Dynamic Distributed Constraint Satisfaction with Asynchronous Solvers

- work-in-progress paper -

Georg Ringwelski and Hans Schlenker

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
{georg.ringwelski,hans.schlenker}@first.fraunhofer.de

Abstract. In this paper we show that the dynamic execution model of Asynchronous Constraint Solving can be applied to solve distributed constraint satisfaction problems. Asynchronous Constraint Solvers are general purpose solvers that are designed as black-box constraint reasoning components for complex object-oriented software systems. Multiple solvers coordinate the execution and re-execution of their associated constraints. We give an introduction to ACS and describe its applicability for distributed problems such as automated meeting scheduling.

1 Introduction

Asynchronous Constraint Solving (ACS, [13]) is a dynamic execution model to solve Constraint Satisfaction Problems (CSP). A CSP formally consists of a set of variables with an associated domain and a set of constraints. Each variable v may take a value out of its domain D_v. Each constraint defines allowed instantiations for a subset of the variables. A solution to such a CSP is a mapping of all variables v into D_v, such that all constraints are satisfied. In a Distributed CSP (DCSP [17]) the mentioned components are distributed among several physically separated entities. In some applications, e.g. the one we will describe in section 5, CSPs or DCSPs are defined in an infinite process, where new knowledge occurs continuously. This new knowledge can include new variables and constraints to be satisfied as well as the information, that a formerly posted constraint is deleted and does not have to be considered anymore. Such continuous CSPs that allow the deletion of constraints are called Dynamic Constraint Satisfaction Problems or in the distributed case Dynamic DCSP.

In constraint programming, a real world problem is transformed into a CSP such that a solution to the CSP can be mapped into a solution to the original problem. The CSP is then solved using two techniques: *propagation* within a constraint-solver and *search* within the actual application. Propagation extracts information from the given CSP and applies specialized *look-ahead* [10] algorithms to simplify the problem. This incremental solving usually leads to reductions of variables' domains or early detections of inconsistencies in unsatisfiable CSPs. By invoking search in a so-called *labeling*-constraint, the constraint-based application tries to find a feasible variable assignment. In general, CSPs are NP-complete, such that search algorithms have to explore the search-space non-deterministically. If an erroneous instantiation is detected during *forward-checking* [10], i.e. propagation, the system must create a consistent state and try another instantiation. In most constraint solvers this is done through backtracking. A former state is restored from a global history (e.g. in CLP [10]) or recomputed in an encapsulated search-space (e.g. in Mozart [6]) that is taken as a basis for new assignments. In non-incremental solvers that do not support propagation, reversing erroneous instantiations can be performed locally by restoring an initial domain and trying another instantiation. This propagation-free algorithm (*generate-and-test*) is very inefficient with conventional search algorithms, but it can be implemented quite efficiently in *local-search*-approaches [5]. Local search is, however, not complete and can thus not be used to prove the non-existence of solutions or to find all solutions to a CSP.

Dynamic constraint solvers like our ACS-implementation J.CP [14] or DJCHR [15] do not require state restoration techniques and are still applicable for dynamic problems. These solvers provide a reverse-operation to constraint posting, called constraint retraction [16, 8, 7]. Thus arbitrary non-deterministic search algorithms can be implemented that post instantiation constraints and reverse this operation, if an inconsistency has been detected [15]. During search, dynamic solvers compute states, that *would* be the current state, if something *would not* have been done in the past.

There are some recent approaches to solve DCSPs with incremental solvers. These approaches use one standard incremental solver component in every agent that communicates with other agents. Hannebauer [9] for example proposes the use of such local solvers that are connected through some meta-algorithm to cooperatively solve a distributed problem. However, completeness of propagation and search seem to be very complicated tasks in such a system, because much less communication between variables and constraints can be allowed.

