Information and Software Technology xxx (2008) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Original papers

Model-checking for adventure videogames

Pablo Moreno-Ger, Rubén Fuentes-Fernández, José-Luis Sierra-Rodríguez *, Baltasar Fernández-Manjón

Dpto. Ingeniería del Software e Inteligencia Artificial, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain

ARTICLE INFO

Article history:
Received 8 January 2008
Received in revised form 5 August 2008
Accepted 5 August 2008
Available online xxxx

Keywords:
Adventure games
Domain-specific languages
Verification of games
Model-checking
Temporal properties

ABSTRACT

This paper describes a model-checking approach for adventure games focusing on ⟨e-Adventure⟩, a platform for the development of adaptive educational adventure videogames. In ⟨e-Adventure⟩, games are described using a domain-specific language oriented to game writers. By defining a translation from this language to suitable state-based models, it is possible to automatically extract a verification model for each ⟨e-Adventure⟩ game. In addition, temporal properties to be verified are described using an extensible assertion language, which can be tailored to each specific application scenario. When the framework determines that some of these properties do not hold, it generates an animation of a counterexample. This approach facilitates the collaboration of multidisciplinary teams of experts during the verification of the integrity of the game scripts, exchanging hours of manual verification for semi-automatic verification processes that also facilitate the diagnosis of the conditions that may potentially break the games.

1. Introduction

Videogames constitute an increasingly important resource for entertainment and education from a social and economic point of view. They provide exciting and engaging experiences that capture the attention of users and motivate them to solve the challenges posed by the game [53]. Among the different videogame genres, adventure games focus specifically on exploration and problem-solving, presenting challenging puzzles and riddles, and connecting them through appealing narratives [2].

The implementation of adventure games, which are complex and costly software applications, requires the ability to study and verify the validity of certain constraints in sophisticated and complex interactive narratives. A common problem in the design of such applications is how to prove that a game is correct (i.e. it satisfies certain properties that the writers consider relevant). For example these properties may refer to the presence of certain elements, the reachability of states, or the acceptable sequences of actions. Allowing the writers to actively participate in the development process enriches the results but does not necessarily allow for the verification of certain runtime properties regarding the potential ways in which a game can be played. These games can include complex chains of events in which certain interactions unlock further actions, often without a linear structure and using puzzles with different potential solutions. In fact, these chains of non-linear actions are sometimes so complex that it is not feasible for writers to consider every possible path of action and make sure that no undesirable states are reached. For instance, solving some game situations may require using a certain object (potentially breaking the game if the player reaches the objective without the object) and certain paths of actions may be incompatible with an adequate game solution. Simplifying the complexity of these narratives is not a good solution since players expect elaborate chains of interactions.

The traditional approach is therefore to be very careful when writing the game and specifying the puzzles, and to have a thorough and expensive quality-assessment process in which several testers spend hundreds of hours exploring every possible combination of actions that may eventually break the storyline. The process is even more challenging when we introduce the notion of *adaptive games* (i.e., games that can be customized with different initial states for different player profiles). As stated in [40], these games are especially relevant for educational uses in e-learning. An adaptive game that has been validated through a detailed testing process can be broken when applying a new adaptation profile, thus increasing the need for sophisticated quality assurance procedures, which must necessarily rely on some sort of automatic support.

In this research, we propose to improve the mentioned process by employing automatic or semi-automatic verification techniques, and, in particular, *model-checking* approaches [13]. These approaches introduce verification techniques for concurrent systems, which are abstracted as state-based models with a given initial state. The properties to be verified are usually expressed as temporal logic formulae. Then, the checker uses efficient algorithms to traverse the model defined by the system and to check whether the properties hold, reporting violations when counterexamples are found. The model-checking approach is especially

* Corresponding author. Tel.: +34 913947548; fax: +34 913947547.
E-mail addresses: pablom@fdi.ucm.es (P. Moreno-Ger), ruben@fdi.ucm.es (R. Fuentes-Fernández), jlsierra@fdi.ucm.es (J.-L. Sierra-Rodríguez), balta@fdi.ucm.es (B. Fernández-Manjón).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.08.003

interesting for games subjected to change and evolution (and, in particular, for adaptive games). Indeed, the properties that must hold in the game can be formalized at the initial design stages and can be maintained throughout the lifecycle of the game. If the game changes (or is adapted to meet a particular user's need), the testing effort required to check the suitability of the new version is substantially less, since the properties already formalized can be automatically checked to ensure their consistency.

However, model-checking for arbitrary videogames is a very complex process because it would require dealing with arbitrarily complex programs developed in general-purpose programming languages. Fortunately, in well-defined genres such as adventure games it is possible to formulate suitable *domain-specific languages* [35,51,52] for describing the games in the genre. The games will actually be generated from these descriptions using suitable application generators. From a model-checking perspective it also presents a fundamental advantage, since the state-based models can also be automatically generated from the domain-specific descriptions.

It was precisely from the ideas of domain-specific languages that we created the ⟨e-Adventure⟩ platform [41] to facilitate the development of *point and click* adaptive educational adventure games.[1] The definition of the ⟨e-Adventure⟩ language (an XML-based domain-specific language for adventure games) simplifies the specification of these games. As a language targeted to game writers, it was designed to provide the elements required to describe these kinds of games in a format that is very close to the common practices of these designers. The language is specified with a document grammar (in particular, with an XML Schema) and writers describe a game by editing XML files with authoring tools (i.e. the ⟨e-Adventure⟩ editor). These XML files contain all the information needed to make an executable game, and also to automatically extract a state-based model of the game, which can be used to model check the game.

