Integrating Constraints with an Object-Oriented Language

Bjorn N. Freeman-Benson[1] and Alan Borning[2]

[1] University of Victoria, Department of Computer Science, P.O. Box 3055, Victoria, B.C., V8W 3P6, CANADA
bnfb@csr.uvic.ca

[2] Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195, USA
borning@cs.washington.edu

Abstract. Constraints are declarative statements of relations among elements of the language's computational domain, e.g., integers, booleans, strings, and other objects. Orthogonally, the tools of object-oriented programming, including encapsulation, inheritance, and dynamic message binding, provide important mechanisms for extending a language's domain. Although the integration of constraints and objects seems obvious and natural, one basic obstacle stands in the way: objects provide a new, larger, computational domain, which the language's embedded constraint solver must accommodate. In this paper we list some goals and non-goals for an integration of constraints and object oriented language features, outline previous approaches to this integration, and describe the scheme we use in Kaleidoscope'91, our object-oriented constraint imperative programming language. Kaleidoscope'91 uses a class-based object model, multi-methods, and constraint constructors to integrate cleanly the encapsulation and abstraction of a state-of-the-art object-oriented language with the declarative aspects of constraints.

1 Introduction

A constraint describes a relation that should be maintained. In a constraint-based programming language, constraints are declarative statements of relations among elements of the language's computational domain, e.g., integers, booleans, strings, and other objects. These constraints are solved by delegating to the language's embedded constraint solver the task of determining the algorithms to use and the order in which those algorithms should be applied. As demonstrated by the wide variety of systems and languages that use them, constraints are useful in programming languages, user interface toolkits, simulation packages, and many other systems. (See [16] or [29] for an overview.) Their usefulness stems from the fact that constraints are declarative: constraints emphasize the relation itself rather than the procedural steps necessary to maintain the relation. Furthermore, in general they are multi-directional; for example the single constraint $c = a + b$ can be used to find a value for any one of the variables a, b, and c, or (in combination with other constraints) values for several of the variables.

Orthogonally, the concept of an object provides a important mechanism for extending a language's computational domain. Normally, an object-oriented programming language provides both primitive objects (such as integers, booleans, and arrays), and the means for the programmer to create new application-specific objects. Most existing object-oriented languages are imperative, that is, they include state-changing operations, either provided explicitly using an assignment statement, or as primitive methods that change the state of an object.

We believe that combining the declarative relations of a constraint-based language and the extensibility of an object-oriented one leads to an interesting and useful mixed-paradigm programming language. The integration seems obvious and natural, as evidenced, for example, by the large number of object-oriented languages and environments that include some sort of constraint-like mechanism, such as a one-way dependency maintenance system (see e.g. [2, 28, 32]). However, one basic obstacle stands in the way of a thorough integration: objects provide a new, larger computational domain, which the language's embedded constraint solver must accommodate. Providing a general solver for arbitrary constraints over arbitrary domains is a completely unreasonable goal. There are good algorithms for particular classes of constraints (such linear equations over real numbers, or acyclic constraints over arbitrary domains), but not for arbitrary constraints over arbitrary domains.

The primary purpose of this paper is to describe both the obstacles to, and some solutions for, the integration of constraints into object-oriented programming languages. First, we list some reasonable goals and non-goals for such an integration. Second, we describe a number of previous approaches to the integration of constraints and objects, and why each fails to meet our goals. The five that we survey are the ones that recur most regularly in the literature, usually without a full discussion of their weaknesses. Third, we describe the design and implementation of the integration techniques we are using in Kaleidoscope'91, our second-generation object-oriented constraint imperative programming language. Kaleidoscope'91 contains a number of significant differences from its predecessor, Kaleidoscope'90 (as described in [15]). For example, Kaleidoscope'91 uses multi-methods whereas the original Kaleidoscope'90 did not; Kaleidoscope'91 has separate type and implementation inheritance hierarchies whereas Kaleidoscope'90 combined the two; and Kaleidoscope'91 has only one kind of constraint as opposed to the bewildering variety of kinds in Kaleidoscope'90. Finally, we demonstrate, using a small number of examples, the importance of multi-method dispatching in a language that integrates constraints and object-oriented features.

1.1 Goals

Previous integrations of constraints and objects have compromised the benefits of either one or the other (or both). We propose that a successful integration of the two should meet the following goals. In subsequent sections of this paper, we discuss how well Kaleidoscope'91 succeeds at this task.

1. The language design should preserve all the flexibility and expressiveness of modern object-oriented languages. Among other things, it should not rule out such features as multiple inheritance, or separate type and implementation inheritance hierarchies.

2. It should support constraints in an object-oriented style. For example, it should allow constraints to be placed on the result of sending a message to an object, rather than on the object's concrete implementation. Also, it should support user-defined constraints on user-defined objects, in the same way that existing object-oriented languages allow the user to define new kinds of objects, and methods for those objects. Constraints should interact correctly with the language's inheritance mechanism.
3. It should allow standard imperative object-oriented programs to be written without forcing an extensive alteration of programming style. It should be possible, if the constraint mechanism is not used, to write code much like that in a standard imperative language. In the imperative portions of programs that do use constraints, the programmer should not be forced to trigger the constraints manually using e.g. `self changed` messages.
4. It should solve useful collections of constraints—in other words, the integrated language should aid in solving problems that were difficult to solve before. Our original and continuing motivation for integrating constraints and objects is to support the development of interactive graphical user interfaces. Based on our experience with such interfaces, we suggest that the minimal useful collection of constraints and solvers for supporting graphical user interfaces include arbitrary constraints solved by local propagation, and linear equalities and inequalities over real numbers (which might be simultaneous).
5. It should have a clean, declarative semantics. A particular issue concerning semantics is that many of the early interactive constraint-based systems, such as the original ThingLab [5] and Magritte [19], used a *perturbation* model of constraints. In this model, the constraints describe the relations that should hold. The user or some other outside influence perturbs the system, which must then adjust itself to re-satisfy the constraints. The problem with this model is that there are often many ways of adjusting the system to re-satisfy the constraints, but no declarative specification of which way should be selected, so that this choice was often embedded procedurally into the constraint solver. In contrast, in the *refinement* model of constraints, the set of possible values for each variable is refined during the execution of the program (but never altered in other ways). This model is more declarative, and is invariably used in the constraint systems rooted in logic programming (e.g. [4, 11, 21, 35]). We therefore set as a subgoal the use of a refinement model in the integrated language.
6. Finally, it should be reasonably efficient.

