A Toolkit for the Realization of Constraint-Based Multiagent Systems

Federico Bergenti

Consorzio Nazionale Interuniversitario per le Telecomunicazioni
AOT Lab
bergenti@ce.unipr.it
http://www.ce.unipr.it/people/bergenti

Abstract. Autonomy is largely accepted as a major distinctive characteristic of agents with respect of other computation models. This is one of the main reasons why the agent community has been investigating from different perspectives constraints and the tight relationship between autonomy and constraints. In this paper, we take the software engineering standpoint and we exploit the results of the research on constraint programming to provide the developer with a set of tools for the realization of constraint-based multiagent systems. In detail, the purpose of this paper is twofold. In the first part it presents a model that regards multiagent systems in terms of constraint programming concepts. This model comprises an abstract picture of what a multiagent system is from the point of view of constraint programming and a language for modeling agents as solvers of constraint satisfaction and optimization problems. The second part of this paper describes an implemented toolkit that exploits this model to support the developer in programming and deploying constraint-based multiagent systems. This toolkit consists of a compiler and a runtime platform.

1 Introduction and Motivation

Autonomy is largely considered a characteristic feature of agents that differentiate them from other computation models [13]. Many researchers claim that autonomy is the one and only distinctive features of agents [3] and the large amount of work about goal-directed behavior [2] or, more generally, about rationality [9] has the ultimate goal of providing a scientific understanding of autonomy. Along these guidelines, the research that motivates the results described in this paper has been devoted to study the opposite side of the coin: if we accept that agents are inherently autonomous, then we need to face the engineering problem of constraining this autonomy in a reasonable way. Even if some researcher have a radically different opinion, see, e.g., [14], we believe that a major step we need to undertake to regard agents as applicable abstractions in the engineering of everyday software system is to find a reasonable way to constraint the autonomy of agents.

Rationality can be regarded as one way to achieve this purpose, but facts have already proven extensively that the use of rationality in real system is still remote. Nevertheless, we need some way to guarantee (at least) some very basic properties of engineered systems, e.g., safety and liveness, and we cannot simply accept that something, which is not under the control of the developer, might emerge and damage the system.

In this paper we consider a method for constraining the autonomy of agents that exploits the direct intervention of the developer in defining what are the boundaries of an acceptable behavior for an agent. This method relies on the results of the research on constraint programming because of two very basic reasons. The first is that constraint programming treats constraints, and therefore autonomy, as a first class metaphor. This allows the developer to manipulate constraints directly. Moreover it offers the great advantage, over other forms of management of the autonomy, of being independent of any grand principle, like rationality. Such principles are elegant and they permit to treat a large set of different problems homogeneously. Nevertheless, they often become subtle and difficult to manage for the developer because of their inherent problem-independence. Rather, constraint programming puts the focus on the problem at hand and it uses only the constraints that are embedded in the problem itself, and no other grand principle is required.

The second reason for choosing constraint programming for managing the autonomy of agents is that it is sufficiently powerful to allow the description of another very basic characteristic of agents: goal-directed behavior. The success of constraint programming in problems like scheduling [10] and planning [12] demonstrates that it can provide good results in supporting the desired goal-directed behavior of agents.

This paper presents the results of the work that we have been doing during last year, and that is based on the guidelines that we have just stated. The aim of this project is to deliver a set of constraint-based tools that everyday developers could adopt for the realization of their multiagent systems. We did not choose any reference application domain for such tools because we intended to provide an enabling technology capable of providing its benefits in many cases. Actually, we believe in the position that Freuder stated in its celebrated truism [4]: *"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

The description of our tools starts, in the following section, with an introduction to what constraint-based multiagent systems are for us. Then, in section 3 we present a programming language that exploits the model of constraint-based multiagent systems to provide the developer with a direct means for implementing agents as solvers of constraint satisfaction and optimization problems. This language is the core of the development toolkit that we present in section 4. This toolkit consists of a compiler and a runtime platform and it allows for a seamless integration of constraint-based multiagent systems with everyday applications. This work is summarized in section 5, where we outline some conclusions and we present some future direction of research.

