.Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 11–119, Jan 1987.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, W. Germany, January 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[JMSY] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP($\Re$) Language and System. *ACM Transactions on Programming Languages and Systems*. To appear. Preliminary version appeared in *Proc. Fourth International Conference on Logic Programming*, Melbourne, May 1987.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic Non-Determinism in Concurrent Constraint Programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[Koh91] Wolf Kohn. Declarative control architecture. *Communications of ACM*, 34(8):64–79, August 1991.

[LS91] Patrick Lincoln and Vijay Saraswat. Proofs as Concurrent Processes: A Logical Interpretation for Concurrent Constraint Programming. Technical report, Systems Sciences Laboratory, Xerox PARC, November 1991.

[Mah87] Michael Maher. Logic Semantics for a Class of Committed-Choice Programs. In *4th International Conference on Logic Programming*. MIT Press, May 1987.

[Sar87] Vijay A. Saraswat. The Concurrent Logic Programming Language **CP**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1991.

[Sar92] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science* (to appear), 1992.

[Sha87] Ehud Shapiro, editor. *Concurrent Prolog, Volume* 1 *and* 2. MIT Press, December 1987.

[Sim89] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[SL92] Vijay A. Saraswat and Patrick Lincoln. Higher-order, linear concurrent constraint programming. Technical report, Xerox Palo Alto Research Center, August 1992.

[Smi91] Doug R. Smith. KIDS: A Semi-Automatic Program Development System. *IEEE Transation on Software Engineering*, 1991.

[SND88] H. Simonis, H.N. Nguyen, and M. Dincbas. Verification of Digital Circuits Using CHIP. In *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988.

[SR90] Vijay A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL*. ACM, 1990.

[SRP91] Vijay A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of concurrent constraint programming. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, 1991.

[SS83] William L. Scherlis and Dana S. Scott. First Steps Towards Inferential Programming. In R.E.A. Mason, editor, *Information Processing 83*, pages 199 – 212. Elsevier Science Publishers B.V. (North Holland), 1983.

[Stu90] P.J. Stuckey. Incremental Linear Arithmetic Constraint Solving and Detection of Implicit Equalities. Submitted for publication, 1990.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986.

[VH89] P. Van Hentenryck. A Logic Language for Combinatorial Optimization. *Annals of Operations Research*, 21:247–274, 1989.

[VHD86] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, Philadelphia, PA, August 1986.

[VHG92] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 1992. To Appear. A preliminary version was presented at The International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Fl, January 1990.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical report, Artificial Intelligence Center, SRI International, August 1983.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[WWC92] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of ACM*, 35(11):89–99, November 1992.