1.2 Non-Goals

We also suggest a number of "non-goals" for such an integration:

1. Although it should be reasonably efficient, it need not be 100% as efficient as an optimized imperative-only object-oriented language, such as C++, Eiffel, or SELF. The additional expressiveness of the integration will offset some loss of efficiency, just as the additional expressiveness of an object-oriented language offsets the cost of doing additional pointer manipulation and run-time dispatching.

2. While we require that the language preserve object encapsulation from the programmer's point of view, we do not require that its internal runtime structures do so, as long as the language semantics is preserved. (Requiring the internal runtime structures of the constraint satisfier to preserve encapsulation would be analogous to requiring an optimizing compiler for an object-oriented language to respect encapsulation, which would preclude such optimizations as directly accessing instance variables, inlining methods, caching pointers, and producing customized versions of methods. Such optimizations are key to efficient implementations of pure object-oriented languages [10, 13].)
3. The work in the logic programming community on integrating constraints with logic programming, and on doing object-oriented programming in a concurrent logic programming framework [23, 24, 36], is quite relevant to the task at hand. However, for the current experiment, we wish to remain rooted in the imperative, object-oriented world; we do not set as a goal supporting logic variables or backtracking. Also, we wish to represent objects in a more-or-less standard manner as encapsulated chunks of state and behavior, rather than in the concurrent logic programming style as perpetual processes consuming a lazily-produced stream of messages. Our reasons for making these choices in the current design are: first, as noted above, we believe that the integration of constraints with an imperative object-oriented language is itself a natural and useful step; and second, we believe such languages will make it easier for more conventionally-trained programmers to move from existing object-oriented languages to constraint imperative languages, thus making it more likely that such languages could achieve widespread use. In future versions of Kaleidoscope, we will likely attempt to integrate more of the features of logic programming with the language.

2 Constraints

Formally, a constraint is a relation over some domain $\mathcal{D}$. In an object-oriented language, the domain $\mathcal{D}$ includes arbitrary user-defined objects, as well as primitives such as integers and real numbers. A ***constraint graph*** $G = \langle C, V, \mathcal{D}\rangle$ is a set of constraints C, their variables V, and their domain $\mathcal{D}$. A ***valuation*** θ is a function that maps the variables of a constraint graph to elements of the domain $\mathcal{D}$. A ***solution*** S to a constraint graph is the set of all valuations that satisfy all of the constraints in the graph. A ***flat constraint system*** is one in which all valuations in a solution must satisfy all of the constraints completely. For example, if the constraint graph contains the canonical Celsius–Fahrenheit–Kelvin graph over real numbers:

$$G = \left\langle \left\{ \begin{array}{l} c * 1.8 = f - 32.0 \\ c = k - 273.13 \end{array} \right\}, \{c, f, k\}, \mathcal{R} \right\rangle$$

then one possible valuation is: $\theta = \{c, f, k \mapsto 0.0, 32.0, 273.13\}$
and the solution is the infinite set of valuations:

$$S = \left\{ \begin{array}{lrr} \{c, f, k \mapsto & 0.0, & 32.0, 273.13\} \\ \{c, f, k \mapsto & -40.0, & -40.0, 233.13\} \\ \{c, f, k \mapsto & 100.0, & 212.0, 373.13\} \\ & \ldots & \end{array} \right\}$$

2.1 Constraint Hierarchies

We have found it useful to extend the constraint paradigm to allow both *required* and non-required, or *preferential*, constraints. The required constraints must hold for all solutions, while the preferential constraints should be satisfied if possible, but no error condition arises if they are not. A *constraint hierarchy* can contain an arbitrary number of levels of preference, in either a total or partial order. The solutions to a constraint hierarchy are the valuations (mappings from variables to values) that best satisfy the constraints in the hierarchy, respecting their relative strengths. References [4, 18] present a formal description of the theory of constraint hierarchies, while [16, 17] describe two algorithms, DeltaBlue and DeltaStar respectively, for finding solutions to them.

Extending the Celsius–Fahrenheit–Kelvin example with a constraint hierarchy results in the constraint graph:

$$G = \left\langle \left\{ \begin{array}{l} \texttt{required}\ c * 1.8 = f - 32.0 \\ \texttt{required}\ c = k - 273.13 \\ \texttt{strong}\ c = 0.0 \\ \texttt{weak}\ f = 0.0 \end{array} \right\}, \{c, f, k\}, \mathcal{R} \right\rangle$$

The solution to this graph is the singleton set of the best valuation (note that this valuation satisfies the three strongest constraints at the expense of the single weakest one):

$$S = \{\{c, f, k \mapsto 0.0, 32.0, 273.13\}\}$$

3 Previous Constraint–Object Integrations

As we mentioned earlier, there are a number of obstacles to the successful integration of constraints and objects.