This paper details the model-checking process supported by ⟨e-Adventure⟩. This process promotes a minimally invasive collaboration model involving different experts: game writers, programmers and experts in model-checking. For this purpose, ⟨e-Adventure⟩ includes an extensible property description language, which can be tailored to each specific application scenario. Then, a model checker verifies the properties with the automatically-extracted state-based model of the game and provides counterexamples if any property does not hold. Although the paper focuses on ⟨e-Adventure⟩, in our opinion the process could be applied to other game development frameworks.

The rest of the paper is organized as follows: Section 2 presents an overview of the theoretical foundation of the checking support for ⟨e-Adventure⟩ by introducing transition systems in the context of model-checking to verify properties specified with a temporal logic. The ⟨e-Adventure⟩ framework for the development of educational adventure videogames is briefly reviewed in Section 3. In Section 4, we introduce a case-study regarding an educational videogame that will be used in the rest of the article to illustrate the different concepts. Section 5 presents the checking support developed in this research and includes examples of its use. A comparison with related work can be found in Section 6. Finally Section 7 discusses several conclusions about the potential advantages, limitations, and future work.

2. Preliminaries

The dynamic properties of ⟨e-Adventure⟩ games can be seen as properties expressed in a temporal logic about a finite labeled transition system automatically generated from the game, which can be verified with model-checking techniques. Formally, a labeled transition system (or labeled *Kripke* structure) [8], is a tuple $\langle Q, \sum, \rightarrow, AP, P, Q_0 \rangle$ where Q is a set of *states*, $\sum$ is a set of *transition labels*, $\rightarrow \subseteq Q \times \sum \times Q$ is the *transition relation*, AP is a set of *atomic propositions*, $P \in Q \rightarrow 2^{AP}$ is an assignment of atomic propositions to states, and $Q_0 \subseteq Q$ is the set of *initial states*. In order to prove temporal properties chosen an initial state, the transition graph of the transition system can be unwound into a usually infinite computation tree rooted in such a state. Every node in the tree has as many child nodes as transitions leave its corresponding state in the system. With this representation, the paths in the tree from a given node represent the possible alternatives in the evolution of the system from the state that corresponds to that node. Therefore, the transition from a node to its child nodes carries out an implicit notion of time.

Computation tree logic (CTL) [12,23] is a branching-time temporal logic widely used to verify properties of these computation trees. CTL interprets transitions in the computation tree as time progresses, making explicit the idea already present in the tree. If there is a path from a node q_i to another q_j in the tree, it is said that q_j belongs to a possible future of state q_i. CTL is defined as an extension of the usual Boolean propositional logic with additional temporal connectives. As shown in [12], the complexity of model-checking in CTL is polynomial in the size of the model's state space, contrary to what occurs with more general temporal logics [23]. Still it is expressive enough to describe many interesting properties on the dynamics of the underlying transition systems. In Table 1 the basic CTL temporal connectives, together with their meanings, are introduced. As usual in logics, a richer vocabulary of connectives can be defined in terms of the pre-existing ones. In Table 2, other common temporal connectives are introduced, together with their rewriting in terms of the basic ones.

The abstraction of ⟨e-Adventure⟩ games as labeled transition systems and the use of CTL with these systems to describe dynamic properties provide the foundation for the checking support proposed in this paper. This support is based on model-checking using the NuSMV tool [11], which is based on the former tool SMV [34]. These tools have been extensively used in checking properties of hardware systems with CTL. Regardless of the polynomial complexity of CTL model-checking with respect to the size of the state space, it should be noted that the size of this state space can become exponential in some parameters of the modeled problem [45]. In order to cope with this complexity, tools like SMV and NuSMV use *symbolic model-checking* to deal with huge state spaces [7]. The idea is to encode the search frontier as a boolean formula, as well as the transition relation, and to use efficient representations of these formulae with binary decision diagrams (BDD) [1]. A BDD represents a boolean formula as a rooted, directed, acyclic graph which is initially a binary tree. Each node of the tree is labeled with a variable of the formula. The child nodes are referred to as low and high child, where the low child represents an assignment of 0 to the variable and the high child a 1 value. BDD can be efficiently reduced by merging isomorphic sub-graphs (i.e. with the same label and identical children) and by removing redundant nodes (i.e. those where their two children are identical).

A model for NuSMV is specified with a set of variables whose types are finite ones (i.e. boolean, scalars, and constant sized arrays). The possible combinations of values for all these variables determine the states existing in the modeled transition system. The transition relation is specified with expressions from the propositional calculus.

Fig. 1a shows an excerpt of code for NuSMV. These elements model a *parity* machine, which detects when a sequence of bits has an odd number of 1's (see Fig. 1b). The specification introduces an *input* variable called `bit`, as well as a *state* variable called

[1] ⟨e-Adventure⟩ was formerly known as ⟨e-Game⟩.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

Table 1
Basic temporal connectives in CTL together with their semantics. M denotes a transition system $\langle Q, \sum, \rightarrow, AP, P, Q_o \rangle$

