GUIDE: Games with UML for Interactive Design Exploration

Jennifer Tenzer Perdita Stevens

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

Abstract

In this paper we present our design tool GUIDE, which allows the user to explore a design in UML interactively by playing a game. The game incorporates both the design model and a specification of what it means for the design to be correct. The central idea of this approach is that the designer can increment the game during a play and gradually add more detail to it. Specification and design are refined by repeated plays of the game. The designer stops playing when design and specification are detailed enough for his purpose and match each other. The interactive game approach helps to cope with incompleteness and informal definition of UML models, which make strictly formal verification techniques difficult. The designer may resolve these problems when they arise during a play or let the GUIDE tool determine how the play should proceed. We discuss the potential impact of GUIDE and tools like it on software development.

Key words: Interactive software design, UML, formal games

---

1 Introduction

The Unified Modeling Language (UML) [28] is a standard language for modelling the design of object oriented software systems. There exists a variety of UML tools, most of which focus on support for drawing UML diagrams and generating code from them. None of the currently existing UML tools provides much support for experimentation and evaluation of different design options. The hard tasks of deciding whether a design fits to the specification, improving it and comparing it to

[1] jennifer.tenzer@gmail.com
[2] Perdita.Stevens@ed.ac.uk (corresponding author)

other solutions still has to be accomplished by the human modeller without much guidance from a tool.

Our design tool GUIDE (**G**ames with **U**ML for **I**nteractive **D**esign **E**xploration) [10] aims at filling this gap. The foundation for this tool are *exploration games* which are an extension of two-player games as used in formal verification. The GUIDE tool supports the user in defining an exploration game, playing it in different roles, and incrementing the game definition.

An exploration game involves four different participants: two players called Verifier and Refuter who compete with each other, a Referee, and an Explorer. The game contains the design model of the system under consideration and a specification of what it means for the design to be correct. The aim of Verifier is to show that the design fits the specification, while Refuter tries to find a flaw in the design. All moves are performed in several stages. The responsibility for each stage can be assigned to one of the players or the Referee. The responsibility assignments allow the game participants to resolve non-determinacy during a play when they are faced with incompleteness or informality of the UML model. The Explorer has the power to modify or *increment* the game definition at any point during a play. The incrementations can affect both the design and the specification of the system and may improve the chances of winning for one of the players.

The exploration game framework can be applied to UML in many different variants. A game variant has to specify how exactly the exploration game is defined, i.e. what its positions, moves, and winning conditions look like, which parts of the UML model are used for the definition, how the responsibilities for the players are assigned and how the Explorer may increment the game. Game variants can either be used to check one design solution with respect to desired or undesired properties, or to compare two different designs.

The GUIDE tool does not expect the user to have any knowledge of formal games or verification and helps him to set up a game on the basis of a UML design model. Once the game is defined the modeller can start a play. The part of the Explorer always has to be played by the human designer, who may play any number of the other game participants in addition. Taking on the role of Verifier or Refuter provides the modeller with a specific perspective and goal for the design exploration. As Refuter he will concentrate on detecting flaws in the design, as Verifier he will attempt to demonstrate that the design is correct. GUIDE makes the moves for all game participants that are not played by the user, evaluates the winning conditions and guides the user through a play.

GUIDE is itself a framework which can be extended in various ways. Additional game variants can be implemented, new kinds of expressions for winning conditions and responsibilities may be defined, and alternative algorithms for computing the moves that are made by the GUIDE tool can be integrated.

The remainder of this paper is organised as follows. Section 2 gives an overview on related work. In Section 3 we argue that games are a suitable foundation for an interactive UML design tool like GUIDE which supports design exploration. Section 4 contains an informal summary of the exploration game framework and a description of an example game variant. The core functionality of the GUIDE tool is presented in Section 5. In Section 6 we explain the architecture of GUIDE and some of the technical solutions that have been chosen. Section 7 provides a brief insight into how the GUIDE tool can be extended and customised. Section 8 discusses possible future developments of the GUIDE tools and its successors, how we see this new approach fitting into practical software development, and the issues that will arise. In Section 9 we conclude.

This paper is a revised and extended version of [24]. Section 8 is entirely new; the other sections have been revised in minor ways.

2 Related work

The idea of using games as basis for a tool that allows design exploration has first been introduced in [22] on a general level. Exploration games are extensions of two-player verification games as described in [8] and [26]. A precise definition of the formal exploration game framework appears in [25]. An application of exploration games to UML activity diagrams has been presented in [23].

On a very abstract level the whole software development process can be regarded as an interactive game. In [2] agile software development is considered as a "cooperative game of invention and communication". The software developers play the game with two goals in mind. The primary goal is the delivery of a working software product, and the secondary goal is to prepare the next game. Non-cooperative games can be used as metaphor for design reviews [13] or inspections [4], where one player defends the design, and the other players try to find a flaw in it. An adversarial attitude has also been successfully adopted by the Black Team [3] for testing software systems. As far as we know games have not yet been applied in a more formal way to the software design process or used as foundation for tools. In particular there exist no approaches where games are used to explore a UML model as introduced in this paper.

The work of Harel et al. on "play-in play-out scenarios" [11], [12] 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 he expects of a system by operating the system's 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. 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 which fulfils the requirements. Thus play-in play-out scenarios do not aim to help in defining intra-object behaviour, as our exploration games do, but remain on the higher level of interaction between objects and user.

When UML tools are considered, the commercial tools Rhapsody [21] and Real Time Studio [20] are most related to our work, because they allow the designer to "play through" and examine state machines by animation. However, these tools require very precise design models. They do not permit informally specified constraints or interruptions of the animation process in order to add more information or change the design. There is also no notion of a system specification in these tools against which the design is checked. The user merely observes the animation and decides on his own whether the system behaves as he expected.

The tools HUGO [14] and vUML [15] allow the designer to check general properties of UML state machines, such as the possibility for deadlocks. HUGO additionally verifies whether desired (or undesired) behaviour specified by UML interaction diagrams can be realised by state machines. Both tools translate a UML model which has been created by an external tool into a formal model which is then used as input for the model checking tool SPIN. They differ from our GUIDE tool in two respects. First, they require a UML model as input which is precisely defined. Informal guard conditions, undefined object attributes and non-deterministic state machines are not permitted. Second, they concentrate on the evaluation of a UML model, while our exploration games are focused on interactive modification of the design.

3 Motivation

Games have been chosen as foundation for GUIDE because they offer several advantages. First, playing a game does not require background knowledge in formal methods and is fairly intuitive. The basic idea of two players Verifier and Refuter who compete against each other to prove the correctness of the design or detect a flaw in it is easy to grasp. Since the GUIDE tool is targeted at mainstream software designers, this has been an important factor for choosing the tool foundation.

Another advantage of games with respect to tool support is that they are an *interactive* technique, which allows the modeller to influence a play. The progress of a play is determined by the decisions of the players who react to each other's moves, and by the Referee. These roles can be played by the designer, who may also increment the game definition as Explorer during a play.

Such an incremental development of a game is the central idea of this work. Since the design and the specification of the system are both incorporated in the game, the designer can increment each of those parts. For example, suppose that the design model is complete, but that there is only limited understanding of what it means for the design to be correct. 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. The game as initially defined by the modeller may incorporate only a small amount of information about what it means for the design to be correct: it may be “too easy” for Verifier to win the game. The modeller should be able to improve the game to make it a better reflection of the correctness of the design. This might include, for example, 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 modeller should also be able to change the game by adding more information about the design.

This idea fits very well to iterative and incremental software development processes where the specification is updated along with the artifacts describing the system design. Thus games are a very natural choice with respect to how software is usually developed. The designer increments the game while it is being played and has to ensure that the game is challenging for the two players. A sequence of such incrementations, which may be part of different plays, is an exploration of the design and its specification. Even though the incrementations of the game may be beneficial for one player, the modeller is not necessarily biased. For example, during one play the designer may first increment the game such that it becomes easier for Refuter, and later explore a different part of the game which increases Verifier’s chances of winning.

The advantage of permitting game incrementation during a play is that the designer does not have to start a new play from the beginning but can continue the improved game from the current position. However, the disadvantage is that an incrementation may invalidate the play. For example, moves of the old game may not be part of the incremented version anymore. Even if all moves of the play still exist after the exploration, there is no guarantee that the players will select the same moves as before when the play is repeated from the beginning. Thus a winning strategy for the old game does not necessarily work in the incremented game and may have to be adapted.

