InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming [⋆]

Dan Barak, David Harel and Rami Marelly

The Weizmann Institute of Science, Rehovot, Israel

Abstract. We describe **InterPlay**, a simulation engine coordinator that supports cooperation and interaction of multiple simulation and execution tools, thus helping to scale-up the design and development cycle of reactive systems. InterPlay involves two main ideas. In the first, we concentrate on the inter-object design approach involving LSCs and the Play-Engine tool, enabling multiple Play-Engines to run in cooperation. This makes possible the distributed design of large-scale systems by different teams, as well as the refinement of parts of a system using different Play-Engines. The second idea concerns combining the inter-object approach with the more conventional intra-object approach, involving, for example, statecharts and Rhapsody. InterPlay makes it possible to run the Play-Engine in cooperation with Rhapsody, and is very useful when some system objects have clear and distinct internal behavior, or in an iterative development process where the design is implementation-oriented and the ultimate goal is to end up with an intra-object implementation.

1 Introduction

The goal of this work is to enrich the scale-up possibilities in the development cycle of reactive systems, when working in an inter-object, scenario-based paradigm, such as that described in [5]. We do this by introducing and implementing a methodology of distributed design, which involves two related ideas. The methodology is intended to supply a new level of flexibility in system development, and to help ensure that the various parts of a system designed by different teams cooperate and integrate into a single working and harmonious system.

The ideas are implemented in what we shall be calling **InterPlay**, a simulation engine coordinator[1] that supports the cooperation and interaction of different simulation and execution tools. These can support different design approaches to the modeling parts of a system or the various levels of abstraction thereof.

There are many proposed approaches to distributed computing, and many feature platform and language independence. This allows connecting applications spanning multiple platforms and operating systems, which have been written by different companies in various languages. Among such solutions are the following: RMI (Remoter Method Invocation) for distributed Java applications [11]; DCOM[2], which is most often associated with Microsoft operating systems but is also supported on Unix, VMS and Macintosh [1]; CORBA [9]; and

⋆ This research was supported by the Israel Science Foundation (grant No. 287/02-1).

[1] In fact, in [5] InterPlay was referred to by the acronym SEC.

[2] Soon to be replaced by .NET[8]

the more recent Web Services using the SOAP communication protocol [10]. While all these approaches apply to the realm of implemented components, there appears to be no solution to the problem of high-level model-driven distributed design that can offer independence of vendors (supporting, e.g., both Rational Rose, and Rhapsody from I-Logix), of overall design philosophy (supporting both an inter-object and an intra-object methodology), and of levels of abstraction. InterPlay can be viewed as an attempt to address these kinds of independence too.

Before discussing the two ideas manifested in InterPlay, we briefly recall the dual approaches to specifying reactive behavior, described, e.g., in [2, 5]. The first approach is an inter-object, scenario-based one, which is based on specifying cross-object scenarios of various modalities, one at a time. This approach is particularly natural for discussing behavior and specifying requirements, and is exemplified by the language of **live sequence charts** (LSCs) [2] and the **play-in/out** method with its supporting **Play-Engine** tool [5]. The second approach is the more conventional intra-object one, which is usually state-based, and is naturally suited for the specification of objects that have clear internal behavior. This approach specifies all possible behaviors for each object in the system, and it leads directly to implementation. It is exemplified by the language of **statecharts** [3] and the **Rhapsody** tool [4, 6], or by conventional object-by-object code. The conceptual duality between these approaches is illustrated visually in Figure 1.

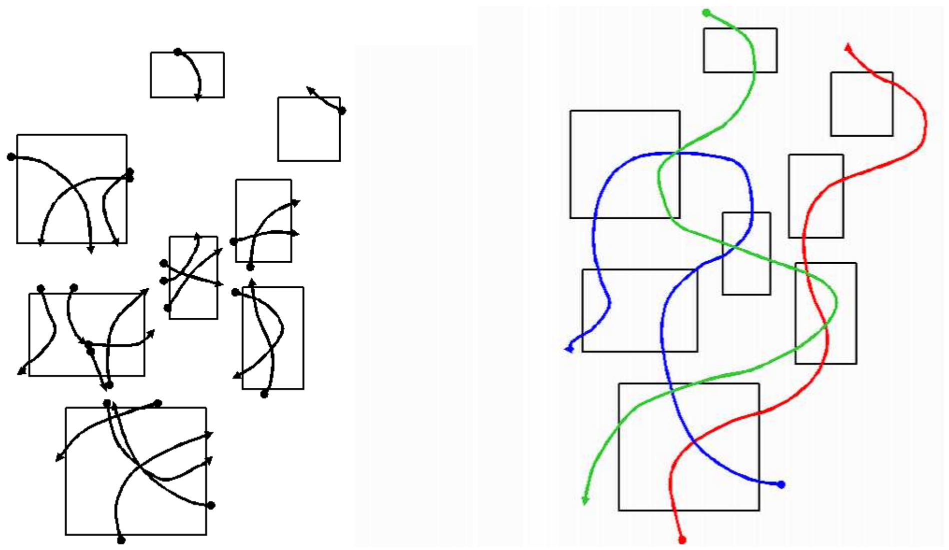


Fig. 1. A visual description of the intra-object and inter-object design approaches respectively

Let us examine the design and development cycle of a system, observing how the two approaches may be used within it. In the early stages of transforming the client's requirements into a formal specification, the overall functionality of the system is the most important. Here, the main logical components of the system will typically appear, with no specific implementation-related details. This bird's-eye point of view is best described using the inter-object design approach, where we ignore inner mechanisms of system components and focus on the overall behavior of the system, concentrating on interactions among the user, the environment and the system components. Complex systems may have a very large number of objects, practically forcing the distribution of the specification effort — and later also the design and implementation efforts — between multiple teams.

Accordingly, the first ability of InterPlay concentrates on the inter-object approach, and enables multiple Play-Engines to run in cooperation. This makes it possible for different teams to specify the inter-object behavior of different collections of objects, and then run these specifications in a fully cooperative manner. It also makes it possible to refine parts of the system using different Play-Engines. Technically, this is achieved by using **external objects**: each team is assigned some part of the system (actually, a set of objects) to design in detail. A particular team's objects may interact with other objects, to which the team refers as external. These external objects are in fact the *interface* of the other subsystems with respect to the current team's subsystem.[3] All other objects are ignored. The objects with which the team's specification interacts are thus outside the assigned scope and responsibility of the team, yet the team is aware of them, recognizing them as being designed and driven by some other team. The first part of the InterPlay methodology allows these different parts to be executed in tandem, by its ability to have multiple Play-Engines execute together. This distributed design method is illustrated in Figure 2.

