A Constraint-Based Framework for Prototyping Distributed Virtual Applications

Vineet Gupta[1], Lalita Jategaonkar Jagadeesan[2], Radha Jagadeesan[3], Xiaowei Jiang[1], and Konstantin Läufer[3⋆]

[1] PurpleYogi.com, 201 Ravendale, Mountain View, CA 94043
{vineet,xjiang}@purpleyogi.com
[2] Software Production Research Dept., Bell Laboratories, Lucent Technologies
263 Shuman Blvd., Naperville, IL 60566
lalita@research.bell-labs.com
[3] Dept. of Mathematical and Computer Sciences, Loyola University Chicago
6525 N. Sheridan Road, Chicago, IL 60626
{radha, laufer}@cs.luc.edu

Abstract. This paper describes the architecture and implementation of a constraint-based framework for rapid prototyping of distributed applications such as virtual simulations, collaborations and games. Our framework integrates three components based on (concurrent) constraint programming ideas: (1) `Hybrid cc`, a (concurrent) constraint modeling language for hybrid systems, (2) `Sisl`, a (discrete) timed constraint language for describing interactive services with flexible user interfaces and (3) `Triveni`, a process-algebraic language for concurrent programming. The framework is realized as a collection of tools implemented in Java. The utility of the ideas are illustrated by sketching the implementations of simple distributed applications.

1 Introduction

The focus of this paper is rapid prototyping in the domain of systems that include hybrid components, concurrency and reactivity, (virtual/code) mobility and distribution. The following systems exemplify the applications of interest:

- Consider the computer simulation aspects of NASA's Airport **S**urface **D**evelopment and **T**est **F**acility (see http://sdtf.arc.nasa.gov/sdtf), an airport operations simulator. A typical virtual simulation in such a context involves large numbers of planes in large sections of airspace around an airport.
- Consider the emerging area of distributed collaborative applications. In their simplest forms (Instant Messaging, MSN Messenger Service, ICQ etc.), this consists of contact/buddy lists and automatic notification of presence of contacts and so on. In more sophisticated virtual world scenarios, e.g., Gelernter's vision of cyberbodies and lifestreams [19], this idea is generalized to mobile and distributed repositories of information called cyberbodies. Chronological streams are the most common kind of cyberbody, since time and causality are natural ways to organize

⋆ R. Jagadeesan, X. Jiang and K. Läufer were supported in part by a grant from NSF.

information. The most common computational task is the exchange of information between different such streams, e.g., exchange credit card information between a shopper cyberbody and a bank cyberbody.

Both of these examples have the four conceptual components: (1) hybrid systems, (2) concurrency, (3) interaction via flexible user interfaces and (4) mobility and distribution.

Hybrid systems: Reactive systems react with their environment at a rate controlled by the environment. In each phase, the environment stimulates the system with an input and obtains a response within a bounded amount of time. Continuous systems, such as mechanical and physical systems, are those in which the system has the potential of evolving autonomously and continuously. The description of this behavior is usually in the form of differential equations that arise naturally in the description and modeling of the behavior of physical systems. In contrast to the discrete notion of time in reactive systems, the appropriate notion of time in continuous systems is dense, i.e. the rationals or the reals. Complex applications are *hybrid systems* that combine both of these ideas. Consider for example the model of the dynamics of the airplanes. It is a reactive system responding to inputs from the sensors, the pilot program etc. Each plane is modeled as an object with dynamics given by differential equations based on the physics of flight. There are discrete changes in the motion of the plane based on inputs from the pilots. Thus, the execution of the plane simulation alternates between open intervals in which the state of the plane (e.g., the position or velocity) changes continuously in a manner prescribed by the laws of flight, and points at which discontinuous change can occur, such as when the pilots take an action. Similarly, the evolution of a lifestream is naturally modeled as a hybrid system. In general, precise virtual simulation of physical artifacts in collaborative spaces naturally leads to hybrid systems.

Concurrency: Concurrency is omnipresent in the above collection of examples. Concurrency arises in two ways. Firstly, it arises in an intrinsic way because of the modeling of several independent activities, e.g., several aspects of the model of an airplane, several virtual participants in a collaborative discussion. Secondly, it arises as an abstraction mechanism useful in the implementation of responsive user interfaces.

Interaction via flexible user interfaces: Modern interactive services are becoming increasingly more flexible in the user interfaces they support. These interfaces incorporate automatic speech recognition (ASR) and natural language understanding, and include graphical user interfaces on the desktop and web-based interfaces using applets and HTML forms. The key role of flexible and varied user interfaces in our target application area is evident.