- **(The Meaning of Constraints)** One obstacle is that standard object-oriented languages provide excellent mechanisms for defining new methods, but (obviously) no mechanisms for defining new constraints. In those cases where constraint definition mechanisms are provided, the mechanisms usually do not respect the object-oriented nature of the base language.
- **(Constraints on Parts and Wholes)** A second difficulty is that constraints can be placed at many different levels of a part-whole hierarchy. For example, consider the following two constraints—one between two whole objects, the other between a part and a constant:

  ```
  var c1, c2 : Circle;
  always: c1 = c2;
  always: c2.center = 10@23;
  ```

  Most existing constraint solving algorithms are tailored to primitive domains such as integers, reals, or booleans, and thus cannot handle this kind of structured information directly.

- **(Declarative versus Imperative)** A third difficulty with the integration of constraints and object-oriented programming languages is that constraints are declarative and object-oriented languages are usually imperative. Thus any useful integration must provide a solution to the interaction between destructive assignment and declarative constraint satisfaction. For example, what happens when the program assigns 42 to a variable that is constrained to contain anything except 42? Does the assignment fail, and if so, what does that mean? What if the program assigns 42 to a variable that is weakly constrained to be 33? Does the variable remain 42 after the assignment, or does it revert to 33?

  Furthermore, the integration must consider not only how the constraints are solved, but when they are solved. Is the solver automatically invoked whenever a variable changes, or does is the programmer given the burden of explicitly triggering the solver when necessary?

In reviewing previous efforts to overcome these obstacles, one finds a small number of approaches, none of which meet all of our goals:

1. Use local propagation exclusively.
2. Limit constraints to primitive leaf objects.
3. Link in a new constraint solver.
4. Use a graph rewriting system.

The following sections discuss these basic approaches.

3.1 Local Propagation

By far the most common technique is local propagation. Local propagation starts with some known set of values and then, using a single constraint, determines another value. This may in turn let yet another value be determined by another constraint, and so forth. The popularity of local propagation is due to its simplicity and speed. Local propagation works very well when local choices of constraint satisfaction can lead to globally satisfactory solutions. Unfortunately, local propagation is incapable of solving cyclic constraints (i.e., simultaneous equations) and partial information constraints (e.g., greater-than). Furthermore, local propagation will not work correctly when constraints are used at different levels in a part-whole hierarchy unless information is available to connect parts to wholes, and wholes to parts.

There are two basic phases in local propagation: choosing which constraints to use, and then executing those constraints. Dataflow, or blind, local propagation systems use local knowledge both for choosing and executing constraints. Planned local propagation systems use global knowledge for the choosing phase, and thus can solve more complex constraint graphs than dataflow propagation. Some local propagation systems (particularly blind ones) interleave the two phases. This has the advantage of simplicity. On the other hand, an advantage of separating the phases is that one particular choice of constraints (a "plan") can be reused until the constraint graph topology changes, thus resulting in better execution speed.

Representative examples of systems that use local propagation include Alien [12], Garnet [33], Smalltalk's MVC mechanism[3] [28], and ThingLab II [31]. Alien, Garnet, and MVC all use dataflow propagation, whereas ThingLab II uses planned propagation. Although these systems are all reasonably efficient (meeting goal 6), they are strictly limited in the kinds of constraints they can solve (not meeting goal 4). Furthermore, these systems all require a different programming style (not meeting goal 3): either the use of special ***constrainable variables*** or the sending of **self changed** messages. None of these systems solve cyclic constraints. However, all but MVC will detect cycles and not go into an infinite loop; with MVC, this detection is the programmer's responsibility.

3.2 Constraints on Primitive Leaves

A less common, but still simple, technique for integrating constraints and objects is to separate the domain into two subdomains: $\mathcal{D}_c$ and $\mathcal{D}_p$. $\mathcal{D}_c$ contains the complex user-defined objects (e.g., employees, loans, and machine tools), and $\mathcal{D}_p$ contains the pre-defined primitive objects (e.g., integers, reals, and booleans). The kinds of constraints available are pre-defined, and constraints can only be asserted on variables that range over the primitive domain $\mathcal{D}_p$. In other words, constraints can only be asserted on the primitive leaf objects in the part-whole hierarchy.

There are a number of well known algorithms for solving constraints over these primitive domains, including local propagation, Gaussian elimination, and the Simplex algorithm. However, because the set of constraints is fixed by the system, the programmer cannot create new application specific constraints. For example, if the programmer needs to assert `cos(x) = y`, but the built-in set does not include `cos`, then the programmer is out of luck. Further, the programmer cannot define constraints over application specific objects. For example, the constraint `x + y = z` is legal if `x`, `y`, and `z` are integers or reals, but not if they are points or sets. The program can explicitly "split" a constraint on compound objects into a set of constraints on leaf objects, but the division is done only once and is based on the concrete implementation of the objects. This works well in a logic programming environment in which there is no mutable state, and is frequently used in programs written in languages such as CLP($\mathcal{R}$) [22]. However, it is problematic in an object-oriented language, since the splitting immediately becomes invalid if either the objects or the implementations change. For example, one could manually divide the `p + q = r` constraint over `CartesianPoints` into two leaf constraints: `p.x + q.x = r.x` and `p.y + q.y = r.y`. However, doing so effectively "freezes" the implementations of p, q, and r—they can no longer store `PolarPoints`, `3DPoints`, or `Pixels`. As another example of this difficulty, suppose we have a constraint that one tree is the mirror image of another. We could split the `mirror` constraint into primitive equality constraints on the corresponding leaves, but this splitting would become invalid as soon as the shape of one of the trees changed.

