Solving CSP with CSP

Peter Barth, Michael Duchmann, Agnetha-Kristin Kraus, and Arne Leonhardt

Hochschule RheinMain, Wiesbaden, Germany
bitbucket.org/gofd/gofd

September 5, 2014

Abstract. We propose to solve Constraint Satisfaction Problems (CSP) with a concurrent finite domain constraint solver (gofd) based on the Communicating Sequential Processes (CSP) paradigm. A central constraint store manages a collection of independently running propagators that work on copies of the available finite domains and post domain reductions back to the store. The store itself accounts for all domains and communicates domain reductions back to potentially affected propagators. The constraint solver is implemented in the Go programming language. The architecture provides for scaling in multiprocessor environments on the propagator level.

Keywords: constraint satisfaction problems, communicating sequential processes, finite domain constraints, constraint solving, propagators, parallelism

1 Introduction

Solving widely applicable "Constraint Satisfaction Problems" [11] is notoriously hard. Fortunately, solvers can benefit from multicore architectures. However, most approaches focus only on distributing labelling to independent agents [13], which happens after consistency is achieved. In contrast, we propose to deploy propagators as independent agents that reduce finite domains to achieve consistency. Thus, parallelisation is also exercised on the propagator level. The independent agents work on copies of the finite domains and communicate domain reductions following the "Communicating Sequential Processes" paradigm [6] as provided in the Go programming language [9]. We use a selection of "Constraint Satisfaction Problems" to analyse the performance of the gofd system on single and multicore machines.

First, we introduce constraint satisfaction problems and their modelling with finite domain constraints. Then we give a broad overview of current solver technologies focusing on achieving local consistency. We follow the "Communicating Sequential Processes" paradigm to introduce parallelism to constraint propagation. Then we detail propagation algorithms that are triggered by and provoke domain reductions following an example. Afterwards, the architecture of the developed concurrent finite domain constraint solver is described. Then, we discuss why sequences of intervals are the preferred choice of the domain representation and why some form of arc consistency is the preferred level of consistency of the implemented propagators. Finally, some empirical results are given followed by the conclusion.

2 Existing Work

First, we present constraint satisfaction problems, which are typically modelled and solved as finite domain problems [11, 12] where sets of possible values of integer variables are reduced and then a solution is searched by means of enumeration. Then, we introduce a concurrent programming paradigm based on coroutines [6] that we employ for the domain reductions.

Solving Constraint Satisfaction Problems Constraint satisfaction problems are used to model and solve many NP-complete practical problems. They consist of three components [11]: variables, domains, and constraints. Variables have finite domains consisting of a finite set of integers such as X and Y that may have the domain $\{1,2,3,4,5,6\}$, representing the allowed assignments. Constraints express the relation between their variables, for example $X+Y=9$. A solution to a constraint satisfaction problem is an assignment of domain values to their variables satisfying all constraints. However, not all values in a finite domain may occur in a satisfying assignment. The ones that do not, can be safely removed without changing the solution space. A set of constraints is consistent, if only values occurring in at least one solution are present and no domain is empty. In our example the values 1 and 2 can be removed in both X and Y. If more than one constraint is given, local consistency can be achieved by checking each constraint separately. Note, that this does not exclude all impossible values. For example, the constraints $X \neq Y, X \neq Z, Y \neq Z$ with $X,Y,Z \in \{0,1\}$ are locally consistent, as individually each constraint has valid assignments. However, the constraints imply that X,Y, and Z must all have different values, which is obviously unsatisfiable as all three domains of the variables have the same two values.

This failure can be detected by trying to achieve global consistency via enumeration or labelling. With labelling, each possible combination is tried until a solution is found or the search space is exhausted resulting in a failure. The search space is implicitly enumerated by backtracking with the aim to detect failure as early as possible. The number of search nodes and therefore the running time is often heavily influenced by the selection of the variable and the selection of the value for that variable suggesting the use of heuristics. In addition, at each search node local consistency is ensured, which again can be strengthened by more sophisticated algorithms or more sophisticated global constraints, which typically reduce more values in one go. Despite their name, they are not achieving global consistency, but they are still running in (low) polynomial time.