Symbol	Meaning	
	Intended	Formal
EX(_)	EX(φ) holds iff there **E**xists a ne**X**t state where φ holds.	$M, q_0 \mid = \mathrm{EX}(\varphi)$ iff $M, q_1 \mid = \varphi$ for some transition $q_0 \xrightarrow{l} q_1$
AX(_)	AX(φ) holds iff in **A**ll ne**X**t states φ holds	$M, q_0 \mid = \mathrm{AX}(\varphi)$ iff $M, q_1 \mid = \varphi$ for all transitions $q_0 \xrightarrow{l} q_1$
E(_ U _)	E(φUψ) holds iff there **E**xists a sequence of states where φ holds **U**ntil a state where ψ holds is reached	$M, q_0 \mid = \mathrm{E}(\varphi \mathrm{U} \psi)$ iff for some sequence of transitions $q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \ldots \exists i(\forall j(0 \leqslant j < i \Rightarrow M, q_j \mid = \varphi) \wedge M, q_i \mid = \psi)$
A(_U _)	A(φUψ) holds iff in **A**ll sequence of states φ holds **U**ntil a state where ψ holds is reached	$M, q_0 \mid = \mathrm{A}(\varphi \mathrm{U} \psi)$ iff for all sequences of transitions $q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \ldots \exists i(\forall j(0 \leqslant j < i \Rightarrow M, q_j \mid = \varphi) \wedge M, q_i \mid = \psi)$

The transitions of the form $(q,l,q') \in \rightarrow$ will be written as $q \xrightarrow{l} q'$.

Table 2
Other temporal connectives and their rewriting in terms of the basic ones

Symbol	Meaning	
	Intended	Formal
EF(φ)	There **E**xists a sequence of states leading to a **F**uture state where φ holds	E(true U φ)
AF(φ)	**A**ll sequences of states lead to some **F**uture state where φ holds	A(true U φ)
EG(φ)	There **E**xists a sequence of states where φ holds for each state of the sequence (i.e. where φ **G**lobally holds)	$\neg$AF($\neg \varphi$)
AG(φ)	For **A**ll possible sequences of states φ **G**lobally holds	$\neg$EF($\neg \varphi$)

`parity`. In NuSMV states are defined by the combination of all the possible values adopted by the state variables, while transitions are labeled by values for the input variables. Notice that boolean values have an integer encoding with 0 being *false* and any other value meaning *true*. In the `ASSIGN` section, the `init` construction is used to assign a value to the `parity` variable for the initial state. The `next` construction is used to assign the value of the variable in the next state. The `case` statement sequentially evaluates their guards. When one of them is true, the corresponding branch determines the value of the expression. Also notice that, since `bit` stays unassigned, it can take any of the two possible boolean values. The keyword `CTLSPEC` allows the introduction of assertions about the execution of the model as CTL formulae. Violations of these formulae are reported to the user. For instance, the `CTLSPEC` assertion in Fig. 1a states that every time (i.e. `AG`) that `parity` variable is *true*, then in some possible future evolution, eventually (i.e. `EF`) `parity` becomes *false*. It should be noted that the language of NuSMV provides a certain level of hierarchical functional abstraction with the definition of modules (i.e. `MODULE`) and their instantiation as processes, but given that the models are automatically generated from the game descriptions, these structuring features will not be used in the present work.

3. The ⟨e-Adventure⟩ framework

⟨e-Adventure⟩ is a framework for the development of educational *point and click* graphical adventures [41]. In the ⟨e-Adventure⟩ framework, the development of these kinds of games is the result of the collaborative work among three groups of people with orthogonal skills: game writers, artists, and programmers [39]. Game writers create the storyboard that describes the story and how the different elements of the game are integrated in it. Artists produce the artworks used in the final product, like music, graphics or videos. Finally, programmers implement and customize the software infrastructure.

Among these roles, ⟨e-Adventure⟩ considers that producing and maintaining the storyboard are the primal activities in the development process of this kind of games, and thus it adopts a document-driven approach [48,49]. In this approach, the writers use a descriptive markup language (the ⟨e-Adventure⟩ language) to make the structure of the story explicit. Given the focus on the storyboard, the markup language is close to the common practices of game writers, by using their jargon and structures to model games. Additional markup is provided to refer to the assets created by the artists and integrate them into the story. The result is a formalized storyboard (the ⟨e-Adventure⟩ document). In turn, the programmers implement a processor for the ⟨e-Adventure⟩ language (the ⟨e-Adventure⟩ engine) that can produce executable games from the ⟨e-Adventure⟩ documents and their associated art assets. The process is outlined in Fig. 2.

The framework focuses on the educational applications of these games and, for this reason, it includes a number of education-

(a)

```
MODULE main
 IVAR
    bit: boolean;
 VAR
    parity: boolean;
 ASSIGN
    init(parity) := 0;
    next(parity) :=  case
                      bit & !parity: 1;
                      !bit & parity: 1;
                      1:             0;
                     esac;
 CTLSPEC
    AG(parity -> EF( ! parity))
```

(b)

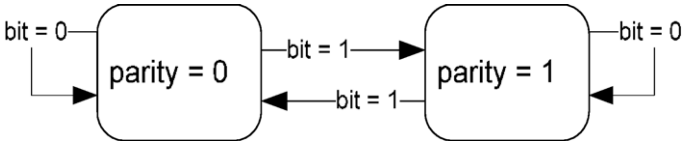

Fig. 1. (a) A very simple specification in NuSMV of a parity machine; and (b) graphical representation of the transition system for the parity machine.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