Mobility and distribution: In the collaboration and lifestreams examples, distribution and mobility issues arise naturally in the context of geographically spread-out groups of participants. In the airplane simulation, the potential for large number of airplanes in large sections of airspace mandates the modular organization of the objects of the simulation into "logical locations", also termed "ambients" [8, 18]. For example, there are ambients for each airplane object and for each section of the physical airspace. The

computations of each ambient are performed by (a collection of) computing nodes, and the physical motion of the airplanes naturally leads to considerations of mobility in the simulation.

1.1 Current Programming Practice and Shortcomings

The class of applications in which we are interested is implemented using concurrency, say in the form of threads, and some form of distributed programming. For concreteness, we phrase the following discussion about current programming practice in terms of the Java language; however, we note that the Java language largely reflects current popular practice. Monitors guard shared memory between threads, and these monitors cause additional indirect communication between threads; threads can wait for access to a monitored region, yield control of the monitor, and notify other threads waiting on the monitor. Threads are equipped with priorities that facilitate scheduling of threads. Finally, the thread groups of Java provide rudimentary structuring facilities for building collections of threads that are controlled in unison. Java supports two different mechanisms for distribution: RMI (remote method invocation) is Java's RPC (remote procedure call) mechanism, integrated with Java's object-oriented approach; the Servlet API provides infrastructure for the special case of client-server programming, especially when the communication (between client and server) is handled via web-based infrastructure such as the hypertext transfer protocol (HTTP). These approaches suffer from the following shortcomings.

- The model lacks support for a modularity notion of abstract *behavior*, which in the systems of interest is essentially the interaction of the system with its environment. For example, suppose one is given a computer model A of an airplane that emits position readings, `Pos(t)` events, from time to time (perhaps among others), and accepts `Turn(dir)` requests (perhaps among others). Consider the task of implementing a controller for the plane, e.g., to accept `Pos(t)` events and emit `Turn(dir)` events when appropriate. Furthermore, we would like to *compose* our controller with A to yield an activity *Controlled*A that internalizes the `Turn(dir)` event.
  For modularity reasons, we would like to design and implement the controller knowing only the interface of A described above, without access to the implementation fact that a behavior could be realized by one or more threads, potentially based at several processors. The operations in Java, as briefly described above, do not *directly* support this kind of software design.
- The model fails to provide coherence between the concurrent and distributed aspects. For instance, a user of a system cannot reason about system properties (such as safety specifications and deadlock behavior) independently of the nature of distribution of the system (such as the location and communication of the individual planes A and the other components of the system). We are not demanding uniformity of performance across different kinds of distribution; we are merely demanding a uniform view of distribution that does not violate the semantics of the concurrent aspects of the programming language.
  However, the operations in Java expose the programmer to the *details* of the distribution of the program components. For instance, it is possible that a Java program

involving threads and RMI will be deadlock-free if the processors reside on the same file system, but will deadlock if the processors reside on different file systems.
– Finally, the two different mechanisms for distribution in Java, namely RMI and Servlets, are not presented in a unifying distributed programming model. There is no direct support for parameterizing the basic program logic over the mode of distribution. This in turn leads to duplication of effort/code and serious problems in software maintenance.

1.2 Our Approach

In our prior work, we have explored declarative programming languages, inspired by the constraint programming paradigm, targeted at some of these application areas.

– `Hybrid cc` [26, 25, 24] is an executable specification/programming language for hybrid systems in the concurrent constraint programming framework [41]. `Hybrid cc` can be viewed as a declarative high-level programming notation for the appropriate operational model for this context, namely hybrid automata [2].
– `Triveni` [12–14] is a programming methodology for concurrent programming with threads and events. `Triveni` has its operational basis in formalisms from concurrency theory, such as process algebras [37, 31] and synchronous programming languages [6, 30, 23, 29, 42]. The logical semantics of `Triveni` permits viewing `Triveni` programs as formulas in a fragment of linear-time temporal logic.
– `Sisl` (Several interfaces, single logic) [3] is a deterministic constraint language [32] for the description of interactive services with multiple user interfaces. `Sisl` utilizes its constraint programming foundations to allow users considerable flexibility in the way they input their requests to such services.

In this paper, we present an architecture that combines these components smoothly in a coherent framework for prototyping distributed applications of the kind described earlier. This integration yields the following advantages with respect to the current (pragmatic) state of the art.

