Constraints on Objects, Conceptual Model and Implementation

Richard H. M. C. Kelleners
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, and
Technical University of Eindhoven, Dept. Computing Science
Den Dolech 2, 5612 AZ, Eindhoven, The Netherlands
E-mail: richard@win.tue.nl

Remco C. Veltkamp
Utrecht University, Dept. Computing Science
Padualaan 14, 3584 CH Utrecht, The Netherlands
Remco.Veltkamp@cs.ruu.nl

Edwin H. Blake
University of Cape Town, Dept. Computer Science
Rondebosch 7700, South Africa
e-mail: edwin@cs.uct.ac.za

Abstract

This paper presents the design and implementation of a model for combining object oriented programming and constraint programming. This model aims to be an aid in the development of computer graphics applications that use these two programming paradigms.

We first identify the typical aspects of the paradigms and how they conflict with each other. Next, we show how these conflicts are solved in the model by radically separating the object oriented paradigm from the constraint paradigm. The communication between these two systems is then managed by a third party.

A prototype of the model was implemented, based on a coordination language a language for managing concurrent, independent processes. This allows us to build an ideal implementation. Although this ideal implementation does not yield the necessary performance in speed, it clearly demonstrates that the model provides an appropriate way for setting up an application that incorporates object oriented programming as well as constraints.

1 Introduction

In today's interactive graphics applications, such as drawing applications, CAD/CAM systems, systems for visualization, simulation, animation, etc., we can often recognize two basic design methodologies. These two are object oriented programming and constraint programming. Object oriented programming is widely used to design and implement graphics systems, both in commercial and research fields. On the other hand, the usage of constraints and constraint solving techniques is not so widespread in the commercial world, but it has a long history in research and dates back to the first days of interactive graphics ([Sut63]).

Both methodologies are powerful techniques for building computer graphics applications. Object oriented methods provide sound software engineering principles needed to cope with the design and implementation of large, complex software systems. The use of constraints allows for the declarative modeling of the behaviour of animations and interactions with many components or objects.

The power of both approaches justifies for a combination where objects and constraints are unified in a harmonious and coordinated whole. However, integration leads to conflicts in programming methodologies ([FBB92]), which obstructs the application of constraints in the object oriented approach. We distinguish two incompatibilities between constraints and object oriented concepts:

- A constraint solver looks at and sets the internal data of an object.
- Object oriented programming is imperative, and constraint programming is declarative.

The first incompatibility conflicts with the information hiding principle in the object oriented paradigm. The second incompatibility identifies a difference in programming methodology: object oriented programming specifies the actions to be taken (imperative programming), while constraint programming specifies a set of constraints that have to be maintained (declarative programming).

A number of ways have been described to combine objects and constraints. Some methods are completely based on message passing. In [LvdB91], the methods of an object that may violate constraints are guarded by so-called propagators. The propagators send messages to other objects to maintain the constraints. This technique is similar to the pre- and postcondition facilities in Go [Dav91] [GoP93]. This approach is limited to constraint maintenance, starting with a consistent situation.

Equate [Wil91] uses *term rewriting* as a guide to find solutions. Constraints are specified as equations, and rewrite rules convert equations into equivalent sets of equations that can be solved by messages to an object. The rewrite rules which rewrite the equations are provided by the classes and are similar to the program clauses of logic programs. The Object-Oriented Constraint System (OOCS) [HB94] is similar, does not use term rewriting. Instead, an object supplies a set of *solution program segments* for each constraint that has been imposed upon it. The object guarantees that execution of any of these segments will leave the object in a state which satisfies the constraint. OOCS then solves a set of constraints by determining which program segment steps interfere with each other. By arranging the solution steps, the OOCS solver is able to decide which are feasible solutions, if any.

These systems often cannot solve global constraints, due to the local character of the satisfaction mechanism. More powerful solutions are necessarily global in nature. The danger is that all objects need methods to get and set their internal data. This however, allows every other object to get and set these values, which is clearly against the object-oriented philosophy. One way to restrict this, is to have an object allow value setting only when its internal constraints remain satisfied (see [Ran91]). A constraint could be made internal by constructing a 'container object', which contains the constraint and the operand objects, but this does not solve the basic problem. Another approach is to limit access to private data to constraint-objects or the constraint solver-objects only. For example, C++ provides the ‘friend’ declaration to grant functions access to the private part of objects. This is also comparable to the approach taken by [CBL91], where special variables (slots) are accessible by constraints only. However, encapsulation is still violated, and the C++ friend construct could be easily misused.

On the other hand, under strict information hiding, constraint satisfaction on objects cannot be guaranteed [VK95]. If one is to sacrifice strict information hiding in order to facilitate constraint satisfaction, care should be taken not to allow abuse. The proposed solution is to separate the object oriented application from the constraint framework and let the two systems then communicate via a communication scheme that is different from the message passing activity.

The separation provides a remedy to the above mentioned incompatibilities. Firstly, the communication scheme takes place orthogonal to the objects’ message passing activity, which guarantees that the information hiding conflict *among objects* will not be violated. Secondly, separation of the object oriented application from the constraint framework creates a decoupling between the imperative and the declarative paradigms. This facilitates the design, implementation and maintenance of the systems, without sacrificing any of the power of both methodologies.

In this document, we present a conceptual model that uses this separation for building computer graphics systems that use both objects and constraints. Section 2 present the conceptual model. In section 3, we describe an implementation of the model using the coordination language **Manifold**. The last section discusses directions for future research.

2 Conceptual model

The conceptual model offers a controlled way of dealing with the violation of the information hiding principle by separating the object oriented framework from the constraint framework. The two frameworks communicate by a separate third-party communication protocol that is different from the typical object oriented message passing activity.