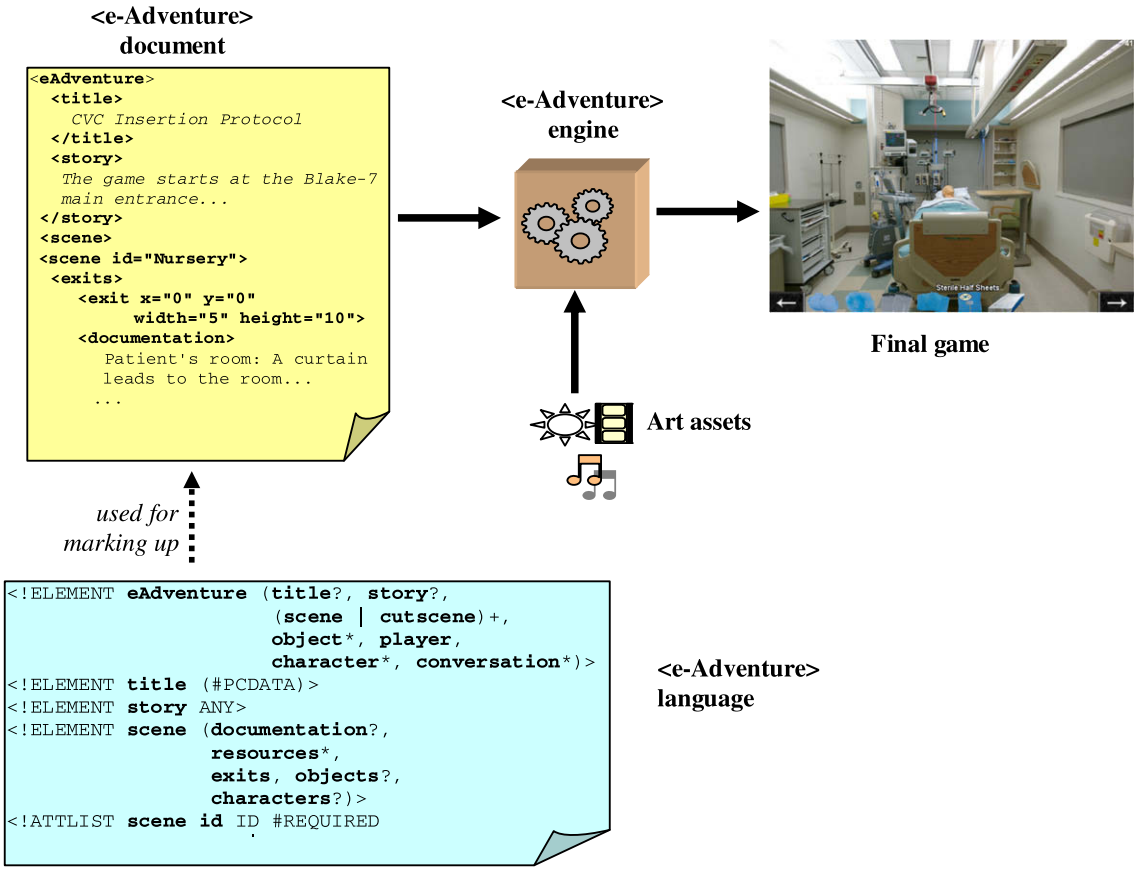

Fig. 2. The ⟨e-Adventure⟩ document-driven approach to adventure game development.

specific features, such as the user's assessment, support for adaptive learning, and integration with e-learning environments. However, the presence of these educational features is not directly relevant for this work. ⟨e-Adventure⟩ can be used as an authoring tool for traditional adventure games with or without educational purposes. Nonetheless, the adaptation features included in the framework are particularly relevant in this context, given that they can suppose the invalidation of previously established properties with each new adaptation of the game.

The rest of this section describes the features of ⟨e-Adventure⟩ that are relevant from a property-verification perspective. Section 3.1 briefly introduces the markup language for adventure games provided by the framework. The characteristics of the ⟨e-Adventure⟩ engine are discussed in Section 3.2. Finally, Section 3.3 describes the adaptation features of the framework.

3.1. The ⟨e-Adventure⟩ language

The purpose of ⟨e-Adventure⟩ language is to allow writers to formally describe the storyboard of the game. The language conceives these games as sets of interconnected *scenes*. In this context, a scene is a certain static environment populated by dynamic objects. It usually represents a room (or a perspective of a room) in the game world. The scenes contain exits that allow the player to engage with them, and they are also populated by objects and characters. The player can navigate the scenes by activating the exits, interact with objects (examine them, grab them, combine them with other objects, etc.) and interact with characters (giving them objects or maintaining interactive conversations).

The language provides constructs to define all these elements (scenes, exits, objects, characters, interactions, and conversations). All these components include at least a representational description and most of them (except for scenes and conversations) have a behavioral description. The representational description defines the appearance of the object by including references to art assets while the behavioral description considers how the component is used in the game.

However, simply describing maps of scenes and populating them yields static games where every exit leads to the same place, every object is always available and every in-game character always says the same thing. In order to provide a narrative flow, it is necessary to introduce a notion of state. This is carried out by means of *flags*. These flags are boolean variables that can be activated as a result of performing an action, and which can be also used to formulate preconditions for other actions. Thus, if the player has not performed specific actions that activate the appropriate flags, some exits may not be traversable, some objects or characters may not appear in the scene, and some actions may be forbidden or their result may be different.

As will be described later in the paper, the domain-specific nature of the ⟨e-Adventure⟩ language makes it possible to *automatically* extract games' verification models as finite transition systems. In addition, the dynamic properties regarding the execution of games can be expressed as temporal logic formulae in which, from an initial state, some statements are made about the potential future states of the game. Thus, as stated before, ⟨e-Adventure⟩ games are suitable for model-checking techniques.