Figure 1 illustrates how different plays of game *G0* initiate different explorations. The new versions of the game are further improved which leads to new variations and combinations of design and specification. Incrementing the game will in most cases correspond to adding more detail to its parts. That means both specification and design become gradually more precise. Hence this approach provides a smooth

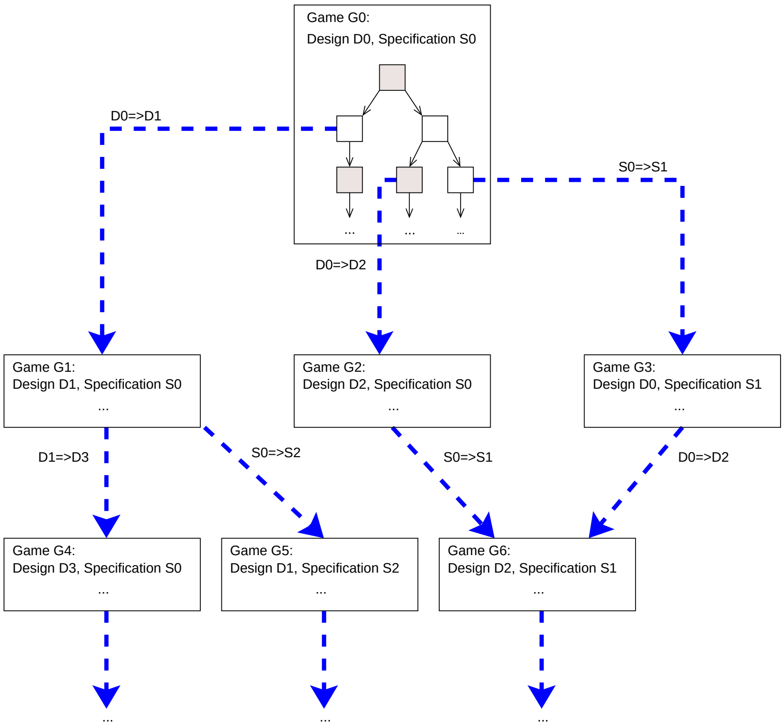


Fig. 1. Repeated game incrementation

progression from informal exploration of decisions to full verification. This has the potential to lower the commitment cost of using formal verification. The designer can stop exploring if he believes that design and specification are precise enough for his purpose. Playing the game again from the beginning without further incrementations verifies the current design against the current specification. The designer may still have to provide information during the verification process if the game is too incomplete at some points, which is very likely with UML design models as basis. However, it is not necessary to improve the game so far that the complete system can be verified formally without the help of the designer.

Finally, games may have another advantage: games which people play in their free-time are played for fun. The question is whether games that are played with the purpose of exploring design decisions are also to some extent entertaining. If so, a game-based design tool like GUIDE may actually make the work of software designers more enjoyable.

4 Exploration games

In this section we describe exploration games informally and introduce an example variant which is based on UML state machines. However, the concepts of exploration games are very general and can easily be applied to other UML diagram types or combinations thereof. A full formal definition of the exploration game framework and several applications to UML appears in [25].

An exploration game is defined by a game arena, an initial position, responsibility assignments for the different stages of each move, game settings, winning conditions for the players, and possible incrementations by the Explorer. The game arena consists of positions and moves. It draws information from particular parts of a UML design model – in our example from UML state machines and class diagrams. A move may have a precondition and parameters. The preconditions of the move do not have to be specified formally. If they are based on constraints in the UML model, such as guards in state diagrams, they are very likely to be formulated in natural language. The participants in an exploration game are the two players Verifier and Refuter, the Referee and the Explorer. A move is selected in the following steps:

(1) Precondition evaluation. The set of legal moves from the current position is determined by declaring which informally specified preconditions are assumed to be true.
(2) Choice of move shape. A move shape is a set of moves which have the same source position, name, precondition and parameter signature. The moves belonging to a move shape only differ in their parameter values and target positions. Only legal move shapes may be selected in this move step.
(3) Parameter provision. The move shape is reduced by fixing the parameter values for the move. If only one single move is left, the next move has been selected and the last step is obsolete.
(4) Resolution of non-determinism. There may be more than one move which belongs to the selected move shape and has the chosen parameter values. These moves only differ in their target positions and one of them has to be picked as next move.

In contrast to formal models that are normally used as basis for verification games, the UML model is most unlikely to define a unique system, complete in all detail. In the exploration game framework the game participants resolve any kind of non-determinacy during a play. The responsibility for performing the four different move steps are assigned to Refuter, Verifier or the Referee. In contrast to the players the Referee does not benefit from decisions about the progress of the play.

The game settings can be general or specific for one variant. They are used for two purposes. First, they fix an interpretation of the UML semantics where necessary. UML contains “semantic variation points” for some of its features which provide a degree of freedom in its interpretation. Since the possible moves in a game depend to a great extent on the UML semantics, the designer has to decide how such semantic variation points should be treated. Second, the game settings may impose restrictions on how the game is played and incremented. For example, the game settings can specify a move limit and thus determine the maximum length of a play. Furthermore the game settings define whether the Explorer may increment

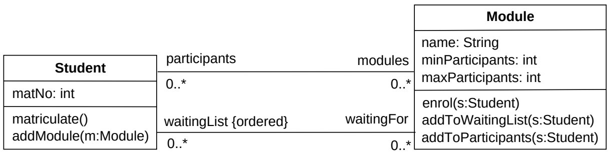


Fig. 2. Example class diagram

the game in a way that violates the play history.

The Explorer's goal is to make the game more precise and keep the balance between Refuter and Verifier. He is allowed to adjust the difficulty of the game for the players by incrementing the game definition during a play. Apart from incrementing the game the Explorer may also backtrack in the play history or change the current position. The role of the Explorer is always played by the human designer who has enough knowledge about the system to choose sensible incrementations and make the model more precise. Additionally the modeller may take on other parts in the game, such as, for example, the role of one of the players to examine the design from a particular perspective.

Incrementations are defined with respect to the UML model for all parts of the game where this is possible. In this case the designer does not work directly with the game definition, but increments the game indirectly via changes in the UML model. As we shall see, the designer can also modify the game directly, for example, by changing the winning conditions; this kind of incrementation corresponds to modifying the specification of the system, as opposed to the design model. Thus during the course of play, both the design and the specification can be improved. After he has performed such an incrementation as Explorer, the play is continued according to the new game definition.

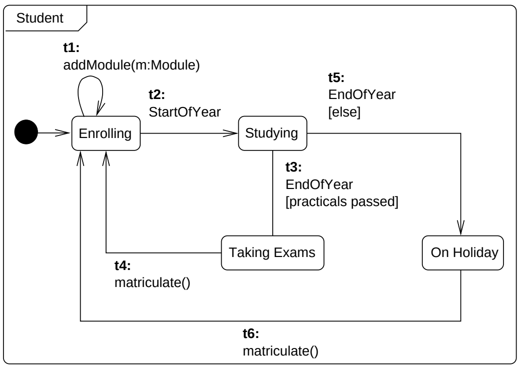
4.1 Example of an exploration game

For our example game variant we assume that a UML model for a university course registration system consisting of the class diagram in Figure 2 and the two state machines shown in Figure 3 is given. Notice that the guard conditions in the state machines are informally specified. Whenever we refer to one of the players we use the female form for Verifier and the male form for Refuter.

During a play of this game a collection of objects is observed with respect to state changes. The objects constitute the system which is verified. Here we consider a student *Joe* and a module *CS1*.

Positions: A position represents a snapshot of the system and consists of the following parts:

- For each object



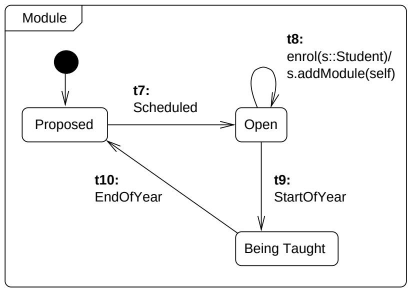


Fig. 3. State machines for *Module* and *Student*

  · a state configuration,
  · and an event pool.
- A set of parameters which are in scope at the position.

The positions where all event pools are empty belong to Refuter, all others to Verifier. At the initial position all objects are in their default states and the event pools are empty.

Moves: In this example the moves from Refuter's positions correspond to generating events. All events which refer to an operation in the class diagram are regarded as call events and are targeted at a specific object. When a call event is generated a target object has to be specified and the event is put into its event pool. The call events in our example are *matriculate*, *addModule*, *enrol*, *addToWaitingList* and *addToParticipants*. All other events that occur in the state machines are considered as signal events. They are broadcast to all objects and put into their event pools when generated. If an event is parameterised, suitable parameter values have to be provided for it.

A move from one of Verifier's positions corresponds to firing a set of state machine transitions according to the UML semantics. For each object the first event in its pool is dispatched and an enabled state machine transition is fired, if there is any. An event which does not trigger any transitions is discarded as specified in the UML standard [28][p.492]. Whether a transition is enabled or not depends on the evaluation of the guard condition at the time when the event occurs. Thus the legality of a move is determined by the evaluation of the guards, which are considered as preconditions of the move.

If a transition is fired and an effect is attached to it, another event is generated. The new event is put into the appropriate event pool and the object completes its transition. This corresponds to the idea of *asynchronous* actions in UML, where the object does not have to wait until the new event has been processed before it reaches the next stable state.