There are the following types of entities: constraints, constrainables, solvers, coordinators, and a single Communication Server, see Figure 1 for an overview. Informally, the constrainables correspond to objects in the object oriented application and can be subjected to constraints. The constraints specify relations among the constrainable entities they operate on. A solver computes values for (the attributes of) constrainables that satisfy the constraints imposed on them. Coordinators are responsible for the coordination of possibly multiple, concurrently operating constraint solvers, depending on the type of constraint solvers. For example, constraint satisfaction by local propagation needs other coordinators than constraint satisfaction by solving a set of equations. The communication between constraints, constrainables, solvers, and coordinators is done by event raising and data flows. Those events are handled by the Communication Server. It sets up and destroys data flows when needed.

2.1 Communication Server

The Communication Server manages the communication between the entities in the conceptual model. The general outline of the Communication Server was inspired by the language

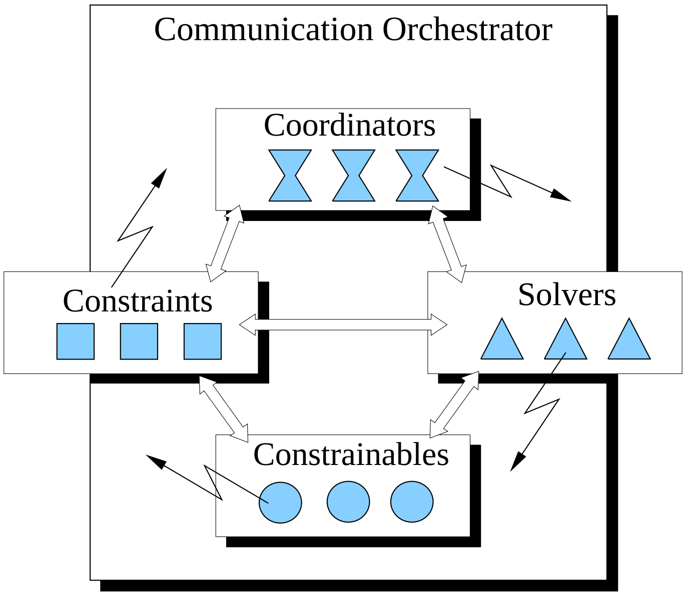

Figure 1: Constraints, constrainables, solvers, and coordinators communicate with each other via events (zigzag arrows) and data flows (double arrow), which are provided by the Communication Server.

MANIFOLD and its underlying concepts (see section 3.1 and [Arb96]). The two most important notions are event and data flow.

event An event is an asynchronous, non-decomposable (atomic) message, broadcasted by the Communication Server or by a constrainable, constraint, solver, or coordinator entity.

- *Raising* an event means that it is broadcasted to the environment. An entity that raises an event is called the *event source*.
- *Posting* an event means that it is detectable only to the entity itself. We call a posted event *internal*.
- An event contains a data structure holding the *name* (or *type*) of the event and a reference to the source that raised it.

An event can be raised by the Communication Server and by any entity (constrainable, constraint, solver, or coordinator). An entity can show its interest in an event by registering itself to the Communication Server. It can register itself for a certain event type raised by a certain entity, for a certain event type raised by any entity, for any event raised by a certain entity, or for any event raised by any entity. All raised events are picked up by the Communication Server and are forwarded to entities that are interested in these events. Entities have an event memory where the Communication Server places the event. Posted events are placed only in the event memory of the posting entity. The entities are polling the event memory to see if they have received an event. They can identify these events and take proper action. An entity that raises an event can make no assumptions on who receives the event and when or in which order received events are processed.

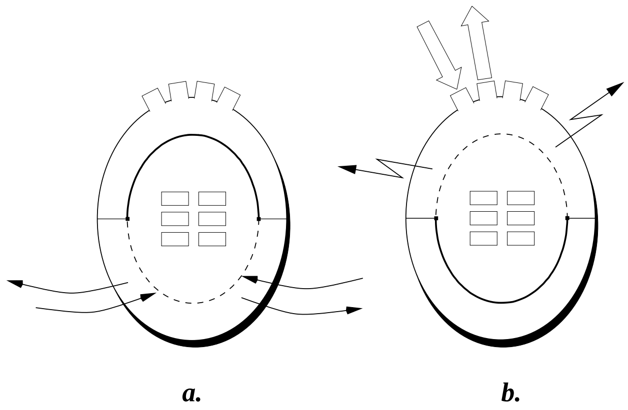

Figure 2: A constrainable can communicate with other objects via message passing (a) and with entities of the conceptual model via events and data flows (b).

data flow A data flow is the transport of units of data. This flow is always in one direction and has a beginning and an end.

- A data flow has an input side, called the *source*, and an output side, the *sink*.
- When data enters a data flow at the source, it is delivered to the data flow's sink, without loss, error, or duplication, and the order preserved.
- The data is transported through a *channel*, which is created and deleted by the Communication Server.

The moments at which channels are created, and between which entities depends on the actions that entities take. An entity can request the Communication Server to create a channel by raising an event.

At the beginning and end of a channel are entities. The connection point for a channel is a port:

port A port is a named 'opening' to an entity, through which data can be exchanged.

- Ports can be guarded by a condition, for example to see if there is a data flow connected to it. If the condition is true, then an event is posted to notify the entity.

By means of events and data flows, entities can communicate in a way that is completely orthogonal to the message passing framework of the object oriented application.

2.2 Constrainables

Objects on which constraints can be imposed, need some extra functionality to make them suitable for constraint solving. It is necessary that they can notify the environment of state changes and that they provide access to internal data for a constraint solver. Objects with this extra functionality are called *constrainable objects* or just *constrainables*.