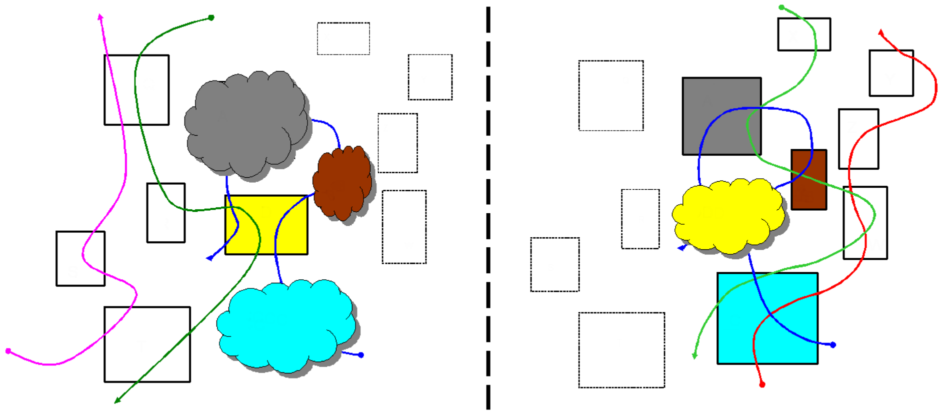


Fig. 2. Distributed design with external objects: External objects are drawn as clouds and each external-internal pair share the same color. Each team specifies a part of the system using the inter-object design approach, and refers to other relevant objects as external.

Let us now turn to the second ability of InterPlay. Following detailed specifications and refinement of requirements, we would like to carry out a transition to design and implementation. While the Play-Engine can indeed execute inter-object specifications, including multiple engines playing together through InterPlay, this is still within the inter-object approach. There will often be objects that have clear and distinct internal behavior which we

[3] For more details about external objects, interfaces and distribution to subsystems, see Section 3

would like to specify in a more conventional state-based intra-object fashion, using, say, statecharts or code. Moreover, the ultimate goal might be to end up with a complete intra-object implementation, which could be achieved by an iterative development process, during which objects will be gradually provided with intra-object implementation-oriented behavior. The Play-Engine would be useful at the very beginning of this process, and a standard intra-object tool like Rhapsody would be useful at the end, but we want something for the interim, when we have a combination of inter-object and intra-object specifications.

The second feature of InterPlay allows just that: the cooperative execution of a *mixed* system, some parts being specified in a scenario-based fashion, e.g., in LSCs, and others specified in an intra-object state-based fashion, e.g., in statecharts or code. Technically, InterPlay allows the Play-Engine and Rhapsody to execute simultaneously, each taking care of some of the objects. Figure 3 illustrates this, by showing an inter-object specification, with one object designed using the intra-object approach.

The two InterPlay ideas combined enable what we call **horizontal scale-up**, whereby a large system can be split up into parts, each specified in an inter-object or intra-object fashion, at will, and then executed as a whole by Play-Engines cooperating among themselves and/or cooperating with the Rhapsody tool. We view this as a crucial step towards the ability to incorporate the inter-object approach into the development of large and complex systems.

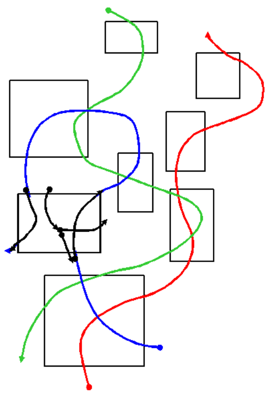


Fig. 3. An inter-object specification with one object designed using the intra-object approach

The rest of this paper is organized as follows. Section 2 gives a brief overview of the LSC language and the Play-Engine, illustrated using a take-out service system, which serves as a running example throughout the paper. Section 3 discusses the changes introduced in the Play-Engine to support InterPlay and explains their relevance to horizontal scale-up. Section 4 introduces in more detail the InterPlay tool and techniques. Section 5 elaborates on the take-out service example, illustrating the usefulness of InterPlay in integrating the various parts of a system. Section 6 concludes with a discussion of future work, including related research we are carrying out on **vertical scale-up**.

2 The Play-Engine and LSCs

This section provides a short introduction to the language of **live sequence charts** (LSCs) and the *Play-Engine*. The discussion, however, is very brief, and we strongly suggest referring to [5] for more details.

The language of LSCs [2] is a scenario-based visual formalism, which extends classical message sequence charts (MSCs) with logical modalities, thus achieving a far greater expressive power, comparable to that of temporal logic [7]. The Play-Engine supports LSCs, by enabling a system designer to capture behavioral requirements by **playing in** behavior using a graphical interface (GUI) of the target system or an abstract version thereof. As the behavior is played in, the formalized behavior is automatically generated by the Play-Engine, in the form of LSCs.

LSCs have two types of charts, **universal** and **existential**. Universal charts are used to specify restrictions over all possible system runs, and thus constrain the allowed behaviors. A universal chart typically contains a **prechart**, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. Existential charts, on the other hand, specify sample interactions between the system and its environment, and are required only to be satisfied by at least one system run. They thus do not force the application to behave in a certain way in all cases, and can be used to specify system tests, or simply to illustrate longer (non-restricting) scenarios that provide a broader picture of the behavioral possibilities to which the system gives rise.

We borrow an LSC from our running example, a take-out system described in detail in section 5, to illustrate the main concepts and constructs of the language of LSCs.

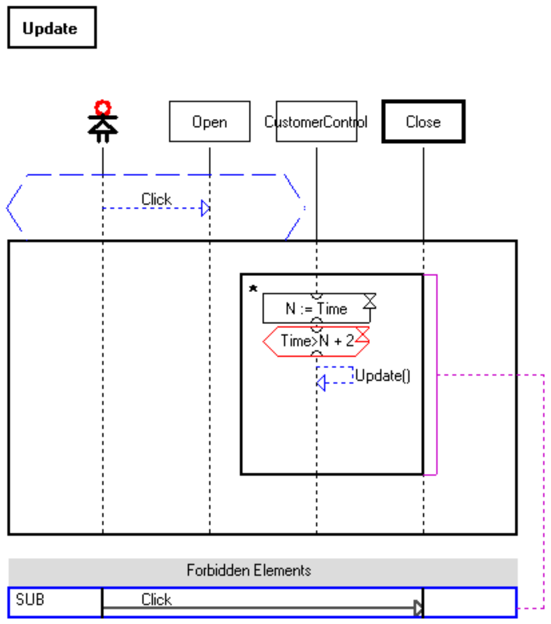


Fig. 4. An LSC example: Updating the occupancy of a restaurant

In the universal LSC of Figure 4, the prechart (top dashed hexagon) contains the event of the user clicking the `btnOper` button. If this indeed occurs, the chart body then requires the `CustomerConrtol` object to update the occupancy of the restaurant by means of a method call that changes the number of customers in the restaurant. However, we want this update to happen only after a fixed time interval — three clock ticks in our case. The chart body consists of an unbounded loop construct (denoted by '*'), which is repeated infinitely many times, unless interrupted. The loop contains an assignment in which the variable `N` is assigned the current time. It is important to note that the assignment's variable is local to the containing chart and can be used for the specification of that chart only, as opposed to the system's state variables, which may be used in several charts.

After the assignment comes a **hot** condition, requiring the time to advance 3 ticks before continuing. Hot conditions are mandatory, and must always be true; if not, the requirements are violated and the system aborts. However, when dealing with time, the system simply waits until the specified condition holds. On the other hand, if a **cold** condition is false, the surrounding (sub)chart is exited. This is one example of the way the logical modalities are incorporated into LSCs.

An LSC can have **forbidden elements**, listed in a separate area underneath the main chart. Hot and cold elements work similarly there too; e.g., if a hot forbidden condition becomes true, the requirements are violated and the system aborts, whereas a cold one becoming true causes the chart or subchart which is its scope to be exited. In our example in Figure 4, there is a cold forbidden message associated with the loop subchart, the effect being that if the user presses the `btnOper` button again the loop and the chart terminates.