Concurrent composition as a first class primitive. Our approach takes the point of view of formalisms from concurrency theory, such as process algebras and concurrent constraint programming — these formalisms are designed from the start around behaviors, termed *processes*. These paradigms then describe a algebra of processes in which, for instance, the concurrent composition of two processes yields a process. Extending the viewpoint of `Triveni` to distributed situations, combinators in our framework operate on behaviors and the result of the combinators are behaviors: the implementation yields the correct combination of behaviors. Thus, our framework enables concurrent composition to be used freely for the modular decomposition of designs. The concurrent composition of programs yields programs that are indistinguishable from simple ones (in much the same way that a complex function in functional programming has the same status as a simple function). The correct dispatch of events sent by concurrent components is done automatically by our framework, and thus, the implementation of a program can closely reflect its design.

Parameterization over distribution. With regard to distribution issues, our architecture provides a framework to describe the essential features of the program logic without specifying the distribution mechanism. Thus, in our framework, we can informally think of a program P as the concurrent composition of its logic $P_{\texttt{logic}}$ and its distribution mode $P_{\texttt{dist}}$. Thus, the distribution becomes a "pluggable" parameter to the program and enables the sharing of the program logic across different distribution mechanisms. We re-emphasize that our aim is not to ensure identical performance or even behavior across different distribution mechanisms. Rather, our approach enables us to achieve:

- Reuse of the extensive work done in the protocols and frameworks for distributed communication, such as Ensemble [40], which provide the ability to combine modularly basic protocol layers that implement simple properties.
- Parametric, with respect to distribution mode, reasoning about program logic by providing a clear "slot" for assumptions about the guarantees (e.g., total order or causal broadcast) provided by the distribution model.

Enlarging domain of applicability of `Hybrid cc`. `Hybrid cc` is a synchronous programming language that operates on a "global clock" assumption. This tunes it for modeling systems of tightly coupled hybrid components where subcomponents evolve at approximately similar rates. Thus, the following desiderata remain.

- There is tremendous overhead in using the models constructed by `Hybrid cc` as subcomponents in a general asynchronous environment. For example, in our work on the model of the Sprint AERCam [1], the model is interfaced with an animation interface to allow a user to interact with the model as it evolves. This interaction was achieved by traditional concurrent programming, and the synchronization code consumed almost half the total time spent in implementation.
- Secondly, we have observed (e.g., [1]) that the synchronous hypothesis of `Hybrid cc` leads to inefficiencies when the model has several loosely coupled hybrid subcomponents. For instance, in the airport simulations, each airplane mainly interacts with nearby airplanes, thus any discrete actions it takes affect only nearby planes. However the synchronicity assumption underlying `Hybrid cc` forces each such discrete action to cause a *global* synchronization point involving all the objects in the simulation, including those that are not affected by the action.

The framework of this paper provides a structured methodology to arrange the interaction of `Hybrid cc` components in a general context of asynchrony and distribution.

1.3 Rest of the Paper

The rest of the paper is organized as follows. In Section 2, we review our earlier work on `Hybrid cc`, `Triveni` and `Sisl`, and we present a comparison with related work. In the following section, we describe the architecture and give an overview of its current implementation. We illustrate the ideas in the context of some concrete examples that have been realized using the framework: an n-player variation of the board game Battleship (using two different distribution mechanisms), as well as a multi-user instant messaging application.

2 Background

2.1 Hybrid cc

Hybrid cc is an executable specification/programming language for hybrid systems [26] in the concurrent constraint programming framework [41]. Hybrid cc can be viewed as a high-level programming notation for hybrid automata [2], much as synchronous programming languages are high level notation for discrete automata [1].

Hybrid cc incorporates two key ideas — (1) Continuous constraint systems (ccs) and (2) extending (concurrent) constraint programming over (real) time. We sketch these ideas here, referring the reader to [26] for a precise foundational description. Intuitively, continuous constraint systems express the information content of initial value problems in integration. Continuous constraint systems support an integration operation, $\int^r \texttt{init}(a) \wedge \texttt{cont}(b)$ which determines the effect of b holding continuously in the interval $(0, r)$, if a held at time 0. For example, $\int^7 \texttt{init}(x = 3) \wedge \texttt{cont}(x' = 4) \vdash x = 31$. [26] describes a set of axioms for continuous constraint systems — these include intuitive properties of integration such as the monotonicity and continuity of integration, and some computability axioms to enable finite description and implementation. We add a single temporal control to the untimed (concurrent) constraint programming: hence A. Declaratively, hence A imposes the constraints of A at every time instant after the current one. [26] shows how hence can be combined in very powerful ways with ask operations to yield rich patterns of temporal evolution, e.g., do A watching P — execute A at every time point beyond the current one until the first time instant at which P is true, assuming that there is, in fact, a first time instant at which P is true.