A constrainable object communicates with its environment in two ways. It can send messages to other objects, which is the normal way of communication in an object oriented environment. Message passing is prohibited when constraint solving takes place. When constraints are being solved the constrainable communicates with other entities through data flows. This is illustrated in Figure 2. The curved arrows in the figure are messages sent or received by the objects. The zigzag arrows depict events. The rectangles inside the constrainable object represent its internal data, the squares on the object boundary depict the ports to which data flows can be connected.

A constrainable is in one of two modes, the solving mode, caused by an event, called E-solving-mode, or the normal mode, caused by the event, called E-normal-mode. When the constrainable is in normal mode, it communicates with other objects via messages. When it is in solving mode, it communicates with other entities through data flows. In both modes the constrainable can raise and detect events.

A constrainable has the following ports:

- P-state-in. Through this input port, the constrainable receives a new state from a solver. There is a guard that posts the event E-state-in when data arrives.
- P-state-out. Through this output port, the constrainable transmits its current state to a constraint or to a solver. There is a guard that posts the event E-state-out when a connection to the port has been made.

A constrainable reacts to the following events:

- E-solving-mode. This event is raised by a constraint on this constrainable, or by a coordinator. In reaction to this event, the constrainable delays the handling of messages from other objects until reception of the E-normal-mode event.
- E-normal-mode. This event is also raised either by a constraint on this constrainable or by a coordinator. Upon reception of this event, the constrainable resumes the handling of messages.
- E-state-out. This is an internal event. The constrainable puts its internal data on the P-state-out port.
- E-state-in. This is also an internal event. The constrainable copies data from the P-state-in port to its variables. The source of the channel connected to P-state-in is the solver that has computed a new state for this object.

A constrainable can raise the following event:

- E-changed. This event is raised when its internal state is changed by methods of the object. No event is raised when the state changes by copying data from the P_state_in port.

2.3 Constraints

A constraint specifies a relation among a set of constrainables. It does not compute a solution to satisfy the relation it describes, this is done by a solver entity (see section 2.4). In an actual application, a constraint and a solver for that constraint may be implemented as a single object, but in the conceptual model they are distinguishable entities with a specific functionality.

A constraint has the following ports:

- P-state-in. Through this input port, the constraint receives the state of an operand from that constrainable. There is no guard on this port. Data only arrives at this port after the constraint has requested it.

- P-param-out. Through this output port, the constraint sends additional data (to the coordinator), e.g. the size of the violation error, or the changed operands. There is a guard that posts the event E-param-out when a connection to the port has been made.
- P-constrainable-out. Through this output port, the constraint sends constrainable identifiers of its operands to a solver. There is a guard that posts the event E-constrainable-out when a connection to the port has been made.
- P-commserv-out. Through this port, the constraint sends constrainable identifiers to the Communication Server in order to establish a channel between the constraint and that constrainable (see below).

A constraint reacts to the following events:

- E-changed. This event is raised by a constrainable that is its operand. Upon reception of this event, the constraint collects the states from its operands and checks whether the constraint relation still holds. Collecting the data is done by raising the event E-state-needed (see below).
- E-constrainable-out. This is an internal event. The constraint puts constrainable identifiers on the P-constrainable-out port, which will be received by a solver.
- E-param-out. This is an internal event. The constraint sends data through the P-param-out port.

A constraint can raise the following events:

- E-violation. This event is raised in order to let the coordinator know when the constraint relation does not hold.
- E-solving-mode. This event is raised in order to put the constrainables that are its operands in solving mode.
- E-normal-mode. This event is raised in order to put the constrainables that are its operands in normal mode, again.
- E-state-needed. This is raised to let the Communication Server make a channel between the P-state-in port of the constraint and the P-state-out port of each of its operands.

A constraint entity, and similarly solver and coordinator entities, can raise events to request the Communication Server to create channels. Depending on the event type, the Communication Server knows which ports have to be connected. E.g., when a constraint raises E-state-needed, a channel has to be created between P-state-out of an entity (in this case, a constrainable) and P-state-in of the constraint. After an entity has done the request (i.e., raised the event), it sends an identifier of the entity through the P-commserv-out port. This identifier is then received by the Communication Server and used to create the channel.

2.4 Solvers

A solver calculates values for the collection of constrainables that satisfy the constraints that the solver operates on. It takes one or more constraints as input and, after performing the necessary calculations, it sends values to the constrainables that satisfy the constraints. A solver may be a dedicated solver that solves only one constraint, but it can also solve a set of constraints.

A solver has the following ports:

- P-constrainable-in. Through this input port, the solver receives constrainable identifiers from the constraint. There is no guard on this port, since data only arrives after the solver has requested it.
- P-constraints-in. Through this input port, the solver receives constraint identifiers from the coordinator. There is a guard that posts the event E-constraints-in when data arrives.
- P-state-in. Through this input port, the solver receives the state of a constrainable. There is no guard on this port. The solver waits for data at this port after he has requested it.
- P-state-out. Through this output port, the solver sends the state of a constrainable. Again, there is no guard on this port. The solver sends the data after it has requested the Communication Server to create the channels.
- P-param-in. Through this input port, the solver receives additional parameters which may be needed during solving from the coordinator. There is a guard that posts the event E-param-in when data arrives.
- P-param-out. Through this output port, the solver sends results to the coordinator concerning the outcome of its calculations. There is a guard that posts the event E-param-out when a connection to the port has been made.
- P-commserv-out. This is the output port for sending constrainable and constraint identifiers to the Communication Server.

A solver reacts to the following events:

- E-constraints-in. This is an internal event. The solver reads constraint identifiers from the P-constraints-in port.
- E-param-in. This is an internal event. The solver reads data from the P-param-in port.
- E-param-out. This is an internal event. The solver puts data on the P-param-out port.