We shall not discuss the play-in process here, but play-out is very relevant. In the play-out phase the user plays the GUI application as he/she would have done when executing a system model (or, for that matter, the final system) but limiting him/herself to 'end-user' and external environment actions only. While doing so, the Play-Engine keeps track of the actions taken, and causes other actions and events to occur as dictated by the LSCs, thus giving the effect of working with a fully operational system or an executable model. It is actually an iterative process, where after each step taken by the user, the play-engine computes a **superstep**, which is a sequence of events carried out by the system as response to the event input by the user. Only those things it is required to do are actually done, while those it is forbidden to do are avoided. This is a minimalistic, but completely safe, way for a system to behave exactly according to the requirements. It is noteworthy that no code needs to be written in order to play out the behavior, nor does one have to prepare a conventional intra-object system model, as is required in most system development methodologies (e.g., using statecharts or some other language for describing the full behavior of each object, as in the UML, for example). We should also emphasize that the behavior played out is up to the user, and need not reflect the behavior as it was played in; the user is not merely tracing scenarios, but is executing the specified behavior freely, as he/she sees fit.

This ability to execute inter-object behavior without building a system model or writing code leads to various improvements in building reactive systems. It enables executable requirements, for example, whereby the Play-Engine becomes a sort of 'universal reactive machine', running the requirements that were played in via a GUI or written directly as LSCs.[4]

[4] In principle this could have been done using any other sufficiently powerful scenario-based language, such as timing diagrams or temporal logic.

You provide the global, declarative, inter-object ways you want your system to behave (or to not behave), and the engine simulates these directly. It also allows for executable test-suites, whose executions can then be compared with those of the actual implementation.

As we shall explain later, enabling the cooperation of multiple Play-Engines and these cooperating with conventional tools, allows both distributed design and refinement of such specifications, as well as the gradual introduction of implementation-oriented details in advanced design stages.

3 External Objects in Preparation for InterPlay

Some time ago we introduced external objects into LSCs and implemented them in the Play-Engine along with their respective mechanisms; see Chapter 14 in [5]. However, that introduction was made bearing in mind the idea presented here. In fact, on their own, without InterPlay, external objects are rather hollow, providing little substantial enhancement to the design and development cycle[5]. In this section, we briefly survey the addition of external objects, stressing their role in the scheme we present.

When dealing with reactive systems we distinguish between the system proper and other elements that interact with it, to which we refer as the **environment**. The system's user is separated from the environment and can interact with the system through the GUI, while the other elements of the environment can affect external settings of the system, mainly through changing object properties. Since most reactive systems work in the presence of such external/environmental objects and can affect them and be affected by them, it is necessary to express the interaction with them.

Technically, we have added to the LSCs language and to the Play-Engine a new kind of object, the **external object**, which will be considered as part of the system's environment. External objects are recognized by the system, but are driven externally by another modeling tool, or by code. What will become extremely important, however, is the fact that external objects allow other systems to interact with the one we are working on.

Having external objects within the specification entails more than just breaking up the environment into individual pieces. These pieces are objects in their own right, they have properties, they can be in different states, they can call other objects, etc. However, as we shall see in a moment, in terms of what the Play-Engine knows when 'working on' a particular system with its environment, an external object is abstract; it is not considered to be an ordinary object, and, for example, cannot be triggered (by our Play-Engine specification) to call other objects.

In the LSCs themselves (and also during play-in) external objects are treated much like other objects, and the fact they are external is merely indicated by a little cloud attached to the object-name box. Any object can be made external easily, by flipping the appropriate property in its definition. Thus, objects can be considered internal throughout some portion of the system development process, and then made external later on, whether for refining its design elsewhere, or to implement and test it. We shall see later how this ability can be exploited.

The main difference between internal and external objects occurs during play-out. Usually, property changes of objects, and calls between them, are performed by the Play-Engine

[5] Without InterPlay no more than two Play-Engines can run cooperatively, and they must always use the exact same system model

as a part of its super-steps. This, however, is not what we want for external objects. The way they are controlled in a simple one-engine use of the Play-Engine is by the system's end-user, but the ultimate goal is for them to be controlled by some other modeling tool, possibly another Play-Engine, or implemented in code. And this is what InterPlay is all about. Consequently, the execution mechanism of the Play-Engine has been modified, so that it does not initiate events that originate from external objects, just as it does not initiate events from the user, or the environment.

Appropriate sets of external objects serve as a commitment between the different teams and their respective parts of the system. They can be compared to an interface in object-oriented programming. The team that sees a specific object as external uses it as a part of its communication mechanism with the outside world. As such, the team relies on this object having certain properties and methods. Hence, the team that 'owns' the object as internal can add properties or methods to it, but not change the original ones. All the added properties and methods added in such a way are for the internal use of that specific team and are not reflected outside on the other external views of the object.

Our methodology is, in a sense, backward compatible, since it can be applied to any existing specification set, even if it was prepared before the introduction of external objects. One of the benefits of this compatibility is that even if two systems have been specified separately, they can later be joined, without any pre-planning. If the two different specifications have referred to some common part, even if slightly differently and by different names, they can still be considered jointly, by choosing the common part to be external in one of the specifications and remaining internal in the other.

In order to support the external objects mechanism, we added to the Play-Engine an **external event manager**, which deals with the technicalities of remote connections to other computers (e.g., IP, ports, etc.) and conveys messages to and from external objects. In fact, once the external manager is activated, the Play-Engine transmits to the outside world the entire sequence of events that occurs among its GUI and internal objects. The Play-Engine also receives via the external manager events and messages from other Play-Engines, or other modeling tools. Since external objects reflect elements specified or implemented outside the scope of the local Play-Engine, events (e.g., property changes or method calls) that originate in those objects also arrive through the external manager. Upon receiving such an event, the Play-Engine acts as if the event originated from the external object itself. In short, the external object is recognized by the local system, but is driven by a remote one.

In order to best serve the InterPlay techniques, the external manager has various operation modes, allowing either cooperation between two Play-Engines or execution by a single Play-Engine and monitoring its run by another. Such a connection was possible between only two Play-Engines having the exact same system model. However, using InterPlay any number of Play-Engines, with different system models, can be connected, as we shall see shortly.

4 InterPlay: Cooperation of Various Design Tools

InterPlay operates in two stages, a preprocessing offline stage, and a main online execution stage. In the first stage a mapping is set up, which associates each internal object with all of its images as external objects in other tools, making them all seem as a single object. During the execution stage InterPlay uses the mapping to translate and transmit messages and events among the connected models and their respective tools, so that whatever happens to an object during play-out is reflected in all its external views.