For non-dynamic DCSPs, where no incremental constraint propagation is required, Yokoo et al propose some distributed search algorithms [17], without constraint propagation. The only existing general purpose constraint solvers (we are aware of), that directly supports incremental solving of DCSPs is Mozart [6]. Propagation and search can be executed with cooperating physically distributed solvers, i.e. domain changes of a variable within one agent is communicated to all other agents that this variable belongs to. In Mozart, however, propagation between distributed solvers is only done for variable assignments (only *forward-checking* and no *look-ahead*) for efficiency reasons. It is not practicable to propagate domain restrictions in the synchronous execution model of Mozart, that uses chronological ordering in search spaces, because a program cannot continue its execution, until the propagation of a constraint is completed. To provide *look-ahead* propagation, a program would have to await the slow inter-process communication for every domain reduction. In ACS inter-process propagation can be done without significant performance drawbacks, because due to its asynchronous execution model it can be implemented with non-blocking message passing.

2 Constraint Satisfaction with ACS

Given implementations of the used constraints, any CSP can be formulated in an imperative program by defining variables and posting the respective constraints over them in declared solvers.

Definition 1 (Constraint Program). *A constraint program is an imperative program using the following primitives.*

1. *solver definitions* `s = new Solver()`
2. *variable definitions* `v = new Var(p)`*, where* `p` *is a feature vector describing the initial domain of* `v`
3. *domain queries* `p = v.getDomain()`
4. *constraint definitions* `c = new Constr(v,T)`*, where* `T` *implement* `c`*'s propagation methods and* `v` *is a vector of variables*
5. *constraint postings* `c.post(s)`*, where* `s` *is a defined solver*
6. *constraint retractions* `c.retract()`
7. *wait statements* `s.awaitReliability(val)` *that suspend the application-thread until* `s` *has reached the reliability-value* `val`*.*

2.1 Constraint Propagation

The operational semantics of the primitives given in Definition 1 is described in detail in [13]. Here we give a short outline of the execution model defined with ACS and will point out the relevant properties for the use in distributed systems in Section 3.

With the creation of a constraint solver object a concurrently running thread is started that waits for solver-tasks to be processed. Such tasks wrap method-calls that perform constraint propagation, consistency tests or retract previously posted constraints. If a constraint is posted or retracted in the constraint program a corresponding task is created and buffered by the solver. A constraint may define several propagation tasks that are associated to priority-values. These serve the solver-thread to schedule the tasks from the buffer for execution. The progress of constraint propagation, i.e. the consistency that is currently ensured is represented by the reliability value of the solver. It increases, if propagation tasks have been executed and decreases if ne tasks are posted by the application or by the variables during propagation. The reliability of the solver is determined by the amount of the un-exectued tasks and their priority-values. The application, however, can continue before top-reliability is reached.

Constraints are implemented as objects that hold all necessary information for constraint retraction and provide one or more propagation tasks. These tasks may post or retract other constraints, restrict the domains of the constraints' variables and perform consistency tests. If a variable domain is restricted, other constraints, that depend on it are re-posted by the variable in order to reach the required consistency. In difference to solvers without task-priorities we can also use inefficient or ineffective algorithms here because they will only be executed if the solver has some spare time to do so. Such algorithms may for example implement domain restrictions in global constraints thar are rather unlikely to reduce the search space, but can do so in some cases. In [1] K. Apt showed, that such a propagation process (a so-called chaotic iteration) will result in a stable state where no further domain reductions can be deduced from the posted constraints. This state is uniquely determined such that the order of the constraint execution is irrelevant for the result and can be varied by a task scheduler like we use it in ACS. We showed in [13] that the execution model of ACS is sound and complete wrt. chaotic iteration. This means that the invocation of propagation tasks will lead to the correct result and that retraction does not interfere with this property.

Constraint retraction creates a system state, where all relevant changes that resulted from the posting of a constraint are taken back. Such changes are domain restrictions of the constraint's variables and all variables that are reached through propagation. Furthermore the thus reduced domains are used by constraints that were posted after the retracted constraint. The retraction is performed by rebuilding former variable domains from a history snapshot stored in the constraints that were affected and by reposting all these constraints. Our algorithm is an adaption of retraction in CLP(FD) [8] where we abandon the use of a centralized history management by storing consequences of constraints locally (similar to [16]).