2 Constraint-Based Multiagent Systems

The interplay between autonomy and constraints has been the subject of a large work in various research communities and under various points of view, e.g., classic works about this are [6, 7, 8]. In our work we take the point of view of software engineering and we see *agent-oriented constraint programming* as a particular approach to realize multiagent systems, rather than, e.g., a distributed approach to solve constraint satisfaction problems.

In order to give a more precise meaning to these words, and to show the advantages of this approach, we start from the foundations of constraint programming and we show how these can be used as guidelines for the realization of multiagent systems. It is worth noting that even if some objectives are in common, our approach differs significantly from concurrent constraint programming [11] because we decided to start from a more operational, and more modern, approach to constraint programming.

A *constraint satisfaction problem* (CSP) [1] is a tuple $< X, d, C >$, where X is a set of n variable names, $X = \{x_1, \ldots, x_n\}$, $d : X \rightarrow D \subseteq \mathbf{R}$ is a mapping between a variable name and the domain of its possible real values, and C is a collection of m constraints $C = \{C_1, \ldots, C_m\}$. Each constraint is a proposition over a subset of the available variable names $S \subseteq X$, called the *scheme* of the constraint.

In this definition we consider only real variables because all other sets we need to deal with, e.g., sets of strings enumerated extensively, can be mapped bi-directionally to subsets of real numbers.

A solution to a CSP is an assignment of values that maps all variable names in X with a value compatible with d that satisfies all constraints in C.

An interesting property of this definition of CSP is that it allows for an easy definition of sub-problem. This feature is of singular importance in our agent-oriented approach because we will exploit it in treating goal-delegation.

We say that CSP_1 is a *sub-problem* of a CSP_2 if $X_1 \subseteq X_2$, and all constraints in CSP_2 whose scheme is a subset of X_2 are in CSP_1.

For the sake of simplicity, in this paper we do not take into consideration the serious problem of possible mismatches between the representations of different problems. If two problems contain the same variable name, we assume that these names are two appearances of the same variable, which is then shared between the two problems.

In order to exploit these ideas in an agent-oriented fashion, we need to introduce some notion supporting the embodied nature of agents and their inherent goal-directed behavior. This is the reason why CSPs are not sufficient and we need to rely on a well-known extension of them.

A *constraint satisfaction and optimization problem* (CSOP) [1] is a CSP with an associated targets T, i.e., it is a tuple $< X, d, C, T >$ where $T : S \subseteq X \rightarrow \mathbf{R}$.

A solution of a CSOP is a solution of the underlying CSP that maximizes the target T.

We say that $CSOP_1$ is a sub-problem of $CSOP_2$ if the CSP associated with $CSOP_1$ is a sub-problem of the CSP associated with $CSOP_2$ and any solution of $CSOP_1$ is a partial solution of $CSOP_2$ for the set of variables of $CSOP_1$.

This definition of sub-problem relies on a very demanding condition and we need to alleviate it with the introduction of the notion of *weak sub-problem*. We say that a $CSOP_1$ is a weak sub-problem of $CSOP_2$ if the CSP associated with $CSOP_1$ is a sub-problem of the CSP associated with $CSOP_2$, and nothing is said on the two targets. This definition allows decomposing a problem into sub-problems and to solve the various sub-problems independently, without requiring the decomposition of the target of the original problem.

The approach that we propose to exploit these ideas for the realization of multiagent systems is to state that an agent is nothing but a solver of a particular CSOP and that each agent in a *constraint-based multiagent system* may have a different CSOP to solve.

Agents acquire from their sensors:

1. Constraints, that are inserted or removed from the set of constraints of their problems; and
2. Values for the variables of their problems, that can be asserted or retracted, i.e., particular constraints of the form $X = x$.

Agents act in accordance of the solution, even partial, they find for their problems. Once an agent has definitely found a partial solution to its problem, i.e., once it comes to know that it will not backtrack the partial assignment of its variables if nothing changes in its internal state, then it can act on the environment or it can perform communicative actions in the direction of other agents. The selection of the action to perform depends only on the partial solution it found.