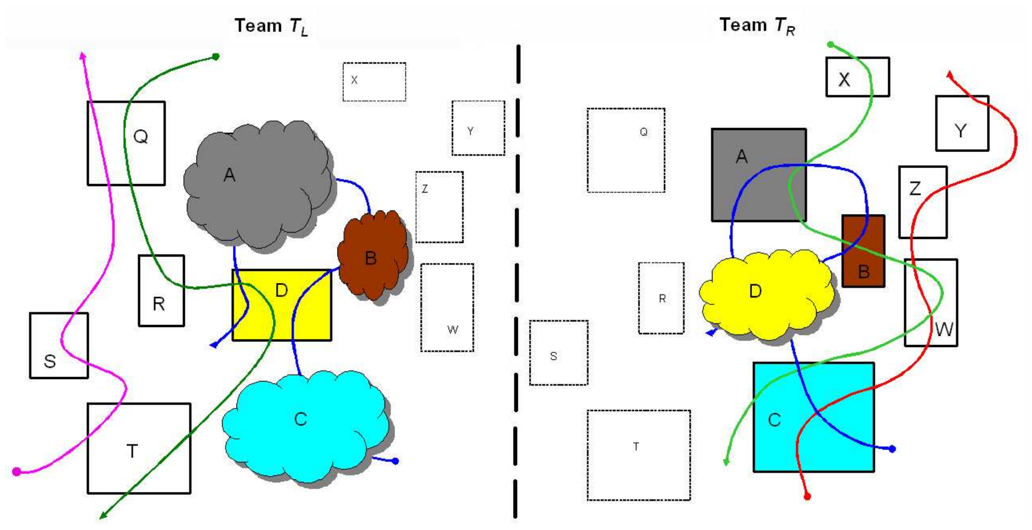


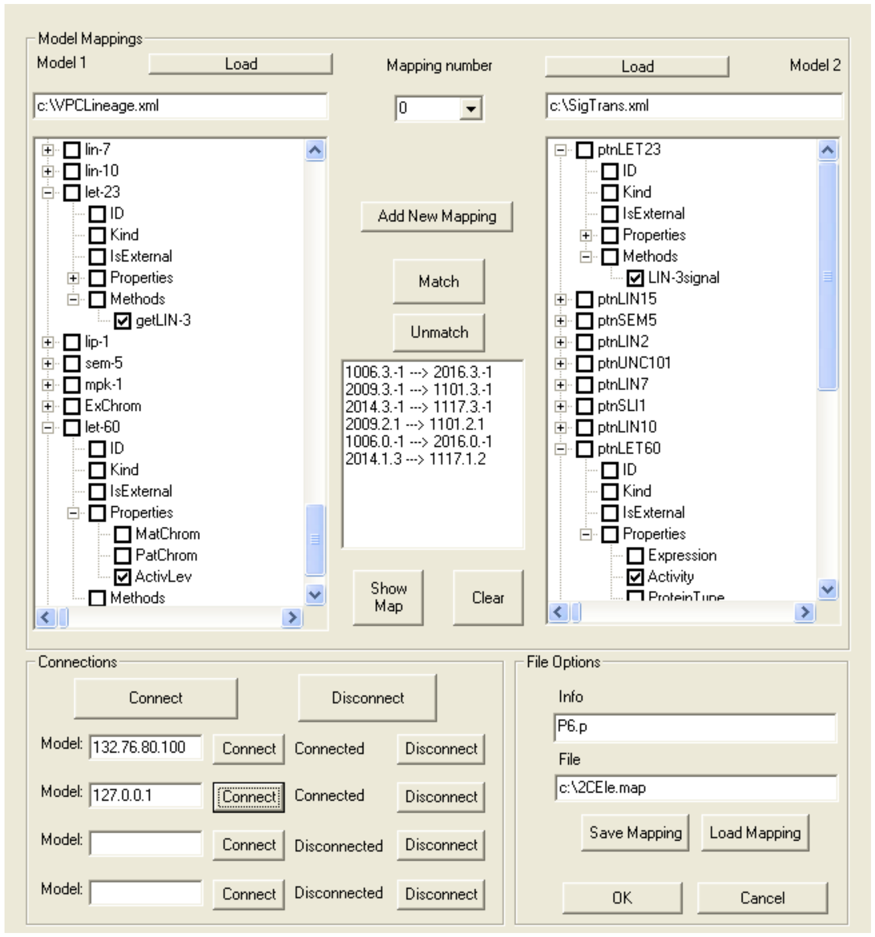
Fig. 5. Inter-object specifications of a system from the points of view of two teams. Objects with thin dotted lines do not actually appear in the relevant specification and are included for better illustration only.

InterPlay's mapping stage is really part of the system's specification, in which one indicates how the different parts of the system fit together. We use the two specifications in Figure 5 throughout this section as a specific example, and concentrate on connecting only multiple Play-Engines.

Consider object D in the figure. It is internal to the left-hand team T_L and external to the right-hand team T_R. Although both teams do deal with this common object, they might refer to it by different names[6] and team T_L might have added to it additional properties or methods. Thus InterPlay works on mapping two system parts together, in order to overcome such naming differences while matching an object to its external view. This, of course, does not limit the number of specifications of systems parts and their respective tools that can be fused together. Figure 6 displays a screenshot of InterPlay mapping two system models to each other. These are two parts of a biological system, which communicate using two common proteins Let-23 and Let-60. Although both parts refer to the same proteins, their descriptions are very different in the two models. The common interface is a method in Let-23 and an activity measurement property in Let-60, which are mapped to each other through InterPlay.

When using InterPlay to bridge different levels of abstraction one has to pay particular attention to the specification refinement from coarse to fine. Objects described on the coarse level are **interface objects** for some subsystem that interacts through them. Hence, on a coarse level we describe interactions among interface objects, while when refining the

[6] Had there been another team containing object D as external, it could have referred to it by yet a different name than do teams T_L and T_R.



Model 1: c:\VPCLineage.xml

Mapping number: 0

Model 2: c:\SigTrans.xml

```
1006.3.-1 ---> 2016.3.-1
2009.3.-1 ---> 1101.3.-1
2014.3.-1 ---> 1117.3.-1
2009.2.1 ---> 1101.2.1
1006.0.-1 ---> 2016.0.-1
2014.1.3 ---> 1117.1.2
```

Model: 132.76.80.100	Connect	Connected	Disconnect
Model: 127.0.0.1	Connect	Connected	Disconnect
Model:	Connect	Disconnected	Disconnect
Model:	Connect	Disconnected	Disconnect

Info: P6.p

File: c:\2CEle.map

Fig. 6. InterPlay screen shot, mapping two biological models through an interface of a method and a property of two common to both.

specification we implement[7] the subsystems that interact through them. This rather subtle difference from actually refining an object is further illustrated in Figure 7: The right-hand side of the figure is the coarse level system, in which there is an external interface object I_X. The left-hand portion of the figure is a refinement of the I_X, and within it the internal object X implements the interface object on the coarse level. All other objects on the fine level constitute the subsystem behind the interface X. Thus, all interactions between this subsystem and other subsystems are conducted through object X. When mapping the two specifications through InterPlay, X is matched to I_X, allowing events on the finer abstraction level to be reflected on the coarser level, and vice versa.

[7] By "implementation" in this context we still refer to inter-object design, used to specify in detail a subsystem which has been declared on the coarse level.