2.2 Search in ACS

In our prototypical ACS implementation J.CP we implemented two generic *labeling*-constraints. The first one is Backtracking that allows complete search and is thus used, if all solutions of a CSP are to be found. The other algorithm is a very simple *local-search* that is faster than the first one, but incomplete. Both algorithms are implemented as global constraints that post other constraints (so-called agents) the are concerned with the instantiation of the variables to be labeled. The agents communicate in order to find valid assignments and ensure synchronization where necessary. In complete search algorithms the agents additionally hold information on used values to ensure the complete exploration of a search space. Just as any ACS constraint such labelings are applicable for physically distributed variables as far as the algorithms themselves do not require centralization. In table 1 we compared the performance of these algorithms to some standard solvers for constraint-based Java programs with the well known n-queens benchmark. The benchmark was implemented with $\neq$-constraints only and one solution was searched. We did not use a global *alldifferent* constraint in the benchmark in order to compare the solvers wrt. their execution model and search algorithms and not to compare the propagation algorithms of their constraints.

Chronological Backtracking Our current implementation of backtracking explores a previously defined search space by synchronization of the instantiations. It is not an asynchronous algorithm like [17]. The idea is that the variables that

Table 1. Comparison of J.CP using backtracking (with fixed variable ordering and random value selection) and simulated annealing to other Java constraint solvers. JCL 2.1 [11] is used with its "forward checking arc consistency" solver (which was the fastest in our tests), JaCK [12] uses standard labeling with backtracking and SICStus/jasper [4] v3.8.7 was tested with and without the "first fail" labeling strategy. All numbers are average values (in milliseconds) of 3-10 tests, ran on a dual Pentium III-500 Linux PC.

	J.CP BT	J.CP SA	JCL	JaCK	SICStus	SICStus ff
5 queens	308	307	49	1061	739	736
8 queens	323	312	23	7814	768	760
12 queens	913	556	29	30274	835	828
20 queens	3617	2015	18575	> 5 min	7962	1061
30 queens	10164	2893	> 5 min	> 5 min	> 5 min	1458

are to be labeled are ordered, such that the first variable is instantiated first, the next second and so on. For every variable an agent is started that communicates with the agent of its previous and of its subsequent variable. The communication is performed by regular or remote method invocation depending on the distribution of the variables among a physical network. The agents are integrated in the system as constraints that post and retract instantiations (i.e. other constraints) during search. If such an agent detects an inconsistency, it notifies its previous variable such that this will try another instantiation. If an agent could successfully instantiate its variable it notifies the agent of the subsequent variable to proceed with the labeling. If the agent of the last variable has successfully posted an instantiation, a solution is found. The values that have already been explored during the current search are stored locally in the agents to ensure the complete traversal of the search space.

Due to the asynchronous execution of constraint postings and their propagation we have to explicitely ensure that the consistency check of every instantiation is finished before the next is posted. This can be done in ACS with priority values of propagation tasks that influence the scheduling of the tasks in the buffer of the solvers (similar to ECL^{i}PS^e [2]). The agents that post instantiations are scheduled with very low priority, such that other tasks, like the propagation and thus the consistency check, of other constraints are executed first. In the distributed application this has to be synchronized between different solvers. An agent cannot be sure that his instantiation did not lead to an inconsistecy until all solvers have reached a certain reliability level. This requires a lot of extra communication which makes this labeling uneligible for distributed applications. However, this algorithm is just as any other constraint also applicable in distributed applications, because it does not require any global control or data like constraint stores or history trails.