Achieving different forms of local consistency is called propagation and is done with propagators that check in their simplest implementation if every possible value in the variable's domain suits the constraint and remove the ones that do not. Note, that values are only removed from, but never added to domains by propagators. Continually removing values might cause one domain to become empty, which causes the entire store be inconsistent and represents failure. In case there remains only one value in the domain the variable is called fixed or ground. Fixing a variable typically provokes domain reductions in other variables. In the above example fixing X to 0 removes 0 from Y and from Z via propagation. If only 1 remains as value for Y then Z becomes empty or vice versa. Thus, propagation results in a failure. Via backtracking, X is then

fixed to 1, which forces Y to 0 and Z to have the empty domain. The search space is exhausted and global inconsistency is shown.

In order to propagate, propagators differentiate between input and output variables. During propagation the domains of the input variables do not change. In contrast, output variables are affected by domain changes which are normally processed centrally. Therefore, it is possible to divide the propagator further into one independent agent per output variable, so that this unit only examines the domain of the output variable and the resulting changes.

Communicating Sequential Processes As we want to parallelise the propagation process, we introduce a concurrent programming paradigm that is well suited for massively concurrent processing based on coroutines, messaging, and local data [6]. With the principles of communicating sequential processes we can get rid of locking at the program level altogether. To this end, we need to pay the price to copy data and let coroutines safely work on those local copies. The independent agents running as coroutines communicate through a safe message passing primitive. This paradigm together with the sacrifice of preemptive scheduling provides an environment for massively concurrent programs [5, 10].

Google developed its programming language Go for solving big problems in large software landscapes [9]. It is designed for solving those problems in a distributed and parallelised way with its concept based on the communicating sequential processes paradigm [4]. Coroutines are called goroutines and are native to the language. By nature, they are more lightweight than threads [5], which can communicate and synchronise their states over a built-in concept named channels. Instead of paying attention on subtleties required to implement correct access to shared variables, shared values should be passed around on channels and may never be concurrently shared by different threads in Go [4]. The principle is placard with Google's slogan [4]: "Do not communicate by sharing memory; instead, share memory by communicating.".

3 Propagation and Messaging

In order to initiate propagation, every propagator has to be executed at least once upfront. Then, the propagators will be rerun several times which is induced by domain reduction messages. For example, there could be two different propagators [12], one for the constraint $X + Y = 9$ (propagator 1) and one for $2 \cdot X + 4 \cdot Y = 24$ (propagator 2). Both propagators are interested in the same two variables X and Y. Therefore, if one propagator changes the variables' domains, the other one has to be notified via a message and propagate again. Every propagator has its own local copy of the domains.

The propagation process is depicted in figure 1: At first, propagator 1 is executed which results in no changes, since every value of X can be paired with a value of Y so that the sum is nine and the corresponding constraint is satisfied. In step 2, the second propagator is started and values unsuitable for the constraint are removed. These changes are collected and wrapped up into a message containing the values which have to be deleted from the variables' domains. For the propagation step 2 a message could look like this: $[X \neq 1,3,5,7,9, Y \neq 0,1,7,8,9]$. Consequently, propagator 1 receives the

message and updates its copy of the locally held domain. Propagation is started again and the values $2, 8$ are removed from the domain of X and the values $2, 4, 6$ are removed from the domain of Y. Accordingly, a message with the changes is sent to the interested propagator 2. Propagator 2 updates its local domain copy, removes values, sends messages to propagator 1 and vice versa until no more changes are made to the domains and propagation is finished. In our example there exists only one solution in the end, being $X = 6$ and $Y = 3$.