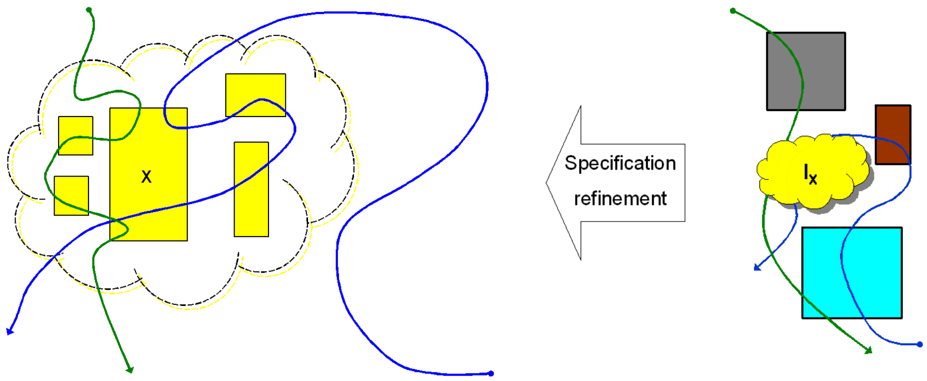


Fig. 7. Multiple levels of abstraction. The left figure represents a specification refinement of the external interface object I_X in the coarse level.

Here's how the mapping is set up. InterPlay loads a system model from a Play-Engine specification and displays it to the user. Only the components of the system are loaded (i.e., the GUI and internal objects, with their properties and methods), without any behavior (LSCs) attached. There are several levels of mapping between objects. Assuming object A has not been extended with new methods or properties by team T_R, the mapping can be completed as is, by simply associating (using an appropriate form that pops up) the two versions of A on the object level. This implies that all the object's properties and methods are also mapped.

Assume that object B has been expanded by team T_R. InterPlay allows partial mappings of selected properties and methods, leaving some unmatched. Thus, only the properties and methods common to the two teams will be mapped to each other and we do not allow splitting; e.g., mapping some properties of B in team T_R's specification to object B of team T_L and others to object C therein. This kind of splitting up of an object is closely related to aggregation, and is the central aspect of **vertical scale-up**, which we discuss briefly in Section 6. Nevertheless, InterPlay does allow mapping multiple objects to a single one on the object abstraction level. Going back to Figure 5, it might be the case that the left-hand team T_L considers objects A, B and C as having the same functionality. For example, D might be a department manager with a direct phone line connection to his/her bosses A, B and C. As only these bosses can call this line, D is impervious to which of them assigns him/her a task. Team T_L can thus use a single external object only, say, A, which will be mapped to the group of objects A, B and C in team T_R's specification. This raises the question of whether any event involving object A in team T_L's specification would have to be reflected in all of its mapped variants on the right. Currently, InterPlay broadcasts such an event to all internal objects mapped to an external one, but other possibilities are mentioned in Section 6.

During play-out, InterPlay carries out the ramifications of the mappings set up in the preliminary phase. Each Play-Engine connects to InterPlay through its external manager. Once connected, played out events (user operations, property changes and method calls) are transmitted to InterPlay, which translates them according to the mappings and sends them to all the relevant Play-Engines. Consider the blue (rightmost) scenario in Figure 5. Play-out starts with team T_R's Play-Engine, involving object C. Since C is internal to T_R's, its Play-Engine performs the necessary events, operating it. InterPlay translates and transmits these events to the T_L's Play-Engine, which traces the scenario as well. The scenario moves on to object D, which is external to T_R's scope, and thus T_R's Play-Engine goes idle. Object D is now 'driven' by the T_L's Play-Engine and through InterPlay the respective events are sent to the T_R's Play-Engine. This initiates an event coming from D, allowing the scenario to proceed. The scenario continues in a similar fashion, with each Play-Engine running and driving its own internal objects, and waiting to receive input from the other one if necessary.

As mentioned above, this description concentrates on several Play-Engines, but a similar process is carried out when the Play-Engine is connected to Rhapsody. More on this later.

5 An Example: The Food Take-Out System

In this section we illustrate InterPlay by a simple example of a food take-out service that enables clients to order food from diverse restaurants through a single ordering center.

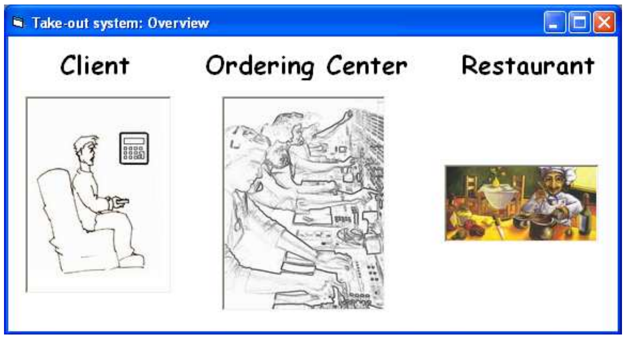


Fig. 8. The three high-level components of the food take-out system

The development process starts with specifying an inter-object overview of the system's overall functionality. This coarse specification identifies the system's main components — a client, the ordering center and a restaurant component — as illustrated in the GUI of Figure 8. Using the Play-Engine and LSCs, we describe the functionality of the system by interactions among these components, as exemplified in Figure 9. One LSC therein describes the simple process of acquiring a menu from the ordering center, while the other concerns placing a take-out order. Before we explain the latter LSC, note that the `Client` and `Restaurant` were internal at this stage and became external only in later design stages. The prechart contains the event of the `Client` ordering a dish by calling the `Center`'s `Order(Dish)` method. Should this occur, the main chart specifies the `Center` asking for a time estimate on the `Dish` from the `Restaurant`, by calling the `Restaurant`'s `Estimate(Dish)` method. The `Restaurant`'s resulting estimated time is conveyed to the `Center` via the `Time(T)` method. (In accordance to the inter-object design approach we do not specify at this stage how the restaurant calculates this estimated time.) After receiving the estimated time to delivery, the `Client` responds by calling the `Confirm` method with its `ID` and `Decision`. Should the `Client` agree, depicted by the cold `Decision=True` condition, a series of method calls follows, confirming the order to the `Restaurant` and getting an

`OrderID` in exchange. If for some reason the `Customer` doesn't wish to order, the chart is simply exited, in effect cancelling the order.

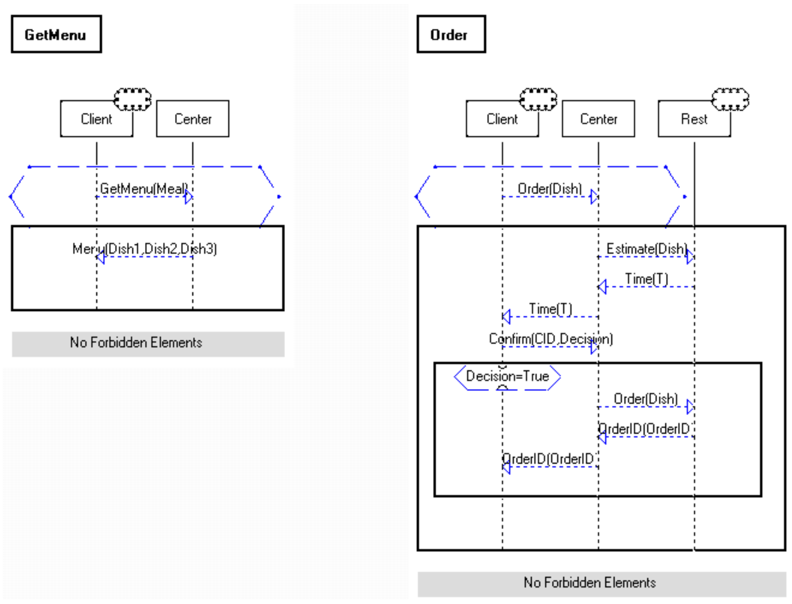


Fig. 9. Two LSCs describing an overview of the behavior of the take-out system.

