Games for UML software design

Perdita Stevens* and Jennifer Tenzer**

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. In this paper we introduce the idea of using games as a driving metaphor for design tools which support designers working in UML. We use as our basis a long strand of work in verification and elsewhere. A key difference from that strand, however, is that we propose the incremental development of the rules of a game as part of the design process. We will argue that this approach may have two main advantages. First, it provides a natural means for tools to interactively help the designer to explore the consequences of design decisions. Second, by providing a smooth progression from informal exploration of decisions to full verification, it has the potential to lower the commitment cost of using formal verification. We discuss a simple example of a possible game development.

1 Introduction

The Unified Modeling Language (UML)[10] is a widely adopted standard for modelling object-oriented software systems. It consists of several diagram types providing different views of the system model. UML is a semi-formal language defined by a combination of UML class diagrams, natural language and formal constraints written in the object constraint language (OCL).

There are many of tools available which support, or claim to support, design with UML. They aid in drawing UML diagrams, generation of code fragments in different object-oriented languages and documentation of software systems. Some of them are able to perform consistency checks such as for example the accessibility between packages. However, these features seem to be useful for the recording and verifying of a chosen design, rather than for the design activity itself. There is nothing available to the mainstream object-oriented business software developer which will actively help him or her to explore different design decisions and work out which is the best option. It is natural to look to verification to fill this gap.

Ideally, a designer would be able to make use of verification technology whenever s/he is faced with a difficult decision, say between two design solutions. S/he might take two models representing the design with each of the solutions applied,

* Email: Perdita.Stevens@ed.ac.uk Fax: +44 131 667 7209
** Email: J.N.Tenzer@sms.ed.ac.uk Fax: +44 131 667 7209

together with some desired properties, and check in each case whether the desired properties are fulfilled. Then s/he would adopt the design that best met the requirements.

Unfortunately this situation is far from the truth at present. The decision to verify cannot be taken so lightly, for several reasons. It generally requires the building of a special purpose model, which has to be complete in an appropriate sense. (There are a few tools which can work directly with UML models, but most work only with one diagram type, typically state diagrams.) Writing properties which should be fulfilled by a design is a specialist job requiring knowledge of an appropriate logic: even for someone with that knowledge, identifying the properties at an appropriate level of granularity can sometimes be even harder than the design problem itself. One of the issues is that a good design does not only meet the external, customer requirements. It also does so in a clear, understandable way so that the software will be maintainable in future. The designer's choice often involves understanding the implications of each decision and then making essentially aesthetic judgements about them. That is, the desirable characteristics are often a combination of technical, formalisable features and features pertaining to how future human maintainers will most easily think. For this reason, it is unlikely that the process of design will ever be fully automated: verification and synthesis will not replace design. Instead tools should support the human designer in making use of formal technology alongside his/her own skills.

Thus the ideal tool support should do two things: first, it should help make verification in the traditional sense, against explicit requirements, available at any stage of design and even, as far as possible, in the presence of incomplete models; second, it should help the designer to explore consequences of design decisions so that choices can be made even when the criteria for the choices have not been formalised.

2 Games in verification

Our thesis is that mathematical, formal games may be a suitable basis for improving tool support for software design. In this section we introduce such games by means of a simple example, the *bisimulation game* as explained by Stirling in [12]. Such a game is used for checking whether two processes are equivalent under the equivalence relation known as bisimulation. Essentially this captures the idea that two processes can each simulate the other, and that during the simulation their states remain equivalent, so that either process can "lead" at any time.

A bisimulation game is defined over two processes E and F and played by players Refuter (abbreviated R) and Verifier (abbreviated V). The aim of player R is to show that E and F are different from each other while player V wants to prove that E is equivalent to F. At the beginning of the game player R picks one of the two processes and chooses a transition. After that player V has to respond by choosing a transition with the same label from the other process.

This procedure is repeated and each time player R can choose a transition from either process. Furthermore all moves in the game, no matter by which player they were made, can be seen by both players.

If one of the players is stuck and cannot choose a suitable transition, the other player wins the game. In the case that the game is infinite player V wins. A player can play the game according to a *strategy*, which is a set of rules. These rules tell the player how to move and may depend on earlier decisions taken in the game. A strategy is called *winning strategy* if a player wins every game in which he or she uses it.

Figure 1 shows the classic example of two vending machines. Imagine a machine E which has only one coin slot, and a machine F which has separate slots for tea and coffee. E and F are not equivalent because player R has a winning strategy consisting of the following rules:

1. Pick transition 20p from E.
2. If player V responds with the left transition 20p in F, choose selCoffee in E. Otherwise select transition selTea in E.

If player R follows this strategy, player V gets stuck and thereby player R wins the game. Notice that playing the game yields a counter-example, i.e. a particular sequence of moves from which we can see that E and F are not equivalent, which is an advantage of using formal games for verification tasks.