Figure 4 shows some of the positions and moves of our example game. Refuter's

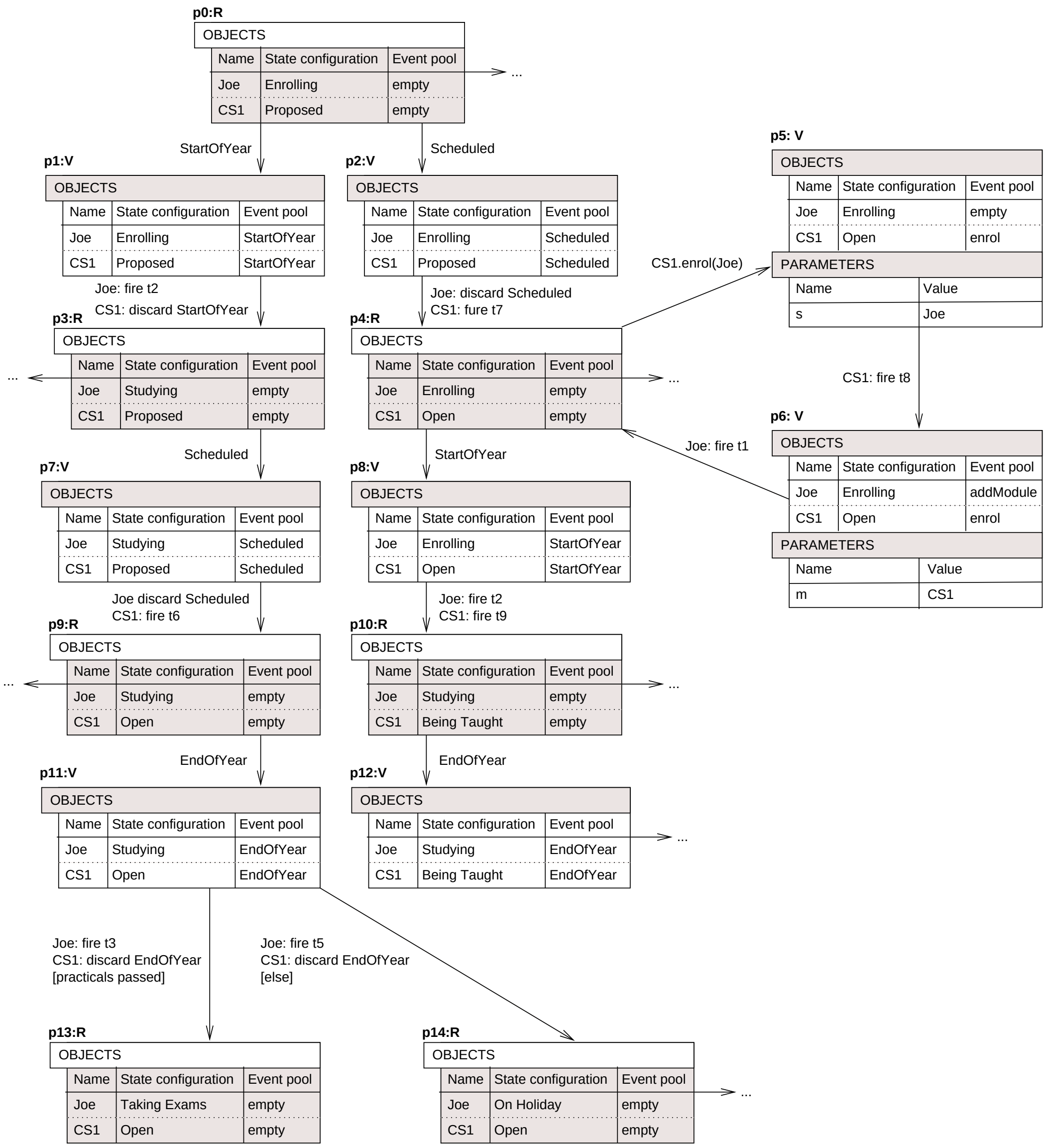


Fig. 4. Positions and moves in the example game

positions are shown in grey-shaded rectangles and are labelled by "R", Verifier's are labelled by "V". The position shown as *p0* is the initial position.

Winning conditions: Refuter wins a play if a position is reached where *CS1* is in state *Open* and *Joe* is in state *Taking Exams*. He also wins all plays which end at a position belonging to Verifier because no further moves are possible. Verifier wins all other plays.

Because of the informally defined guard conditions at the transitions it is unclear for some of the moves whether they should be regarded as legal or not. For this example we assign the responsibility for deciding about the legality of moves to Verifier. Furthermore we assume that the players select move shapes at their own positions. They also provide parameters and resolve non-determinism for all moves emerging from the positions belonging to them.

Responsibilities:

- Verifier decides whether an informal precondition of a move is assumed to be true.
- Verifier and Refuter fulfil all other tasks at their own positions and for the moves emerging from these positions.

Game settings: According to the UML semantics an event is always discarded if it does not trigger a transition. However, this semantic rule has been criticised as being inappropriate in some cases, especially where the events are the calling of methods; in practice, many UML users use the alternative rule that call events may not be discarded. Here we introduce a setting which specifies whether the official rule should be applied to call events. For this example game we assume that discarding call events is forbidden.

Incrementations:

- Add or delete a state transition.
- Add or delete a state. If a state is deleted, all transitions which emerge from or lead to it are also deleted.
- Add or delete an event or operation.
- Change the guard condition at a transition.
- Change the winning conditions.
- Change the responsibility assignments.
- Change the game settings.

Notice that the first four incrementations all operate directly on the parts of the UML model which have been used as basis of the game definition for our game variant. The last three incrementations are more general and transferable to other game variants which do not use UML state machines and class diagrams as foundation.

The game variant which has been introduced here is a simple example of an application of exploration games to UML in order to illustrate the approach. We have abstracted from details, such as, for instance, how exactly the winning conditions and responsibilities are defined. Since the positions of the game only record the abstract states of the objects, it is difficult to evaluate sophisticated guard conditions. For example, it is not possible to decide whether the number of students who are enrolled in a module exceeds the maximum number of participants. In order to evaluate a precondition like this we would have to define a more complex game variant whose positions contain the objects' attribute values and links. For our example game preconditions whose evaluation is undefined because the positions do not contain enough information are treated as if they were informally defined. The game participants have to decide whether their evaluation is assumed to be true or false.

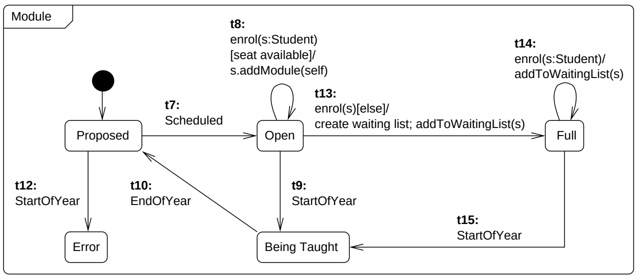


Fig. 5. State machine for *Module* after exploration

4.2 Example plays without exploration

We examine some example plays of our game, consisting of the positions shown in Figure 4. Here we only consider plays which do not involve incrementations by the Explorer. Assume that Refuter challenges by *StartOfYear* at position *p0*. Verifier has only one choice to respond to this challenge and moves to *p3*. From there Refuter's next moves are *Scheduled* and *EndOfYear*. Again Verifier has only one possibility to react to these events and the play continues from *p3* via *p7* and *p9* to *p11*. Here Verifier has for the first time the chance to actually select a move. Before she can do this she has to decide which of the moves are legal. The guard condition *else* at transition *t5* indicates that exactly one of the transitions triggered by *EndOfYear* must be enabled, i.e. the guard conditions are mutually exclusive. That means only one move emerging from *p11* can be declared as legal. Verifier realises that she will lose the play if she moves to *p13*, because this position fulfils the winning condition for Refuter. Hence a rational choice for Verifier is to declare that the move to *p14* is legal. If she selects this move, she can avoid losing the play. In fact, if Verifier applies this strategy every time position *p11* is visited, she can win all plays. That means Verifier has a winning strategy and the design is considered to be correct with respect to the specification under the current game definition.

Verifier wins this variant of the game so easily because she can always avoid firing transition *t3*. At an early stage of the design phase, it may be useful to give Verifier so much power. This game variant is suitable for playing through the desired scenarios of the system without being forced to consider preliminary parts of the design or special cases. A variant of the game which is maximised for detecting flaws is to allow Refuter to decide about the validity of informal preconditions. In the example play described above Refuter will then declare that the move to *p14* is illegal. Thus Verifier is forced to move to position *p13* where Refuter wins the game. If the Referee is responsible for evaluating informally defined preconditions, the outcome of each play is uncertain. None of the players has a safe winning strategy because the decision of the Referee at *p11* determines who can win.

4.3 Example plays with exploration

In this section we use the example game from Section 4.1 to show how a game is repeatedly changed by incrementations. During the explorations the state machine for *Module* will be altered. The resulting state machine after all explorations considered here is shown in Figure 5. In the descriptions of the example plays we will again refer to positions which are part of the arena excerpt shown in Figure 4.

Assume that Refuter challenges by *StartOfYear*, *Scheduled* and *EndOfYear* from the initial position. Verifier applies her winning strategy and moves to *p14*. At this point the designer realises that the game is too easy for Verifier: she is permitted to forbid ("dodge"!) a reasonable challenge, *EndOfYear* when practicals have been failed. This discovery urges him to increment the game as Explorer such that the disadvantaged player has a better chance of winning. The Explorer backtracks to position *p11* and changes the responsibility assignments such that Refuter is responsible for the evaluation of all informal preconditions. The play is continued with these modifications and Refuter declares that the critical move to *p14* by which Verifier can avoid losing the game is illegal. Now Verifier has no other choice except to move to *p13*, where Refuter wins the game.

The designer decides to play the incremented game again from the beginning to see how the players move under the changed circumstances. It becomes obvious that it is now Refuter who can always win the game easily by declaring that the move to *p14* is illegal. The modeller realises that Verifier loses because she cannot respond adequately when Refuter raises *StartOfYear* before *Scheduled*. There are several alternatives of how he can improve the game as Explorer such that Verifier has better chances of winning. Here we consider the following three possibilities:

(1) Backtrack to *p1*, add a new state to the state machine for *Module* and add a transition *t11* from *Proposed* to the new state which is triggered by *StartOfYear*. With these changes Verifier must fire *t11* for *CS1* in response to *StartOfYear*. The state of object *CS1* changes to the new state and the critical state combination is avoided as the play continues.

(2) Backtrack to *p1*, add a new state *Error* and add a new transition *t12* from *Proposed* to *Error* with trigger *StartOfYear*. Then change the winning conditions such that Verifier wins the game if state *Error* is reached. Verifier must fire *t12* for *CS1* in response to *StartOfYear*. After that move the winning condition holds and Verifier wins the play.