Step	Domains		Propagators	
1	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	[x + y = 9]	2x + 4y = 24
2	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	x + y = 9	[2x + 4y = 24]
3	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	[x + y = 9]	2x + 4y = 24
4	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	x + y = 9	[2x + 4y = 24]
5	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	[x + y = 9]	2x + 4y = 24
6	X = {0,1,2,3,4,5,6,7,8,9}	Y = {0,1,2,3,4,5,6,7,8,9}	x + y = 9	[2x + 4y = 24]

Fig. 1. Simple propagation example

4 Architecture

The gofd constraint solver consists of two major parts as depicted in figure 2: the store acting as the central control unit of the solver and the propagators as independent agents computing domain reductions concurrently. The store manages the master copies of the finite domain variables containing a unique identifier and a domain representing a set of possible values. To ensure safe concurrent access, there is a variety of events and corresponding channels to communicate domain reductions and exchange other information between store, propagators, and calling application.

The propagators represent the constraints that are imposed on the store and contain the variable identifiers of the employed finite domain variables and local copies of their domains as well as propagator specific code. This code produces domain reduction messages by computing local consistency according to the constraint the propagator is describing. Each propagator holds exactly one output- and one input-channel to send and receive these domain reduction messages to and from the store. These channels as well as the local domain copies are retrieved from the store during the propagator's registration process. This triggers an initial consistency check executing the propagator code. Future domain reductions on input variables potentially cause further propagation. Each propagation signals completion by producing a single event message, which may contain multiple domain reductions or may be empty, that is sent to the store. There, the central domains are updated and changes are again distributed to dependent propagators. Only these change events cause updates of the propagators local copy of the affected domains and subsequently provoke a new consistency check. Note, that even the source of the domain reduction may not eliminate the domain values from its local

copy in advance. An equilibrium is reached if no further domain reductions are pending, meaning that no messages on the input channel nor output channel of any propagator remain.

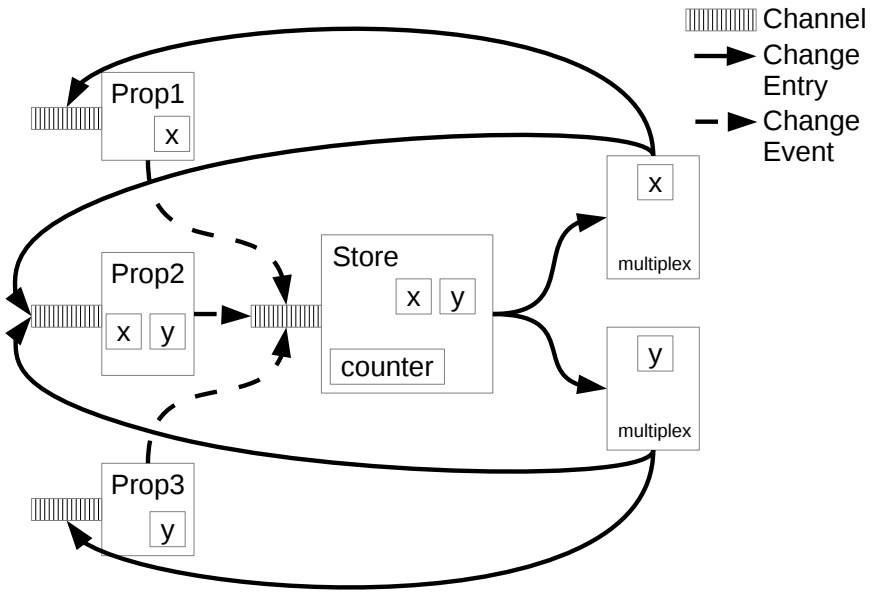

Fig. 2. Propagators and Messaging

The store manages all propagators and the global copies of the variables including their domains. It also is an independent agent with a single coroutine receiving domain reduction and control events and processes them sequentially. The store listens to a control channel for control events like registering a new variable or propagator and requesting internal state and data. Since all those events are processed one at a time in a central loop no further synchronisation is needed when accessing the stores private data. Control events implement a callback function, which is called during event processing inside the stores central coroutine. All actions triggered by control events shall not do expensive computations as propagation may be hindered otherwise.