Worse, from an object-oriented point-of-view, this constraints-on-leaves scheme forces the programmer to violate encapsulation when asserting inter-object con-

[3] As one anonymous referee noted, MVC is more primitive than the other systems listed here, and perhaps should be called a communication mechanism rather than a constraint system.

straints. Constraints must be asserted on the leaves, and the leaves can only be accessed by explicitly descending the part-whole hierarchy and accessing instance variables. For example, consider two **Rectangle** objects constrained to have the same center. Using constraints-on-leaves the necessary constraints are:

```
r1.origin.x + r1.corner.x = r2.origin.x + r2.corner.x;
r1.origin.y + r1.corner.y = r2.origin.y + r2.corner.y;
```

However, the correct, encapsulation-preserving constraint would be:

```
r1.center = r2.center;
```

Note that the undesirable set of constraints forces (i) the **Rectangle** class to have **origin** and **corner** instance variables and (ii) those instance variables to contain **CartesianPoint** objects. Assuming reasonable abstraction principles and encapsulation mechanisms, the programmer should not need to know anything about the internal implementation of **Rectangle**, much less about what implementations of **Points** are used.

A representative example of a system that uses this two-domain, constraints-on-leaves technique is the COOL system [1]. This system appears to be reasonably efficient (meeting goal 6), but does not preserve the object-oriented programming style when defining or using constraints (not meeting goal 2).

3.3 New Constraint Solvers

The most powerful of the common techniques for integrating constraints and objects is to define a new constraint solver to handle the new domain. The new solver can be provided by extra procedures in the program itself or, more often, by linking a new solver into the runtime system. In either case, the result is two solvers, one for each of the two domains: the primitive domain $\mathcal{D}_p$ and the user-defined objects $\mathcal{D}_c$. Typically, each of the two solvers will be an efficient algorithm tailored to the domain, i.e., based on domain-specific knowledge.

Unfortunately, in general, this technique suffers from the same inflexible nature as the constraints-on-leaves technique. For example, if an extra solver has been written to solve constraints over the user-defined objects in $\mathcal{D}_c$, that solver will not solve constraints over a different set of user-defined objects $\mathcal{D}_d$. Second, and perhaps worse, having separate solvers for separate domains usually precludes any inter-domain constraints! Thus, if $\mathcal{D}_p$ is the real numbers and $\mathcal{D}_c$ is geometric figures, the following figure-to-real constraint would not be supported by a two-solver system:

```
myCircle.radius = 15.2*x - 9.4*y;
```

A single solver for the unified domain of figures and real numbers could handle that constraint, but not a system with separate solvers for separate domains. The problem is that the integration of two domains requires the integration of two solvers. However, constraint solvers are typically not written to support integration—their algorithms assume an internal database representing the complete state of the world, rather than an incomplete state that must be communicated to and from other solvers.

Representative examples of systems that use this new solver approach are the constraint systems for graphical objects described in [25] and [37]. These systems are efficient (meeting goal 6) and provide constraints at a higher level of abstraction than just the real numbers (partially meeting goal 4), but do not allow the programmer to define new kinds of constraints (not meeting goal 2).

3.4 Graph Rewriting

A number of systems have integrated constraints and objects by using a graph rewriting system to rewrite the constraint graph with the goal of producing a single solution object, or a local propagation dataflow plan. Some of these systems are implemented on top of an existing object oriented language, whereas others define the object model using rewrite rules as well. The most "object oriented" of these systems encapsulate the rewrite rules within the objects (or their classes). Thus the definition of an object includes its state (instance variables), its behavior (methods), and the rewrite rules that are applicable to it.

The biggest advantages of the graph rewriting technique are that encapsulation is preserved (goal 2), and that the set of objects and their constraints are easily extendible (goals 1 and 2). However, the biggest disadvantage is that the constraints over $\mathcal{D}_c$ are not solved directly. The constraint solver rewrites the graph to a canonical representation, and then passes this residual graph to a primitive constraint solver. This two-level constraint solving algorithm can be inefficient (not solving goal 6). More importantly, graph rewriting is a fundamentally different execution model from the standard imperative one, bringing this technique in conflict with goal 3.

The first programming language to use graph rewriting to integrate objects and constraints is Bertrand [29]; later systems of this type are Equate [39], and Siri [20]. A primary design goal of Equate is preserving object encapsulation in the internal operation of the constraint satisfier, which, as noted above, we view as an inappropriate goal. Siri has a completely integrated object model similar to that of BETA [27, 26] and thus better satisfies goal 2.

3.5 Other Techniques

Another technique that has been used to integrate constraints and objects is the *path* approach used in the original ThingLab [5]. The ThingLab constraint solver was essentially an augmented planned local propagation system—it used both local propagation and iterative relaxation. The planning phase was based on virtual variables. Each virtual variable was linked to a real variable through a path: an ordered sequence of part names that defined a path from the root object to the variable being constrained. The planner would use the paths to avoid conflicts, cycles, and various dataflow problems that are common to local propagation systems. However, because the planner had no information about the implementation of the constraints or the objects, using local propagation alone, it was unable to handle simultaneous constraints on an object, unless it was explicitly told that those constraints applied to separate parts of the object. For example, it could not handle the following set of constraints (which might occur in a multi-user user interface) using local propagation:

```
var p, q, r : Point;
horizontal( q, r );
vertical( p, q );
mouse1 = p;
mouse2 = r;
```

Thus, the ThingLab constraint solver was reasonably efficient (meeting goal 6) and did not require a new programming style (meeting goal 3). However, it required a static part-whole hierarchy, and (in the original version) required the parts on the paths to be stored as instance variables rather than computed; both of these restrictions violate goal 2.