(3) Backtrack to *p7* and change the winning conditions such that Verifier wins if Refuter challenges with *Scheduled* immediately after *StartOfYear*. With these changes position *p7* becomes a winning position for Verifier, because the two events have been generated in the forbidden order.

The first two options indirectly extend the set of moves for Verifier in the arena of the game. If the first solution is chosen, Verifier has the chance to circumvent a position which leads to a win for Refuter in the old game by using one of the new moves. The last two possibilities involve changes of the winning conditions such that Refuter is discouraged from making the critical sequence of moves which causes problems for Verifier. Here we assume that the Explorer chooses the second alternative. If played without exploration, the improved game is always won by Verifier.

We can continue in various ways, with the designer gradually improving both the system design and its specification. A way of incrementing the game which has not been considered yet is to alter the guard conditions at transitions. For example, the designer can refine the conditions under which *t8* may be fired by adding a guard condition *seat available*. When Refuter challenges by *Scheduled* and *m.enrol* from the initial position, Verifier now loses the play if Refuter declares that *seat available* does not hold. Verifier cannot find a transition which is triggered by *m.enrol* and the game settings forbid her to discard call events. That means there are no moves possible from the current position which belongs to Verifier, and Refuter wins the play.

A simple way to improve Verifier's chances of winning the game is to change the game settings such that call events may be discarded. Another approach which preserves the strictness of the game settings is to add more detail about what should happen if there is no seat available when a student attempts to enrol to the model. One solution is to add the student to a waiting list. In order to follow this approach the Explorer adds a new state *Full*, and transitions *t13*, *t14* and *t15* as shown in Figure 5 to the diagram. After this exploration Verifier has again a winning strategy for the current game.

4.4 Significance of explorations

Explorations can be regarded as possible answers to design questions. Sometimes very concrete design questions arise during a play. For example, the fact that Verifier loses the game at position *p13* after the first incrementation leads to the following questions;

- What should happen if the year starts before the module is scheduled?
- Is the sequence *StartOfYear*, *Scheduled* legal or out of the system's scope?

Often it may be enough that the play evolves in a way which was not expected by the designer to make him think about certain design issues. For example, the designer may realise during a play that a feature which he assumed to be part of the system is missing both in the specification and the design. The designer discovers this flaw because he misses corresponding moves in the play. In our example the designer could for instance ask himself whether a module can be cancelled at any time.

In other cases the idea for a new exploration is not triggered directly by a play, but comes up when the designer thinks about how to improve the game for one of the players. For example, attaching a guard condition to *t8* is just one possibility to improve the chances of Refuter that the designer decided to follow.

It is also possible to think of Explorer's incrementations as independent proposals for system changes which are not inspired by plays of the exploration game at all. On this more general level exploration games can for instance be used to explore the evolvability of a system. In this case the incrementation is hypothetical and serves to show that a game can be extended as desired without breaking functionality that was present in the initial design.

5 Playing a game with the GUIDE tool

GUIDE is a prototype tool written in Java which is based on the exploration game framework. Before a game can be set up with GUIDE, a UML model has to be created. GUIDE does not contain a visual editor and requires an .xmi file which is compliant to the UML1.4 metamodel [27] from an external UML tool as input. We have chosen an old version of the UML metamodel, because most UML tools still use this version. The test models for GUIDE have been created with the community edition of the Poseidon tool [19].

Figure 6 shows the main window of GUIDE after a project with the example game of Section 4.1 has been opened. There is a menubar on top of the window, a model tree which displays the UML model on the left hand side, a game panel with six different views on the right, and a message window at the bottom. The views in the game panel are controlled by the game tabs on top of the panel. Each view shows a particular part of the game definition. The definition of a new game in GUIDE consists of the following steps using the menubar:

(1) Open the UML model game by *File→Open Model*, which displays a file dialogue.
(2) Set the arena type by *Edit→Arena type*, which opens a dialogue where the user can select one of the types that are currently available in GUIDE. The arena type specifies the game variant and determines which parts of the UML model are used within the game. Once the user has performed this step, the model tree is displayed.
(3) Set the initial position by *Edit→Initial position*. The dialogue which is invoked by this operation is customised for the arena type that has been selected. For the example game variant which is considered in this paper the designer first enters object names and then selects classes from the UML model for

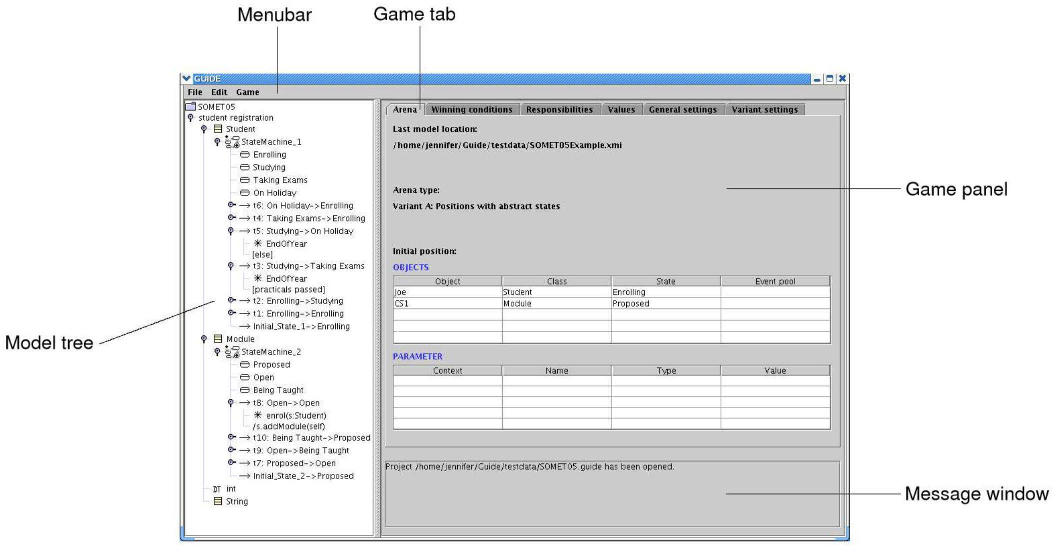


Fig. 6. GUIDE main window

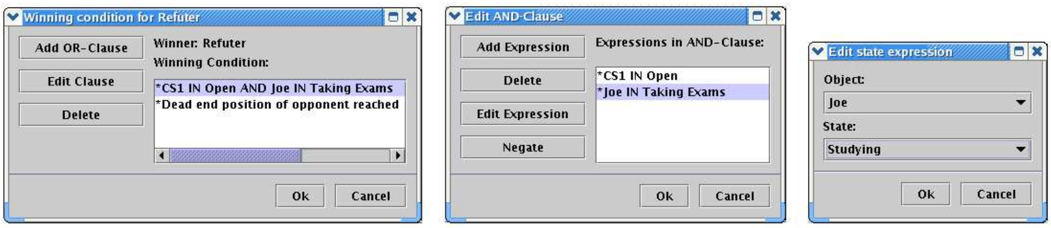


Fig. 7. Dialogues for editing an expression in Refuter's winning condition

them. After that he chooses a state from the appropriate state machine for each object. Since the user cannot enter arbitrary class and state names, the initial position is always valid. The user can also specify parameters which are known at all positions of the game and their initial values. The parameter values have to be updated manually during a play and allow additional information about the system to be recorded.

(4) Define the winning conditions for the players by *Edit→Winning condition Refuter* and *Edit→Winning condition Verifier*. Most of the dialogues which are displayed on selecting these menu items are the same for each variant because GUIDE contains a general expression framework. A typical sequence of dialogues is shown in Figure 7. A winning condition consists of one or more AND-Clauses. Each AND-Clause is a conjunction of expressions, which can be applicable to all variants, such as for instance *Dead end position of opponent reached*, or to just one game variant. The only dialogue that is variant specific in Figure 7 is the last one in the sequence where the state expression is defined. This expression may be part of a winning condition because it "fits" the selected arena type.

GUIDE uses default values for all other parts of the game definition which can be changed by the user. The responsibilities of the two players and the Referee are