3 Distributed Constraint Satisfaction

In DCSP variables and constraints may be distributed among different physical entities. For ACS this means that variable- and constraint-objects as well as the solvers may be located in different processes. With objects in other processes (so called remote-objects) we can either communicate directly, which may be expensive because of the remote-communication effort, or use a local copy of the object instead. The latter case does not allow 'call by reference' on remote objects and will not lead to any action taken by the remote object. Such indirect access, however, can provide information on the object that was available when the copy was created. Furthermore, information on how to communicate with remote objects can be provided. The communication can be implemented as non-blocking message channel such that asynchronous remote method calls do not necessarily require extra communication time.

In distributed ACS we assume that constraints are located in the same process as their corresponding solver. This is no restriction to generality because the constraints are associated to a unique solver, after they were posted, anyway. Thus the invocation of the propagation tasks by the solver does not require any network communication. The methods *post* and *retract*, that cause the re-insertion of a task in a solver, may also be called for existing constraints by remote objects. In Figure 1 we give an example for the architecture of a typical scenario of our distributed constraint-based application, which will be described in Section 5.

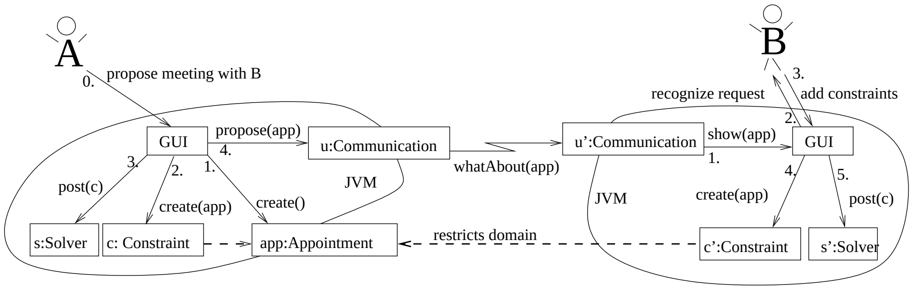

Fig. 1. Scenario in a distributed meeting scheduling system, where A proposes a meeting to B. A creates an Appointment-object app containing constraint variables and posts constraints over them in her local solver. The reference to these variables is sent to B's VM such that she can post additional constraints in her local solver.

Since the remote methods for constraint retraction and re-posting are executed asynchronously anyway they can be realized with non-blocking message-passing. This requires that the local references to remote constraints provide a communication channel to the constraints themselves. In ACS such local references occur in the stored consequences of the constraints (necessary for our retraction algorithm) and in the variables, that store constraints for re-posting during propagation. In both cases the only necessary communication to the remote objects is an asynchronous call to their *post*-methods. The history that is stored in the constraints for retraction does not have to be read from the remote object, because it is invariant and can thus be stored in a local copy together with the reference. In ACS constraints can be used as local copies of remote objects that provide the possibility to send a non-blocking message, that causes the remote constraint to re-insert its tasks in its local solver.

Variables may be located in any process (even if there is no solver running). The constraints over these variables use the methods *getDomain* and *setDomain* in their propagation tasks and retraction method. Since our propagation and retraction algorithms require current variable domains to operate correctly [13] we need direct access to the variables from the constraints. Using the remote objects themselves ensures the soundness of our execution model in distributed applications and requires synchronization of read/write access to the domains. If a constraint uses the domain of a remote variable for propagation, it has to communicate with this variable. This may either be for reading or for restricting the domain. The latter operation will then cause asynchronous propagation. I.e. for the restricting constraint, this operation is finished, before the propagation has been performed. In difference to distributed Mozart [6] we can thus perform *look-ahead* propagation across process-borders without getting significant performance problems. On the other hand, inconsistencies can, in our execution model, not be detected immediately, because propagation is not awaited. But due to the possibility of asynchronous constraint retraction this is not a big drawback. The algorithm will retract constraints that were posted long ago in similar time as recently posted constraints.

This kind of distributed execution of ACS is reliable as far as the communication with the variables works. Any change of a variable's domain will be known in any constraint that may propagate due to this change, because the variable itself is accessed. However, as we already described, we do not have to wait until the propagation is finished and can continue executing tasks in the solver before the restriction of the (remote-) variable's domain is completely propagated.