While conceptually simple to understand, hence A requires the execution of A at every subsequent real time instant. Hybrid cc is made computationally realizable by exploiting the basic intuition we exploit is that, in general, physical systems change slowly, with points of discontinuous change, followed by periods of continuous evolution. Computation at a time point establishes the constraint in effect at that instant, and sets up the program to execute subsequently. Computation in the succeeding open interval determines the length of the interval r and the constraint whose continuous evolution over $(0, r)$ describes the state of the system over $(0, r)$. We recall:

- Hybrid cc is declarative [26] — programs can be understood as formulas that place constraints on the (temporal) evolution of the system, with concurrent composition regarded as conjunction.
- Hybrid cc is amenable to the tools developed for the verification of hybrid systems [24] — for any Hybrid cc program, there is a hybrid automaton whose valid runs are precisely execution traces of the program; and for any given safety property expressed in (real-time) temporal logic, there is a Hybrid cc program that "detects" if the property is violated.
- We have implemented Hybrid cc [9, 25] and used this implementation for several examples, e.g., an (executable) model of the paper path of a photocopier [27], an (executable) model and controller for a robotic camera of the Space Shuttle [1].

[1] **F**unctional **R**eactive **AN**imation [17], built on functional programming, has similar goals.

2.2 Triveni

Triveni [12–14] is a programming methodology for concurrent programming with threads and events. Triveni has its basis in formalisms from concurrency theory, such as process algebras [37, 31, 38] and synchronous programming languages [6, 28, 30, 42]. Communication is via broadcast of (labeled) events that are abstractions of names of communication channels. In addition to the usual process-algebraic combinators of event emission, concurrent composition and waiting for events, Triveni supports exceptions via preemption combinators in the style of synchronous programming languages.

The semantic study of Triveni [14] includes an operational semantics that includes a precise formalization of the fairness assumptions of the current implementation, and a denotational semantics based on (fair) traces. Furthermore, the logical semantics of Triveni proceeds via a compilation of the Triveni control combinators as constructions on Buchi automata. In the light of the automata-theoretic approach to temporal logic [44], this shows that Triveni programs constitute a carefully chosen fragment of linear time (temporal) logic [35].

The current implementation of Triveni (in Java) makes Triveni compatible with existing threads standards such as P-threads and Java threads, and includes an integrated specification-based testing environment that automates the testing of safety properties. We have used this implementation to perform a case study from telecommunication [13], to prototype a domain specific language for writing flexible interactive services [3], and in the classroom as an environment and tool for teaching the rudiments of designing, implementing, and reasoning about concurrent programs [15].

2.3 Sisl

The context of Sisl is modern interactive services. It is now common for such services to have more than one interface for accessing the same data, e.g., personal banking services from an automated teller machine, a bank-by-phone interface, or a web-based interface. Furthermore, telephone-based services are starting to support automatic speech recognition and natural language understanding. In this context, it is desirable for the programming methodology to provide the following capabilities:

- allowing requests to be phrased in various ways (e.g., needed information can be provided in any order),
- prompting for missing information,
- lookahead (to allow the user to speak several commands at once), and
- backtracking to earlier points in the service logic.

In Sisl, the service logic (i.e., the code that defines the essence of the service) of an application is specified as a *reactive constraint graph*, which is a directed acyclic graph with an enriched structure on the nodes. The traversal of reactive constraint graphs is driven by the reception of events from the environment: these events have an associated label (the event name) and may carry associated data. In response, the graph traverses its nodes and executes actions; the reaction of the graph ends when it needs to wait for the next event to be sent by the environment.

The key kind of node is $\texttt{constraint}(P_{\texttt{next}}, \langle(\phi_i, \sigma_{\phi_i}, P^i_{\texttt{viol}})\rangle i = 1 \ldots n)$, where the ϕ_i are predicates on events. Intuitively a constraint node is awaiting *all* the events in $\cup_i \sigma(\phi_i)$. These events can be sent to the constraint node in any order. When control reaches the constraint node, the `Sisl` service logic automatically sends out a prompt event for every event that is still needed in order to evaluate some constraint. In addition, it automatically sends out a optional prompt for all other events mentioned in the constraints — these correspond to information that can still be corrected by the user. In every round of interaction, the constraint node waits for the user to send any event that is mentioned in its associated predicates. Each predicate associated with a constraint node is evaluated as soon as all of its events have arrived. If an event is re-sent by the user interfaces (i.e, information is corrected), all predicates with that event in their signature are re-evaluated.