A solver can raise the following events:

- E-state-needed. The event is raised in order to let the Communication Server create a channel between the P-state-in port of the solver and the P-state-out port of a constrainable. The solver communicates the constrainable identification to the Communication Server through the P-commserv-out port.
- E-state-available. The event is raised in order to let the Communication Server create a channel between the P-state-out port of the solver and the P-state-in port of a constrainable.
- E-constrainable-needed. The event is raised to let the Communication Server create a channel between the P-constrainable-out port of a constraint and the P-constrainable-in port of the solver.

A solver is triggered when a coordinator puts constraint identifiers on its P-constraints-in port. Additional data may arrive on the P-param-in port (from the coordinator), which the solver can use in its solving algorithm.

Before starting the calculations, a solver requests constrainable identifiers from the constraints by raising the E-constrainable-needed event. The Communication Server will create channels from the P-constrainable-out ports of the constraints to the P-constrainable-in port of the solver. The constraint identifiers needed by the Communication Server to create the channels, are transmitted through the P-commserv-out port.

When the solver has all identifiers, it raises the E-state-needed event to fetch the states of the constrainables. Again, this event is picked up by the Communication Server, which creates channels between the P-state-in port of the solver and P-state-out ports of the constrainables. The identifiers of the constrainables are communicated to the Communication Server through the P-commserv-out port.

When a solver has calculated values that satisfy the constraints, it raises the event E-state-avail and sends the states through its P-state-out port. The Communication Server creates the channels in the same way as when the states were fetched.

If the solver has finished its calculations and there is a connection to the P-param-out port (which is created by the coordinator), the solver will put the results of the calculations on this port.

2.5 Coordinators

A coordinator is concerned with conducting a number of constraint solvers. Any particular coordinator has a specific solving strategy to solve a collection of constraints. For example, there can be a coordinator which guides local propagation, a coordinator for relaxation, one for solving numerical constraints, etc. Each strategy requires a different way of triggering the solvers. The task of the solver coordinator is to trigger the proper solver at the proper time.

A coordinator maintains a data structure in which it stores all constraints and their operands. This data structure, we will call the *constraint network*. From this network, it can derive how constrainables and constraints are interrelated. Furthermore, a coordinator stores for each constraint which solver types can solve the constraint.

A coordinator has the following ports:

- P-constraints-in. Through this input port, the coordinator receives constraint identifiers from another coordinator. There is a guard that posts the event E-constraints-in when data arrives.
- P-constraints-out. Through this output port, the coordinator sends constraint identifiers to a solver or other coordinator that has to solve the constraints. There is no guard on this port, since the coordinator requests the Communication Server to create the channels.
- P-param-in. Through this input port, the coordinator receives additional parameters from a constraint. There is no guard on this port.
- P-param-out. Through this output port, the coordinator sends additional parameters to a solver. Again, there is no guard on this port.
- P-commserv-out. This is the output port for sending constraint and solver identifiers to the Communication Server.

A coordinator reacts to the following events:

- E-violation. Raised by a constraint. When the coordinator detects this event, it will start its specific strategy in order to satisfy all constraints.
- E-constraints-in. This is an internal event, posted by a guard when there is data on P-constraints-in.

A coordinator can raise the following events:

- E-solving-mode. This event is raised in order to put the constrainable entities this coordinator handles in solving mode.
- E-normal-mode. This event is raised in order to put the constrainable entities this coordinator handles in normal mode.
- E-param-needed. This event is raised in order to retrieve data from a constraint or a solver or another coordinator.
- E-param-avail. This event is raised when the coordinator wants to send data to a solver or another coordinator.
- E-constraints-avail. This event is raised when the coordinator wants to send constraint identifiers to a solver or another coordinator.

When there is no constraint solving taking place, a coordinator waits for the events E-violation or E-constraints-in. When the event E-violation is detected, the coordinator will start its specific strategy to solve all the constraints in its constraint network. When the coordinator detects the E-constraints-in event, it will receive constraints on the P-constraints-in port and starts solving this network of constraints.

When a violation event E-violation from a constraint is received, the coordinator will put all constrainables, that are managed by the coordinator in solving mode by raising the event E-solving-mode. Next, it will analyze the network to decide which constraints will be solved by which solvers. A coordinator can also decide to let a part of the network be solved by another coordinator.

The coordinator triggers a solver or coordinator by raising the event E-constraints-avail and sending an entity identification through the P-commserv-out. The Communication Server will then create a channel between port P-constraints-out of the coordinator and P-constraints-in of the entity. If the coordinator also raises the event E-param-avail, a channel will be created between the P-param-in/out ports of both entities.

Finally, the coordinator will raise the event E-param-needed, in order to retrieve results from the solvers and/or coordinators. The channels are created in the similar way as described above. If the coordinator wants to end all constraint solving, it raises the E-normal-mode event, which places all constrainables in normal mode.

2.6 Example

We will now demonstrate the operation of the model by means of an example. In the example, there are two coordinators. One coordinator uses a local propagation algorithm to solve a constraint network. We will call this coordinator, the *lp-coordinator*. The other coordinator solves a constraint network by putting the constraints in a form that can be handled by a numerical solver. We will call this one, the *num-coordinator*.

Suppose, that there is a collection of constrainables, constraints, and the two coordinators, and that the constraint network corresponding to the existing entities is known to the lp-coordinator. Furthermore, we assume that for every constraint entity in the constraint network, there is a solver entity that can calculate a local solution for that constraint and there is a separate solver that uses some numerical technique to solve a set of constraints.

When the state of a constrainable changes, this entity will raise the E-changed event. This event is received by the constraints that operate on this constrainable. (The raised E-changed is picked up by the Communication Server and placed in the event memories of all constraints that operate on the constrainable.)

