Test-case generation driven constraint solving for object-oriented languages

No Author Given

No Institute Given

1 Research objectives

The main objective of the current proposal is to investigate how constraint solving capability can be added to an object-oriented language while diminishing the need to write custom solvers or propagators. In particular we aim to investigate the following:

1. To what extent can the basic concepts underlying the object-oriented programming paradigm (classes, objects, methods) be used to express constraints over a class' possible instantiations.
2. To what extent can techniques from automated test-case generation be used for deriving class instantiations satisfying these constraints.

2 State of the art

Over the years, constraint programming has been developed as a programming paradigm particularly tailored towards solving constraint satisfaction problems. Arguably the most interesting advantage of the paradigm lies in the fact that it allows one to write down the key characteristics of a solution to the problem of interest, while the search for a solution itself is left in the hands of the system. This declarative way of problem solving is inherently different from a more classical algorithmic approach in which the programmer needs to write down the series of subsequent steps that are needed to compute the solution.

Formally, a constraint satisfaction problem (CSP) is represented by a triple (Z, D, C) where Z is a finite set of variables, D is a function mapping the variables in Z to a set of objects of arbitrary type (the *domain* of the variables) and C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z. A solution to a to a CSP (Z, D, C) is a simultaneous assignment of values (from the respective domains of each variable provided by D) to the whole set of variables Z in such a way that the constraints in C are satisfied []. The main techniques for solving CSPs are based on search and backtracking [] often combined with propagation or domain reduction techniques such as node-consistency [] and (hyper) arc consistency [].

Whereas constraints offer a simple, neat and thus attractive way to model and solve CSPs, they are usually mixed with other programming language constructs

– be it declarative, functional or imperative – to create a real programming language in which constraint solving can be combined with traditional algorithmic problem solving and that provides support for interfacing with the outside world.

Although programming languages exist in which constraints are fully embedded such as Kaleidoscope [3], Oz [4], Curry [2] and different dialects of Prolog [], mainstream imperative and object-oriented languages usually allow for adding constraint-solving capabilities by interacting with software libraries. Examples of the latter include Choco [] for Java, Gecode [] for C++, and Numberjack [] for Python.

While creating and solving constraints through a library offers certain advantages – the most visible arguably being the freedom of choice among several existing libraries and/or solvers – it also comes with a number of severe drawbacks. First of all, constructing and solving constraints through the library's API typically involves non-trivial syntactic manipulations in order to construct constraints using the host language's primitive data types, call the solver, and interpret the answers. Subsequently, constraint manipulation typically resides in an isolated portion of the code, which makes it hard to develop an application in which constraint solving is fully integrated.

A somewhat particular position is occupied by CHR [1] (Constraint Handling Rules) language. While CHR can be used as a standalone, general-purpose, high-level declarative language, it is most often used to extend a host language with constraints. While not being a library in the strict sense, using CHR from within a host language often makes a similar syntactic handling necessary.

3 Research proposal

Often, constraint systems are defined over a particular kind (or type) of values, with typical examples being the set of integers, floating point numbers, or, more generally, a finite domain of (symbolic) values. Moreover, solving constraints over these different types most often requires different solvers or, at least, different solving techniques. This is for example clear in constraint logic programming (CLP) where the different types of constraints give rise to what are formally considered to be different languages, examples being CLP(I), CLP(R) and CLP(FD).

Fully integrating constraints into an object-oriented language demands for dealing with the complex type structure of these languages. Integrating constraints in object-oriented languages has received a substantial amount of interest. (* Check literature on what has been done *)

In this work, we take a different approach. Given a class definition and a set of (boolean) methods describing the desired characteristics of the objects (i.e. the constraints), let the system find an instantiation of the class that satisfies these characteristics. In other words, we aim to develop a system in which the mere creation of an object (being an instantiation of a constrained class) implies that the created object satisfies the constraints imposed on its class.

The main strength of the approach is the fact that the constraints, being simply boolean methods, can use the host language's constructs in order to express – at least in principle – any characteristic of any structure. Consequently, constraints can be expressed on data structures such as lists, trees, maps and the like, without the need to introduce particular constructs in the underlying constraint language. From a programmer's perspective, the approach allows one to use the object-oriented host language to encode what is essentially a declarative approach to a constraint problem. Indeed, rather than encoding the algorithmic machinery needed to construct the desired (constrained) objects, it suffices to write a set of methods that verifies whether a given object is properly instantiated.

In order to compute an instantiation that satisfies the constraints, one needs to create an object with values for its data fields that are such that an invocation of the boolean methods representing the constraints all return true. Deriving data values that make a method execute along a desired path can be done by using well-known techniques from test-case generation. Although different approaches towards test input generation exist, so-called constraint-based test-case generation techniques translate a desired execution path into a set of basic constraints that allow in turn to compute values that make the method execute along the specified path.

The main research questions that will be addressed in this project are the following:

Firstly, to what extent can techniques from test-case generation be extended and adapted in order to compute class instantiations that verify the set of given constraints. Although seemingly rather straightforward in theory, it remains to be seen whether limits need to be imposed on the language constructs that can be used in the constraints in order to make value generation feasible in practice. Moreover, while test-case generation is typically concerned with a single method at a time, for object instantiation multiple boolean methods need to return true at the same time. It is also to be expected that the techniques need to deal with argument values provided from the object's constructor.

Secondly, to what extent can the above mentioned base technique profit from defining propagation rules [] that are logically redundant but that allow to replace complex constraints by simpler ones as such possibly aiding the resolution process. Moreover, constrained classes possibly being placed in an inheritance hierarchy, it needs to be investigated how constraints and possibly propagators (or even complete solvers) can exploit this hierarchy.

Thirdly, in a typical setting multiple (different) instantiations need to be computed for a given constrained class and thus a search strategy needs to be incorporated. It is an open question how this search strategy (i.e. basically the solver) can be represented in such a way that it can easily be adapted and manipulated for a particular application.

Fourthly, being part of a running system, the objects generated from a constrained class could be altered during their lifetime. Strategies need to devised that allow to deal with these changes. In particular, it will be interesting to

investigate whether it is feasible to propagate these changes through the class' constraints in order to further alter the object in such a way that the constraints are again satisfied.

4 Plan

References

1. Thom Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends, LNCS 910*, pages 90–107. Springer-Verlag, 1995.
2. Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, pages 583–628, 1994.
3. Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language, 1994.
4. Gert Smolka. The oz programming model. In *COMPUTER SCIENCE TODAY, LECTURE NOTES IN COMPUTER SCIENCE*, pages 324–343. Springer-Verlag, 1995.