The multiagent system as a whole is not associated with any CSOP, rather its behavior emerges from the interactions of all agents. Such interactions can occur through the environment, i.e., through shared variables, or though communication.

Communication between agents is represented in our model using the standard approach of communicative acts. But, differently from many other approaches, we largely reduce the problems of communicative acts because we take a minimalist approach to agent communication languages. We say that an agent communicates with another agent only to ask to this agent to perform some action, whether communicative or not, whose outcome would help it solving its problem. Basically, we say that an agent communicate with another agent only to delegate a weak sub-problem of its CSOP to it.

Exploiting this simple model of communication we have the advantage of decoupling the problem of actually choosing what to say and to who from the CSOP the agent solves. If we work under the assumption that all agents provide to a central repository the (true) list of problems they can solve, then an agent A_1 communicate with an agent A_2 only if:

1. The problem that A_2 solves is a weak sub-problem of the problem of A_1; and
2. A_1 cannot, or does not want, to solve this sub-problem on its own.

From a rather superficial point of view, this approach to communication seems too poor with respect of the standard approaches that one may expect from an agent model. This is only partially true, because this approach is a direct implementation of goal-delegation, which is largely considered the ultimate rationale of communication between agents. Moreover, this model is not limited to cooperative agents only, because agents are associated with different CSOPs and optimal solutions to such problems may be conflicting. In particular, the target of the CSOP of each agent can be assimilated to the utility function of that agent, and each agent has potentially a different target for its problem.

The model that we have just outlined is obviously ideal for a number of reasons. First, it does not include time as a first-class concept. Time can be modeled as a variable in the CSOP of an agent, but the tight coupling between such a variable and the resolution process, through the time spent for actually solving the problem, can cause severe problems. Moreover, our model allows actions only in result of a partial solution of the problem of an agent. This implies that an agent cannot perform actions for a part of the duration of the resolution process, which inherently depends on the problem the agent is facing. This is a classic inconvenience found in many other agent models and it raised crucial problems like belief and intention revision [5].

Such limitations of our model are significant, but for the moment we decided not to take them into consideration because we adopted the standard approximation of quasi-stationary environments. If the time needed to solve the problem is sufficiently small with respect of the expected time of reaction of the agent to a change in the environment, then the agent would behave as expected. We decided to adopt this assumption because our experience suggests that it is applicable in many realistic situations, especially if we target common software systems where, if it is sufficiently reactive for the user, then performances is not an issue.

Our model of agents as CSOP solvers has many resemblances with more traditional agent models, especially with rational agents. Anyway, we put a strong focus on the internals of the agent, rather than on its behavior seen from the outside, and therefore the ascription of a mental state to such agents seems a difficult problem. This difficulty justifies the assumption that our agents are not easily framed into other agent models and therefore we decided to find a new nickname for them capable of capturing their nature of atomic computational entities whose interaction animates the multiagent system as a whole. This is the reason why we use to refer to this particular sort of agents as *quarks*.

3 A Language for Agent-Oriented Constraint Programming

The model of quarks that we outlined in the previous section has a number of interesting properties. Among them, it is worth noting that it does not depend on any particular (and possibly implicit) restriction that a specific language for constraint programming might impose. Therefore, it offers a good level of generality and expressive power. Moreover, it is a good approach for studying the algorithms capable of controlling the behavior of the quark.

Nevertheless, we cannot simply give to the developer the notions of variable, domain and constraint. It is not a reasonable approach for a developer that is already familiar with the sophisticated modeling techniques that object orientation has promoted in the last twenty years. The basic problem is that the effort required for mapping a reasonably complex application domain into a set of variables, domains and constraints seems excessive. We definitely need to define a language supporting higher level abstractions and the rest of this section addresses this problem.