The store as the central control unit of propagation receives change events containing multiple entries of values that have to be deleted from a variable's domain as seen in figure 2. These entries are retrieved from the event, the change is made to the stores central copy of the affected domains. In case the domain is actually reduced, the change is forwarded to all dependent propagators that are managed by a multiplexer mapping the variable to propagator input channels. The propagators then update their local copy of the domain and re-initiate propagation. If an event results in an inconsistent domain, an equilibrium is reached and all propagators terminate (their input channel closes).

Reaching an equilibrium is a distinct event that must be made available to other components such as labelling. It signals failure by returning false and a consistent idle state by returning true. The store lapses into idle state as soon as there are no more domain reduction messages on the input and output channels. For every outgoing change entry the store expects exactly one incoming change event. To keep track of pending change events the store maintains an internal event counter that reaches zero if propagation has ended. Only external stimuli, such as adding further propagators or domain reductions induced by labelling, may cause further propagation.

For labelling or in order to give the user feedback that propagation has ended, we have to provide a notification that an equilibrium is reached. We do that by providing a distinct event on request. Note, that providing this event is by the nature of concurrent propagation asynchronous. Consequently, after such a notification is requested, several other messages may be processed before its answer is sent. However, from a consumer perspective all these control events are blocking. This is done by providing a result channel for each control event.

5 Domain Representation

A domain such as $\{1,2,3,4,8,9,10\}$ can be represented as a set of values using hashing, which we support. However, such an explicit representation uses a lot of memory. As we have to copy domains more often than in classical solvers, a more compact representation is useful. We favour the representation of domains as increasing sequence of consecutive intervals [3, 12]. For example, the domain $\{1,2,3,4,8,9,10\}$ is represented as the interval domain $[(1,4),(8,10)]$. Any set of integers can be represented as a unique interval domain. Most operations on domains involve set union, difference, or intersection operations, which can be effectively implemented using interval domains. Even if there are large number of intervals, triggered by "holes" in the range of values, in an interval domain, most operations perform faster than with explicit representation and hashing [3]. Only the containment test of a single value may be faster using the explicit representation. Therefore, and because explicit representation is also easier to understand, we provide both domain representations in gofd with a clear preference for the interval representation. Propagator implementations may use the specific advantages of the different domain representations. However, for all communication with the store we use an interface for the domain representation. It is the responsibility of the propagator to check the type of domain representation and convert to a more suitable representation if necessary or refuse to work with a unsuitable representation. It is recommended to stick with interval domains and if not, at least model the entire problem using one other alternative domain representation. Conversion between domain representations is supported, but due to copying during backtracking not recommended.

6 Propagators

The goal of a propagator is to eliminate values that, given the current domain of the involved variables, may not lead to a solution of the constraint the propagator represents. To this end, the propagator reacts on changes of the domains and computes possible

changes (removals of values) of the domains of the depending variables. Most commonly this local consistency is achieved by providing arc consistency. A value may remain in the domain of a variable only if it is part of a solution assignment. Thus, for each value it is tested, whether an assignment of all other variables of that constraint can be found that constitutes a solution. Otherwise, if a value cannot possibly constitute a valid assignment, the value is eliminated from the domain. For most constraints, we provide an implementation providing arc consistency.

However, for some constraint satisfaction problems arc consistency is too expensive. For example, if each domain of a constraint with three variables contains thousands of values, then billions of combinations may need to be checked. In these cases it may be more effective to use a weaker form of consistency, bounds consistency [12]. There, we only check the bounds of the domains and not every value in between. Consequently, we may eliminate a large chunk of consecutive numbers quickly. Obviously, bounds consistency makes most sense when used together with the interval representation of the domains. Some problems such as the crypto-arithmetic puzzles as described in the next section are only computationally solvable with bounds consistency. Thus, we also support some propagators for constraints that employ bounds consistency only.