3.2. The ⟨e-Adventure⟩ environment

In the ⟨e-Adventure⟩ framework, the running games are automatically produced from XML documents. This way of working is

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

mainly based on two tools, an editor and an engine. The ⟨e-Adventure⟩ editor eases the game designers' task of describing games as documents, by avoiding direct XML manipulation. The editor is an author-oriented tool in which most of the XML document is specified by graphically editing the game components. The ⟨e-Adventure⟩ engine is essentially a language processor that is able to interpret the marked up storyboards and produce the running games. The implementation of the engine relies on the operational semantics defined for the ⟨e-Adventure⟩ language [41]. This semantics provides a formal description of the dynamic behavior of a game described in the previous section. Roughly speaking, in ⟨e-Adventure⟩ the game behaves as a state-machine in which the actions of the player trigger state-transitions. Acts such as traversing an exit or grabbing an object represent implicitly a state-transition (the location of the player changes and the object moves from the scene to the inventory). Nevertheless, as ⟨e-Adventure⟩ is targeted to game writers, even this formal description tries to closely reflect practices in game development. This means that the specified semantics is based on the usual interactions governing games in the genre of adventure games instead of being based on programming concepts. In this way, authors can get enough intuitive knowledge to understand how the engine interprets the game although they are not experts at programming. For instance, they know that some actions change the state of the game by modifying the flags or that a grab action inserts an object in the inventory.

This formalization of ⟨e-Adventure⟩ games to describe their operational semantics is the departure point of the work in this paper. The verification framework builds its models over a more abstract view of these semantics that will be introduced later on in this paper. This verification framework is also integrated in the ⟨e-Adventure⟩ environment as a support tool for game writers allowing them to check properties of the produced games.

3.3. Adaptation of ⟨e-Adventure⟩ games

The ⟨e-Adventure⟩ framework is focused on supporting educational adventure games. For this reason, it includes a number of features that enhance its educational value [38]. One of these features is the support of adaptive learning patterns. As described in [40], ⟨e-Adventure⟩ games can be adapted to suit different player profiles. These adaptations include omitting some portions of the game (for students with different levels of initial knowledge) or changing parts of the content to suit different learning scenarios.

The behavior of this mechanism targets precisely the formal notion of game state on which this work is based. By forcing an initial game state when the game is launched, it is possible to achieve completely different behaviors when the game runs (see Fig. 3).

Having a separation between the game definition and the initial states that can be forced into the game improves the flexibility of the ⟨e-Adventure⟩ framework. For instance, it allows the modification of game itineraries to fit emerging needs without modifying the description of the game itself. However, it increases the complexity of maintaining coherent games. Forcing an alternative initial state when the game is launched may introduce new side-effects in games that were already tested thoroughly from their original starting point. Manually checking the relevant properties from the beginning *each* time the game is adapted is not a sustainable approach to the production and maintenance of this kind of videogames. For this purpose, an automatic support is essential for the successful application of the approach.

4. Case-study

This section introduces an example game that is used to show how the verification framework for ⟨e-Adventure⟩ works. The framework has been applied to different fields, such as medical training (both for undergraduate students and residents), workplace safety regulations, the teaching of History (for mid-school level) and some student-created non-educational games. Among those, we have based our case-study on an educational *point and click* adventure game created to train medical doctors jointly developed by the Complutense University of Madrid and researchers from the Harvard Medical School and the Massachusetts General Hospital. The game walks the doctor through the insertion procedure of central lines as described in [37]. The process takes place in a Medical Intensive Care Unit, with different perspectives of the room being represented as ⟨e-Adventure⟩ scenes. During the procedure, the player must follow a 98-step protocol in which some steps must be performed before others. We have used this game as a test-bed for the verification procedures. However, in order to better illustrate the verification process and allow its complete study, we will strip the example to a bare minimum. The simplified game document can be observed in Fig. 4.

In the simplified game, the player is initially located at the entrance to the patient's room (scene `RoomEntrance`, lines 2–12). From here, it is possible to hold a conversation with the patient. Whenever he or she wants, the player can activate a transition (indicated with an `exit` element, lines 5–7) that leads to another game location, the header of the bed (scene `BedHeader`, lines 13–31). In this second location, the player can grab the ultrasound probe (indicated with a `grab` element, lines 39–46) or activate the exit that leads to the third location, where the clinical procedure happens (scene `Examination`, lines 32–34). The actual examination process is not relevant for the proposed example. The case-study actually focuses on just one element of the game: the notion of maintaining a proper communication with the patient.

There is a chain of dependencies forming the basis of this example that prevents starting the examination of the patient without first having had a reassuring conversation. Using the flag system described in the previous section, we see how it is necessary to have previously activated the flag `ReadyToExaminePatient` in