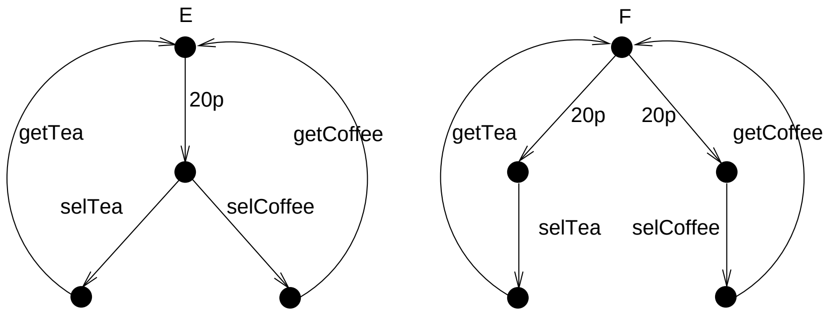

Fig. 1. Bisimulation game for two vending machines E and F

In this game a winning strategy for Refuter can be viewed as a proof that no bisimulation could exist. Similarly, a winning strategy for player V *is* a bisimulation relation, that is, it is a proof that the answer to the question "are these processes bisimilar?" is Yes.

Similarly, model-checking can be expressed as a game: this one is somewhat more complicated, and we will not go into details here (the interested reader is referred to [3]). Two differences are particularly worth noting. First, in a model-checking game it is not necessarily the case that the players alternate turns. The current game position determines who is to move. Depending on the position and the move chosen by a player, the next game position may be one from which the same player is to move again. Second, the winning conditions are more complex than the simple fact that V wins all infinite plays. Instead, there is a predicate on plays which determines which player wins a given play.

In fact, it seems that all verification questions can be expressed as games, although sometimes the game is "trivial" in the sense that a play will always be over after two moves.

Tools such as the Edinburgh Concurrency Workbench[1] exploit this game view of verification questions. The user asks the question; the tool calculates a winning strategy for the game; it then offers to take the winning part in a game against the user. The user finds that, no matter which moves s/he chooses, the tool always has an answer: the user can only lose. This seems to be an effective way of getting an intuition about *why* the answer to the question is as it is.

3 Beyond verification games

When we use verification games, we notice a curious fact. A typical scenario is as follows: we have one process, representing a system, and we want to verify that it is correct. We may choose to do this in any one of several ways. We may develop a second process, which is supposed to stand in some formal relation to our system process; perhaps the two are supposed to be bisimilar or perhaps one is supposed to be a refinement of the other according to one of the many different refinement relations used in process algebra. Alternatively, we may choose to develop a logical formula in a temporal logic such as the modal mu calculus.

In either case, the verification problem can be expressed as a game. The positions of the game incorporate some information about the "current state" of the model, and also some information about the "current state" of the specification. If the two specifications, by process and by logic, intuitively succeed in expressing the same notion of correctness, they correspond to (essentially) *the same game.*

Now, games are rather natural: the idea of defining a game between Verifier and Refuter by describing the valid challenges that Refuter may make, how Verifier may respond, and who wins under which circumstances, is quite easy to grasp. It is easier than understanding the semantics of one's chosen relation between two processes, or understanding the semantics of the modal mu calculus. (Indeed in many institutions, including Edinburgh, this fact is often used to help students grasp the concepts of bisimulation and the semantics of the mu calculus.)

Thus the central idea of this work is to allow the user to define, directly, a verification game. The game should involve the existing design of the system, together with the information that determines whether the design is correct. It should be the case that the player Verifier has a winning strategy for the

[1] http://www.lfcs.ed.ac.uk/cwb/

game if and only if the design is correct. The rules of the game incorporate the challenging circumstances in which the design must work as challenges by Refuter; correct and incorrect behaviour is captured by the winning conditions.

Once we have decided to let the user define a game, we may observe that the game can in fact be defined incrementally. For example, suppose that the design is complete, but that there is only limited understanding of what it means for the design to be correct. (We assume that, as usual, there is no formal specification. Perhaps it has not yet been understood how the informal specification of overall system requirements should be translated down into precise requirements; or perhaps the informal specification is itself incomplete or incorrect. In mainstream business software development, which is our main focus of concern, both are likely to be the case.) In this case the game as initially defined by the user will incorporate only a small amount of information about what it is for the design to be correct: it will be "too easy" for Verifier to win the game. The user should be able to explore the game and improve the rules to make it a better reflection of the correctness of the design. This might include, for example, changing the moves of the game to permit Refuter to make new challenges, or changing the winning conditions so that plays which would have been won by Verifier are won by Refuter in the new game.

At the same time, it is likely that the design itself is too incomplete to permit full verification. The user should also be able to change the game by adding more information about the design.

In order to work more formally with this basic idea, let us begin by defining what is meant by a game in general.

3.1 Game terminology and formal definition

For the purposes of this paper, a game is always played between two players Verifier (abbreviated V) and Refuter (abbreviated R). We refer to players A and B to mean Verifier and Refuter in either order.

Definition 1. *A game G is $(Pos, I, moves, \lambda, W_R, W_V)$ where:*

- *Pos is a set of positions. We use $u, v, \ldots$ for positions.*
- *$I \subseteq Pos$ is a set of starting positions: we insist that $\lambda(i) = \lambda(j)$ for all $i, j \in I$.*
- *$moves \subseteq Pos \times Pos$ defines which moves are legal. A play is in the obvious way a finite or infinite sequence of positions starting at some $p_0 \in I$ where $p_{j+1} \in moves(p_j)$ for each j. We write p_{ij} for $p_i \ldots p_j$.*
- *$\lambda : Pos \to \{Verifier, Refuter\}$ defines who moves from each position.*
- *$W_R, W_V \subseteq Pos^\omega$ are disjoint sets of infinite plays, and (for technical reasons to do with working with abstractions of games) W_A includes every infinite play p such that there exists some i such that for all $k > i$, $\lambda(p_k) = B$.*

Player A wins *a play p if either $p = p_{0n}$ and $\lambda(p_n) = B$ and $moves(p_n) = \emptyset$ (you win if your opponent can't go), or else p is infinite and in W_A.*

Notice that our general definition does not insist that $W_R \cup W_V = Pos^\omega$; that is, it is possible for a play to be a draw. The games we consider in this paper will have no draws, but when we go on to consider abstractions of games (see e.g. [11]) it is necessary to permit them.