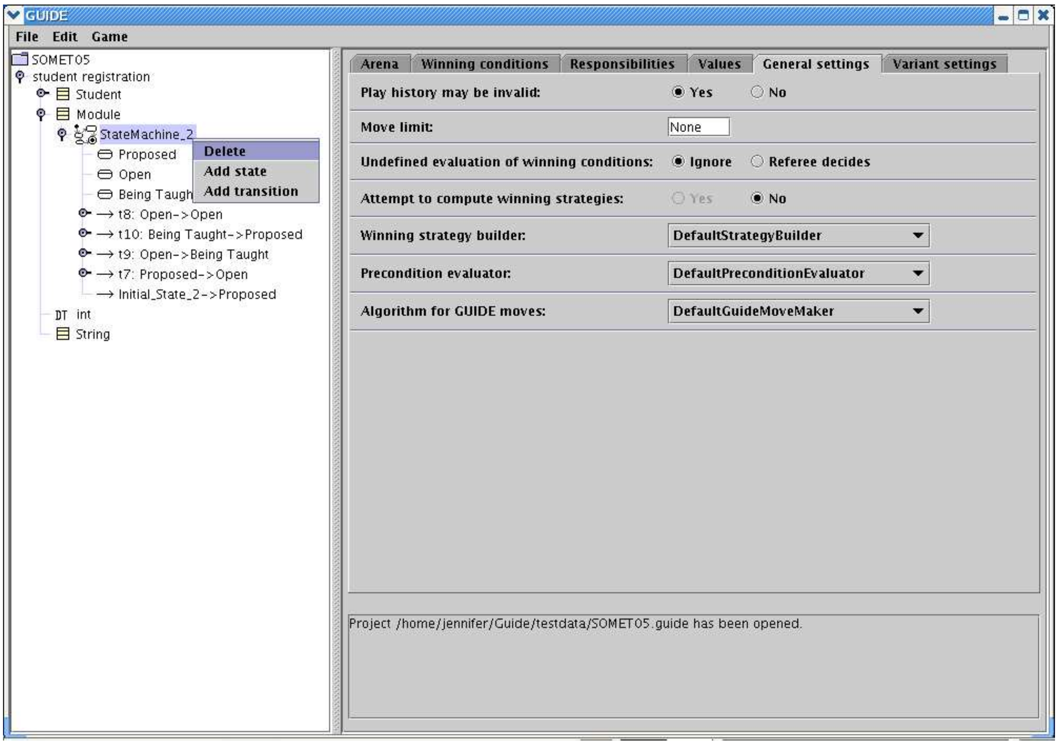


Fig. 8. GUIDE main window with general settings tab and a context menu for a state machine node

edited via dialogue sequences that are very similar to those for the winning conditions. Moreover the game settings may be modified directly in the corresponding game tabs. The tab for the general game settings, which are the same for all game variants, is shown in Figure 8. The same figure also shows a context menu in the model tree which pops up when the user clicks on a node representing a state machine with the right mouse button. The model tree contains context menus like this for all other node types. Each item of the context menus opens a dialogue which allows the user to increment the corresponding part of the UML model at any time. In the *Values* view the user can specify which values should be used for parameters that are not objects but primitive data types. This is a mechanism to enforce finiteness of the number of moves that emerge from a position, which we do not consider further here.

Notice that a game cannot be defined in arbitrary order. For example, it does not make sense to define a winning condition before a UML model, which serves as foundation for the game, has been opened. Therefore some of the menu items in GUIDE are not always enabled and the user is forced to perform the different steps of the game definition in a reasonable order.

Once the user is satisfied with the game set up, he can start to play by *Game→Play*. A dialogue as shown in Figure 9 appears and asks the user for the distribution of tasks during the play. After that the play window which contains the current position and play history is displayed and the players start to move. Figure 10 shows

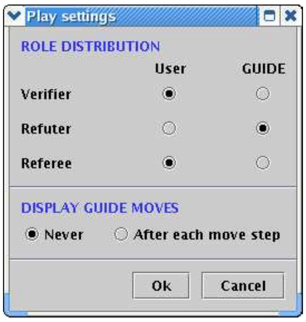


Fig. 9. Preparation of a play

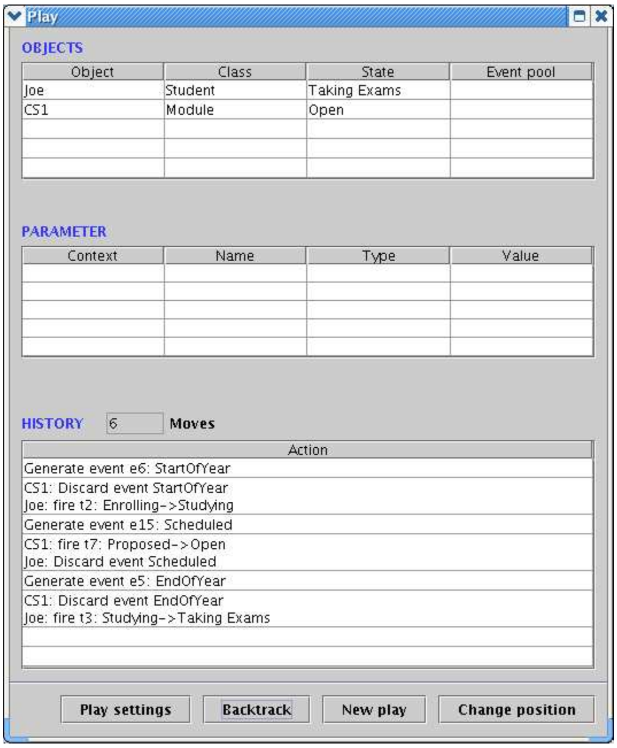


Fig. 10. Play window showing a play without exploration

one of the plays that was discussed in Section 4.2 in the play window of GUIDE. When the last position of this play is reached, GUIDE discovers that Refuter's winning condition holds. The play is finished and GUIDE announces that Refuter is the winner.

Each move consists of the four stages that were explained in Section 4.1. The algorithm for GUIDE moves, which is selected in the general settings tab, computes how GUIDE performs these move steps in the role of the players or the Referee. If the settings specify that GUIDE should attempt to compute winning strategies for the players, the algorithm can use these strategies during its computation. The user can choose in the dialogue for the preparation of a play whether the move steps that are made by GUIDE should be displayed. If the user has to perform a part of

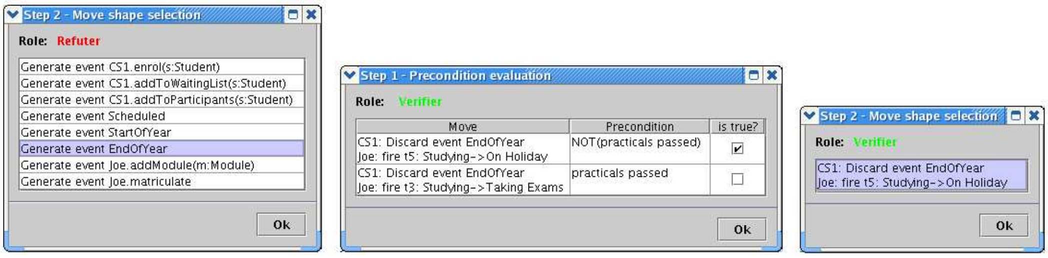


Fig. 11. Dialogues for move steps

the move, he is asked for input by different dialogue windows. Figure 11 shows a sequence of move steps in the example game where the user plays both Refuter and Verifier.

At any time during a play the user can increment the game definition as specified by the formal exploration game framework. He is not forced to answer the dialogues that are displayed immediately and can use the context menus in the model tree, the items of the *Edit* menu and the features in the game settings tabs to increment the game. Furthermore the *Change position* button in the play window permits modifications of the current position during a play. Each change of position results in an incrementation of the game arena because the new position becomes the target of the last move. Whenever the modeller performs an incrementation, a short description is displayed in the play history. Figure 12 shows the history of the example play in Section 4.3 during which incrementations have been performed.

The *Backtrack* button in the play window allows the modeller go back to an earlier position of the play which can be selected in the history. Backtracking includes the incrementations of the game, i.e. the game definition is also changed back to an earlier version. For example, clicking the *Backtrack* button in the play window shown in Figure 12 would restore the old version of Refuter's winning condition and remove transition *t12* from the UML model. After backtracking the play would continue at the second item of the play history, which is selected.

6 GUIDE Architecture

This section is intended as an overview on GUIDE's architecture and concentrates on the most important classes and methods. Figure 13 shows the package structure of GUIDE.

Package *io* consists of classes for saving and loading GUIDE projects and UML models. As storage mechanism for the UML model we have used the Metadata Repository (MDR) by NetBeans [16]. The MDR project is aimed at persistent storage and manipulation of metadata. It is based on the Meta Object Facility (MOF) standard [18]. The solution based on MDR has two main advantages. First, differ-

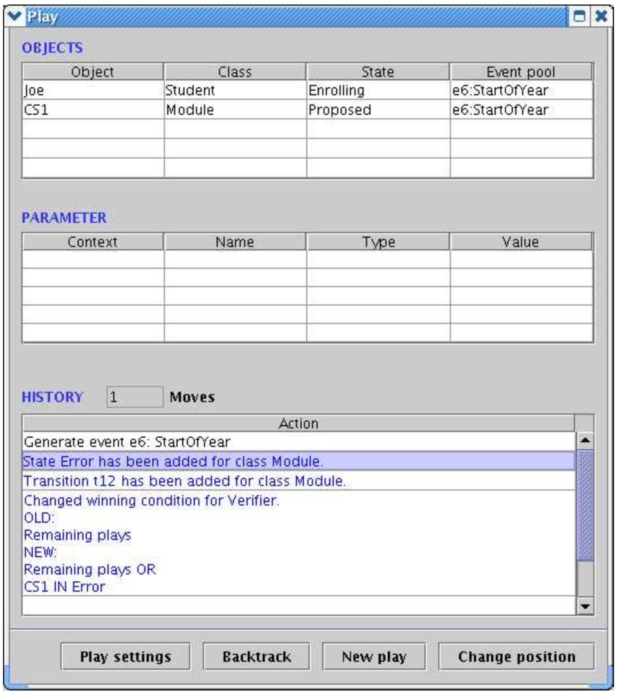


Fig. 12. Play window showing a play with exploration

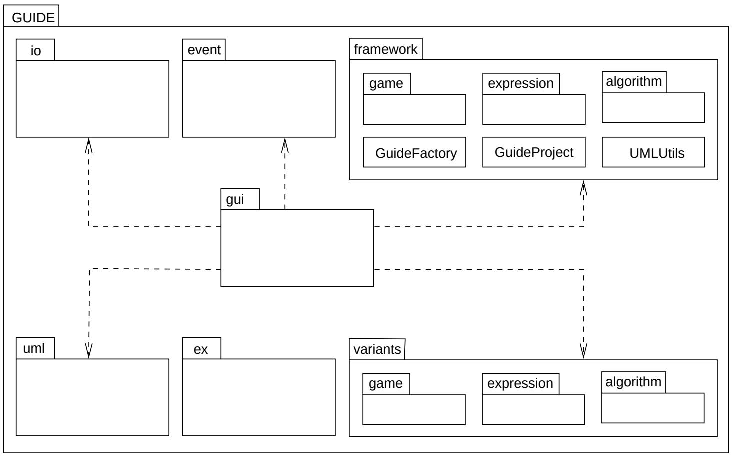


Fig. 13. The package structure of GUIDE

ent UML tools can be supported because MDR is not bound to a particular XMI version or tool-specific saving format. That means the XMI output of any UML tool that is compliant with the UML1.4 metamodel can be read into GUIDE. Second, MDR's interfaces for accessing and manipulating the UML model have reduced the amount of code that had to be written.