`Sisl` is implemented as a library in Java and supports the development of services that are shared by multiple user interfaces including automatic speech recognition-based or text-based natural language understanding, telephone voice access, web, and graphics-based interfaces. `Sisl` has been integrated with VeriSoft [20], a systematic state-space exploration tool, and hence supports automated and efficient testing of `Sisl` applications [21]. `Sisl` is currently being used to prototype a new generation of call-processing services for a Lucent Technologies switching product.

2.4 Related Work

We have already referred to several pieces of work that have inspired and influenced this paper. [43] is an eminently readable survey of concurrent logic programming languages. This line of work has now developed into extensive work on temporal logic programming languages with perhaps some notions of distribution (e.g., [7, 4, 5, 39, 36, 22]). Our work differs from this literature in that our approach has tended to emphasize the reuse of the extensive existing work in the design, implementation and analysis of (concurrent) programming languages. For example, the compatibility of our work with existing threads standards and event models, such as P-threads and JavaBeans, has enabled us to easily use our languages and framework in the context of concrete applications. More substantially, our methodology is significantly influenced by ideas from process algebras and synchronous programming languages. This is revealed in the explicit treatment of operational notions such as fairness in our framework, and has permitted our study to be compatible with the extensive analysis methodologies/tools developed for testing and verifying concurrent systems in this context, such as computer-aided verification via model checking (e.g., [20, 11, 34, 10] to name but a few) and specification-based testing of temporal properties (e.g., [16]). Indeed, both `Triveni` and `Sisl` support a systematic and efficient testing architecture based on these methods.

3 Examples

In this section, we discuss two examples with respect to their hybrid, reactive, and distributed characteristics. `MyMessaging` is an multi-user instant messaging application. This system has has rich multi-modal reactive and distributed behavior. `Battle`,

an n-player variation of the 2-player board game Battleship, is a modified version of an example from [13]. This game has a significant hybrid component to model the motion of the ships, in addition to the evident reactive and distributed behavior. We have used both examples as projects in courses.

Contact List. Each user has a contact list ("buddy list") of people with whom he or she usually communicates. The contact list can be managed by adding or removing users and organizing them by category. The contact list can be viewed in different ways, including by current online status.
Online Status. Each user can change between different online statuses: online, do-not-disturb, offline, etc.
Messaging. A user can send different types of messages to one or more users. Message types include text messages, files or URLs, and chat invitations.

Fig. 1. Features of `MyMessaging`

Loss. A player loses when all his/her ships are destroyed.
Oceans. Each player has a collection of ships on an individual ocean grid. The n ocean grids are disjoint. Each player's screen displays all n oceans, but a player can see only his/her own ships. A player's ships are confined to the player's ocean. Each ocean has a surface current that causes its ships to drift in the direction of the current.
Ships. Each ship occupies a rectangular sub-grid of the player's ocean and sinks after each point in its grid area has been hit. Ships can move on the surface of the player's ocean. Once set in motion, a ship moves along a straight line that factors in the surface current. If a ship hits an edge of the ocean, it bounces back. If it collides with another ship, the usual conservation of energy and momentum laws apply.
Moves. A player can move as fast as the user-interface/reflexes allow. Player i can make 3 kinds of moves:

1. Fire a round of ammunition on a square of another player j's ocean by clicking on it. The ammunition may hit a previously unmarked point on one of player j's ships, in which case a mark is displayed at that point in player j's ocean on all players' screens. No information is reported in case of a miss.
2. Impart a velocity to a battleship that lasts until it receives another velocity command or until it collides with an edge or another battleship.
3. Raise a shield over his/her entire ocean for a game-specific interval of time, during which player i's ships are invulnerable. When a player raises an ocean-wide shield, his/her ocean becomes dim on the screens of all players. Each player has a limited supply of shields.

Fig. 2. Features of `Battle`

4 Architecture

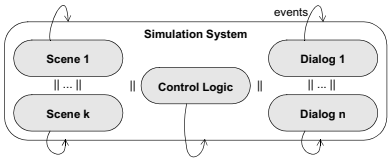
This section describes the architecture of our framework and its use of multiple interacting constraint-based paradigms. The framework exploits the capabilities of each paradigm and decouples application-specific from application-independent aspects.