In contrast, other problems greatly benefit from a more compact representation and providing a stronger propagation than the local consistency of fine grained constraints. Therefore, we also support global constraints [1]. Global constraints allow to express in a single constraint dependencies that would normally require many classical constraints. The most prominent example is the AllDifferent constraint that expresses that all variables of the constraint must contain different values. Thus, the example in section 2 which states that $X \neq Y$, $X \neq Z$, and $Y \neq Z$ can be more compactly represented as AllDifferent(X,Y,Z). This representation is not only more compact, it may also provide for a stronger propagation as it is possible with a set of classical constraints. A propagator may detect inconsistency with the domain constraints $X,Y,Z \in \{0,1\}$ without resorting to enumeration. We expect global constraints to be ideally suited for our concurrent approach, as global constraints combine a compact representation with a more expensive computation agent. Thus, eliminating the need to copy domains as well as avoiding communication. To this end, we have implemented one global constraint, the Among constraint [7, 2]. The Among constraint can be used as a basis for many other global constraints and supports among others all kinds of scheduling problems, where typically disjunctive instead of conjunctive collections of constraints are needed.

Finally, there are problems, that can be naturally modelled only if one can reason over the truth value of the constraint itself. Thus, for each constraint C_i, we introduce a Boolean variable B_i with the domain $\{0,1\}$ for which $C_i \Leftrightarrow B_i$ holds. This principle is called reification [12]. By using the introduced variables B_i in other constraints one can easily express for example disjunction with the constraint $\sum B_i \geq 1$. In order to support reification we need to support negation and entailment, which is complicated to do in general. But it works much better for some classical constraints, if their implementation is based on a standard library of set operations called indexicals [12]. Based on these primitives and reification also many global constraints can be more easily implemented. gofd provides for indexicals as well as reification [3].

7 Evaluation

In order to avoid style and naming discussions, we follow the style and naming convention of another constraint library that is embedded in a non logic programming language, JaCoP [8]. The gofd system is available in source code under

```
https://bitbucket.org/gofd/gofd
```

and can be directly installed using standard Go conventions. A simple, complete Go

```
1  package main
2  import (
3        "bitbucket.org/gofd/gofd/core"
4        "bitbucket.org/gofd/gofd/propagator"
5        "fmt"
6  )
7  func main() {
8        store := core.CreateStore()
9        X := core.CreateIntVarFromTo("X", store, 0, 9)
10       Y := core.CreateIntVarFromTo("Y", store, 0, 9)
11       equation1 := propagator.CreateC1XplusC2YeqC3(1, X, 1, Y, 9)
12       equation2 := propagator.CreateC1XplusC2YeqC3(2, X, 4, Y, 24)
13       store.AddPropagator(equation1)
14       store.AddPropagator(equation2)
15       fmt.Printf("consistent: %v\n", store.IsConsistent())
16       fmt.Printf("X: %s, Y: %s\n",
17             store.GetDomain(X), store.GetDomain(Y))
18 }
```

Listing 1: Example gofd program: two propagators

program using gofd then looks like listing 1 implementing the example in section 3. We need to instantiate a store, then create finite domain variables and assign an initial domain at the same time. Propagators for the respective constraints, in this case providing local consistency, need to be created and then added to the store. With `IsConsistent` we force one consistency round to finish. Afterwards, we print the results of the propagation.

```
consistent: true
X: [6], Y: [3]
```