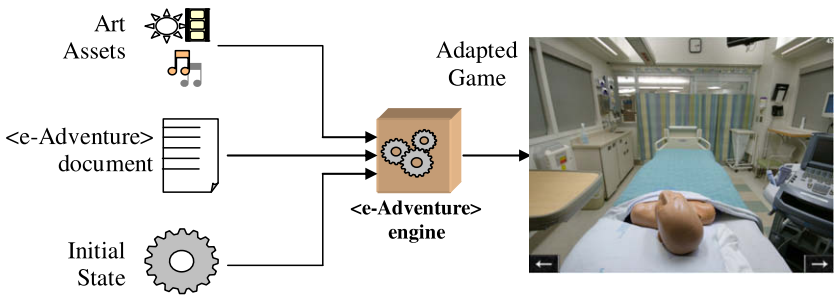

Fig. 3. When the game is launched, the engine receives the description of the game, the art assets and an initial state. The initial game scenario and the game behavior will depend on the initial state.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

```
01 <eAdventure>
02   <scene id="RoomEntrance" start="yes">
03       <resources> …  </resources>
04       <exits>
05           <exit height="253" width="640" x="66" y="68">
06               <next-scene idTarget="BedHeader"/>
07           </exit>
08       </exits>
09       <characters>
10           <character-ref idTarget="Patient"/>
11       </characters>
12   </scene>
13   <scene id="BedHeader">
14       <resources> … </resources>
15       <exits>
16           <exit height="159" width="383" x="220" y="385">
17               <next-scene idTarget="Examination">
18                   <condition>
19                       <active flag="ReadyToExaminePatient"/>
20                   </condition>
21               </next-scene>
22           </exit>
23       </exits>
24       <objects>
25           <object-ref idTarget="UltrasoundProbe" x="9" y="6">
26               <condition>
27                   <inactive flag="UltrasoundNotAvailable"/>
28               </condition>
29           </object-ref>
30       </objects>
31   </scene>
32   <scene id="Examination">
33       …
34   </scene>

35   <object id="UltrasoundProbe">
36       <resources> … </resources>
37       <description> … </description>
38       <actions>
39           <grab>
40               <condition>
41                   <active flag="PlayerSpokeWithPatient"/>
42               </condition>
43               <effect>
44                   <activate flag="ReadyToExaminePatient"/>
45               </effect>
46           </grab>
47       </actions>
48   </object>
49
50   <character id="Patient">
51       <resources> … </resources>
52       <conversations>
53           <conversation-ref idTarget="CalmPatient"/>
54       </conversations>
56   </character>
57
58   <tree-conversation id="CalmPatient">
60       <speak-player>Hello Mr. Smith</speak-player>
61       …
62       <end-conversation>
63           <effect>
64               <activate flag="PlayerSpokeWithPatient"/>
65           </effect>
66       </end-conversation>
67   </tree-conversation>
68 </eAdventure>
```

Fig. 4. Excerpt of the specification of a game in ⟨e-Adventure⟩. It corresponds to the simplified game of the case-study that only includes three scenes. The arrows indicate the indirect relation between executing the conversation with the patient and unlocking the transition that allows the player to initiate the medical procedure.

order to start examining the patient (lines 18–20). This condition is only satisfied after grabbing the ultrasound probe (lines 39–46) and having talked to the patient is a pre-condition in order to perform this action (lines 41 and 64). In other words, the game prevents the player from arriving at the scene `Examination` without first speaking with the patient through a chain of events (displayed in Fig. 5) in which there are different connected steps.

Even though this example is simplified and the verification could be easily performed without support tools, it illustrates the kind of problems that these games might pose. In this example, the requirement of having talked to the patient before the examination holds, but the relation is indirect and depends on other steps in a chain of actions.

Additionally, as mentioned in Section 3.3, ⟨e-Adventure⟩ includes an adaptation mechanism that allows game writers (i.e. the medical instructors in this case) to define new initial states for the games in order to cover alternative learning scenarios. The example game has different adaptation options, including the possibility of altering the availability of the ultrasound machine. The engine can receive an adaptation rule as shown in Fig. 6 that eliminates the ultrasound machine. In order to unlock the examination scene, through this rule the well-intentioned author also activates the `ReadyToExaminePatient` flag, so that the elimination of the ultrasound machine does not render the game unusable. It can be observed that the introduction of this adaptation rule introduces an undesired side-effect that makes it possible to examine the patient without previously talking to him/her, by breaking the dependency between grabbing the ultrasound probe and speaking with the patient. Even though this is a rather simple example, it must be noted that the chain of events

1. Initial scene *RoomEntrance*

2. Holding a conversation with the patient

3. Changing position to *BedHeader* and grabbing the *UltrasoundProbe*

4. Initiating the examination

Fig. 5. Playing the game in the ⟨e-Adventure⟩ engine.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

```
<adaptation>
  <adaptation-rule>
      <description>Ultrasound not available</description>
      <game-state>
          <activate flag="UltrasoundNotAvailable"/>
          <activate flag="ReadyToExaminePatient"/>
      </game-state>
  </adaptation-rule>
</adaptation>
```

Fig. 6. An adaptation rule in ⟨e-Adventure⟩.

connecting the conversation with the patient and the examination could be much longer, and the instructor that particularizes the learning scenario may not be aware of the potential side-effects of this apparently localized change.

5. Verification of game properties in ⟨e-Adventure⟩ games

The checking support introduced in this work describes an ⟨e-Adventure⟩ game as a model for a model checker. For this purpose, ⟨e-Adventure⟩ games are abstracted as labeled transition systems, which are subsequently encoded into models for one of those checkers. These encodings also include the game properties to be verified in the form of asserts. This section describes how this checking support is built and how to use it with NuSMV, [11] (our model checker of choice). Section 5.1 presents the overall process for the verification, including its stages/tools, the data, and participants. Section 5.2 introduces the language to specify the properties for the execution of the games that have to be verified. These properties are stated as CTL formulae about games. Section 5.3 reports how to conceive the labeled transition system for a game. Section 5.4 describes how to encode these transition systems as NuSMV models, which therefore can be extracted automatically from the original ⟨e-Adventure⟩ specifications, and which also incorporate the properties to check. Section 5.5 discusses the verification of the model and the generation of counterexamples when a property does not hold. Finally, Section 5.6 shows how violations of such properties can be shown to game writers by animating the corresponding counterexamples using the ⟨e-Adventure⟩ robot.