Definition 2. *A (nondeterministic) strategy S for player A is a partial function from finite plays pu with $\lambda(u) = A$ to sets of positions (singletons, for deterministic strategies), such that $S(pu) \subseteq moves(u)$ (that is, a strategy may only prescribe legal moves). A play q follows S if whenever p_{0n} is a proper finite prefix of q with $\lambda(p_n) = A$ then $p_{n+1} \in S(p_{0n})$. Thus an infinite play follows S whenever every finite prefix of it does. It will be convenient to identify a strategy with the set of plays following the strategy and to write $p \in S$ for p follows S. S is a* complete *strategy for Player A if whenever $p_{0n} \in S$ and $\lambda(p_n) = A$ then $S(p_{0n}) \neq \emptyset$. It is a* winning *strategy for A if it is complete and every $p \in S$ is either finite and extensible or is won by A. It is* non-losing *if it is complete and no $p \in S$ is won by B. It is* history-free *(or* memoryless*) if $S(pu) = S(qu)$ for any plays pu and qu with a common last position. A game is* determined *if one player has a winning strategy.*

All the games we need to consider are determined, and this is an assumption of this work.

In this paper we focus on the informal presentation of the idea of using and modifying games for software design. It should be clear, however, that there is scope for defining relationships between games in which the existence of a winning strategy for one game implies the existence of a winning strategy for a related game. Some early work in this direction was reported in [11] in the context of verification games. The study of these relationships between games will be important in the context of tools to support games for software design. A simple example is that increasing the number of options open to Refuter – e.g., adding requirements that a design should satisfy – should make it harder for Verifier to win the game: if Refuter had a winning strategy for the original game, the same strategy will work in the extended game.

3.2 How to manage games and their improvement in a tool

A tool will always have a notion of "current game". Part of what the tool should do is to allow the user to play the current game. The tool could operate in two modes:

1. Tool as referee. The user chooses moves both for Refuter and for Verifier (indeed, there might be several users playing against one another). The tool's role is simply to ensure fair play and declare who wins (in the case of a finite play which is won at all).
2. Tool as opponent. The user chooses whether to play Verifier or Refuter and the tool takes the other player's part. If it is possible for the tool to calculate a winning strategy for the current game, then the tool might play this winning strategy, or use it to suggest better moves to the user, as appropriate.

Otherwise, the tool might use random choices and/or heuristics to play as well as possible.

It is not immediately obvious how to incorporate the improvement of games into the tool use. Should improving a game be thought of as part of the game, or as a separate process? It is undoubtedly easier for formal reasoning to regard the improvement of the game as a separate process: then we do not have to worry about what's true of strange composite games in which the rules are changed part way through a play. For practical purposes though, if the user plays a significant number of moves and then realises that there is a problem with the rules which affects how play will proceed from then on, but not what has happened up to this point, it would be annoying not to be allowed to alter the rules and carry on. A suitable compromise is probably to say that a play in which the user changed the rules cannot be formally won by either player. (The tool might announce "You won, but you cheated so it doesn't count".) It is possible to formalise such rule changes as being the actions of a third player, but we have not so far found this very helpful.

4 Example: Incremental definition of simple game

The overall aim of this work is to show how games akin to verification games can be used for software design, and incrementally designed as part of the software design process. Thus we have two aspects to address: the incremental definition of games, and the use of games for software design. In the interests of clarity we devote this section to demonstrating how a simple game, without any special relation to software design, can be defined incrementally.

Our example is deliberately very simple. We will follow a hypothetical user through the design of a vending machine process.

At any stage, there will be both a current system design and a current game, which will include the currently permissible challenges and winning conditions. Our "system design" will be simply a labelled transition system. We initialise this to the LTS shown in Figure 2.

A position in the game may take either of two forms:

1. A state (s_i say) in the system design: Refuter to move. We notate such a position simply s_i.
2. A state s_i in the system design, plus a challenge c from Refuter. Verifier to move. We notate such a position (s_i, c).

Note that it is immediate from the form of the position who is to move: we do not have to specify λ separately.

Initially the winning conditions are that Verifier wins any infinite plays; other than this, we have only the default rules, that any player wins if their opponent is supposed to move but cannot legally do so.