The classes in *event* specify the actions which are invoked via the GUI, and package *ex* contains exception classes. The *uml* package provides classes for the UML elements that appear in the model tree. The classes in the *gui* package are Java Swing components, such as for instance a file dialogue, which are customised and used by different parts of the tool. The main frame of the GUI is also located in

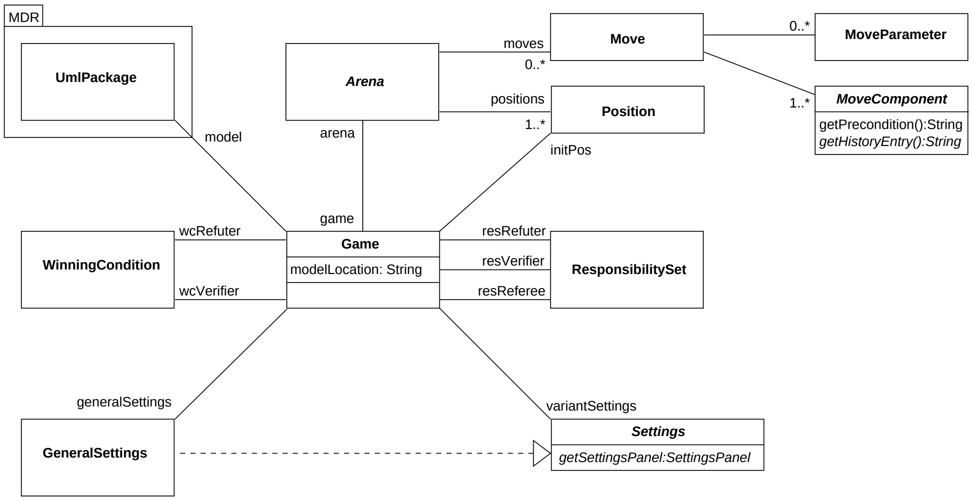


Fig. 14. GUIDE game framework – Game structure

the *gui* package. All other GUI components are stored further down in the package hierarchy in subpackages of the *game*, *expression* and *algorithm* packages.

The most interesting packages in the context of this paper are the *framework* and *variants* package. As shown in Figure 13 they both have the same structure. The *framework* package provides general classes and interfaces which can be refined and realised in the *variants* package. The class *GuideFactory* consists of methods for finding classes in the *variants* package which implement particular interfaces or are subclasses of *framework* classes. Java's reflection mechanism is used for this purpose. The relation between the *framework* and *variants* package is further discussed in Section 7.

Figure 14 shows the part of GUIDE which contains the game structure. As in the formal exploration game framework, a *Game* consists of an arena, an initial position, winning conditions, responsibility assignments and game settings. The UML model of a game is given by a *UmlPackage* which is a class in MDR. The abstract classes *Arena*, *Settings* and *MoveComponent* are specialised by concrete classes in the *variants* package. The concrete subclass of *Arena* for the game variant which we have considered in this paper is based on UML state machines. As in the example game discussed in Section 4.1, the *Settings* subclass refers to the UML state machine semantics. The concrete subclasses of *MoveComponent* for the game variant considered here stand for generating events, firing transitions and discarding events.

The part of GUIDE's framework package which is essential for playing a game is shown in Figure 15. The *GameEngine* is invoked by the GUI and controls the play. It is linked to a *Play* which consists of *ExplorationPosition* instances. Each exploration position is a tuple of a *Position* and an *Incrementation*. If a move is made during a play, an exploration position with *null* as incrementation and the target position of the move is added to the history. In case the game is incremented, a

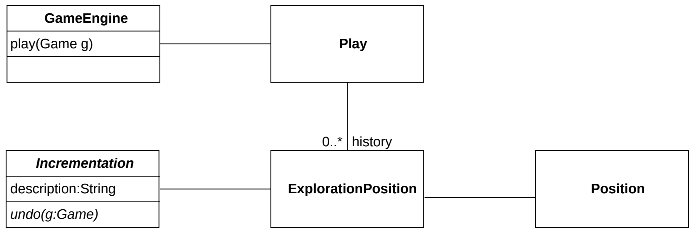


Fig. 15. GUIDE game framework - Playing a game

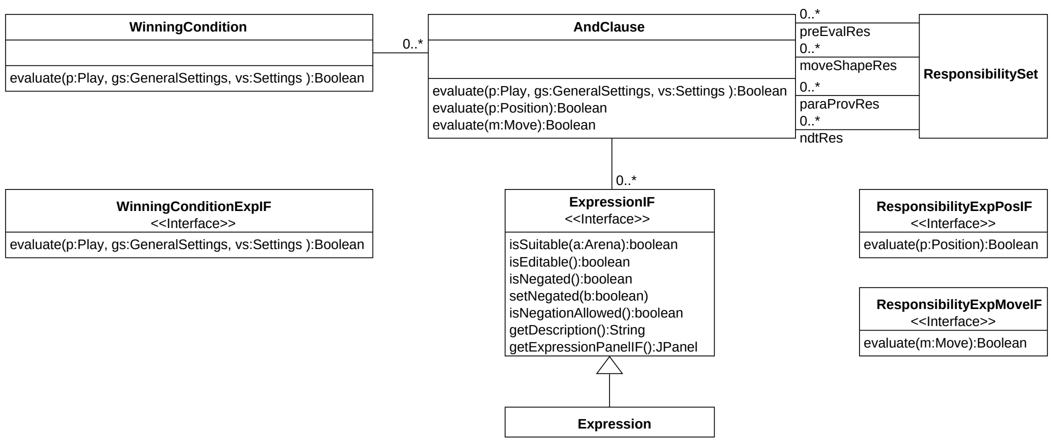


Fig. 16. GUIDE expression framework

concrete instance of the abstract *Incrementation* class and *null* as position constitute the new exploration position.

There already exist all necessary concrete subclasses of *Incrementation* in GUIDE, which represent incrementations of the model, winning conditions, responsibilities, and game settings, respectively. They all implement the abstract *undo* method which restores the game that is provided as parameter to the state before the incrementation has happened.

The general expression framework of GUIDE is shown in Figure 16. Both *WinningCondition* and *ResponsibilitySet* are associated with collections of *AndClause* instances. When a winning condition is evaluated, the result is true if one of its AND-Clauses is true. The *evaluate* method of *AndClause* which has a play, general settings, and variant specific settings as parameters is invoked for each AND-Clause to perform the evaluation. Within this method the expressions that constitute the clause are cast to *WinningconditionExpIF* and evaluated. The GUI ensures that an AND-Clause which is part of a winning condition only consists of expressions which implement this interface and are suitable for the arena of the game. If all expressions are evaluated to true, the evaluation of the AND-Clause and of the winning condition also return true.

Instances of *ResponsibilitySet* are evaluated in similar fashion, but make use of two different evaluation methods. Which one is chosen depends on the type of responsibility that is evaluated. A *ResponsibilitySet* consists of four different AND-Clause collections, which correspond to the four responsibilities in the exploration game framework. The responsibilities for precondition evaluation and move shape selection are evaluated over positions, while the ones for parameter provision and resolution of non-determinism are evaluated over move shapes.

GUIDE provides several subclasses of *Expression* which implement the interfaces of the expression framework and define expressions that are usable in all game variants. An example of a general winning condition expression is *Move limit reached*, which refers to the move limit that may be set as part of the general settings. This expression is represented by a constant in a subclass of *Expression*.

Further subclasses can be created for variant specific types of expressions. For the variant which is based on UML state machines, a class *StateMachineExp* has been added to the *variants* package. The class implements all interfaces except for *ResponsibilityExpMoveIF* in Figure 16. Hence an object of this class can be used in winning conditions and for the definition of responsibilities which are based on positions, but not for responsibilities referring to moves.

Notice that there is no separate class which represents move shapes in GUIDE. A move shape is simply a move whose parameters and target position are ignored. Another important point is that the *evaluate* methods in the expression framework return *Boolean* values. That means they can return true, false or a *null* object. The latter is used to indicate that the evaluation has been undefined.

The contents of the algorithm package are shown in Figure 17. Two interfaces and one abstract class are associated with *GeneralSettings*. They define the algorithms that are used by the tool to make moves and evaluate preconditions during a play, and for computing winning strategies. There are two methods for computing a winning strategy in *StrategyBuilderIF*. The first one computes a fresh winning strategy, while the second method adapts an existing winning strategy during a play. The latter is needed to react to game incrementations and decisions by the game participants. The *evaluate* method in *PreconditionEvaluatorIF* uses a return value of type *Boolean* to cater for undefined evaluations.