Building upon existing implementations of `Hybrid cc` (21,000 lines of C code and 1000 lines of Yacc code; used from within Java through the Java Native Interface (JNI)), `Triveni` (7,000 lines of Java code), and `Sisl` (5,000 lines of Java code), the framework consists of about 1,900 lines of application-independent Java code; this includes support for both RMI-based and servlet-based physical distribution. The prototype implementation of `Battle` contains about 2,200 lines of application-specific new code: 700 lines of `Triveni` code for the control logic, 1500 lines of Java code for user interface and animation, 100 lines of `Hybrid cc` code for the physical model, and 400 lines of Java code for hooking the application together with the servlet-based distribution architecture. The prototype implementation of `MyMessaging` contains about 400 lines of application-specific new Java code: 300 lines for the control logic and support classes, and 100 lines for the user interface.

In the remainder of this section, the figures depicting the various aspects of the architecture use shaded shapes for application-specific components and unfilled shapes for application-independent components.

4.1 Logical Architecture

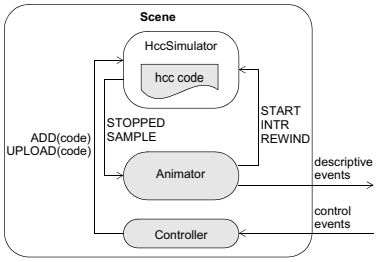
We first consider the logical architecture of a virtual simulation. A simulation consists of zero or more scenes, zero or more dialogs, and a control logic. These components coexist using `Triveni`'s concurrent composition combinator (`||`) and communicate through logical events; the simulation system constitutes the environment in which these events are visible.



- A scene models a collection of closely interacting physical objects. A scene occasionally emits events that describe the scene by providing information such as the position, speed, direction, or other attributes of certain objects. A scene responds to incoming events that control the objects in the scene.
- A dialog provides interaction with outside systems such as users. A dialog receives input from outside systems and emits corresponding events into the simulation. A dialog receives response events from the simulation and transmits the corresponding information back to the outside system.
- The control logic specifies how the scenes and dialogs of the simulation interact.

Scene. A simulation scene has three interacting components. Concurrency and communication between these components is again managed through `Triveni`:

– The application-independent *simulator* simulates a `Hybrid cc` program and communicates with the other components via events and methods. The `Hybrid cc` program provides the physical model for the objects in this scene.
  The `Hybrid cc` simulator instance is wrapped inside a suitable `Triveni` component and can be controlled as needed via the START, INTR, and REWIND events. The ADD and UPLOAD events are used to make changes to the `Hybrid cc` program; this capability is necessary for controlling the physical model from the outside.
  The simulator occasionally emits events that inform the other components about the state of the simulation. SAMPLE is emitted whenever a data sample from the simulation is available. STOPPED is emitted in response to an incoming INTR event to indicate that the simulator has in fact stopped.
– The *controller* is responsible for converting logical events to pieces of `Hybrid cc` code that are uploaded to the simulator for controlling the scene.
– The *animator* is responsible for converting data samples into logically meaningful events that describe the status of the scene.



Example 1. In `Battle`, the `Hybrid cc` program provides physical modeling of the ships in a single ocean using Newtonian laws of motion in the form of differential equations. For instance, the fragment that governs collisions between ships and edges looks as follows:

```
Edges = () {
    always forall Battleship(X) do {
        if (X.px = hw || X.px = xMax - hw) then {
            XEdgeCollision,
            X.ChangeX, X.vx = - prev(X.vx)
        },
        if (X.py = hh || X.py = yMax - hh) then {
            YEdgeCollision,
            X.ChangeY, X.vy = - prev(X.vy)
        }}}
```

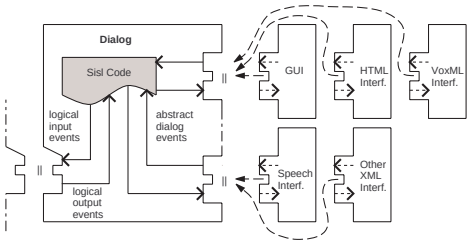
The controller converts move events coming from the player to the corresponding changes in the `Hybrid cc` program. The animator simply renders ships on the screen using the Java 2d package.

Dialog. A dialog has two components:

– The application-independent *pluggable* external user interfaces that convert back and forth between concrete input and output events and the corresponding abstract dialog events. In a model-view-controller architecture [33], these components can be viewed as the view/controller pairs of the interaction.