We now decide to distribute the rest of the specification among three teams, each in charge of one of these components. Each team is required to refine the specification of its assigned subsystem, respecting the interface that was defined on the coarse level. Hence the client has an internal object called `I_Panel`, implementing the interface defined by the `Client` object on the coarse level and serving as its interface with the other system components. It also has an external object called `CommUnit` that implements the ordering center's interface within the client's subsystem. In other words, the entire client subsystem interacts with the rest of the system, represented by `CommUnit`, through its interface, `I_Panel`. Similarly, the restaurant's subsystem has an internal object, `I_Rest`, as its interface with the 'outside world', which in turn is represented by the external `CommUnit`. These objects can be seen in Figures 10 and 12, which show the refined GUIs and additional objects of the client and restaurant subsystems, respectively.

Having the coarse design level available, we then approach the client subsystem and refine its specification using the aforementioned interface and adding to it further objects and internal behavior. Figures 10 and 11 illustrate this specification refinement, with its GUI and a self-explanatory LSC example that describes the process of the client ordering a dish.

Now that the client's subsystem refinement is complete, we make the `Client` object on the coarse level external. As such, the Play-Engine playing out the coarse level can no longer initiate events from the client. Instead, it waits for them to arrive, having been initiated by another Play-Engine playing out the client subsystem. We played out both specifications,

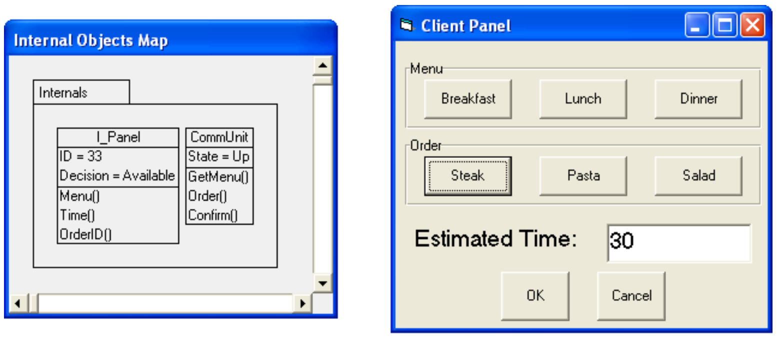


Fig. 10. The client's GUI and internal objects.

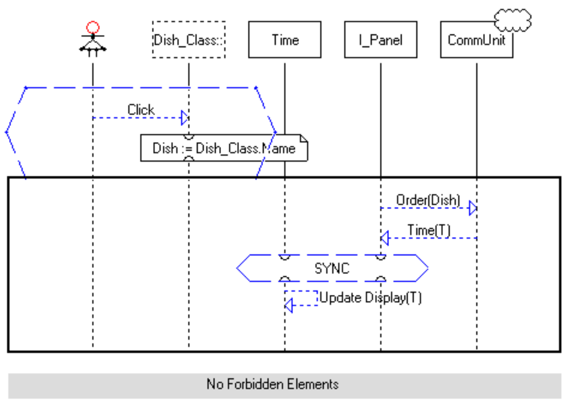


Fig. 11. An LSC that describes the ordering process from the client's point of view. This LSC is invoked by all three buttons referring to dishes (second row in the GUI), which are of class `Dish_Class`.

one fine and one coarse, in cooperation, using InterPlay, as we explain shortly. At this stage the restaurant has not been refined yet, so it continued to be 'driven' by the coarse level specification.

The restaurant's team then starts to refine its specification, deciding that the restaurant has to have some cooks to keep the business running, a few customers who sit inside, and two indicator buttons to capture the opening and closing of the restaurant. The team specifies how these parts of the system should behave, independently of, and in ignorance of, how their 'outside world' operates, but still aware of it and interacting with it through the external `CommUnit`. The restaurant's GUI and additional objects are shown in Figure 12, while an LSC example describing part of its internal behavior is shown in Figure 13.

The LSC in the figure specifies how the restaurant calculates the time estimate for a requested dish. It is activated when the `CommUnit` requests an estimate by calling the method `Estimate(Dish)` of the restaurant's interface, `I_Rest`, as defined in the prechart. In the main chart, using a select-case construct, the basic time required for the requested dish is stored. The number of available cooks is also taken into consideration in the if-then-else

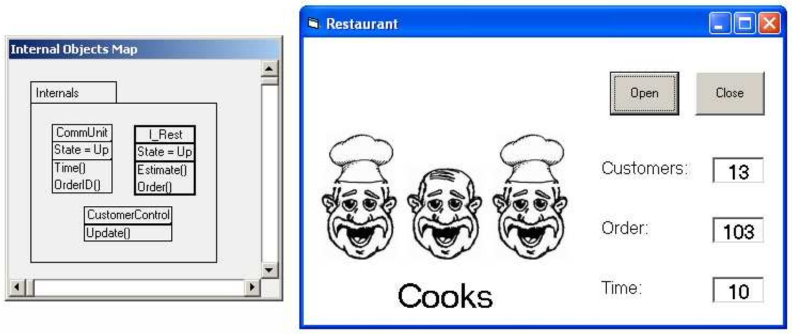


Fig. 12. The restaurant's GUI, including cooks and a reflection of its state. Cooks wearing hats are laboring in the kitchen while the others are on break.

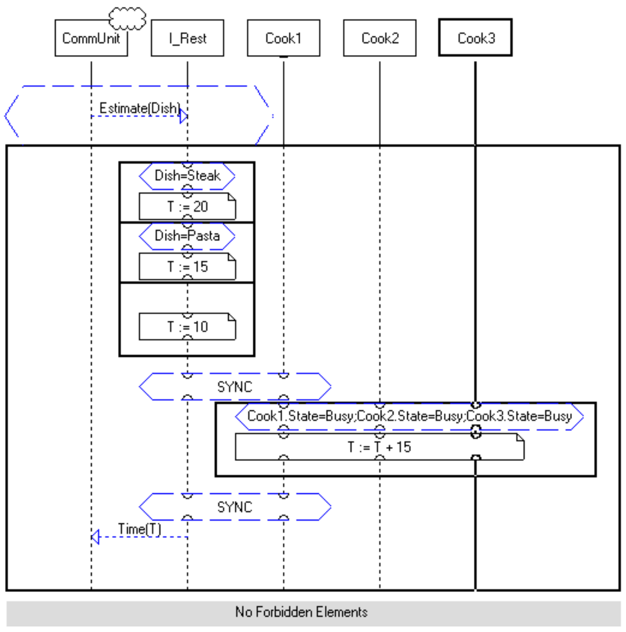
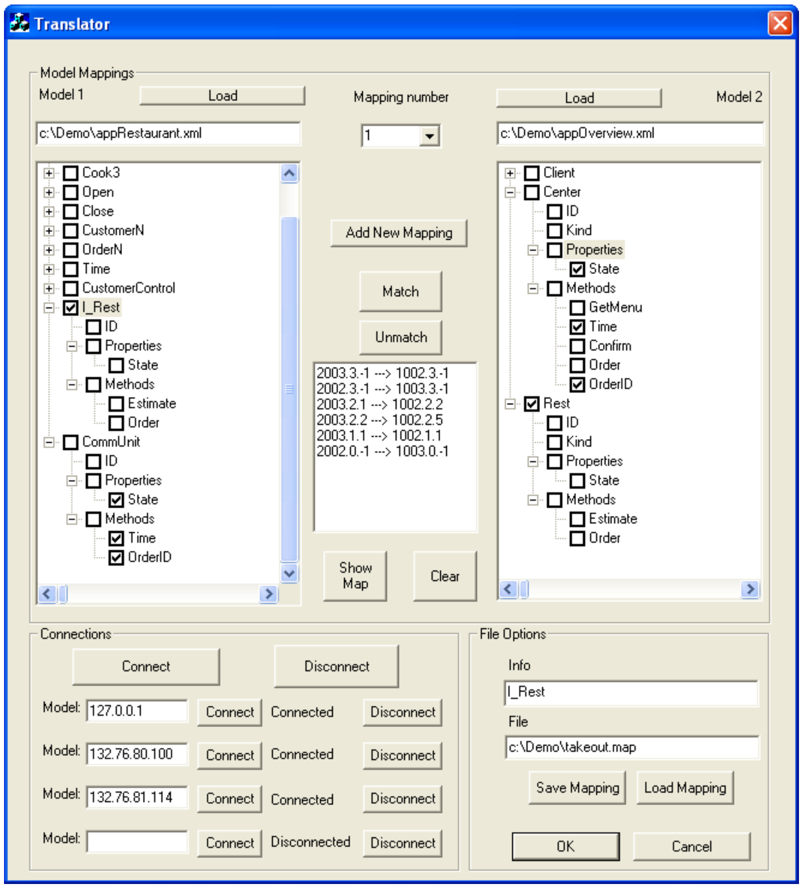