The *Quark Programming Language*, or QPL (pronounced *kju:pl*), was designed to provide the developer with a user-friendly approach to realizing quarks. We defined an abstract model of it and then we mapped this model to a concrete syntax. Exploiting this syntax we realized a compiler, with an associated runtime platform, for a concrete use of QPL in running systems. The compiler and the runtime platform are described in next section and what follows is an informal description of QPL. It is worth noting that the semantics of QPL has already been formalized with two different approaches. The first is an operational mapping between a QPL program and a CSOP. This is a crucial step because it allows mapping a quark written QPL with an algorithm for controlling its behavior. The second approach is based on a description logic and it is useful for understanding the expressive power of the language. For the sake of brevity, none of these semantics is presented here and the subject is left for a future paper.

The realization of a quark in QPL requires to define the class of quarks it belongs to. This class defines the common characteristics of each quark it comprises and it provides a means for the developer to instantiate quarks. A class of quarks is composed of:

1. A name;
2. An import section;
3. A public problem description section;
4. A private problem description section;
5. A targets section; and
6. An actions section.

The name of the class is used to support the creation of new quarks: the developer uses it when he/she wants to instantiate new quarks. The import section declares the external components that the quark will use. Such components are non-agentized, third-party software that a quark may need to exploit in its lifetime.

These two sections of the program of a quark provide a bi-directional link between the quark and the rest of the non-agentized software of the system. This is why they are (intentionally) described vaguely in the specification of QPL. Any possible implementation of QPL has to deal with its own means for instantiating quarks and with its interface with external components. In the implementation of QPL that the following section describes, quarks can be instantiated by means of an API available through a .NET and a WSDL interface. Similarly, external components are imported and accessed through a .NET and a WSDL interface.

After the import section, QPL requires the developer describing the problem that the quark will solve. Such a description deals only with the CSP part of the CSOP that the quark will solve. This description is split into a public section and a private section. The public section provides a description of the problem that is suitable for a publication in the central repository of the system. This description can be used to tell other quarks what this quark is capable of doing, i.e., what are the problems it can solve. This part of the description of the problem is used to support communication and goal delegation. The private section refines what the developer declared in the public section with an additional set of details that he/she needs to introduce to make the quark fully functional. Such details are not essential for other quarks to reason about the problem this quark solves, and these can be assimilated to implementation details that the good old principle of information hiding suggests to keep private.

Both the public and the private sections of the description of the problem are then split into the following sections:

1. A structure section; and
2. A constraints section.

The structure section describes the domain of the problem the quark will solve. This description is based on a classic process of classification that closely resembles the one we commonly use in object-oriented modeling. This section of a QPL program has the ultimate goal of defining a vocabulary for describing the constraints and for naming the variables of the CSOP of the quark. The constraints section uses the vocabulary identified in the structure section to define the constraints of the CSOP of the quark.

The structure section of a class of quarks is described in terms of:

1. A set of classes of objects of the domain of the problem;
2. A set of relations between such classes;
3. A set of catalog objects;
4. A set of enumerative types; and
5. A set of constrained predefined types.

A class of objects is composed of a set of attributes, each of which is described as a name and a type. The type of an attribute is one of the types that the developer defines when he/she declares its enumerative or constrained predefined types (points 4 and 5 in the previous list). An enumerative type is a set of elements expressed extensionally that belongs to one of the predefined types that QPL provides, i.e., string, double, integer, and boolean. All elements of this list belong to the same predefined type.

In cases where an extensive enumeration of values is not practical, the developer can define a subset of the values of a predefined type through a constrained predefined type. This is a predefined type plus a constraint that restricts its possible values. For the moment, QPL allows constraining only doubles and integers and it provides only three constraints: one for setting a minimum value, one for a maximum value and one for a step in the series of values.

A class of objects in QPL can be a:

1. Catalog class;
2. Configurable class; or
3. Abstract class.

A catalog class is a class of objects whose elements are extensively enumerated in the program of the quark. These enumerated values are called *catalog objects* and they are listed as point 3 of the features that the developer uses in the structure section of a class of quarks. In the concrete implementation of QPL that the following section describes, catalog objects can be enumerated directly in the program of a quark, or they can be imported from the tables of a relational database.