Thus in order to define the game we need to specify, for each state of the system design, what the legal challenges that Refuter may make are, and for each

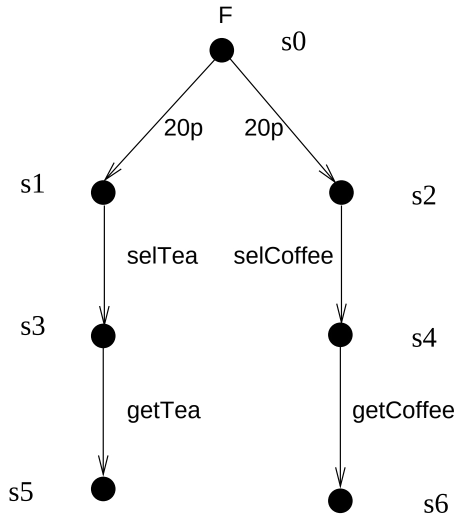

Fig. 2. Initial system design

state and challenge, what the legal responses from Verifier are. Initially, we decide simply to record the requirements that from the initial state, $s0$, it is possible to insert 20p and (maybe many events later) receive tea, respectively coffee. We express this as the challenge to "pick a trace" with certain characteristics.

State	Challenges
s0	pick a trace $s0 \xrightarrow{\mathsf{20p}} \ldots \xrightarrow{\mathsf{getTea}} s_i$
	pick a trace $s0 \xrightarrow{\mathsf{20p}} \ldots \xrightarrow{\mathsf{getCoffee}} s_i$

Note that such a challenge is easy to express in a table as above, and it would be easy for a tool to provide an interface for defining such challenges; however, expressing the property required to hold at $s0$ in a temporal logic is already slightly tricky, because of the need to permit actions other than those named in the traces. For example, the existence of the first trace from state $s0$ is equivalent to state $s0$ satisfying the mu calculus formula $\langle \mathsf{20p} \rangle \mu \mathsf{X}.\, \langle \mathsf{getTea} \rangle \mathsf{T} \vee \langle - \rangle \mathsf{X}$.

After Verifier successfully picks such a trace, we specify that the new game position will be s_i (i.e. system state s_i, with Refuter to move). There are (so far) no legal challenges from any system state other than $s0$, so if Verifier picks such a trace, which will necessarily not end in $s0$, she will win because it will be Refuter's turn to pick a move, but there will be no legal moves.

This is a boring game so far: Verifier easily wins. Suspending disbelief in the inability of the user to understand this, suppose that the user plays the game, taking either Verifier or Refuter's part, and finally is satisfied of this. However, the user realises that the system should not simply stop after the tea or coffee is collected. S/he decides to capture the liveness of the system by permitting

Refuter a new challenge, valid from any system design state, which is simply "pick a transition". At this point, playing the game confirms that Refuter has a winning strategy. So the user refines the system design by merging s_5 and s_6 with s_0. We get the game described by Figure 3 and the table below.

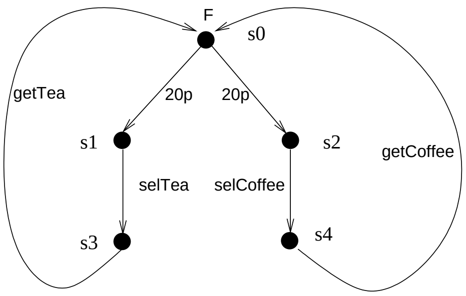

Fig. 3. Revised system design

State	Challenges
s0	pick a trace $s0 \xrightarrow{\mathsf{20p}} \dots \xrightarrow{\mathsf{collectTea}} s_i$
	pick a trace $s0 \xrightarrow{\mathsf{20p}} \dots \xrightarrow{\mathsf{collectCoffee}} s_i$
s_i	pick a transition $s_i \longrightarrow$

We could continue in various ways, with the user gradually improving both the system design and the challenges and winning conditions which (in this example) constitute its specification. So far we have only described adding challenges. Changing the winning conditions is the other main way of altering the game. We could imagine, for example, that instead of winning every infinite play, we might want to say that Verifier won every infinite play on which the number of 20p actions so far was infinitely often equal to the number of getTea actions plus the number of getCoffee actions so far, thus capturing the idea that that the machine should not systematically swindle the beverage-buyer, or vice-versa. (A technical point is that we cannot specify arbitrary winning conditions without losing known determinacy of the game, that is, the existence of a winning strategy for one player: however, in practice, any reasonably specifiable winning condition will fall within the class known to lead to determined games.)

5 Games in the context of object orientation/UML

In this section we address the question of how games for software design using UML may differ from standard verification games; we do not consider the incremental definition of such games. In the next section we consider an example that brings together aspects of the incremental definition considered in the previous section with the games for design issues considered here.

The chief question is what constitutes the "system" being designed, the part analogous to the LTS in the previous example. The design artefact being produced by the designer is the UML model, so obviously the UML model is part of the system. Does it suffice?

In order to explore the dynamic behaviour which is implied by a UML model, we will need to be able to record a "current state" of a prototypical system described by the model, and here we meet interesting issues. The UML model is most unlikely to define a unique system, complete in all detail. This is of course the usual challenge of doing any formal work with an incomplete specification. In the game framework, a promising approach is to allow one or both players, when faced with an incompleteness in the specification, to resolve the nondeterminacy. Different choices about exactly how this is done will yield different games with different strengths. For example, if a transition is guarded by an informal guard which cannot be formally evaluated, perhaps the player choosing the transition may decide to assume that the guard is satisfied. If the game is to be maximally powerful for detecting flaws, we probably wish to make sure that we record enough information to ensure that subsequent choices about whether that guard holds are consistent. For example, we might insist that the player writes a formal specification of the guard, and use this in place of the original informal guard to annotate the model for the rest of the play. On the other hand, a weaker treatment in which the rules of the game did not ensure consistency could still be useful for exploring possibilities. There are many such game-design choices to be made.