4 The Kaleidoscope'91 Approach

After discovering that none of the existing approaches to the integration of constraints and objects met our requirements, we considered modifying an existing object-oriented language or designing our own. We began by defining the basic semantics of such an integration: the object-oriented constraint imperative programming framework. As we were unable to find an existing object-oriented programming language that had both a simple semantics and support for multi-methods, we designed and implemented a new language, Kaleidoscope'90, as an initial instance of this framework. As described in the introduction, we are now implementing a successor language, Kaleidoscope'91, based on the lessons learned from this initial experiment. The remainder of this paper describes the interesting features of our second language, in particular those aspects that deal with the integration of constraints and objects.

4.1 Object Model

One task in the design of a language that integrates constraints and objects is the specification of the object model. Most of the choices here (for example, prototypes versus classes, single versus multiple inheritance, static versus dynamic type checking, combined or separate type and implementation hierarchies) are independent of the particular problems of making the integration. One attribute that is not independent, however, is single dispatching versus multiple dispatching. Kaleidoscope'91 adopts the multiple dispatching techniques of Cecil [9], rather than the single dispatching of Smalltalk and C++. Multi-methods are particularly appropriate in a constraint language, since the multi-directional nature of constraints means that, in general, any argument could be either an input or an output. Thus dispatching on all arguments makes more sense than on any single argument. For example, single dispatching on the message receiver does not work when the message receiver is unknown. Consider the following situation: variables `x`, `y`, and `z` are constrained with `x + y = z`, i.e., the constraint `x.plus(y,z)`. Also, it is known that `y = 3` and `z = 5`, but `x` is unknown. Obviously, one would conclude that the `Integer plus` constraint should be used, but that conclusion could only be arrived at using the types of the arguments y and z, i.e., using multi-method dispatching[4].

When a message is sent, the implementation inheritance graph of each argument is searched for applicable methods. All applicable methods (both local and inherited) are placed into a partial order by the inheritance graphs. If there is a unique lower bound to this partial order, then that method is executed. If there is no unique lower bound, then an "ambiguous message send" error is raised and the program halts. (In other words, we take a conservative approach to multi-method selection: if it is unambiguous which method is to be used, that method is selected; otherwise it is a runtime error. This approach is like the one taken in the Smalltalk multiple inheritance extension [6] and in Cecil, and in contrast to the more elaborate approach in CLOS, in which a linear ordering is imposed on the inheritance graph.)

Kaleidoscope'91 has also inherited Cecil's optional and incremental static type checking system. Type declarations on variables and method arguments are strictly optional, although if they are used, the compiler will ensure that all uses of that variable or argument are type safe. The type inheritance hierarchy in Kaleidoscope'91 (and in Cecil) is separate from the implementation inheritance hierarchy. Kaleidoscope'91 uses a class-based object model (in contrast to Cecil), since the choice of prototypes versus classes is orthogonal to our research focus of integrating constraints with object-oriented programming, and since the class-based design seemed more stable.

Constraints are applied to objects during the execution of the program as constraint statements are encountered. An example statement in Kaleidoscope'91 is:

```
always: c1.radius = 1.0;
```

Executing this statement permanently asserts a constraint that specifies that the result of sending the `radius` message to `c1` will be equal to `1.0`. Note that asserting a constraint is different than solving it—the asserted constraints define a constraint graph, which is solved either when time is advanced (see Sect. 4.2), or when an actual value of a variable should be necessary, e.g., for output to the screen or for use in a conditional branch.

A class definition can include constraints that are automatically applied to its instances. For example, `Circle`, `UnitCircle`, and `FilledUnitCircle` classes could be defined as follows, giving `FilledUnitCircle` three instance variables and one constraint:

```
class Circle inherits from Object
  var radius : Float;
  var center : Point;
end Circle;

class UnitCircle inherits from Circle
  always: required radius = 1.0;
end UnitCircle;
```

[4] Note that `x.plus(y,z)` is a statement of a constraint (a relation) rather than a statement of a functional computation. Thus there is no "result" as there is in functional or imperative expression.

```
class FilledUnitCircle inherits from UnitCircle
  var color : RGBColor;
end FilledUnitCircle;
```

In our effort to explore the limits of the constraint imperative programming framework, Kaleidoscope'91 uses constraints to implement type declarations. Thus the declaration:

```
var p : Point;
```

defines both the variable p and the type constraint "`protocol_conforms(p,Point)`". Thus both the compiler and the run-time system use the same constraint solver: the compiler for type inference, and the run-time system for solving dynamically created constraints.

4.2 Time and Assignment

The second task in the design of an integrated language is devising a solution to the conflict between declarative constraints and imperative assignment. One of our goals (3) is to allow standard imperative object-oriented programs to be written without forcing an extensive alteration of programming style. Therefore, Kaleidoscope'91 must smoothly integrate these two apparently incompatible paradigms. The standard imperative assignment statement is a destructive operation: executing the statement `left := right` causes the value of the `right` expression to be stored in the `left` variable. However, in a constraint imperative language, the `left` variable might be connected, through constraints, to other variables. To treat assignment cleanly and uniformly in Kaleidoscope'91, assignment is defined as an equality constraint between the *previous* value of the `right` expression and the *current* value of the `left` variable. Thus, all expressions denote constraints, and so all computation is handled by a single constraint solving mechanism—neither the constraints nor the imperative object-oriented code is treated specially. Semantically, variables in Kaleidoscope'91 contain streams of pellucid values, in analogy with Lucid [38]—one pellucid value for each integral time interval. In Kaleidoscope'90, the programmer could reach arbitrarily far into the past to reference an old state, requiring the implementation to store an unbounded number of pellucid variables. This was expensive to implement, and allowed confusing (although interesting) programming styles. In reaction to this, Kaleidoscope'91 has been restricted so that the programmer can only refer to two states of a variable: the current state and the previous state.