– The application-specific `Sisl` code that describes the logical interaction with an outside system. In a model-view-controller architecture, this component can be viewed as the model of the interaction. The `Sisl` code is responsible for mediating between abstract dialog events and logical simulation events.

`Sisl` provides the ability to switch between or combine multiple concrete user interfaces without changing the abstract dialog. Examples of such interfaces include graphical user interfaces, speech recognition and synthesis, web browsers using applets or HTML, voice browsers using VoiceXML, and other XML-based interfaces.



Example 2. In `Battle`, the player can interact with ships by clicking on or dragging them with the mouse. Alternatively, the player can use speech to cause ships to move or to shield his/her ocean.

Control Logic. The control logic mediates between the scenes and the dialogs via events and is provided as a `Triveni` component.

Example 3. In `Battle`, the control logic is responsible for a number of tasks, including enforcement of the rules of the game. The following code defines the top-level logic for one player.

```
   ReadyAbortButton
|| await Ready -> Shield(container, numOfShields, shieldDuration)
|| await Ready -> suspend Shield [playerOcean] resume Unshield
|| await Ready_0 -> OpponentOcean(0)
   ... (except this player's ocean)
|| await Ready_n -> OpponentOcean(n = maxNumOfPlayers-1)
```

The following code fragment ensures that a player can use a shield only as long as shields are still available.

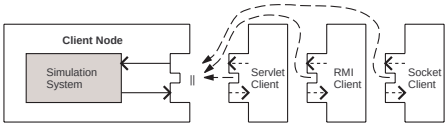
```
Shield(numOfShields, shieldDuration) =
    local OutOfShields in
        do shieldBtn Watching OutOfShields
     || loop
            await Shield -> if (--numOfShields == 0) emit OutOfShields
     || rename Start, Finish to Shield, Unshield in Act(Timer(duration))
```

4.2 Distribution — Logical and Physical

As stated above, the components of a simulation system communicate through logical events. There are two conceptually different degrees of communication coupling between components. Communication between components on a single node is *synchronizing*, that is, components can wait until an emitted event has been received by all

other components in the sub-system — all communication inside `Hybrid cc` and a significant portion of the communication inside `Triveni` is of this kind. On the other hand, communication between components on two distinct nodes is is better carried out asynchronously. Components that require synchronizing communication logically belong to the same subsystem, whereas components that do not require tight coupling logically belong to distinct subsystems. The logical distribution is a partitioning of scenes, dialogs, and control logic into logical nodes and involves breaking the control logic up into suitable pieces — this process is facilitated by the explicit concurrent/parallel composition combinator supported in our framework.

Consider now the physical architecture of a virtual simulation, which allows distribution over (possibly) multiple physical nodes. In our architecture, communication between nodes is provided in a way that is completely transparent to the local simulations. This allows us to decouple the logical architecture of the simulation system from its physical distribution architecture and topology. Concretely, on each node, the local simulation system is paired with a communication component that is responsible for transmitting nonlocal events between the local simulation and other nodes. The remote client hides the details of the specific distribution mechanism used. The pairing between the simulation and communication components occurs as concurrent composition at the `Triveni` level. Thus, the physical distribution architecture becomes a *pluggable* parameter of the former in the following two senses.



- Firstly, it is possible to switch between distribution mechanisms without making changes to the simulation system. For example, we support the following distribution architectures: applet/servlet, remote method invocation (RMI), and sockets.
- Secondly, mechanisms that guarantee reliable communication over asynchronous networks, particularly those based on protocol stacks such as Ensemble [40], are naturally incorporated in this architecture as layers around the communication components.

Example 4. In `Battle`, each logical node consists of a single scene for modeling the player's ocean, a single dialog for interaction with this player, and the control logic. In the RMI-based implementation, the physical nodes use RMI for communication and are arranged in a clique topology with a central server for initial client registration. In the servlet-based implementation, the physical nodes use HTTP and sockets for communication and are arranged in a star topology around a central server for client registration and routing.

References

1. L. Alenius and V. Gupta. Modeling an AERCam: A case study in modeling with concurrent constraint languages. In *CP'98 Workshop on Modeling and Computation in the Concurrent Constraint Languages*, October 1998.