Fig. 13. An LSC describing part of the restaurant's inner behavior

construct. Finally, the restaurant's interface `I_Rest` returns the preparation time to the ordering center, through the `CommUnit`. The restaurant's specification refinement involved a few other LSCs that deal with its internal behavior, such as one describing the working routine of the cooks in the restaurant, depending on the amount of clientele patronizing it. For lack of space we will not show these here. Recall also Figure 4, which updates the number of clients in the restaurant every 3 clock ticks, by calling the `Update` method.

Having now refined the specifications of the client and the restaurant subsystems, we make both `Client` and `Restaurant` objects external on the coarse level. The three system



c:\Demo\appRestaurant.xml

1

c:\Demo\appOverview.xml

Model 1 tree: Cook3, Open, Close, CustomerN, OrderN, Time, CustomerControl, [✓ I_Rest] (ID, Properties: State; Methods: Estimate, Order), CommUnit (ID, Properties: [✓ State]; Methods: [✓ Time], [✓ OrderID])

Model 2 tree: Client, Center (ID, Kind, Properties: [✓ State]; Methods: GetMenu, [✓ Time], Confirm, Order, [✓ OrderID]), [✓ Rest] (ID, Kind, Properties: State; Methods: Estimate, Order)

```
2003.3.-1 ---> 1002.3.-1
2002.3.-1 ---> 1003.3.-1
2003.2.1 ---> 1002.2.2
2003.2.2 ---> 1002.2.5
2003.1.1 ---> 1002.1.1
2002.0.-1 ---> 1003.0.-1
```

Model: 127.0.0.1	Connect	Connected	Disconnect
Model: 132.76.80.100	Connect	Connected	Disconnect
Model: 132.76.81.114	Connect	Connected	Disconnect
Model:	Connect	Disconnected	Disconnect

Info: I_Rest

File: c:\Demo\takeout.map

Fig. 14. InterPlay screenshot, showing the mapping of the refined restaurant's subsystem model to the coarse level overview of the take-out system.

models, with only the objects and their respective properties and methods, are loaded into InterPlay. We map the refined subsystems to the coarse specification, in turn, by associating their appropriate interface objects: `I_panel` is mapped to `Client` and `I_Rest` is mapped to `Rest`, while both `CommUnits` on the fine level are mapped (separately) to `Center` on the coarse level. Notice that the latter two mappings are made based only on a subset of the methods and properties, while the former two are made on the object level. The mapping of the refined restaurant to the overview of the system is shown in Figure 14.

The entire system can now be run in cooperation by three different Play-Engines, one for each of the two refined subsystems and the third running the coarse specification, providing the functionality of the yet unrefined ordering center and monitoring the entire run. Since the Play-Engine can record a run and later display it as an LSC, we have attached in Figure 15 the three recordings of the respective Play-Engines.

After all of this, and assume we have executed, revised and verified the inter-object specification, we might want to make a transition to design, or in other words, to move towards an

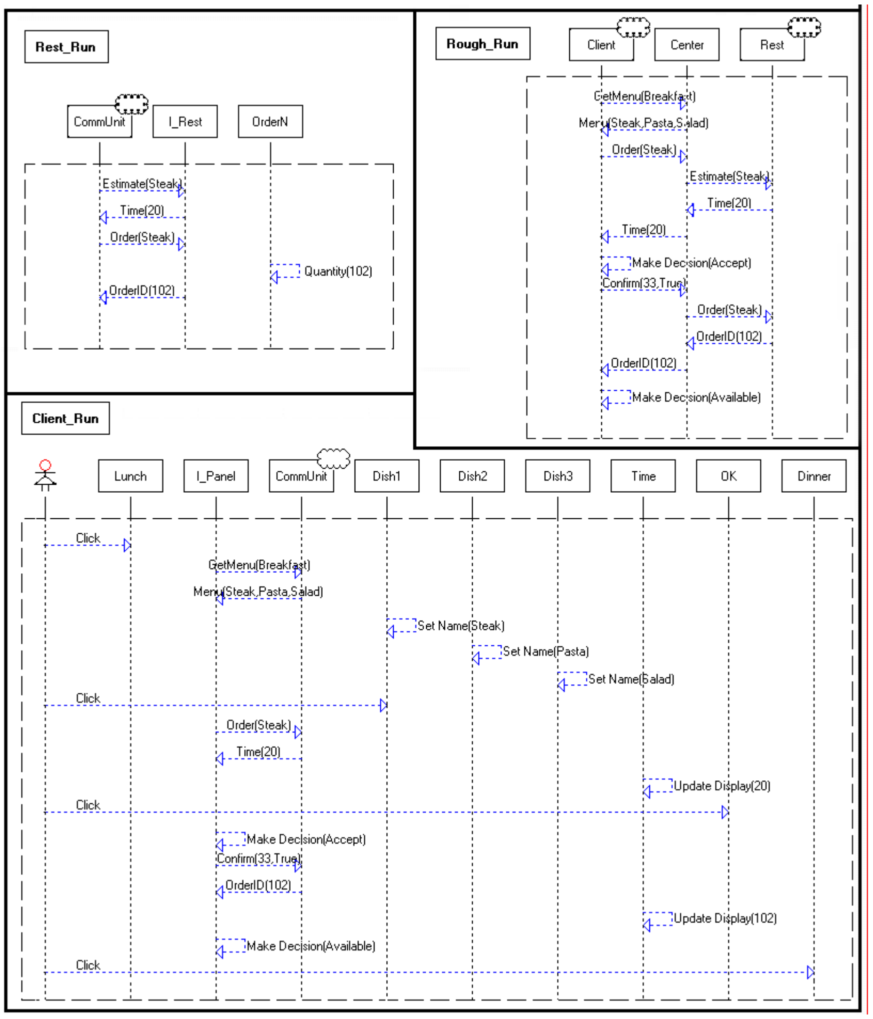


Fig. 15. The results of running the take-out system using three cooperating Play-Engines. Each LSC is the trace of the run from the point of view of one Engine's model.

intra-object implementation. We could pick the restaurant's interface unit (`I_Rest`), for example, which has clear internal behavior. We would make it external to the inter-object specification of the restaurant, and proceed to define its internal behavior in a state-based fashion using statecharts and Rhapsody. We can now load the unit's system model from Rhapsody into InterPlay and map it to the restaurant's LSCs specification. This would then allow running the intra-object design, or implementation, of the panel both against its specification and in cooperation with the rest of the take-out service system.