The triggered constraints will start their validity checking procedure. They will first raise the E-solving-mode event to put their operands in solving mode and lock them from message passing. After that, each constraint will raise the event E-state-needed and send identifiers to its operands through P-commserv-out, in order for the Communication Server to create the channels.

When a constraint has received the object states, it will check its validity. If it is still valid, it will raise the E-normal-mode to unlock the constrainables and no further action is taken. If the constraint is violated, however, it will raise the event E-violation which is picked up by the lp-coordinator.

The lp-coordinator will first raise the event E-solving-mode to put all constrainables in solving mode. Next, for each constraint that raised the E-violation event, the corresponding solver (that can calculate a local solution) has to be triggered. A solver is triggered by a connection to its P-constraints-in port. The coordinator requests the Communication Server to create this channel by raising the E-constraints-avail event and sending the solver identifier through P-commserv-out.

A local solver should leave the state of the changed constrainable untouched. Therefore, the identifier for this constrainable is sent to the solver. The coordinator retrieves this identifier from the constraint that raised E-violation, via the P-param-in port, and sends it to the solver via the P-param-out port (by raising the appropriate events for the Communication Server).

A solver that is triggered reads a constraint identifier from P-constraints-in and one or more constrainable identifiers from P-param-in. Next, it fetches the constrainable identifiers from the constraint (via the E-constrainable-needed event) and subsequently the states of these constrainables (via the E-state-needed event). When the solver has calculated a solution, it assigns the new states to the constrainables (via the E-state-avail event). If the solver could not compute a solution, no assignment to the constrainables is done.

Meanwhile, after having triggered all solvers, the lp-coordinator starts fetching the results of the calculations. A result is obtained via the P-param-out port of a solver and contains the identifiers of the constrainables that the solver changed or, in case no solution was found, an indication that solving failed.

Consulting its internal constraint network, the lp-coordinator (based on the changed constrainables) determines which constraints have to be solved next. It then triggers the local solvers that should calculate a solution for the newly found constraints. These steps are repeated, until no more constraints are determined.

When the lp-coordinator is finished with its algorithm, it raises the E-normal-mode event to put all constrainables back to normal mode.

In general, a local propagation algorithm cannot handle cycles in a network. Local propagation only tries to find local solutions for each constraint, but does not maintain a global overview of the network. As a result, the algorithm can wind up in an endless loop if it continuously tries to find local solutions for constraints in a cycle. A way to avoid this, is to analyze the network and isolating constraints in a cycle, before initiating the local propagation algorithm. These isolated constraints can then be solved in some other way, using more sophisticated solvers.

The lp-coordinator in our example will use the num-coordinator for solving constraints that are in a cycle. Before executing the local propagation algorithm, it analyses the network and marks all constraints that are in a cycle. When, during propagation, a marked constraint is reached, all constraints in the cycle are collected and send to the num-coordinator. The num-coordinator is triggered by the connection to its P-constraints-in port and will subsequently trigger the numerical solver to try and find a solution for the constraints.

The sending of constraints and parameters to the num-coordinator, as well as fetching the results afterwards, is done in the same way as triggering a solver. This implies that for a coordinator, there is no difference in triggering a solver or another coordinator. In this way, hierarchies of coordinators can be easily built and solver and coordinator entities can be interchanged without having to change a coordinator that triggers these entities.

3 Implementation

We have implemented a prototype of the conceptual model. This prototype was built on the language MANIFOLD which enabled us to build an *ideal* implementation. Ideal meant (1) a clear separation between the object oriented world and the constraint world and (2) communication among the entities of the model via events and data flows. Using the language constructs of MANIFOLD (see below), a one-to-one mapping between the conceptual model and implementation could be achieved.

Because focus was aimed at building an ideal implementation in order to study the behaviour of the model, performance issues, such as the efficiency and speed of the system, were not considered during the design process.

In the next section, we will describe MANIFOLD. Section 3.2 presents an overview of the implementation.

3.1 Manifold

MANIFOLD is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes ([AHS93]). The language is based on the IWIM (Idealized Worker Idealized Manager) model of communication ([Arb96]). This model describes a communication protocol which makes a distinction between *worker* processes, i.e. processes that perform a computational task, and *manager* processes, i.e., processes that manage communications. The IWIM model separates computation from communication concerns and establishes that no process is responsible for its own communication with other processes. It is always the task of a manager process to arrange necessary communication among a set of worker processes. Furthermore, a manager process may itself be considered as a worker process by another manager process, which allows hierarchies of communicating processes to be built.

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. A *process* is a black box with well-defined *ports* of connection through which it exchanges units of information with other processes in its environment. Interconnections between ports are made through *channels*, which are communication links that carry units of information. Independent of the communication via channels, there is an *event* mechanism for information exchange in IWIM. Events can be broadcasted into the environment and picked up by any process.

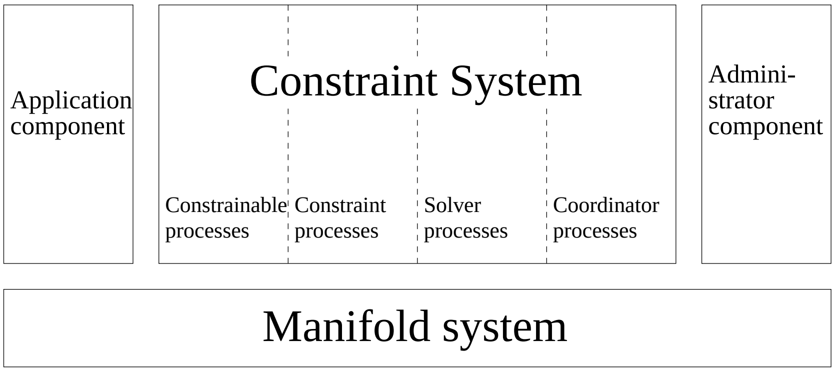

Figure 3: Overview of the implementation.