2. R. Alur, C. Coucoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
3. T. Ball, C. Colby, P. Danielsen, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, P. Mataga, and K. Rehor. Sisl: Several interfaces, single logic. *Intl. J. of Speech Technology*, 2000. Kluwer Academic Publishers, to appear.
4. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: A framework for programming in temporal logic. In J. W. de Bakker, W. P. de Roevere, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems– Models, Formalisms, Correctness*. Springer- Verlag, 1990. LNCS 430.
5. M. Baudinet. Temporal logic programming is complete and expressive. In *Proc. ACM Symp. on Principles of Programming Languages*, 1989.
6. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Special Issue on Another Look at Real-time Systems*, Proc. IEEE, September 1991.
7. C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In V. A. Saraswat and K. Ueda, editors, *Logic Programming: Proc. 1991 Intl. Symp.*, pages 661 – 677, 1991.
8. L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, LNCS 1378, 1998.
9. B. Carlson and V. Gupta. Hybrid CC and interval constraints. In T. A. Henzinger and S. Sastry, editors, *Hybrid Systems 98: Computation and Control*, LNCS 1386. Springer Verlag, April 1998.
10. E. M. Clarke and R. P. Kurshan. Computer-Aided Verification. *IEEE Spectrum* **33***(6)*, pages 61–67, (1996).
11. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Prog. Lang. and Systems*, 15(1), 1993.
12. C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Design and implementation of Triveni: A process-algebraic API for threads + events. In *Proc. IEEE Intl. Conf. on Computer Languages*. IEEE Computer Press, 1998.
13. C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Objects and concurrency in Triveni: A telecommunication case study in Java. In *Proc. Fourth USENIX Conf. on Object Oriented Technologies and Systems*, 1998.
14. C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Semantics of Triveni: A process-algebraic API for threads + events. *Electronic Notes in Theoretical Computer Science*, 14, 1999.
15. C. Colby, R. Jagadeesan, K. Läufer, and C. Sekharan. Interaction, concurrency, and oop in the curriculum: a sophomore course. In *Proc. 1998 OOPSLA Educators Workshop*, 1998.
16. L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, 19(5):140–153, December 1994. Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering.
17. C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Intl. Conf. on Functional Programming*, 1997.
18. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th Intl. Conf. on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.
19. D. Gelernter. Now that the PC is dead. *Wall Street Journal*, Jan 1 2000. http://www.mirrorworlds.com/.
20. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symp. on Principles of Programming Languages*, pages 174–186, 1997.

21. P. Godefroid, L. J. Jagadeesan, R. Jagadeesan, and K. Läufer. Automated systematic testing for constraint-based interactive services. In *Proc. 8th Intl. Symp. on the Foundations of Software Engineering*, November 2000. To appear.
22. S. Gregory. A declarative approach to concurrent programming. In *Proc. 9th Intl. Symp. on Programming Languages, Implementations, Logics, and Programs*, 1997.
23. P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In *Special Issue on Another Look at Real-time Systems*, Proc. IEEE, September 1991.
24. V. Gupta, R. Jagadeesan, and V. Saraswat. Hybrid cc, hybrid automata, and program verification. *LNCS 1066*, 1996.
25. V. Gupta, R. Jagadeesan, V. Saraswat, and D. G. Bobrow. Programming in hybrid constraint languages. *LNCS 999*, 1995.
26. V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30((1,2)):3–49, 1998.
27. Vineet Gupta, Vijay Saraswat, and Peter Struss. A model of a photocopier paper path. In *Proc. 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.
28. N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.
29. N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special Issue on Another Look at Real-time Systems*, Proc. IEEE, September 1991.
30. D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.
31. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
32. J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proc. 14th Annual ACM Symp. on Principles of Programming Languages*, 1987.
33. G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface. *J. Object-Oriented Programming*, 1(3):26–49, August/September 1988.
34. R. P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton U. Press, 1994.
35. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. 427 pp.
36. R. Merz. Efficiently executing temporal logic programs. In M. Fisher and R. Owens, editors, *Proc. of IJCAI*, 1993.
37. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
38. R. Milner, J. Parrow, and D. Walker. Mobile processes. Technical report, U. Edinburgh, 1989.
39. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge Univ. Press, 1986.
40. R. V. Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.
41. V. A. Saraswat. *Concurrent Constraint Programming*. Logic Programming and Doctoral Dissertation Award Series. MIT Press, March 1993.
42. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *J. Symbolic Computation*, 22:475–520, 1996. Extended abstract appeared in the *Proc. 22nd ACM Symp. on Principles of Programming Languages*, San Francisco, January 1995.
43. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
44. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.