A configurable class is a class of objects whose elements are described intensionally. These classes are modeled only in terms of their attributes: each attribute has a set of valid values as it is associated with an enumerative type or with a constrained predefined type. The characteristic feature of these classes is that we do not provide any restriction on the values that various attributes might take in a single instance. The constraints that we will introduce later, in the constraint section of the class of quarks, will provide the conditions that the values of the attributes of a single instance of a configurable class must respect. This should make clear that the quark regards the attributes of configurable classes as variables of its CSOP. The ultimate goal of the quark is to assign a value to any attribute of any instance of any configurable class in its current solution.

The third type of class we can define in the structure section of a quark is that of abstract classes. An abstract class is a class that we use to collect a set of attributes common to a set of catalog classes into one single container. An abstract class is modeled only in terms of its attributes (just like a configurable class) but we need to subclass it with another abstract class, or with a catalog class, in order to give a meaning to it. Abstract classes are useful to provide an abstract view of (a superset of) a set of catalog classes. Moreover, they allow structuring the domain of the problem and expressing constraints on attributes shared by a set of catalog classes.

QPL supports the assembly of classes of objects though relations. Such relations between classes can be of three types:

1. Generalization/specialization, that expresses a superset/subset relation between the objects of two classes;
2. Association, that expresses a shared composition of the objects belonging to two different classes: each object of the container class is made of a number of objects of the contained class, and such objects can be shared among a number of relations; and
3. Containment, that expresses a private composition of the objects belonging to two different classes: each object of the container class is made of a number

of objects of the contained class, and such objects cannot be shared among relations.

The latter two relations are qualified with a cardinality that models the number of objects of the contained class that take part of the relation.

The structure section of a QPL program provides all features we need in order to define the variables and the domains of the variables of the CSOP. Each attribute of any instance of a configurable class is a variable of the CSOP. The domains of such variables equal the domains of the corresponding attributes. Such domains are specified using enumerative types of constrained predefined types.

In order to complete the description of the CSOP of the quark, we need to declare a set of constraints and a target. QPL provides two ways for describing a constraint:

1. Compatibility/incompatibility tables; and
2. Rules.

A compatibility/incompatibility table is a list of tuples that enumerates the possible values of a group of attributes. If the list contains permitted tuples, we talk of a compatibility table, otherwise we talk of an incompatibility table. The two representations are identical and the choice depends only on the number of entries in the two possible lists.

Compatibility/incompatibility tables offer an extensional means for modeling a constraint. On the other hand, rules provide an intensional form of describing a constraint. A rule is a proposition that must hold for any possible solution to the CSOP of the agent. QPL provide only one type of rule: *if/then/else* clauses. The building blocks that the developer can use to compose the three expressions of an if/then/else clause are:

1. The set of standard operators over integers, doubles, booleans and strings; and
2. A set of *attribute terms* that are found in the vocabulary defined in the structure part.

The attribute terms allow the developer to identify variables that the quark will use to validate the rule. The general form of an attribute term in a rule is:

```
class[#id](.relation[#id])*.attribute
```

If we forget about `#id` for a moment, this form simply allows navigating classes and relations to reach an attribute from an initial class called `class`. From this class we can exploit association and composition `relation`s to reach other classes. Once we reached the class that comprises the attribute we are addressing, we identify the `attribute` through its name.

The use of `#id`s allows narrowing the number of variables that this pattern matches. If we do not use any `#id`, all instance of all classes met during the traversal from `class` to `attribute` are used in the rule. For example, `PC.type` addresses the value of the attribute `type` of all instances of the class `PC`. The use of

`#id` allows choosing a particular instance of a relation or of a class and restricting the application of the rule only to such instance, e.g., `PC#0.hardDrive#1.speed` matches the attribute speed of the second hard drive of the first `PC` only.

QPL allows using `#ids` in conjunction with two general purpose operators: `sum` and `product`. These operators have the standard meaning of repeated sums and products and they can contain expressions where `#id` is replaced with a variable term.

The grammar that QPL provides for rules is completed with two shortcuts for expressing common constraints:

1. `always`, meaning `if true then`; and
2. `never`, meaning `if true then not`.