MANIFOLD is an implementation of the IWIM model and each of the basic concepts of process, event, port, channel in IWIM corresponds to an explicit language construct. Every process has an individual set of ports which serve as connection points for the channels (called *streams* in MANIFOLD). A process can raise events and can also decide for which events it is sensitive.

A manager process, called a *manifold*, is always written in the MANIFOLD language. Worker processes can be written in any other language, such as C, C++, Fortran, etc. Corresponding to the IWIM model, a manager process does not differentiate between worker processes and manager processes. A manager process, written in the MANIFOLD language, has a number of 'states'. A state consists of a *state label* and a *state body*. A state label specifies a pattern of event occurrences that may cause a transition to that state. In the state body, processes and streams can be created and removed. The notions of *manner* (comparable to procedures in conventional languages) and *nested states* provide means to write well-structured code.

3.2 Implementational model

The implementation of the conceptual model is subdivided into four main components (see Figure 3).

- The application component,
- the constraint system,
- the administrator component,
- the MANIFOLD system.

The application component is an object oriented application (OO-application) in which objects communicate with each other via message passing. Some objects, the ones on which constraints should be imposed, are extended with additional functionality, which allows them to communicate with the constraint system (see below). These extended objects are called the *constrainables*. When no constraint solving is at hand, the constrainables behave as ordinary objects. During constraint solving, message passing is locked and communication takes place via events and data flows.

The application component also provides constraint objects. A constraint object specifies a relation among constrainables. All calculations for checking the constraint's validity or constraint solving are done in the constraint system.

The constraint system consists of a collection of **Manifold** processes. Four kinds of processes can be differentiated:

- Constrainable processes,
- Constraint processes,
- Solver processes,
- Coordinator processes.

For each constrainable in the OO-application on which a constraint is imposed, there is a Constrainable process in the constraint system. Similarly, for every constraint in the application there is a Constraint process. There are Solver processes to calculate valid solutions for the constraints and Coordinator processes for guiding the Solvers. Creation and removal of the Constrainable and Constraint processes are dynamically controlled by the OO-application. The management of Solvers and Coordinators (in the current implementation) is done by the *administrator component*.

The administrator component manages all administration concerning creation and removal of Constrainable, Constraint, Solver, and Coordinator processes. This also entails providing some functionality to facilitate the communication between the OO-application and the constraint system.

Finally, the **Manifold** system is the underlying system that manages initialization and termination of the system, all event handling, stream creation and removal, and process creation and removal.

The implementation maps to the conceptual model in the following way. The administrator with the **Manifold** run-time system comprise the Communication Server. The Constraint, Solver, and Coordinator processes of the implementation map directly to the corresponding entities of the conceptual model. The constrainable entity consists of two parts: the part that communicates via message passing is implemented as constrainable object in the application. The part that communicates via events and data flows is implemented as a Constrainable process in the constraint system.

In the next subsections, we will treat each component in more detail. Sections 3.3 and 3.4 present an application that has been built on this model.

3.2.1 Application component

The application is written in an object oriented programming language, in this case C++. There are three C++ classes that enable a programmer to use the constraint system. These classes are `Constrainable`, `Constraint`, and `CommHandler`. Class `Constrainable` provides the functionality for constrainable objects. A constrainable object consists of two parts: the first part is implemented as the C++ object in the application, the second part is the **Manifold** process. During 'normal' operation, the C++ object communicates with other objects via message passing. When constraint solving is at hand, the state of the object is copied to the **Manifold** process which then communicates with other processes via events and data flows.

An application programmer is not concerned with the **MANIFOLD** part of a constrainable object. If a new class has to be created for objects on which constraints will be imposed, this new class has to inherit from `Constrainable`. If there is an already existing class, and a programmer wants to be able to impose constraints on objects of this class, a new class has to be created that inherits from both the existing class and `Constrainable`.

A class inheriting from `Constrainable` has to implement a method `frame_data()` that is required for class `Constrainable` and has to (re)implement methods that must trigger constraint solving. In the method `frame_data()`, messages of super-class `Constrainable` are used to indicate the internal variables that are involved in constraint solving. A method that has to trigger constraint solving must call the message `commit()`. `commit()` is a special message of class `Constrainable` that, when executed, locks the calling object and copies its state to its **MANIFOLD** counterpart.

Class `Constraint` provides the functionality for imposing constraints on constrainable objects. An actual constraint consists of two parts; one part resides as an object in the C++ application and one is a **MANIFOLD** process in the constraint system. The constraints in the application specify the relation among a set of constrainables. When a constraint is specified, the corresponding **MANIFOLD** processes are created in the constraint system. Once these processes are set up, only these are used during constraint solving.

Class `CommHandler` is the class that takes care of the communication between the sequential C++ application and the set of concurrent **MANIFOLD** processes. The application has exactly one object of this class. Created constraints are registered to the `CommHandler` object which communicates them to the constraint system. During run-time, the `CommHandler` object gets control when a new constraint is created or when a constrainable object performs its `commit()` function. When in control, it handles requests from **MANIFOLD** processes. In the case of constraint creation or removal, this involves requesting the constraint object to send its type and operands. In the other case, this involves requesting constrainables to send or receive states.

3.2.2 Constraint system

For each entity that occurs in a component of the conceptual model, there is a **MANIFOLD** process in the implementation. This means that there are constrainable processes, constraint processes, solver processes, and coordinator processes. Constrainable and constraint processes correspond to constrainable and constraint objects in the C++ application. Solver and coordinator processes are instantiated by the administrator component at the initialization of the system.

A constraint process is created as soon as a constraint object is created in the application. At the same time, also the constrainable processes are created that are the operands of the constraint. After successful creation, the newly created constraint processes are registered to the coordinator (by the administrator).