Doing this for all the parts of the system that we want to have implemented in an intra-object way would lead to a full implementation. All remaining parts would be played-out in an inter-object manner, with the relevant Play-Engines handing over control whenever an implemented part is to become active.

6 What Next?

In this section we discuss several issues for future research.

Connecting to Rhapsody: We have repeatedly stated as our goal not only to connect Play-Engines to each other, but also to allow cooperation between many types of design or modeling tools. We have set up an initial connection between Rhapsody and the Play-Engine. Its present status is that of a feasibility test, and was carried out in a tailored fashion for a particular system model, with very encouraging results. We are now in the final stages of making this connection generic through InterPlay.

We do not make any changes to Rhapsody's framework in order to allow this connection. Instead, we offer an API with which a dll plug-in can be created. The plug-in serves as an observer that receives events of interest from Rhapsody as they take place during the system run, and can also interact with the animation module of Rhapsody, generating events that will impact the animation. We then made it possible for these dll plug-ins to communicate with InterPlay, in both sending and receiving events.

Connecting to Other Implementations: There is clearly much value in allowing the Play-Engine to be connected to other kinds of modeling and implementation tools, including standard programming environments. For example, if a project requires designing a new component that has to fit exactly into an existing complex of implemented components or systems, it could be extremely useful to connect the LSC model we build for it using the Play-Engine via InterPlay directly to the real environment, allowing the composite system to be tested and run as an integrated whole.

Moreover, given such flexible connection abilities, modeling tools like the Play-Engine could be used to conduct integration tests of implemented components even if these were designed using other tools. The implemented system could then be executed with a Play-Engine tracing its runs, making sure they fit the requirements (which would have been predefined as LSCs).

To make such broad connection abilities possible, we intend to construct a simple API for connecting to InterPlay, which most implemented systems will be able to incorporate. Since they would all connect to each other through InterPlay, no changes in any of these tools will be required by this addition.

Synchronous Messages: Synchronous messages, supported by the Play-Engine, raise a whole new level of complexity when one uses InterPlay to carry out truly distributed modeling and implementation. Recall that a synchronous messages is one that flows (for all practical purposes in zero time) from the sending object to the receiving object if and when the former is ready to send it and the latter is free to receive it. When both objects are controlled by a single Play-Engine it is relatively easy to determine whether the message can be sent, and if so to make sure nothing changes in the two objects until the message is delivered. This is far more complicated when the two objects are driven by different Play-Engines, and even worse if they are driven by statecharts or code.

Several possible solutions come to mind, such as using a *two phase commit* protocol, of the kind used in certain kinds of transaction processing. We have not yet dealt with this feature, and doing so would probably require subtle changes both in the Play-Engine and in the InterPlay module.

Centralized Clock Ticks: Another complication that InterPlay gives rise to involves time. Recall that the Play-Engine supports time via a single clock, with a tick event that can be advanced through the host computer's clock or via the model itself (e.g., by the user or by other objects). Clearly, different Play-Engines running different specifications cannot be assumed to advance clock ticks at the same (absolute) rate, and the classical problems of distributed time arise in full force. Even running a single Play-Engine will advance time very differently when run with or without visual animation of the LSCs, not to mention different Play-Engines working in tandem or with other modeling tools.

Without getting into the usual controversies and opinions about how to best deal with time in a distributed environment, it is quite obvious that there are several incentives for supplying a mechanism for centralized clock ticks across InterPlay. (For one, we might be using InterPlay to build an ultimately centralized system in a distributed fashion.) We propose to add the option of receiving clock tick signals from InterPlay through the external event manager. This is relatively easily done. We have also looked closely into Rhapsody, which has a special time mode controlled by the user, and conclude that it too can receive clock ticks from InterPlay through the observer dll without making changes to the main program's framework.

Type Mapping: Currently two objects, or their properties and methods, can be effectively mapped to one another if they are of the same type, or receive parameters of the same type. We plan to consider adding more flexibility to InterPlay through a type-mapping feature, allowing system models to enrich their interaction without having to make further adjustments to the model itself.

Delegating to Multiple Objects: Recall that InterPlay allows mapping multiple objects to a single one on the object abstraction level. However, should an event that involves the single object be necessarily reflected onto all of its multiple images? We do not have enough experience with InterPlay to decide on this quite yet. Other than the obvious approach, currently implemented, of broadcasting each message (and relevant event) to all the objects mapped to the source, we could also implement a scheme that sends it to the latest image to have interacted with the source. We could also have a user-driven mode, letting the user of InterPlay decide at run time how to delegate the message. Recently we have been toying with the idea of allowing asymmetric mappings, which might solve this problem more elegantly, but this is still in preliminary stages only.

Vertical Scale-Up: In this paper we have used the term horizontal scale-up to denote the kinds of connections between tools we have discussed. The reason is that what they make possible is the **composition** of collections of objects in a side-by side manner (although in an implicit way a limited kind of refinement can be specified too as we have seen in section 5). Complimentary to this is vertical scale-up, whereby we want to support in LSCs and the Play-Engine the **aggregation**, or **rich refinement** of objects. In other words we want in the large a full notion of hierarchies of objects, complete with multiple-level behavior, even within a single LSC specification. And we want all this related in the play-in and play-out processes. This is a complicated topic, since it is not clear how to best define aggregation in the presence of inter-object behavior. For example, how should scenarios (i.e., LSCs) defined within an object, among its sub-objects, be connected to the scenarios between the parent object and its siblings on the higher level? What kind of mappings should we allow between levels, etc.? We are in the midst of a research project on this, and hope to be able to report on it in a future paper.

References

1. N. Brown and C. Kindel. Distributed Component Object Model Protocol - DCOM/1.0. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.
2. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1), 2001. (Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (*FMOODS'99*), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.).
3. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Prog.,* 8,3, 231-274, 1987. (Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.).
4. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, July 1997, pp. 31-42.
5. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
6. I-logix,inc., Website: http://www.ilogix.com/products/rhapsody/index.cfm.
7. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
8. Microsoft .NET architecture and resources: http://www.microsoft.com/net/.
9. OMG - Object Management Group. *The Common Object Request Broker: Architecture and Specification*. 2.2 ed, 1992.
10. *Simple Object Access Protocol (SOAP) 1.1*. W3C Note 08 May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
11. A. Wollrath, R. Riggs and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, vol. 9, November/December 1996.