Most of the methods in *GuideMoveMaker* require a parameter that specifies which role GUIDE should play. The only exception is the last method which refers to the undefined evaluation of winning conditions. It is used to determine whether the play may be continued or one of the players wins in case the game settings specify that the Referee is responsible for this decision and GUIDE acts as Referee. For the provision of parameter values by method *provideParameterValues* a mapping from types to possible values has to be specified.

GUIDE provides simple default implementations of the interfaces and abstract class in the *algorithm* package. The default precondition evaluator always returns *null* to indicate that the evaluation is undefined and has to be performed by the responsi-

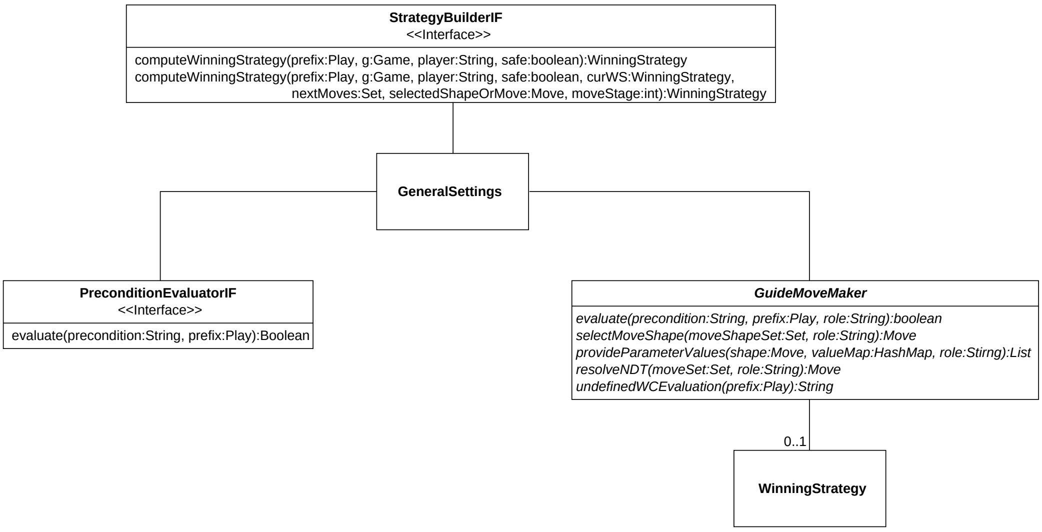


Fig. 17. GUIDE algorithm framework

ble game participant. The subclass of *GuideMoveMaker* which is used by default computes GUIDE's moves on the basis of the associated winning strategies if there are any. Otherwise GUIDE performs the different tasks which it is responsible for randomly. The default strategy builder only attempts to compute a winning strategy if a move limit has been set for the game. It searches for a winning strategy by building the arena up depth first until the move limit is reached.

7 Extensions of GUIDE

The *GuideFactory* class is used to search the *variants* package for realisations of interfaces and subclasses of the *framework* classes while GUIDE is running. The classes that are found are instantiated, and can be selected to be part of the tool via the GUI. This solution permits extensions of GUIDE by adding new classes to the *variants* package. Since the *GuideFactory* attempts to instantiate the classes in this package, new classes should always have a default constructor with no parameters.

In order to define a new game variant, a subclass of the abstract class *Arena* has to be created within the *game* subpackage. The definition of a new variant also requires new subclasses of *Position* and *MoveComponent*, and a panel for displaying and editing positions belonging to the new arena. Moreover methods which compute the next moves emerging from a position and customise the model tree are left abstract in *Arena* and have to be implemented. Any user-defined game variant which follows these rules becomes available for selection in the dialogue that is displayed by *Edit*→*Arena type*.

Another part of GUIDE that can be extended is the *expressions* package. A new kind of expression should be implemented as subclass of *Expression*, and realise at least one of the interfaces for winning conditions or responsibilities shown in Figure 16. The interface *ExpressionIF* contains a method for deciding whether an expression is suitable for an arena and one which yields a panel for editing expressions of this type. The *Expression* class provides default implementations for these methods which should be overridden by its subclasses. The solution for the implementation of the first method in class *StateExpression*, which is part of our example variant, was to define another interface *StateExpressionIF*. This interface specifies which methods should be provided by a suitable arena. The *isSuitable* method in *StateExpression* checks whether the arena implements this interface. All expressions which are suitable for the arena of the game become available for selection in the expression dialogues of the GUI.

It is also possible to define customised algorithms which specify how the GUIDE tool evaluates preconditions, makes moves and computes winning strategies. Classes which contain new algorithms should implement at least one of the interfaces shown in Figure 17 or be subclasses of *GuideMoveMaker*. They must be put into the *algorithms* subpackage of *variants* to be found by GUIDE and are then displayed as options in the general settings panel, where the user can select which algorithms should be used.

8 Adoption issues

The body of this paper has focused on introducing the prototype GUIDE tool as a means of illustrating the use of games to assist the creative software design process. We have briefly discussed future work developing and evaluating the tool. In this section, we discuss how we see GUIDE, and tools like it, eventually contributing to creative software design. We will summarise the issues which we aim to address, current software design practices, and the problems and benefits we anticipate in introducing the GUIDE tool or its successors to real software development environments.

8.1 Supporting creativity in software design

Software design is fundamentally a creative activity. It is not possible to draw up exhaustive rules about how to make good software designs. Current best practices include:

- techniques such as design patterns (see e.g. [6]), to help developers learn how to recognise and solve commonly occurring problems;
- techniques to find and fix bad design after the fact: these include both design review meetings, and individual practices like Extreme Programming's "merci-

less refactoring"[1], which aim to eliminate bad design soon after it has been introduced and before it has a chance to cause many bad knock-on effects.
- heuristics, rules of thumb and coding standards that *tend to encourage* good design: for example, we can use catalogues of "bad smells" [5] to detect possible flaws, such as duplicated code or complex monolithic methods;

Fundamentally, these techniques amount to educating the designer, and/or fixing his mistakes after the fact (although still before the design has been used in a system released to a customer). There is still not much in the way of support to do a good job of a particular design in the first place. It is this gap which GUIDE and its successors aim to fill.

We assert that the fundamental reason why software design is hard to do well is that it generally involves developing a deep intuitive understanding of what will happen *under many different collections of circumstances*. Designers have to take into account, for example, the range of possible data input to the system; the possible interactions the system has with humans or other computer systems; the possible concurrency effects, e.g., race conditions. Even harder, they have to try to design the system in such a way as to minimise the rework necessitated by changes to customer requirements, the technical or business environment, etc., whether the changes occur during initial development or after deployment. It is relatively easy to "walk through" a potential design in a single concrete environment, but to understand the implications of one's design decisions overall, one has in effect to explore a tree of possibilities. Changing a design decision in order to eliminate a problem at one position in the "tree" may introduce problems elsewhere. Creating a design from within a design space is akin to picking the best tree from a forest. In practice, designers undoubtedly use "fast and frugal" heuristics [7]: that is, one part of the skill of the good designer is to identify which sets of circumstances need to be thought about explicitly, while another is the ability to do the mental walkthroughs efficiently and accurately.

One approach to this problem is to attempt to automate the entire forest search. Automating the connection between a UML model and the code, and then using automated testing techniques, is one way to automate the part of the search that concerns different input values. In certain restricted circumstances, it may be possible to define the design space and the desiderata for the design formally, and have a tool search for the best design. This is a promising approach to some aspects of the problem, most notably the management of concurrency issues. However, it can never solve the entire problem. To see why, let us reconsider the fact that the design needs to be robust against future changes to the customer's requirements and other changes to the hardware, software, human and organisational context.

This is one of the reasons why high-level design is, and will remain, a fundamentally human-driven activity. No design can be robust against *every* possible change to the customer's requirements. Indeed, attempts to maximise flexibility, for example by using design patterns at every opportunity, can backfire by leading to excessively complex and hence error-prone designs.

This danger of excess complexity is the rationale behind the Extreme Programming (XP) mantra “You ain’t gonna need it”. “Agile methodologies” [2], of which XP is an example, react against what is called Big Design Up Front (BDUF), the practice of trying to anticipate all possible requirements and requirement changes. They advocate an environment in which, rather than try to make one design which can meet all needs and anticipate all changes, one ensures that the design is kept clean and simple enough that a skilled developer can quickly adapt it to actual changes when they arise. They tend to reject building detailed models in languages such as UML, in favour of focusing on the code and doing everything possible to facilitate informal development among a small team of developers.

The agile methodology movement is an important step forward for software development methodologies, but it works best for software which can be developed by a small team and where every part of the software can be modified as occasion demands. Where this is not the case, the original designer may have to anticipate future changes – to do *some* BDUF – to maximise the chance that required changes can be made without needing to alter hard-to-alter parts of the system. For example, if a system involves some software deployed to customers and some held at a central site, it might be straightforward to alter the latter, but very awkward and expensive to alter the former. In general, the skilled designer discovers enough about the environment in which the system operates to make good guesses about where it is worthwhile to introduce a little extra complexity, for example a layer of abstraction, in order to localise changes that are quite likely to be required. Almost always, which changes will actually be required is unknown at the time of design.

We envisage that a tool such as GUIDE could be useful in cases where a certain about of Big Design Up Front is required, e.g. because modifying the design later will be impractical. Where agile methodologies reject BDUF because historically it has been almost impossible to do well, we aim to make it easier to do, when it must be done. The creativity still comes from the designer, but the tool can help the designer to explore the consequences of a decision.