As an example, the following is a very simple rule in QPL that states that if the type of PCs is games, then any hard drive in the needs to be at least 20GB.

```
if PC.type = ‘Games’ then HardDrive.size > 20
```

Each rule in QPL can be relaxable and relaxable rules are assigned a priority that drives the CSOP solver in deciding which rule to relax in order to find a solution. In the standard semantics of QPL, rules are relaxed only to find a solution and no rule is relaxed if a solution is already available, e.g., in the attempt to find a better solution.

The definition of the CSOP of a quark is completed with the definition of a target. This is done in QPL through a list of expressions that evaluate to a number. Each one of these expressions is assigned a priority and a direction, that can be maximize or minimize. The standard semantics is that the quark tries to maximize/minimize the target expression with a given priority only if all other target expressions with greater priority are already maximized/minimized.

The expressions used to indicate targets are composed with the same building blocks that QPL allows for the expressions of an if/then/else clause, with the sole restriction that they must evaluate to a number.

A quark can act on the outside world when it finds a partial solution to the problem it is managing. QPL allows the developer to specify actions in terms of a fire condition and a concrete act. The fire condition is an expression that evaluates to a boolean value and that exploits the vocabulary and the grammar used for rules and targets. When a fire condition holds, i.e., when a partial solution that verify a condition is found, the quark performs the associated act. Fire conditions are prioritized and the quark performs the act associated with the fire condition with the topmost priority that currently holds.

The concrete description of acts is kept out of the language specification because it relies on the concrete implementation of QPL and on how quarks are enabled to interact with non-agentized, third party software. In the implementation presented in the following section, acts can be invocations of methods of .NET components (previously imported), or they can be invocations of Web services (previously imported). In both cases, the partial solution found can be used to supply the arguments to the invocation.

The definition of actions closes the definition of a class of quarks in QPL. This definition allows enumerating all building blocks that the quark needs to know which problem to solve and what to do during the resolution process. Only two aspects seems missing: some means to allow quarks sensing the environment, and some other means to allow quarks communicating. The problem of sensing the environment is intentionally left out of the specification of the language because of the same reason we mentioned for concrete acts. Quarks does not sense the environment actively, they come to know of any change in the environment because of changes in their problem. How external software, e.g., the manager of a sensor, can actually achieve this is part of the concrete implementation of QPL. In the implementation presented in the following section, a QPL program is compiled to a .NET class or to a WSDL interface and both provide an API for pushing information directly in the problem of the quark.

The problem of communication is basically the same as the problem of sensing. As we briefly mentioned in the previous section, our model uses a minimalist approach to communication that allows hiding the process of information exchange from the developer. Each concrete implementation of QPL will have its own way to exchange weak sub-problems between quarks. In the implementation presented in the following section, the runtime platform provides the central repository for publicizing the capabilities of quarks, and it exploits .NET and WSDL interfaces for concretely exchanging messages in the multiagent system.

4 A Toolkit for Agent-Oriented Constraint Programming

In this section we introduce the *Quark Toolkit*, or QK (pronounced *kju:k*), a toolkit we realized to support the developer in implementing multiagent systems based on the ideas we described in the previous sections. QK is made of two parts:

1. A compiler, that compiles the QPL program of quarks to executable modules; and
2. A runtime platform, that provides all facilities we need to deploy a multiagent systems.

The compiler of QK is a command-line tool that takes a set of QPL programs, one file for each single class of quarks, and generates a set of .NET classes and a set of WSDL interfaces. These classes and interfaces are dual and classes implement the relative interfaces. This approach allows using quarks in two ways:

1. As .NET components exposing a fairly simple API for two-way communication with the rest of the .NET system;
2. As Web services that can be integrated in any system capable of exploiting their WSDL interface.