Provided the respective variable-objects are reachable via some physical network our execution model works in distributed as well as in monolithic environments. Remote variable objects must be reachable by reference, while constraints and solvers can also be handled as local copies that provide a non-blocking communication channel.

4 Implementation

A prototypical solver for finite domain constraints that uses the ACS execution model has been implemented. It is a regular Java package called J.CP that can be used by any Java application and runs on almost any hardware. Constraints are implemented as instances of an abstract prototype class, such that new constraints can be integrated into the system as implementations of this interface. The abstract constraint class provides all necessary methods for the execution in the ACS scheme. Thus only propagation algorithms (i.e. tasks) and their priority have to be defined with each new constraint that inherits all other properties from the prototype-class. The system is even extendable with other constraint systems by using the strict typing of Java. The combination of constraint systems [3] can be performed by defining constraints over variables of different systems. For example a less-or-equal constraint over a finite domain integer variable x and a real-number variable y can be implemented by: `x.setDomain(intMin(x), (int) floatMax(y)); y.setDomain((float)intMin(x),floatMax(y))`.

Solvers extend the class `java.lang.Thread` and run concurrently to the application. The handling of inconsistencies is implemented with Java's asynchronous event-model. A solver holds references of all objects that implement an event-listener as a dispatcher of inconsistency-events. These events implement the `java.util.EventObject` class and can thus be handled by the corresponding specialization of a `java.util.EventListener`. Furthermore in the event-objects all available information about the reasons for the failure are stored so that the listener can deduce the right action to remove the inconsistency from the information given with the event.

Distribution in J.CP is based on the `java.rmi` package which allows method invocation on any object that is exported from another Java virtual machine (JVM) reachable via some network. In our implementation we use the standard RMI communication protocols and thus get the Internet as platform for the implementation of constraint networks. The concept of RMI is that remote methods are not called in the remote objects, but in stub-classes, that are copied to the local JVM. The stub-classes refer to skeleton-classes on the server side that manage the communication. These classes can be generated automatically with the `rmic`-compiler. The J.CP-related classes themselves must be implemented as serializable objects that can be passed as value-parameters or as references to remote objects. Working on the objects themselves or passing them by reference can be performed with the use of wrappers (so-called *delegation patterns*) for the remote objects such that a stub of the wrapper that refers to the real object is used for method invocation. In our implementation we provide such wrappers for variable objects such that constraints can be posted in local solvers over any remote variables. The necessary classes, i.e. the variable itself, its remote wrapper and the necessary interfaces for the `rmi`-compiler and its resulting classes are shown in Figure 2.

5 Distributed Meeting Scheduling

With the described remote interface of J.CP we are about to implement a distributed meeting scheduling tool. The tool shall be used to solve appointment planning problems semi-automatically, only requesting proposals and commitments from the participants. If a user wants to make an appointment with one or more other users, her planner-agent creates constraint variables, that represent the appointment, and posts constraints over them to describe her demands. All planner-agents of persons involved in the appointment are notified and can set

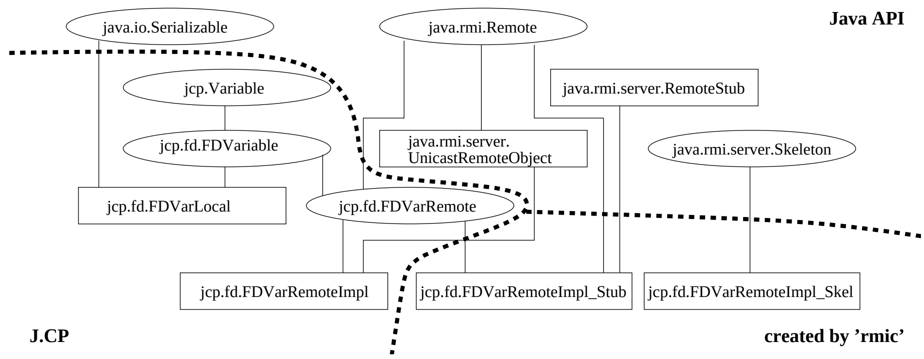

Fig. 2. Inheritance of classes and interfaces describing constraint variables for local and remote use. Classes are illustrated as boxes, interfaces as ellipses.