All expressions are defined relative to a *current time*. For example, if the current time is 6, then the expression `c * 1.8 = f - 32.0` denotes $c_6 * 1.8 = f_6 - 32.0$, and the assignment `x := y + z` denotes $x_6 = y_5? + z_5?$. (The "?" annotations on the variables y_5 and z_5 indicate that they are read-only, so that information can only flow from the past to the present, and not vice versa. The "?" annotation, as well as a write-only annotation "!", is also available at the language level.) The current time is advanced at the end of each statement. In addition, we can create a compound statement, to be executed without advancing the clock between the two sub-statements, by using the `||` operator. Thus, for example, the statement `x := y || y := x;` swaps `x` and `y` without using a temporary.

The duration for which a constraint expression is asserted can vary. By default, assignments are asserted "just once," and normal constraints are asserted "from now on." However, by using the keywords `once:`, `always:`, and `assert...during...` the programmer can specify a different duration for any expression. Semantically, each constraint expression specifies that a new constraint instance be created for each time at which the expression is active, e.g., the constraint `c * 1.8 = f - 32.0` creates $c_7 * 1.8 = f_7 - 32.0$ at time 7, $c_8 * 1.8 = f_8 - 32.0$ at time 8, and so on. Weak stay constraints ($x_i = x_{i-1}?$) are used to propagate values forward in time when no other constraints apply; in other words, in the absence of other stronger constraints, the values of variables will remain the same as time advances.

This semantics for time and assignment, we believe, meets goal 5 (a clean, declarative semantics). In particular, in the description of that goal, we contrasted the perturbation and refinement models of constraint satisfaction, preferring the refinement model. On the face of it, the refinement model is incompatible with an imperative programming style. However, our semantics for variables (a variable holds a stream of pellucid variables, with weak stay constraints to implement the frame axioms relating successive elements in the stream), allows us to provide a refinement-based semantics for our language.

4.3 Constraint Constructors and Procedures

The third task in the design of an integrated constraints-and-objects language is support for user-defined constraints over user-defined objects. Mechanisms are needed both to define the meaning of constraints, and to support constraints at different levels of the part-whole hierarchy. As described in Sect. 3.4, the most object-oriented technique for this support is to use the objects themselves (or their classes) to define the meanings of the constraints. In the graph rewriting systems described earlier, this was accomplished by encapsulating rewrite rules within the objects. In Kaleidoscope'91, however, the meanings of the constraints are defined using the same imperative object-oriented constructs as are used in the rest of the language. Thus, a Kaleidoscope'91 programmer does not need to learn a separate sub-language for each of the two integrated paradigms: constraints and objects.

Constraints in Kaleidoscope'91 are asserted by a statement containing a duration keyword, an optional strength, and an expression. The expression is a normal object-oriented expression, i.e., each operator is dispatched individually and the run-time offers no pre-determined meaning to any operator. For example, the expression in following constraint statement is translated into three sub-expressions—one for each operator:

```
                                          *( c, 1.8, tmp1 )
always: required c * 1.8 = f - 32.0;  ⇒   -( f, 32.0, tmp2 )
                                          =( tmp1, tmp2 )
```

The objects involved in each sub-expression determine the implementation of that sub-constraint through the multi-method dispatch described in Sect. 4.1. However, instead of dispatching to normal object methods, constraint sub-expressions dispatch to *constraint constructors*: side-effect-free multi-methods. For example, the `Point` addition constraint constructor is:

```
constructor +( p, q, r : Point )
  always: p.x + q.x = r.x;
  always: p.y + q.y = r.y;
end +;
```

This constructor splits the **Point** addition constraint into two smaller addition constraints on its component **x** and **y** instance variables. The meanings of these smaller constraints are recursively defined using further constructors and so on, until the objects being constrained are elements from the primitive domain $\mathcal{D}_p$, and a primitive constraint solver can be used. In other words, the Kaleidoscope'91 solution to the problem of having constraints at different levels of the part-whole hierarchy is to decompose all constraints as far as possible, thus allowing the primitive constraint solvers to deal with a flat, unstructured domain.

Because constraint constructors are methods rather than rewrite rules, they can include arbitrary computations—assignment, iteration, recursion, and so forth—in addition to further constraints, thus making them more natural to write for a programmer accustomed to the imperative style. To preserve the declarative properties of constraints, the language prohibits constructor methods from having non-local side-effects. Thus constraint constructors may not constrain out-of-scope variables nor advance the global time. Instead, each constraint constructor is executed in its own *time scope* and thus with its own interpretation of "current" and "previous."

Furthermore, note that `q.x` in the above constraint constructor is not a direct reference to the **x** instance variable of the object contained in **q**. Rather, `q.x` preserves encapsulation by sending the **x** message to **q**, which allows the **q** object to return either a stored instance variable or a computed (or "virtual") variable. Naturally, this computation is done with multi-directional constraints, and thus values can flow in as well as out. For example, if **Points** are represented in polar coordinates, the **x** access method called from the above **Point** addition constructor would be:

```
constructor x( p : Point, x )
  always: x = r * cos(theta);
end x;
```

4.4 Implementation

The Kaleidoscope'90 interpreter included a direct naive implementation of the semantics of time described in Sect. 4.2, using lists of pellucid values, thousands of constraints, lots of garbage collection, and very low performance. Based on that experience, the semantics of Kaleidoscope'91 were restricted (as described in Sect. 4.2) with one consequence being that an implementation need only store two pellucid values for each variable. We anticipate that this restriction will eliminate much of the creation, and subsequent collection, of constraint instances upon each time advance.