8.2 Future developments which would aid adoption

Before GUIDE or a tool like it could be adopted in actual software design practice various advances need to me made. Some of these are intellectually challenging: others are “just” usability enhancements. As usual with UML tools, there is the difficulty that the usability of the tool can be very seriously adversely affected by rather “uninteresting” factors: to develop the tool to the point where one can with full confidence identify the shortcomings of the new technique, as distinct from the shortcomings of the prototype implementation, would require an amount of development work not available in academia. Nevertheless we plan over the next year to undertake initial evaluation attempts. In this section we discuss the main areas of enhancement needed in the medium term.

8.2.0.1 Smoother integration with UML drawing tools The current prototype does not include a full UML editor, but can read in a UML design in XMI format from another tool. Modifications can be made to the design while using GUIDE; GUIDE can write a new XMI file incorporating the modifications which can be read into the designer's UML tool. Ideally what one would want, however, is full integration, allowing changes to be made in one's favourite UML editor and immediately seen by GUIDE.

8.2.0.2 Smoother integration with code and tests More ambitiously, a future GUIDE-like tool could build on the closer integration between models and code allowed by some UML tools to take into account design decisions recorded only in code, as well as those reflected in the UML model. There are intriguing possibilities in the area of incorporating the growing test suite, perhaps by using test scenarios to derive preferred plays that the tool would definitely explore. New tests might be derived from the rules of the game, as one way of recording the designer's developing understanding of the requirements.

8.2.0.3 Integration with an OCL tool Currently the evaluation of move preconditions is defined to be the responsibility of the players, because the move preconditions are derived from constraints in the model, which are uninterpreted in GUIDE, and indeed often written in natural language. Alternatively, constraints can be written in UML's built-in textual specification language, Object Constraint Language (OCL). GUIDE could then be coupled to a tool that can evaluate OCL constraints, such as for example the USE tool [29]. If the designer commits himself to use OCL in the UML model, some of the move preconditions may then be evaluated automatically and do not require interaction with the game participants during a play. One disincentive to pursuing this approach is that OCL has not been as successful in practice as the diagrammatic parts of UML. Designers tend to find it hard to read and write OCL constraints, and OCL also has some remaining semantic problems. For this reason many designers who want to write precise constraints write them in the target programming language; evaluating them automatically then becomes part of the integration with code task mentioned above.

8.2.0.4 More powerful strategy search algorithms The current GUIDE tool incorporates a basic algorithm for searching for winning strategies, which requires that the exploration depth be limited. In the field of verification (e.g. model-checking) games which were the inspiration for this work, there is a literature on algorithms for strategy search, including work on infinite state spaces. Applying this work to the context of design games will not be straightforward, but could be beneficial. An alternative, perhaps complementary, approach is to investigate heuristics that would allow us to calculate a "good" strategy for a player, even if we could not guarantee that it was a winning strategy.

8.2.0.5 Automatic specification extraction Using a GUIDE-like tool the user improves both the software design and the understanding of its specification. The modified design can be extracted as XMI, but there is currently no systematic way to extract the new specification in a way which would allow it to be used easily to modify a specification document. Using the game variants currently provided, it would be quite straightforward, and perhaps useful, to provide such a mechanism. More interesting and challenging would be to investigate the extraction of a specification from a game definition if more flexible game definition is permitted, as discussed above.

8.3 Broader issues

Several of the issues discussed in the previous subsection are related to the central theme of raising the abstraction level at which development decisions are made from code to models. This is a chicken-and-egg situation: in part, GUIDE-like tools should be an enabler of this change; in part, they depend upon that change happening. The topic is a hot one in software engineering: it is the focus of the OMG's Model Driven Architecture movement [3], and of the Agile Modelling movement [4]. Models are currently most successful, in the sense of remaining key artifacts throughout the development process, where essentially they function as high-level code: the model records so much about the system that the generation of code can be fully automatic [17]. The underspecification normal to models is removed, which reduces the cognitive burden on the designer. The disadvantages are that visual modelling languages may not be as well-adapted as code to recording detailed decisions easily, and that we may be in danger of losing the advantage of separating architectural concerns from details of implementation. We hope that by helping the designer to understand the implications of design decisions even when the design is incomplete, GUIDE-like tools can make a wider range of models cognitively manageable.

An important technical issue is that the more one relies on a model the more it matters that there should be a precise common understanding of what the model

[3] `http://www.omg.org/mda/`
[4] `http://www.agilemodeling.com/`

means – what it implies about the code, and (just as importantly) what it leaves open. UML is currently deficient in this respect. Its semantics is imprecise in many important ways, and the method by which its semantics is defined is very hard even for specialists to understand. Arguably this is, in practice, an inevitable result of the increasing complexity and scope of the language, and it might be better to make use of much smaller, simpler languages. Microsoft's Domain Specific Language tool work [9] is interesting from this point of view. Note that although we have so far worked in the context of UML because of its dominance of the software modelling world, the basic ideas of the work are not UML-specific.

9 Conclusion

In this paper we have introduced our tool GUIDE, which is an implementation of the exploration game framework. By repeatedly playing an exploration game, the designer gradually adds more detail to the design and specification in the role of the Explorer. While a game is being played, the designer may resolve non-determinacy by interaction with the GUIDE tool. He can take on the roles of other game participants to examine the design from a particular view and to perform parts of the moves in the play.

At the moment GUIDE supports the example game variant, expressions and algorithms which have been described in this paper. The tool is a proof-of-concept prototype which needs further testing with more complex examples than presented here. Probably the most important task for future work is to give users the opportunity to exercise our tool and to analyse their feedback. Thereby it would be possible to identify which parts of the approach presented here are most interesting in practice and where improvements are necessary. Experiments with students will be used as first step for testing GUIDE more thoroughly before advertising it to a wider community. More advanced investigations could focus on the comparison of exploring a design using the GUIDE tool with traditional reviewing techniques such as design reviews.

In this paper, and in our medium-term suggestions for further developments in the previous section, we have assumed that it is always the designer who plays the Explorer and increments the game. It is undoubtedly fascinating to imagine that the tool could perform the exploration of the design. However, as we have discussed, the exploration generally requires knowledge about the system and design skills. We expect that building a tool which performs the tasks of a designer to a certain degree would involve a large amount of research in the area of artificial intelligence. The GUIDE tool does not aim at substituting for the designer, but at supporting him in using his skills. A very advanced version of GUIDE could try to give the designer feedback about which kind of incrementation is beneficial for a player in specific situations. However, it is then still the designer who has to make a concrete incrementation according to the tool's suggestion.

Acknowledgements

The authors would like to thank the anonymous reviewers of the SoMeT paper [24] for useful comments, and the UK Engineering and Physical Sciences Research Council (GR/N13999/01, GR/A01756/01) for its support of this research.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[2] A. Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

[3] T. DeMarco and T. Lister, editors. *Peopleware – Productive Projects and Teams*. Dorset House Publishing, 1987.

[4] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[7] Gerd Gigerenzer, Peter M. Todd, and the ABC Group. *Simple Heuristics that Make Us Smart*. Oxford University Press, 1999.

[8] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infi nite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[9] Jack Greenfield and Keith Short et al. *Software Factories*. Wiley, 2004.

[10] GUIDE – Games with UML for Interactive Design Exploration. Available from the author's homepage at `http://www.lfcs.informatics.ed.ac.uk/jnt`.

[11] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 342(1):53–60, January 2001.

[12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proceedings of Formal Methods in Computer-Aided Design, FMCAD'02*, pages 378–398, 2002.

[13] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[14] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *5th Workshop on Tools for System Design and Verifi cation, FM-TOOLS'02*, Report 2002-11. Institut für Informatik, Universität Augsburg, 2002.

[15] J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *Proceedings of Automated Software Engineering, ASE'99*. IEEE, 1999.

[16] NetBeans Metadata Repository (MDR). Website at `http://mdr.netbeans.org`.

[17] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[18] Meta-Object Facility (MOF). Available from the OMG at `http://www.omg.org`.

[19] Poseidon for UML, version 3.0. Available from Gentleware at `http://www.gentleware.com`.

[20] Real Time Studio professional, version 4.3. Available from Artisan Software at `http://www.artisansw.com`.

[21] Rhapsody, version 5.0. Available from I-Logix at `http://www.ilogix.com`.

[22] P. Stevens and J.Tenzer. Games for UML software design. In *Formal Methods for Components and Objects, FMCO'02*, volume 2852 of *LNCS*. Springer, 2003.

[23] J. Tenzer. Exploration games for safety-critical system design with UML 2.0. In *Proceedings of the 3rd International Workshop on Critical Systems Development with UML, CSDUML'04*, Technical Report I0415, pages 41–55. Technische Universität München, September 2004.

[24] J. Tenzer. GUIDE: Games with UML for interactive design exploration. In *Proceedings of the 4th International Conference on Software Methodologies, Tools, and Techniques, SoMeT'05.*, pages 364–387. IOS Press, 2005.

[25] J. Tenzer. *Exploration games for UML software design*. PhD thesis, University of Edinburgh, 2006.

[26] W. Thomas. Infinite games and verification. In *Computer Aided Verifi cation, CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[27] OMG Unified Modeling Language Specification, version 1.4, September 2001. Available from the OMG at `http://www.omg.org`.

[28] UML 2.0 Superstructure Final Adopted Specification, August 2003. Available from the OMG at `http://www.uml.org`.

[29] USE – a UML-based Specification Environment. Website at `http://www.db.informatik.uni-bremen.de/projects/USE`.