up constraints over the same variables themselves. When appointments are to be fixed, a *labeling* is executed, that uses dyanamic constraint processing to instantiate the respective variables. Thus a Dynamic DCSP is interactively defined in continuous distributed processes where constraints are asynchronously posted. *Labeling*-constraints can be posted by any agent over any appointment, which will instantiate the variables representing it. Thus the tool does not conventionally solve a CSP, but instantiates variables by request such that a changing set of distributed constraints is satisfied.

Using J.CP for distributed timetabling keeps full privacy for all involved agents. Only the variables and domains located at a user A that describe the appointment with a person B can be read by B. Due to the soundness of the distributed ACS scheme, which is based on the communication via common variables the system is reliable if a certain reliability-level has been reached in the solvers.

6 Conclusion and Outlook

The architecture of ACS is intended to allow the integration of constraint solvers into complex and possibly distributed and strongly interactive systems. Since all information for constraint propagation and retraction is stored locally in constraint-objects, ACS is well suited for the use in distributed systems. Neither the ACS architecture nor the execution model or the history management of the solvers have to be adapted to the distributed case. We showed that this approach is well suited for solving CSPs locally as well as for distributed applications.

Besides further improvements of the Distributed Meeting Scheduling application next steps will be the implementation of a distributed search algorithm like Asynchronous Backtracking [17]. This search algorithm will be integrated as a labeling-constraint in the system, that should be used for search in distributed applications. The agents that communicate "nogood" and "ok?" messages like it is described in [17] are implemented as constraints that are posted by the labeling in remote solvers. Every agent is concerned with the instantiation of all variables that are located in the same process. This architecture with several variables for every agent (which is an extension to regular ABT) was discussed in [17]. Inconsistencies that occurr during propagation are posted by the constraints in their local agent leading to a "nogood" that is sent to the respective other agents. For the implementation of this algorithm we will have to extend J.CP by the possibility to post constraints in remote solvers. This can be done by a method in the solvers' remote interface. We expect that no other changes will have to be made in the system.

References

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.

[2] IC-Parc at Imperial College London. ECL^{i}PS^e internet pages `http://www.icparc.ic.ac.uk/eclipse/`.

[3] Franz Baader and Klaus U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192:107–161, 1998.

[4] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP97*, volume 1292 of *LNCS*, 1997.

[5] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *Proc. SAGA 2001*, LNCS 2264, pages 73–89. Springer, 2001.

[6] The Mozart Consortium. `http://www.mozart-oz.org`.

[7] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37:1-3:185–212, Oct-Dec 1998.

[8] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in CLP(FD). *Constraints*, 4(1):5–42, February 1999. ISSN 1383-7133.

[9] Markus Hannebauer. *Autonomous Dynamic Reconfiguration in Collaborative Problem Solving*. PhD thesis, TU Berlin, 2001.

[10] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[11] EPFL Lausanne. Java constraint library. `http://liawww.epfl.ch/~akira/JCL/`.

[12] LMU Munich. Jack, `http://www.fast.de/~mandel/jack/jack.html`.

[13] Georg Ringwelski. A new execution model for constraint processing in object-oriented software. In *Proc. International Workshop on Functional and (Constraint) Logic Programming (WFLP01)*, 2001.

[14] Georg Ringwelski. Object-oriented constraint programming with j.cp. In *Advances in Artificial Intelligence, Proc. MICAI2002*, LNCS 2313, pages 194–203. Springer, 2002.

[15] Armin Wolf. Adaptive constraint handling with chr in java. In Toby Walsh, editor, *Proc. CP2001*, LNCS 2239, pages 256–270. Springer, 2001.

[16] Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaption of constraint handling rule derivation after constraint deletions. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 2000.

[17] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.