5.1. The approach

Fig. 7 illustrates the proposed checking process, which is integrated in the ⟨e-Adventure⟩ environment. It is composed by the succession of the activities of four tools: the *Verification Model Generator*, the *Model Checker*, the *Animation Generator*, and the ⟨*e-Adventure*⟩ *Robot*.

- The *Verification Model Generator* is the tool that produces the *verification model* used by the *Model Checker*. The data needed by this tool are the description of the ⟨*e-Adventure*⟩ *game* (for the case-study, the code from Fig. 4), the *adaptation specification* for a concrete initial situation (as described in Fig. 6), and the *properties* to verify. These properties correspond to the constraints that the game must satisfy. In the case-study, the property that we will attempt to verify can be informally described as "it is not possible to start the examination process without first talking to the patient". The properties will appear as asserts in the verification model. Thus, the verification model is the encoding of a suitable transition system for the ⟨e-Adventure⟩ game. Currently, it is expressed in the specification language of the NuSMV model checker.
- The *Model Checker*, which is currently NuSMV, takes as input the *verification model* and checks the properties asserted about it. It can ensure that the properties are satisfied or, otherwise, to find a counterexample in terms of a path in the state space described by the transition system of the game (i.e. the *counterexample trace*). The counterexample is therefore a sequence of valid actions that the player can perform in the game, and which vio-

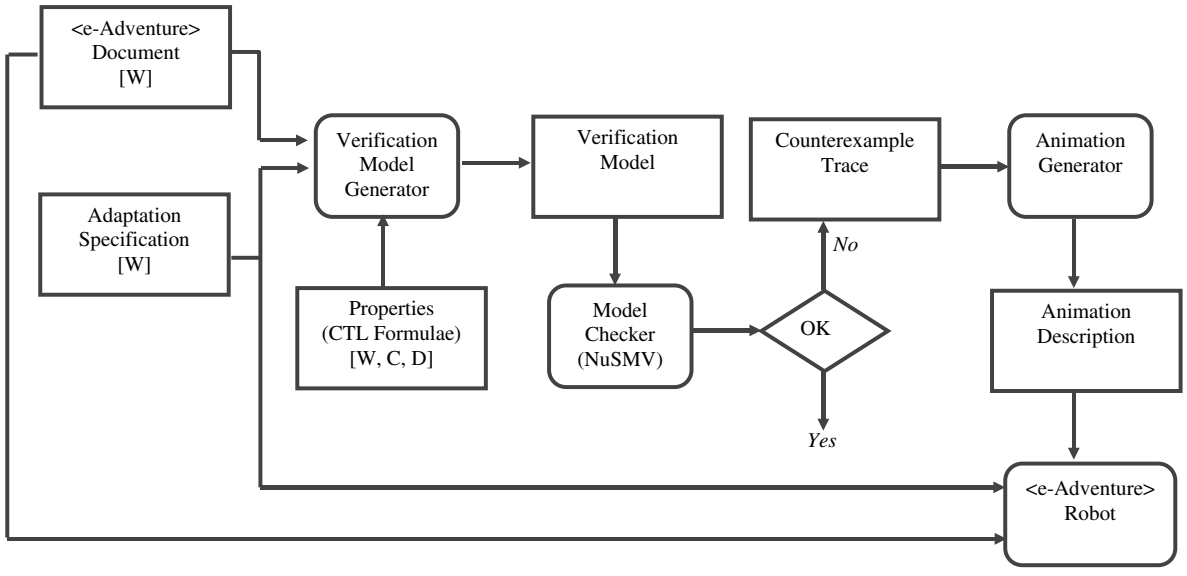

Fig. 7. The checking process for ⟨e-Adventure⟩. Providers of data are shown between square brackets: [W] for Writers, [C] for experts in Checking and [D] for other developers.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

lates the property. As we will see below, in the case-study the *Model Checker* will actually find a sequence of actions that violates the previously mentioned property.

- The *Animation Generator* takes a *counterexample trace* from the *Model Checker* as input and generates *animation descriptions* as outputs. An animation description is a sequence of actions in a suitable format for the ⟨*e-Adventure*⟩ *Robot*. This step is important because the output of the *Model Checker* may be too complex to be directly understood by a human. The animation description generated by this module is a cleaner XML-based documentation of the trace of actions that can lead to the property or properties under study to be violated.
- The ⟨*e-Adventure*⟩ *Robot* is a version of the ⟨e-Adventure⟩ engine that, taking an *animation description* as input, automatically plays a trace of the game's execution. This robot also takes the description of the game and the adaptation rules as additional inputs. It is used to visually illustrate the counterexample to the game writers, so that they better understand the discrepancy identified in the process. The animation hides the details and complexity of the underlying verification from game writers. They do not see the search problem over the computation tree, but a sequence of actions inside the game that violates their proposed constraints. In the case-study, the result would be an automated animation of the game in which the player manages to start the examination process without first speaking with the patient.