More concretely, any particular game for design with UML will draw information from particular parts of a UML model – maybe a complete model, maybe just the parts of it considered most relevant. If the focus is to be on exploring consequences of design decision, it seems likely, though perhaps not essential, that some dynamic diagrams will be involved. The section following gives an example using a simple form of state diagrams, protocol state machines (PSMs)[10] in which transitions are labelled by events, not by actions. (In earlier work we attempted to use more general state machines with both events and actions labelling transitions. However, we found that in normal sequential systems where events and actions are the receiving and sending (respectively) of normal synchronous messages, the UML semantics has anomalies in the presence of recursion. This is discussed in [13], where we propose as a solution that PSMs without actions be used for providing (loose) specifications of the overall behaviour of classes, whilst method state machines are used to define the behaviour of operations.)

Another issue which may arise is the possibility of using parts of the UML for more than simply describing the current system model. For example, it might be convenient to let the user describe certain kinds of challenges as sequence diagrams. We do not consider this further here.

6 Example: definition of a game for UML software design

We now show how a game for software design with UML could be defined. For this purpose we assume that a class diagram and state diagrams for some of its classes are given.

As an example consider the class diagram shown in figure 4 with classes Customer, TravelAgent, Hotel and Flight which models a (very much simplified and not realistic) software system for a travel agency. The most interesting class is Customer which has attributes for the holiday start and end dates of a customer. It contains public methods for booking a holiday, changing a hotel booking and private methods for linking hotel and flight objects.

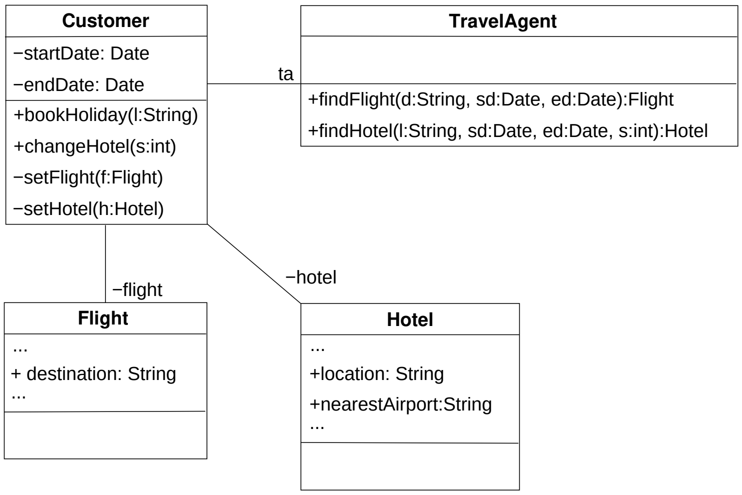

Fig. 4. Example class diagram

In class Customer parameter l represents the desired holiday location and s - which is also used in changeHotel - the requested hotel quality given by the number of stars. The parameters d, sd, ed and s provide values for the flight destination (an airport name), the start and end date of the holiday and, as in Customer, the number of hotel stars.

A protocol state machine for Customer is given in Figure 5. Only the effects of booking a holiday, changing a hotel and setting hotel and flight on the object state are shown in this diagram. For the other classes of the system it is assumed that their objects can only be in a state default which is not changed by any of the methods, i.e. the corresponding state machines have only one state and a loop transition for each method of the class.

For simplicity we only use classes and associations from the class diagram and the (finite) set of abstract states specified in the state machines for the definition

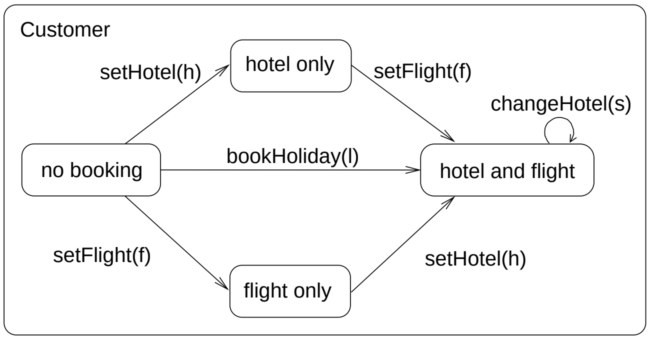

Fig. 5. State machine for Customer

of a state in the UML system design. As we will see later this is restrictive with respect to how the game can be incremented. For a given class diagram CD and a set of state machines SM, where S is the set of all states occurring in SM and S_c the set of states in the state machine for class c, a state in the UML system design consists of

- a collaboration given by a set of objects O and a relation $Links \subseteq O \times R \times O$ respecting the associations in CD where R is the set of role names in CD, including the empty string ε. A tuple (o, r, p) means that o can access p via rolename r.
- a function $state : O \to S$ such that $state(o) \in S_c$ for an object o of class c.
- a callstack $cs = a_1 : \ldots : a_n$ whose elements a_i are of the form $o.m(p_1, \ldots p_n)$ where $o \in O$ is an object of class c and m is a method of c with parameters $p_1, \ldots p_n$ according to the method definition in CD. We refer to o in this context as the *target object* of m.

A position in the game consists of a state in the UML system design and depending on the callstack it is either

- a state in the UML system design with an empty callstack: Refuter to move.
- a state in the UML system design with a non-empty callstack: Verifier to move.