In this example, we have not used labelling yet. But we have already seen one problem instance in section 2, where we wanted to solve the problem that the variables X, Y, and Z each having the domain $\{0,1\}$ must be mutually different. We can solve this problem with the gofd program in listing 2. Up until line 20, this is similar to listing 1. We create three variables X, Y, and Z with the domain consisting of the values 0 and 1. Then

```
package main
import (
	"bitbucket.org/gofd/gofd/core"
	"bitbucket.org/gofd/gofd/labeling"
	"bitbucket.org/gofd/gofd/propagator"
	"fmt"
)
func main() {
	store := core.CreateStore()
	boolVals := []int{0, 1}
	var X, Y, Z core.VarId
	core.CreateIntVarsIvValues([]*core.VarId{&X, &Y, &Z},
		[]string{"X", "Y", "Z"}, store, boolVals)
	equation1 := propagator.CreateXneqY(X, Y)
	equation2 := propagator.CreateXneqY(X, Z)
	equation3 := propagator.CreateXneqY(Y, Z)
	store.AddPropagators(equation1, equation2, equation3)
	fmt.Printf("consistent: %v\n", store.IsConsistent())
	fmt.Printf("X: %s, Y: %s, Z: %s\n",
		store.GetDomain(X), store.GetDomain(Y), store.GetDomain(Y))
	query := labeling.CreateSearchOneQuery()
	result := labeling.Labeling(store, query)
	fmt.Printf("solution found: %v\n", result)
}
```

Listing 2: Example gofd program: mutually different values

we add the three propagators enforcing $X \neq Y$, $X \neq Z$, and $Y \neq Z$, which enforce, that all three variables have different values. Note, that local consistency does not find out, that there is no solution. Furthermore, propagation has not reduced any domain. This is shown by the output of the program.

```
consistent: true
X: [0..1], Y: [0..1], Z: [0..1]
solution found: false
```

To solve the problem, we need to enumerate using backtracking, which is performed with the `labeling` package. As we just want to check satisfiability – whether at least one solution exists – we create a query looking for one solution. Its return value tells us, whether a solution was found, which in this example is as expected not the case.

In order to evaluate the expressiveness as well as the performance of the gofd system, we have implemented classical examples such as the N-Queens problem, the crypto-arithmetic puzzle SEND+MORE=MONEY, and the magic series problem among others in both gofd as well as JaCoP. We have run all examples on a machine with an i7-2600 @ 3.4 GHz Intel processor with 8 Gigabyte main memory on a 64 Bit Linux distribution with kernel 3.13. We use Go version go1.2.1 linux/amd64 for gofd. We use

Java version 1.7.0_65 OpenJDK and JaCoP-3.2 for JaCoP. The results are provided in table 1.

				gofd			JaCoP	
Problem	#C	#V	#S	#P	#N	T	#N	T
Queens-7	3	7	40	1	182	39.56	179	2.63
Queens-8	3	8	92	1	673	63.76	709	12.44
Queens-9	3	9	352	1	2658	370.81	2783	22.69
Queens-10	3	10	724	1	10675	1640.52	11693	76.38
Queens-11	3	11	2680	1	47888	7870.19	52935	375.21
Queens-10	3	10	724	2	10675	1834.99	–	
Queens-11	3	11	2680	2	47888	7304.37	–	
SEND+MORE=MONEY	3	31	1	1	8	5.28	10	1.11
MagicSeries-9	10	10	1	1	309	145.86	881	6.91
MagicSeries-10	11	11	1	1	434	250.82	1265	8.63
MagicSeries-12	13	13	1	1	784	637.63	2343	15.87
MagicSeries-14	15	15	1	1	1294	1476.24	3899	30.03
MagicSeries-12	13	13	1	2	784	469.36	–	
MagicSeries-14	15	15	1	2	1294	1079.50	–	
MagicSeries-12	13	13	1	4	784	507.01	–	
MagicSeries-14	15	15	1	4	1294	1064.66	–	

Table 1. Performance evaluation, #C = number of constraints, #V = number of variables, #S = number of solutions, #P = number of processors/cores (always 1 for JaCoP), #N = number of explored nodes, T = run time in milliseconds,