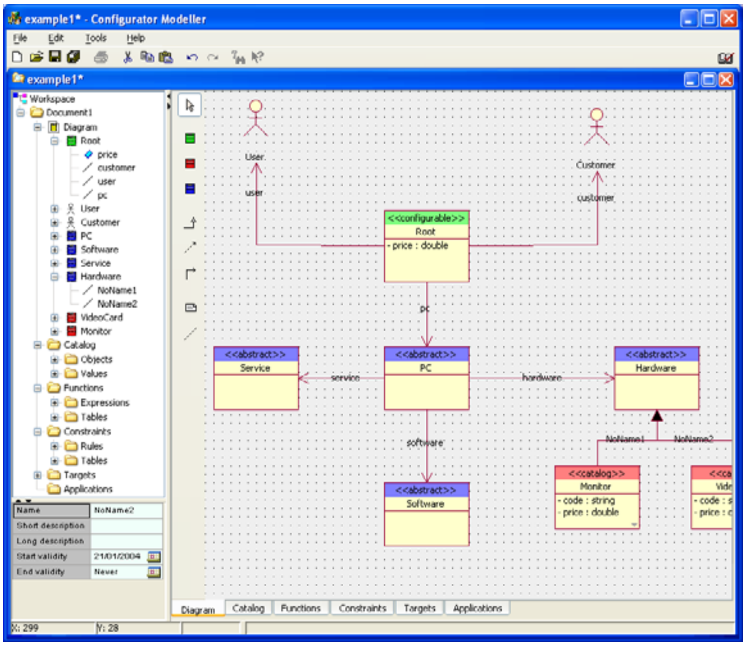
Both approaches are equally valid from the point of view of the developer and the pros and the cons of them have already been discussed largely after the introduction of .NET.

The QK compiler produces one .NET class for each single class of quarks. The interface of these classes does not depend on the problem a quark is designed to face, but it simply enables a bi-directional communication between the quark and an external .NET object. In particular, this interface provides a set of methods for informing the quark of new values for a variable or of new constraints in the problem. Then, it exposes a listener interface, together with a set of management methods, to allow an external component to observe the state of the reasoning process. Finally, this interface provides a few management methods that the QK runtime platform uses to manage the lifecycle of quarks and to interface a quark with the central repository of the platform.

The compiler of QK produces a .NET Intermediate Language (IL) source code, i.e., a source code of .NET IL mnemonics. This compilation is direct and it does not need to pass through a higher level language, e.g., C# or Java. The produced .NET IL is then translated into its executable form exploiting an IL assembler. The system has been tested with the two most popular IL assemblers:

```
agents ShopAssistant {
  uses webservice PriceManager = ‘http://...’;

  ...define what a PC is
  via composition, aggregation and inheritance...

  target minimize PC.delivery

  rule SOLO9100SE
    if PC.code = ‘SOLO9100SE’ then
      // Constraint on processor
      Processor.type  = ‘Pentium’       and
      Processor.clock => 300            and
      Processor.clock <= 366            and
      // Constraint on RAM memory
      RAM.type = ‘SO-DIMM’              and
      RAM.size >= 16 and RAM.size <= 384 and
      ...

  action EstimatePrice
    if PC.code = ‘SOLO9*’ then
      PC.price = PriceManager.
        EstimatePrice9XXX(PC, Customer);
}
```

Fig. 1. A simple QPL source code



Name	NoName2
Short description	
Long description	
Start validity	21/01/2004
End validity	Never

Fig. 2. The visual quark modeler

the one available in Mono (`http://www.go-mono.org`) and the one available in the .NET Framework (`http://www.microsoft.com/net`).

The compilation process is straightforward because it is basically a simple mapping between the QPL program and the equivalent IL source code that exploits the reasoning engine that the runtime platform provides.

The runtime platform is a container capable of hosting a number of quarks that can be loaded and started programmatically or from the command line.

The core of runtime platform is a reasoning engine we developed to efficiently solve constraint satisfaction and optimization problems. The discussion of the techniques we used to implement this engine is out of the scope of this paper, but it is worth mentioning that it uses a mixture of standard constraint programming algorithms, e.g., AC-2001.

The reasoning engine is multi-threaded and all quarks hosted in the same container share the same engine. This is particularly useful because the engine can handle a number of problems concurrently, with the possibility of sharing many internal structures that are possibly common to many problems. This is the reason why the CSOP solver that quarks exploit is not embedded in quarks themselves, rather the runtime platform provides it.