As initial state S_0 for our example we choose

- $O = \{\mathsf{c{:}Customer}, \mathsf{t{:}TravelAgent}, \mathsf{h{:}Hotel}, \mathsf{f{:}Flight}\}$ and $Links_0 = \{(\mathsf{c}, \mathsf{ta}, \mathsf{t}), (\mathsf{t}, \varepsilon, \mathsf{c})\}$. For simplicity the choice of objects is not changed during the game, i.e. we do not consider creation and deletion of objects.
- $state(\mathsf{c}) = $no booking and $state(\mathsf{t}) = state(\mathsf{h}) = state(\mathsf{f}) = $ default
- cs is the empty stack.

Refuter challenges by invoking a method that has public visibility on any of the currently available objects. The choice of the method and its parameters has to conform to the specification by the class diagram, i.e. the method has to be defined in the class of the target object and the parameters have to be accessible and of suitable type. Furthermore the object has to be in a state where an invocation is possible. A challenge is independent of the current linking of objects and the states of t, h, f never change, so table 1 only shows the mapping of *state* for c.

State	Challenge
any state	t.findFlight(d,sd,ed) t.findHotel(l,sd,ed,s)
any state where $state$(c) = no booking	c.bookHoliday(l)
any state where $state$(c) = hotel and flight	c.changeHotel(s)

Table 1. Challenges by Refuter

A challenge by Refuter is pushed on the callstack. Since the callstack is then non-empty that means Verifier has to make the next move. Verifier can respond in two different ways:

– pick a transition in the state machine for the class of the target object of the method on top of the callstack. The call on top of the stack is popped and the state of the target object is updated according to the state machine. The response may also have an effect on how the objects are linked with each other.
– call n other (possibly non-public) methods on objects that are reachable from the target object of the method call on top of the stack. The first new method call is pushed on the callstack. We assume that Verifier's responses are at some point of the first kind, which leads to the call being popped from the callstack. After that Verifier pushes the next method call which is specified in his/her response onto the callstack and again we assume that it is at some point removed from the callstack. This procedure continues until all n new methods calls have been one by one pushed onto and then later popped from the callstack. Finally, after the response has been completed, the call on top of the stack, which is the one that caused the response, is popped.

Notice that in general Verifier and Refuter do not take alternate turns. Verifier responds to the method invocation that is on top of the callstack which can either come from Refuter or Verifier. For our example we could have responses as shown in table 2.

State	Top of callstack	Response
any state	t.findFlight(d,sd,ed)	pick loop transition findFlight in state machine for TravelAgent
any state	t.findHotel(l,sd,ed,s)	pick transition findHotel in state machine for TravelAgent
any state where $state$(c) = no booking	c.bookHoliday(l)	h = c.ta.findHotel(l,c.startDate, c.endDate, 3); f = c.ta.findFlight(h.nearestAirport, c.startDate,c.endDate); c.setFlight(f); c.setHotel(h)
any state where $state$(c) = hotel and flight	c.changeHotel(s)	c.hotel = c.ta.findHotel(c.hotel.location, c.startDate,c.endDate,s);
any state where $state$(c) = no booking or $state$(c) = hotel only	c.setFlight(f)	pick transition setFlight in state machine for Customer whose source is $state$(c) and add (c,flight,f) and (f, ε, c) to links
any state where $state$(c) = no booking or $state$(c) = flight only	c.setHotel(h)	pick transition setHotel in state machine for Customer whose source is $state$(c) and add (c,hotel,h) and (h,ε,c) to links

Table 2. Responses by Verifier

Of particular interest are the responses to bookHoliday and changeHotel which are of the more complicated kind explained above. Notice that by the choice of parameters for the methods some properties are fixed. The last parameter "3" of findHotel in the response to c.bookHoliday, for instance, has the effect that h is always set to a 3-star-hotel. Furthermore the usage of nearestAirport in findFlight within the same response ensures that the flight destination fits well with the chosen hotel.

The specification of responses can be regarded as a strategy for meeting Refuter's challenges. Since the sequence of method calls in the response is pushed on the stack one by one and Verifier has to memorise which ones s/he has already handled, it is a strategy with a history. Tool support for the creation of a game as described here should allow the user to manipulate and refine the strategy for Verifier in a comfortable way. For each system state the tool could display all reachable objects and let the user pick a target object. The user could then proceed by selecting one of the object's methods, which are again displayed by the tool, and stepwise create a valid response. The tool could also contain a functionality to record the chosen strategy in a diagram, such as for example a method state machine (see [13]).

In order to complete our definition of a software design game we have to declare the winning conditions. We assume that a game is won by one player if the other one cannot make a move and that all infinite plays are won by Verifier.

We can now finally play a game, starting with our initial system state S_0. An extract of a play is shown in table 3. The play will be won by Verifier because it is infinite: after the first two challenges Refuter can still continue to make challenges, but Verifier can always respond by simply picking a loop transition.

The table does not record the full system state but only the parts that are relevant for this example (callstack, links, state of c) and the moves of the players. The parameters with which a method is called are sometimes left out in the callstack to save space and because they are specified in the preceding move. Moreover Refuter is abbreviated by R and Verifier by V.