However, the greatest savings in execution time will come from moving as much of the constraint satisfaction problem from run time to compile time as possible. When the compiler can statically determine which constraints will be active at run time, it can use the constraint compiling technology of [14] to produce a short sequence of instructions instead of a run-time constraint solver call. For example, if there is an assignment to an otherwise unconstrained variable, rather than representing the assignment as a run time constraint and solving it, we can compile it as a simple store instruction.

However, not all constraint satisfaction can be done at compile time, no matter how clever the algorithms. For the run time constraint satisfaction that remains, the implementation of Kaleidoscope'91 relies on the efficient implementation of two basic features: the streams of values over time, and a two-level constraint solver. The two levels in the constraint solver are for the user-defined constraints over $\mathcal{D}_c$, and for the primitive constraints over $\mathcal{D}_p$. For both Kaleidoscope'90 and Kaleidoscope'91, the primitive domain $\mathcal{D}_p$ consists of three sub-domains: real numbers, booleans, and bitmaps. Each sub-domain has its own primitive constraint solver: a combination local propagation/Simplex algorithm for real numbers, a finite-domain solver [30] for booleans and type constraints, and a local propagation solver for bitmaps. These three solvers use a variation of the algorithm described in [34] to cooperate when solving inter-domain constraints. The idea behind the algorithm is that the primitive constraint solvers communicate relevant variable bindings via inter-domain constraints. Each inter-domain constraint remains dormant until a sufficient number of its variables become known, at which point the inter-domain constraint is replaced by two or more single-domain constraints. It is these smaller constraints that contain the information (variable bindings) being communicated between the solvers.

The user-defined constraint solver is, itself, divided into two parts: a type constraint solver and a constraint constructor execution unit. Each constraint expression in Kaleidoscope'91 defines two constraints: a constraint on the types of its variables, and a constraint on the values of its variables. For example, an `x + y = z` constraint expression defines a type constraint that restricts `x`, `y`, and `z` to contain objects such that a `+(...)` constraint constructor exists for those objects. It also defines a "value constraint" such that once types are found for those objects, the appropriate constraint constructor is executed. These constraint constructors may assert further constraints, leading to further type constraints and constraint constructor calls until, eventually, the process bottoms out and the constraint constructors create primitive constraints over the primitive domain $\mathcal{D}_p$.

Although this scheme is flexible and powerful, the fact that it splits and then delegates responsibility for solving the constraints can be a drawback. Algorithms that deal with complex constraints at the original level of abstraction (such as those described in Sect. 3.3) can be more efficient than those that solve the same problem using a myriad of lower-level constraints. We are working to rectify this problem by providing a pluggable solver interface to the Kaleidoscope'91 run-time. Thus, while constraint constructors will be used to define most constraints, specialized constraint solvers for application-specific objects could be added when necessary, which would interact correctly with existing solvers.

Thus the overview of the Kaleidoscope'91 constraint solver is: first, solve the type constraints; second, execute the constraint constructors; and third, solve the primitive constraints. The complete algorithm is omitted here due to space restrictions, but is available in [18]. The complete algorithm is incremental, and accommodates the constraint constructors asserting further constraints (or local constraints), which then require solving additional type constraints, calling additional constraint constructors, and so on.

5 Examples

As we stated earlier, our original and continuing motivation for both Kaleidoscopes, and the more general constraint imperative programming framework, is to support the development of interactive graphical user interfaces. Language features of Kaleidoscope'91, such as the constraint hierarchy, multi-methods, and constraint constructors are particularly useful in such programs.

As an example of the sort of code we have written, consider the following code fragment from the `Thermometer` class in the Celsius–Fahrenheit–Kelvin application. Constraints are used, not only for typical arithmetic relations, but also to define the user interaction and the graphical appearance. Note that because the graphical appearance is defined with constraints, it will be updated automatically as the other constraints are solved:

```
always: temperature = mercury.height / scale;
always: white_rectangle( thermometer );
always: grey_rectangle( mercury );
always: text_pos = (thermometer.right)@(mercury.top);
always: display_number( temperature, text_pos );

if( near( mouse, mercury ) ) then
    assert
        medium mouse.location.y = mercury.top;
    while( mouse.button = down );
elseif( near( mouse, menu.top ) ) then
    ...
endif;
```

As a second example, consider the definition of an object dragging routine for an object-oriented graphical drawing program such as [3]. The object being dragged usually follows the mouse. However, if the dragged object is moved within the "gravity" region of another object, the dragged object sticks to the fixed object while minimizing the distance between itself and the mouse. The following constraint hierarchy can be used to specify this behavior:

```
always: weak cursor = mouse;
always: strong stick_to( dragged, near );
always: required near = nearest_or_nil( mouse );
always: required dragged.center = cursor;
```

The **weak** equality constraint provides the default tracking behavior and the `strong stick_to` constraint provides the sticking. However, the implementation of the `stick_to` constraint is different for each pair of graphical objects (circle-circle, circle-line, rectangle-triangle, polygon-spline, triangle-rectangle, etc.) and thus multi-method dispatching is used to invoke different `stick_to` constraint constructors as the mouse moves near different objects.