Figure 1 shows selected pieces of a QPL source code that can be compiled with QK compiler. The class of agents implements shop assistants for a Web-shop that can manage the configuration of PCs for customers.

QK has already been used in a number of experiments, mostly for research purposes. It has been recently adopted as the basis of a product that the company FRAMeTech S.R.L. (`http://www.frame-tech.it`) will deliver to their customer later this year. Figure 2 shows a snapshot of this product. The purpose of this product is to provide a user-friendly approach to fast developing product and service configuration systems. Such systems are meant to provide final users with a self-service mechanism for doing the configuration of complex product and services. Examples of such systems are typically used by large hardware shops to provide their customers with a Web application for configuring and then buying personalized PCs and peripherals. Another typical example regards travel agencies giving the possibility to their customers to have a fine-grained configuration of their trips. This system is basically a graphical front-end for realizing QPL programs. It allows modeling the domain of the problem using an UML class-diagram editor. Then, it allows defining all features available in a QPL program in terms of tables and expressions. Finally, it makes a GUI designer available to provide quarks with simple GUIs that final users will exploit for easily managing the problems of their quarks.

5 Conclusion

In this paper we described a set of tools that we realized to support the developer in the realization of constraint-based multiagent systems. These systems are a subset of ordinary multiagent systems because we require agents composing the system to be ascribable to CSOP solvers. This requirement was the starting point of defining a general-purpose programming language to realize agents as CSOP solvers, and a toolkit supporting the deployment of such agents in running systems. This language concentrates on modeling the problem an agent is in charge of and any other meta-level issues, e.g., governing the process of resolution, or orchestrating interactions between agents, are intentionally left implicit.

This language is concretely supported by a toolkit that provides a compiler and a runtime platform. Moreover, this toolkit supports a seamless integration of such agents with legacy systems.

Interesting future research directions regards understanding the relationship between the agent model that we propose and mentalistic agents. It is quite obvious that constraints plays a crucial part also in mentalistic agents, but a clear mapping between basic concepts of such models is still missing.

Another interesting development of this research is in the direction of understanding how inheritance of quarks can be exploited as a software engineering tool.

References

1. F. Bartak. Constraint programming – What is behind? *Procs. Int'l. Workshop on Constraint Programming in Decision and Control*, 1999.

2. C. Castelfranchi. Guarantees for autonomy in cognitive agent architecture. *Intelligent Agents*. Springer-Verlag, 1995.
3. S. Franklin and A. Graesser. Is it an agent, or just a program? *Procs. ECAI'96 Workshop on Agent Theories, Architectures, and Languages*, pages 21–36. Springer-Verlag, 1997.
4. E. C. Freuder. In pursuit of the holy grail. *Constraints*, 1(2), 1999.
5. S.-O. Hansson. *A Textbook of Belief Dynamics*. Kluwer Academic publishers, 1997.
6. M. Henz, G. Smolka, and J.Würtz. Oz – A programming language for multi-agent systems. R. Bajcsy (Ed.) *Procs. 13th Int'l. Joint Conference on Artificial Intelligence*, volume 1, pages 404–409. Morgan Kaufmann Publishers, 1993.
7. A. Mackworth. Quick and clean: Constraint-based vision for situated robots. *Procs. Int'l. Conference on Image Processing*, 1996.
8. M. J. Maher. Logic semantics for a class of committed-choice programs. *Procs. 4th Int'l. Conference on Logic Programming*, pages 858–876, 1987.
9. A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.
10. C. L. Pape. Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.
11. V. A. Saraswat and M. Rinard. Concurrent constraint programming. *Procs. 7th Annual ACM Symposium on Principles of Programming Languages*, 1990.
12. P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. *AAAI/IAAI*, pages 585–590, 1999.
13. M. J. Wooldridge. Intelligent agents. G. Weiss (Ed.) *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 27–77. The MIT Press, 1999.
14. F. Zambonelli and V. Parunak. Towards a paradigm change in computer science and software engineering: A synthesis. *The Knowledge Engineering Review*, 2004.