Every process is only sensitive for a specific set of event types. At process creation time, this is initialized by the administrator. A constrainable process only detects events from constraint processes and coordinator processes that operate on it. A constraint process detects events from the constrainables it operates on and the coordinator process that is concerned with solving that constraint. Finally, a coordinator process can detect events from the constraint processes it administers.

The operation of the constraint system is as described in the conceptual model. I.e., when the user of the application changes one of the objects, the corresponding constrainable process in the constraint system raises the E-changed event. This event is picked up by the constraints that operate on the object. Constraints that are violated, raise the E-violation event and in this way trigger the coordinator to start its solving strategy. The coordinator, after having put all constrainables in solving mode via the E-solving-mode event, triggers the solvers or other coordinators. When finished, it raises E-normal-mode to put all constrainables in normal mode and let them proceed their 'ordinary' operation.

A constrainable process in the constraint system is responsible that it has the most recent state of the corresponding constrainable object in the application. Therefore, the constrainable process fetches the state of the object when it has to switch to solving mode (i.e., when it receives the E-solving-mode event). When it switches back to normal mode again, it copies the state back to the object. Since during normal operation, the constrainable process doesn't change and, during constraint solving, the constrainable object is locked, this guarantees that the state of the constrainable entity is always consistent.

3.2.3 Communication Server

The Communication Server, as described in section 2.1, is implemented by the administrator component and MANIFOLD. The administrator performs the administration tasks, such as storage of references to all created processes. MANIFOLD takes care of all the event handling, stream creation and removal, and the actual process creation and removal.

The system is initiated by a small MANIFOLD program which invokes the administrator. The administrator then starts the coordinators, solvers, some MANIFOLD processes that handle communication, and finally the application. The MANIFOLD processes that are activated by the administrator are 'manager' processes (see section 3.1), which take care of the communication with the 'worker' processes that represent the entities of the conceptual model. E.g., there is a manager process that handles communication with the administrator itself, a manager process for the application and a process for the coordinator, and there will be manager processes for the created constraint processes.

When a constraint is declared in the application, the `CommHandler` object raises an event which is picked up by the administrator. The `CommHandler` object then exchanges (via streams) the constraint data to the administrator. The latter orders MANIFOLD to create constraint and constrainable processes. After MANIFOLD has created the processes, it also creates streams from the administrator to the newly created processes in order to initialize them. Finally, the coordinator has to be notified of the newly created constraint. This is done by raising an event, upon which MANIFOLD creates a stream between the administrator and the coordinator, and sending the constraint data through this stream (by the administrator). The same sequence is executed when a constraint is removed in the application.

3.3 Example: local propagation

The application component is an object oriented system, written in C++, for drawing and modifying geometrical objects. A user can draw and modify circles, rectangles, lines, etc. and impose constraints among them, such as touch, equal area, etc. The main activities of the application are the interaction with the user and the administration of the user-created objects.

The network that is formed by the objects and constraints is solved by a coordinator that uses a simple local propagation algorithm, similar to the one described in section 2.6. For each constraint, there is a solver which can compute a local solution for that constraint. In the current implementation, we combined a constraint and its solver into one **Manifold** process, for convenience.

The constraint mechanism is activated by a constrainable that raises the E-changed event. Here upon, violated constraints will raise E-violation, which will trigger the local propagation coordinator to start its solving algorithm. The coordinator will subsequently trigger solvers that compute local solutions. Since constraint and solver are combined in one **Manifold** process, it is not necessary for the coordinator to send a constraint identifier to the solver (indeed, in the current implementation, the local solver is a part of the constraint). The coordinator triggers a solver by sending a constrainable identifier to the entity's P-param-in port.

A local solver computes *a* solution that satisfies the constraint. In case more solutions are possible (e.g., a constraint that specifies that a circle and rectangle should touch, has an infinite number of solutions, even if one of the objects is fixed), the local solver picks one. Currently, the solution chosen obeys the 'principle of least astonishment'. I.e., the solver tries to find a solution that the user would expect. E.g., in case of a touch constraint, the solver calculates the shortest vector between the two constrainables, and translates one of them over this vector. This behaviour could be adapted by sending additional data for the solver through the P-param-in port. E.g., data that indicates that the solver should not take the shortest vector, but should resize the object.

If a solver calculated a solution which validated the constraint, the constrainables are updated by the solver and the identifiers of the changed operands are sent to the coordinator. If a local solver did not change any operands (either because it could not find a solution, or the constraint wasn't violated), a notification is sent to coordinator. In that case, the coordinator will not propagate to constraints that come after the corresponding constraint.

The simple local propagation coordinator cannot handle cycles. In case there are cycles in the network, it can happen that the coordinator winds up in a loop, continuously triggering constraints around the cycle. Endless looping is prevented by marking each constraint that is triggered. If a constraint is marked more than a certain number of times (currently, this number is arbitrarily set to 5), the constraint will not be triggered again.

3.4 Example: cooperating coordinators

An extension of the previous implementation demonstrates the cooperation of two coordinators. Again, the application component consists of a graphical system where geometrical objects can be created, modified, and removed, and in which constraints can be imposed on these objects.

The geometrical objects that occur in this application are points and lines. Constraints among these objects state that one point should intersect another, that a point should intersect a line, lines should be horizontal, a cloud of points should have a certain center of gravity, etc. The common characteristic of all these constraint types is that all constraints can be translated into a set of linear equations.

The network that is formed by a set of constraints and constrainables is now solved by two coordinators. The first coordinator tries to solve the network by local propagation, as described in section 3.3. As was pointed out, this coordinator cannot handle cycles in the network. If it encounters a constraint that is in a cycle, it collects all constraints in that cycle and sends them to the second coordinator. To detect, whether a constraint is in a cycle or not, the local propagation coordinator does a pre-processing step before starting the actual local propagation algorithm. In this pre-processing step, the constraint network is analyzed and for each constraint is indicated whether it is part of a cycle or part of a tree.