7 Example: Incrementing a software design game

There are several ways in how the example game from the previous section could be incremented. One way is to permit Refuter to call a public method from additional states, for instance we could permit a challenge by changeHotel when the object is in state hotel only. Another possibility is to add a completely new public method to the class diagram, such as for example a method for the cancellation of a holiday booking. In reaction to these increments the user could add states and transitions to the protocol state machines which offer new possibilities of responding to the Verifier.

As soon as we want to increment the game in a more sophisticated manner it becomes clear that working with abstract states is often not enough. For our example we could require that c.hotel.nearestAirport always equals c.flight.destination when c is in state hotel and flight. In order to express this as an additional winning condition which makes it more difficult for Verifier to win the game, attribute values have to be part of the system state. A more detailed object state also leads to increased expressiveness in state machines since we could for instance specify guards whose evaluation depends on the current attribute values. However, introducing a concrete object state has the disadvantage that we in general have to handle an infinite state space.

8 Discussion and future work

This paper has described very early steps in a programme of work. Much remains to be done.

Most obviously, we are developing tools to support the use of these techniques in practice. In an academic environment these tools will necessarily be limited prototypes, but the experience of building them should help in our explorations and dissemination of the ideas.

One possible objection to the proposal is that it is not obvious that there is a *specification* of the system separate from its design. If the specification is incorporated into the game definition, which also incorporates a particular design, does this not lose clarity and prevent a separate evaluation of the specification?

We can answer this objection in a variety of ways. One answer would be that given appropriate prototype tool support we could investigate what games

Callstack	Move	Links	State of c
empty	R: c.bookHoliday(l)	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
c.bookHoliday(l)	V: h=c.ta.findHotel(l,c.startDate,c.endDate,3)	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
t.findHotel(...) c.bookHoliday(l)	V: pick loop transition findHotel	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
c.bookHoliday(l)	V: f=c.ta.findFlight(h.nearestAirport,c.startDate,c.endDate)	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
t.findFlight(...) c.bookHoliday(l)	V: pick loop transition findFlight	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
c.bookHoliday(l)	V: c.setFlight(f)	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
c.setFlight(f) c.bookHoliday(l)	V: pick transition setFlight from no booking to flight only	$\{(c, ta, t), (t, \epsilon, c)\}$	no booking
c.bookHoliday(l)	V: c.setHotel(h)	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c)\}$	flight only
c.setHotel(h) c.bookHoliday(l)	V: pick transition setHotel from flight only to hotel and flight	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c)\}$	flight only
c.bookHoliday(l)	response to bookHoliday completed	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight
empty	R: c.changeHotel(4)	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight
c.changeHotel(4)	V: c.hotel=c.ta.findHotel(c.hotel.location,c.startDate,c.endDate,4)	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight
t.findHotel(...) c.changeHotel(4)	V: pick loop transition for findHotel	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight
c.changeHotel(4)	response to changeHotel completed	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight
empty	...	$\{(c, ta, t), (t, \epsilon, c),$ $(c, flight, f), (f, \epsilon, c),$ $(c, hotel, h), (h, \epsilon, c)\}$	hotel and flight

Table 3. Example play

people build in practice, and see whether there is in fact a clear distinction between design and specification. For example, the challenges in our examples so far can be seen to be separable and could be expressed as a separate specification. Is this typical, or in real examples would the design and specification be more intertwined?

Another answer would be to say that it is not important to have a separate specification at the design level. A specification that is independent of design is a realistic aim at a high level, where the user requirements are being expressed in the user's vocabulary. Inevitably, though, the verification of a design is done against a more detailed, technical specification that always incorporates many assumptions about the design, even when it is presented as a separate document.

9 Related work

There is, of course, a large amount of work on applying verification techniques to UML models. Space forbids a representative discussion of this crowded field; and since our focus here is on highly interactive tool support for design using games as a basis, we will instead discuss the related games-based work.

Two-player games of the kind we consider here have a long history in mathematics (see for example [6]) and informatics (see for example [3, 12]).

In controller synthesis for open systems two-player games where one player (control) has a strategy which enforces the system to behave according to a given specification independent of what the other player (environment) does are of interest. Finding a winning strategy (the controller) for player control is known as the control problem.

The control problem has been studied for different kinds of open systems such as, for example, discrete systems [9] and systems in reactive environments [7]. The system specification can be given as linear or branching time temporal logic formula. Results on the complexity of solving the control problem depend on the chosen logic and the kind of system that is investigated. In case that a controller exists for a given combination of system and specification it is also of relevance whether the controller is finite and how big it is [9].

Games in this context are often infinite and there are some classes of specifications which are of particular interest. An example for such a kind of specification or "game objective" is that eventually an element of a given target set of system states has to be reached. Some frequently occurring game objectives, corresponding winning strategies and complexity results are presented in [14].

The relation between games in control and verification is pointed out in [1], where a translation of a game objective from control into a fixpoint formula written in the mu-calculus such as used in verification, is defined.

A closely related area is that of system synthesis. Given a specification, the task of constructing a system that satisfies it is known as the synthesis problem. It can be formulated in terms of a game between environment and system, and a winning strategy for such a game represents the desired system. If such a strategy exists we say that the given specification is realisable. Like the control problem system synthesis and realisability have been examined for different kinds of logics and systems, such as for systems in reactive environments [7] and distributed systems [8].