Typically, the run time of gofd varies per run, due to the non-deterministic execution of all involved agents, which also results in a different number of explored nodes. To avoid that during benchmarking, we use a fixed variable selection strategy which is the same for gofd and JaCoP and corresponds to the natural order in which the variables are created during problem construction. We see that JaCoP is around an order of magnitude faster than gofd. For the implementation of the queens problem we use a dedicated constraint expressing that a set of variables to which an individual offset is added shall all be mutually different. Such a constraint is not directly available in JaCoP, but has been implemented manually with by introducing one new variable and constraint per offset. As JaCoP is for this problem still an order of magnitude faster, this does not contradict the overall picture. For MagicSeries we use the Among Constraint which still relies on the much slower domain representation using an enumeration of all values and storing them using built-in hashing, which may partially explain the increased performance difference. For all other examples we use an interval representation of the domains.

We see, that parallelisation does not yet always help and may even hinder performance as shown in the queens problems. However, as soon as the propagators decrease in number and increase in power, parallelisation may make constraint solving faster. In the magic series examples using the global constraint Among we experience an per-

formance improvement using multiple cores. Note, that when using multiple cores the variance in run time increases significantly. The results vary regularly by a factor of about three from run to run. For example, magic series with size 14 may as well run faster using two processor cores and slower using four processor cores compared to the run time on a single core. Leveraging multiple cores for propagation is at this stage erratic at best regarding performance evaluation. Still, the numbers demonstrate, that gofd is a reasonable implementation of a finite domain solver. It is about an order of magnitude slower than mature implementations, but may serve as test bed for experimenting with concurrency in finite domain solvers.

8 Conclusion

gofd is a reasonable complete and powerful finite domain constraint solver implementation, with which it is possible to compactly express a wide range of constraint satisfaction problems. Although a large number of typical problem instances can be solved, the performance is not yet on a par compared to more mature implementations. However, gofd provides an innovative architecture for handling constraint satisfaction problems and a test bed to evaluate concurrent propagation techniques. Fortunately, we see on some problem instances already the benefits of parallelising work on multicore machines. And for some problem types and instances, it may already be fast enough.

Besides more efficient implementations of the propagators, there are a number of ways to improve the current state, of which most focus on reducing copying and reducing communication. Most promising might be to group several propagators into one agent. All propagators in such a group would work sequentially and thus on the same instance of the variable domain, which reduces copying and communication. This flexibility may help to balance distribution of work and limit its overhead.

Following the trend to multicore architectures, there is no way to avoid constraint solvers that exploit that power. Programming environments based on communicating sequential processes seem to be well suited for constraint satisfaction problems. gofd not only provides an finite domain constraint solver for the Go programming language but also an evaluation environment for exploiting parallelism at the propagator level.

References

1. Nicolas Beldiceanu and et al. Global constraint catalogue: Past, present and future, 2006.
2. Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Among, common and disjoint constraints. In Brahim Hnich, Mats Carlsson, Franois Fages, and Francesca Rossi, editors, *Recent Advances in Constraints*, volume 3978 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2006.
3. Michael Duchmann. Konsistenztechniken für ganzzahlige Probleme in einem nebenläufigen Constraint Solver. Master's thesis, Hochschule RheinMain, Wiesbaden, Germany, 2014.
4. Google. Effective Go – the Go programming language. http://golang.org/doc/effective_go.html.
5. Google. Faq – the Go programming language. http://golang.org/doc/faq.
6. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, aug 1978.

7. Agnetha-Kristin Kraus. Das Among-Constraint für einen nebenläufigen Constraint Solver. Master's thesis, Hochschule RheinMain, Wiesbaden, Germany, 2014.
8. Krzysztof Kuchcinski and Radoslaw Szymanek. JaCoP - Java Constraint Programming solver. http://jacop.osolpro.com/, 2001. [letzter Zugriff: 25. August 2014].
9. Rob Pike. Go at Google: Language design in the service of software engineering. http://talks.golang.org/2012/splash.article.
10. A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
11. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
12. Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 14, pages 493–524. Elsevier, 2006.
13. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Knowl. and Data Eng.*, 10(5):673–685, September 1998.