The second coordinator can solve constraint networks in which the constraints are represented by linear equations. When solving the network, it first extracts all equations from the constraints and sends all these equations to a numerical solver. This solver uses some numerical method (LU-decomposition), to find a solution for the variables. Again, in the implementation, for convenience, we combined the numerical solver and the coordinator into one **Manifold** process.

If the numerical solver finds a solution for the constrainables, it assigns the new states to them and sends identifiers through the P-param-port to indicate the constrainables that were changed during solving. In case no solution could be found, no assignments are done and the first coordinator is notified that solving failed.

The first, i.e. the local propagation coordinator, after having triggered all solvers and possibly the numerical coordinator, then collects from all triggered entities the constrainables that were changed and continues the algorithm. Finally, the local propagation coordinator ends, when it doesn't find new constraints to be solved.

3.5 Evaluation

The system we implemented is a simple application for manipulating geometrical objects that allows constraints to be imposed on the objects. When an object is adapted, the constraints are solved by local propagation.

The system maintains a strict separation between the C++ application and the constraint framework. In the application, the objects can only communicate with each other via messages. On the other hand, constraint and solver entities have direct access to the object's internal state through the **Manifold** interface. Furthermore, all of the elements of the conceptual model, such as constrainables, constraints, data flows, events, etc., explicitly appear in the implementation.

Having implemented the system, it turns out that a major drawback for using it as an interactive drawing tool is the long delay time needed each time the constraints have to be solved. This delay is caused by the fact that for each constrainable, constraint, and coordinator entity, several **Manifold** processes are instantiated which have to communicate with each other via **Manifold** streams that are constantly being created and removed. These activities produce an overhead which makes the system take several seconds to solve the constraints.

A solution to reduce the overhead can be achieved by reducing the number of independent processes that need to communicate during constraint solving. However, doing this would obscure the 'one-to-one' mapping between the system and the conceptual model. As was indicated before, efficiency concerns were not taken into consideration when the system was designed. Rather, the goal was to study the behaviour of the individual entities of the model and to demonstrate that the proposed model of communication indeed separated the message passing activity from the data flow communication used for solving the constraints. This has successfully been illustrated.

4 Conclusion

In this paper, we presented a conceptual model for combining object oriented programming with constraint programming, and described an implementation. The combination is achieved by separating the object oriented framework from the constraint framework and let the communication be managed by a third party. In the implementation all elements of the model are clearly distinguishable programming constructs, and independent concurrent processes. The communication takes place via events and data flows.

By separating the two programming paradigms, it is possible to maintain the power of both without sacrificing any of the typical characteristics. Furthermore, it provides a controlled way of dealing with the information hiding conflict and isolates the imperative object oriented code from the code that has to maintain the declarative constraints. Separation enables a modular design, which facilitates the design, debugging, modifying, and maintenance of systems. As a result, code can be re-used more easily, allowing for solvers to be re-used and plugged into other systems. Finally, the model offers a scheme which allows for modelling several solvers that operate concurrently.

Currently, research is concerned with implementing the conceptual model into an existing object oriented animation system. Now, the constraint management system that is developed has to meet the high performance requirements of an interactive animation system. Therefore, the ideal implementation in the separate coordination language will be deserted. The implementational model will still show the conceptual elements, but actually the events and data flows are implemented in the language of the animation system.

References

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23 – 70, February 1993.

[Arb96] Farhad Arbab. The iwim model for coordiantion of concurrent activities. In *Coordination Languages and Models*, Lecture Notes in Computer Science, volume 1061, pages 34–56. Springer, 1996.

[CBL91] Eric Cournarie and Michel Beaudouin-Lafon. Alien: a prototype-based constraint system. In Laffra et al. [LBdMP95], pages 92–110.

[Dav91] Jacques Davy. Go, a graphical and interactive C++ toolkit for application data presentation and editing. In *Proceedings 5th Annual Technical Conference on the X Window System*, 1991.

[FBB92] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92–European Conference on Object-Oriented Programming, Utrecht, 1992*, Lecture Notes in Computer Science 615, pages 268–286. Springer-Verlag, 1992.

[GoP93] Bull–Imaging and Office Solutions. *GoPATH 1.2.0 — A Path To Object Oriented*

Graphics, a public domain environment for graphical and interactive application development, 1993.

[HB94] Quinton Hoole and Edwin Blake. OOCS - constraints in an object oriented environment. In *Proceedings 4th Eurographics Workshop on Object-Oriented Graphics, Sintra, Portugal*, pages 215–230, 9 – 11 May 1994.

[LBdMP95] C. Laffra, E. H. Blake, V. de Mey, and X. Pintado, editors. *Object Oriented Programming for Graphics*, Focus on Computer Graphics. Springer, 1995.

[LvdB91] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. *OOPS Messenger*, 2(2):68–72, April 1991.

[Ran91] John R. Rankin. A graphics object oriented constraint solver. In Laffra et al. [LBdMP95], pages 71–91.

[Sut63] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference, Detroit, Michigan, May 21-23 1963*, pages 329–345. AFIPS Press, 1963.

[VK95] Remco C. Veltkamp and Richard H. M. C. Kelleners. Information hiding and the complexity of constraint satisfaction. In *Remco C. Veltkamp and Edwin H. Blake (eds), Programming Paradigms in Graphics*, pages 49–66. Springer-Verlag, ISBN 3-211-82788-9, 1995.

[Wil91] Michael Wilk. Equate: an object-oriented constraint solver. In *Proceedings OOPSLA'91*, pages 286–298, 1991.