Another kind of two-player games called combinatorial games is used to represent, analyse and solve interesting problems in areas such as complexity, logic, graph theory and algorithms [2]. The players move alternately and cannot hide information from each other. The game ends with a tie or the win of one player and lose of the other. An example for a combinatorial game with a lot of literature within the area of artificial intelligence is chess.

A different, and more complex style of game appears in economics. Here, rather than simply winning or losing, a player receives a variable payoff. A game is played between two or more players with possibly conflicting interests, i.e. a move which leads to a better payoff for one player may have a negative effect on another player's payoff. Thus a strategy which optimises one player's payoff can depend on the strategies followed by other players.

The issue of payoff optimisation can be considered under different circumstances: the players can have complete, partial or zero knowledge of each others moves. It is also possible for players to collaborate in order to gain a better payoff for a group of players rather than for the individual player.

These kinds of games reflect situations in economics such as, for example, competition between different companies, and were first introduced within this context in [15]. Since then a large amount of further work has been published in this area. It would be interesting to explore the applicability of this kind of game to software design, but for now we prefer the simpler games described here.

The work of Harel et. al. on "play-in play-out scenarios" [4], [5] has a similar flavour to our work, and is motivated by similar concerns about the interactivity of tools to support design. Play-in scenarios allow the capture of requirements in a user-friendly way. The user specifies what behaviour s/he expects of a system by operating the system's graphic user interface (GUI) – or an abstract version thereof – which does not have any behaviour or implementation assigned to it yet. A tool which is called the play-engine transforms the play-in of the user into live sequence charts (LSCs), which are used as formal requirements language. The user does not have to prepare or modify the LSCs directly but only interacts with the GUI.

LSCs are a powerful extension of message sequence charts (MSCs). In contrast to sequence diagrams – the variant of MSCs which is part of UML, and which is implicitly existential – they can be either existential or universal. A universal LSC defines restrictions that have to hold over all system runs, while an existential LSC represents a sample interaction which has to be realised by at least one system run.

Using play-out scenarios we can verify whether a set of LSCs – created by play-in scenarios or in any other way – meets the system requirements. Thereby the user feeds the GUI with external environment actions rather as though s/he were testing the final system. For each user input the tool computes the response of the system on the basis of the LSCs in terms of a sequence of events which are carried out. The system response is called a superstep and it is correct if no universal LSC is violated during its execution.

The task of finding the desired superstep can be formulated as a verification problem. In [5] a translation of LSCs into transition systems which allows the usage of model checking tools for the computation of the supersteps is given. Similarly model checking can provide the answer to the question whether an existential LSC can be satisfied.

This approach differs form ours in that its focus is on capturing and testing the requirements while we are mainly interested in helping the user to design a system. Thus play-in play-out scenarios do not aim to help in defining intra-object behaviour, as our games do, but remain on the higher level of interaction between objects and environment. Since our work concentrates on UML we use the diagram types provided by it, i.e. UML sequence diagrams instead of the more expressive LSCs.

10 Conclusion

We have suggested the use of games as a driving metaphor for highly interactive design tools to support designers working in UML. We have proposed that the user of such a tool should define incrementally a game which captures not only the evolving state of the system design but also the properties that the design should satisfy; the tool should support the user both in informal explorations of the resulting game at each stage, and in verification, that is, in the finding of a winning strategy. We have given simple examples of how a design tool based on this idea might operate. We hope that eventually this work may contribute to allowing mainstream business software developers to take advantage of verification technology without giving up their existing incremental development practices.

Acknowledgements We are grateful to the British Engineering and Physical Sciences Research Council for funding (GR/N13999/01, GR/A01756/01).

References

[1] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. From verification to control: Dynamic programs for omega-regular objectives. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS)*, pages 279–290. IEEE Computer Society Press, 2001.

[2] A.S. Fraenkel. Selected bibliography on combinatorial games and some related material. *The Electronic Journal of Combinatorics*, (DS2), 2002. Available from http://www.combinatorics.org/Surveys/ds2.ps.

[3] E. Grädel. Model checking games. In *Proceedings of WOLLIC 02*, volume 67 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[4] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[5] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, pages 378–398, November 2002.

[6] W. Hodges. *Model theory*, volume 42 of *Encyclopedia of Mathematics*. Cambridge University Press, Cambridge, 1993.

[7] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive enfironments: control and synthesis. In Catuscia Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107. Springer, August 2000.

[8] O. Kupferman and M.Y. Vardi. Synthesising distributed systems. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*. IEEE Computer Society, June 2001.

[9] P. Madhusudan and P.S. Thiagarajan. Branching time controllers for discrete event systems. *Theoretical Computer Science*, 274(1-2):117–149, 2002.

[10] OMG. *Unified Modeling Language Specification version 1.4*, September 2001. OMG document formal/01-09-67 available from http://www.omg.org/technology/documents/formal/uml.htm.

[11] Perdita Stevens. Abstract interpretations of games. In *Proc. 2nd International Workshop on Verification, Model Checking and Abstract Interpretation, VMCAI'98*, number CS98-12 in Venezia TR, 1998.

[12] Colin Stirling. Model checking and other games. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea, July 1996.

[13] Jennifer Tenzer and Perdita Stevens. Modelling recursive calls with UML state diagrams. In *Proc. Fundamental Approaches to Software Engineering*, 2003. To appear.

[14] W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science, STACS '95*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, 1995. Springer.

[15] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, third edition, 1953.