6 Conclusion

The Kaleidoscope'91 language and implementation meets five of our six original goals, and provides the power of constraints in the framework of an object-oriented language. The one goal that it does not yet meet is number 6: reasonable efficiency. We hypothesize that the Kaleidoscope'91 compiler-interpreter pair will be much more efficient than the proof-of-concept Kaleidoscope'90 interpreter; this hypothesis will be supported or refuted once our implementation is completed. To this end, our implementation and research effort is divided between improving our constraint solvers and developing specialized compilation techniques.

In general, we believe that the object-oriented constraint imperative programming framework is a promising direction in language design because it combines the declarative aspects of constraint programming with the abstraction, encapsulation, and destructive assignment aspects of imperative object-oriented languages. This marriage allows the programmer to use the correct paradigm (declarative or imperative) for each task, rather than forcing the entire application to be molded into a single model. Furthermore, the availability of two paradigms will encourage the development of novel dual-paradigm algorithms, such as the associative array and set implementations described in [7, 8].

Acknowledgements

This work has been supported by the University of Victoria, and by the National Science Foundation under grant IRI-9102938. Craig Chambers, Denise Draper, Gus Lopez, and Michael Sannella have all been actively participating in the design of Kaleidoscope'91; and, along with the anonymous referees, also provided valuable suggestions on improving this paper. We would also like to thank John Maloney, Brad Meyers, and Bradley Vander Zanden for their suggestions and advice.

References

1. P. Avesani, A. Perini, and F. Ricci. COOL: An Object System with Constraints. In *TOOLS 2*, June 1990.
2. Paul Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.
3. Eric A. Bier and Maureen C. Stone. Snap-Dragging. In *Proceedings of SIGGRAPH'86*, Dallas, Texas, August 1986. Also in *Computer Graphics* 20(4), August 1986.
4. Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.
5. Alan H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.
6. Alan H. Borning and Danial H. H. Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, Pittsburgh, Pennsylvania, August 1982. American Association for Artificial Intelligence.

7. Timothy A. Budd. Blending Imperative and Relational Programming. *IEEE Software*, 8(1), January 1991.
8. Timothy A. Budd. Multiparadigm Data Structures in Leda. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, April 1992.
9. Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1992.
10. Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings of the 1991 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Phoenix, October 1991.
11. Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, July 1990.
12. Eric Cournarie and Michel Beaudouin-Lafon. ALIEN: A Prototype-based Constraint System. In *Preprints of the Second Eurographics Workshop on Object Oriented Graphics*, pages 93–114, Texel, The Netherlands, June 1991. To be published in revised form by Springer-Verlag.
13. L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the Eleventh Annual Principles of Programming Languages Symposium*, pages 297–302, Salt Lake City, Utah, January 1984. ACM.
14. Bjorn Freeman-Benson. A Module Compiler for ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, New Orleans, October 1989. ACM.
15. Bjorn Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages, and Applications, and European Conference on Object-Oriented Programming*, pages 77–88, Ottawa, Canada, October 1990. ACM.
16. Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.
17. Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. In *Proceedings of the Eleventh Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1992. IEEE. To appear.
18. Bjorn N. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as Department of Computer Science and Engineering technical report 91-07-02.
19. James A. Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Department tech report CMU-CS-83-132.
20. Bruce Horn. A Constrained-Object Language for Reactive Program Implementation. Technical Report CMU-CS-91-152, School of Computer Science, Carnegie-Mellon University, June 1991.
21. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987.
22. Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Language and System. Technical Report CMU-CS-90-181, School of Computer Science, Carnegie Mellon University, October 1990. To appear in *ACM Transactions on Programming Languages and Systems*.
23. Kenneth Kahn, Eric Tribble, Mark Miller, and Daniel Bobrow. Objects in Concurrent Logic Programming Languages. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 242–257, Portland,

Oregon, September 1986. ACM.

24. Kenneth M. Kahn. Objects—A Fresh Look. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1990.
25. Glenn Kramer, Jahir Pabon, Walid Keirouz, and Robert Young. Geometric Constraint Satisfaction Problems. In *Working Notes of the AAAI Spring Symposium on Constraint-Based Reasoning*, pages 242–251, Stanford, March 1991.
26. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller Pedersen, and Kristen Nygaard. Object Oriented Programming in the Beta Programming Language. Draft of unpublished book, 1991.
27. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller Pederson, and Kirsten Nygaard. Abstraction Mechanisms in the BETA Programming Language. In *Proceedings of the Tenth Annual Principles of Programming Languages Symposium*, Austin, Texas, January 1983. ACM.
28. Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*, volume II. Prentice Hall, Englewood Cliffs, NJ, 1991.
29. William Leler. *Constraint Programming Languages.* Addison-Wesley, 1987.
30. Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
31. John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering technical report 91-08-12.
32. Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.
33. Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive Graphical User Interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.
34. Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages.* ACM SIGPLAN, 1978.
35. Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.
36. E. Shapiro and A. Takeuchi. Object-Oriented Programming in Concurrent Prolog. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, chapter 21. MIT Press, 1987.
37. Remco C. Veltkamp. A Quantum Approach to Geometric Constraint Satisfaction. In *Preprints of the Second Eurographics Workshop on Object Oriented Graphics*, pages 53–67, Texel, The Netherlands, June 1991. To be published in revised form by Springer-Verlag.
38. William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.
39. Michael Wilk. Equate: An Object-Oriented Constraint Solver. In *Proceedings of the 1991 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 286–298, Phoenix, October 1991.

This article was processed using the $\LaTeX$ macro package with LLNCS style