Of all four tools, only the model checker is externally developed. The other three have been specifically built for the ⟨e-Adventure⟩ framework. Although NuSMV is the current model checker, the approach presented can also use other kinds of model checkers. Given that languages to describe games, adaptation, and properties depend only on the ⟨e-Adventure⟩ language, their specifications do not need to be changed if a different model checker is adopted. Only the *Verification Model Generator* and the *Animation Generator* would change in order to adapt their input/output to those required by the new checker.

These tools use three files as input to perform the verification process: the specification of the game, its adaptation, and the properties to verify. According to the development process in Section 3, the specification of the game and its adaptation are the work of game writers. Artists also contribute to the game, but art assets are not relevant for verification purposes. As observed in Fig. 7, formulating the properties to verify is the result of a joint work of these writers and other experts in Computer Science: experts in checkers (and, more particularly, in temporal logics) and developers. This is, in fact, the most delicate part of the process. Properties must be formalized using an assertion language based on CTL that considers a macro mechanism to tailor this assertion language to specific needs. Ideally, game writers should be able to formulate the properties on their own. However, as will become obvious in the next sections, the task requires a level of proficiency in Computer Science and Math. In fact, the formalization of properties with *pure*, non-sugarized, temporal logics can be a difficult task even for computer programmers, requiring the participation of experts in model-checking.

In order to organize this collaboration, we propose a process in which game writers informally state the properties to be checked on in each specific game. The properties can be formalized by developers using a specification language customized with derived operators using macros. This customization can be carried out by experts in model-checking. Fortunately, as we will later see, the effort of defining this language is primarily a one-time investment. In time, a set of stable operators can be reached for a particular family of games, making it unnecessary for experts in model-checking to participate.

5.2. Describing properties

The ⟨e-Adventure⟩ assertion language includes state propositions and CTL formulae to describe their relations over time. The syntax of the language is depicted in Fig. 8. A specification in this language is a sequence of elements of the following types:

- *Define elements*. Represented by the **D** syntactic variable, they define elements through a simple abstraction mechanism based on the use of *macros*.
- *Include elements*. These elements, which are represented by the **I** syntactic variable, make it possible to include other specifications (usually containing definitions of macros). The ⟨e-Adventure⟩ verification framework also enables the automatic inclusion of macros and definitions of properties by placing the corresponding files in an appropriate directory, therefore making the inclusions transparent for subsequent users.
- *Assertion elements*. They constitute the actual properties to be checked, and they are represented by the **F** syntactic variable. The simplest properties are: (i) boolean constants (`true` and `false`), (ii) parameter names, which can occur in the bodies of definitions of macros, and (iii) three different types of *state propositions*. The types of state propositions contemplated are: (i) *Scene propositions* (for each scene *s* there is a scene proposition S*s* with the intended meaning "the player is in scene *s*"), (ii) *Object propositions* (for each object *o* the object proposition O*o* means "the object *o* is in the inventory"), and (iii) *Flag* propositions (there is a flag proposition F*f* for each flag *f*, whose intended meaning is "the flag *f* is active" — i.e. its value is *true*). In addition to these basic propositions, we can combine formulae with the usual boolean and temporal operators, as well as with the new operators defined using macros. Notice that, for the sake of simplicity, we are not making the precedence and associativity of boolean operators explicit. Indeed, all the operators are associative, with the binary ones associating to the left.

S ::= { **D** | **I** | **F** }+
D ::= **DEF** *op-name* (*param*-name { , *param*-name }) = **F**
I ::= # *spec-uri*
F ::= `true` | `false` | *param-name* | S*scene-name* | O*object-name* | F*flag-name* |
`not` **F** | **F** `and` **F** | **F** `or` **F** | **F** `->` **F** | **F** `<->` **F** | `(` **F** `)` |
`EX(`**F**`)` | `AX(`**F**`)` | `E(`**F** `U` **F**`)` | `A(`**F** `U` **F**`)` |
op-name(**F** { , **F** })

Fig. 8. Syntax of the ⟨e-Adventure⟩ assertion language. For simplicity we do not make explicit the precedence and associativity of the operators.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

Regarding precedence, from higher to lower, the boolean operators are ordered as follows: `not`, `and`, `or`, -⟩ and ⟨- (with -⟩ and ⟨- having the same precedence). All the temporal operators have the lowest priority.

The macro mechanism is particularly useful to accommodate the language to fit the specific needs of each application scenario. Table 3 shows some of the macros currently distributed with the ⟨e-Adventure⟩ verification framework. In particular, the last three have proved very useful to describe many interesting properties in adventure games since they allow for the temporal ordering of two related conditions.

Regarding the case-study proposed in Section 4, game writers are interested in guaranteeing that every time a player begins the examination of the patient, he/she has previously had a conversation with that patient. According to the specification of the game in Fig. 4, these two conditions of the property are described as follows: The player has had a conversation with the patient if the flag `PlayerSpokeWithPatient` is active (lines 41 and 64), that is, if state proposition F*PlayerSpokeWithPatient* is true; the player is examining the patient if he/she is in the scene `Examination` (lines 32–34), which corresponds to the state proposition S*Examination*. The property about the game can be expressed with the macro `before` from Table 3: "If a point where S*Examination* holds is reached, it was done by previously visiting a point making F*PlayerSpokeWithPatient* hold". Therefore, the final statement of the property is "`before` (S*Examination*, F*PlayerSpokeWithPatient*)".

5.3. Labeled transition systems for verification

The description of games in the ⟨e-Adventure⟩ language allows for the automatic extraction of verification models in the form of labeled transition systems. In these models, the states are determined by the flags that appear in the specification of the game, the scene that the player is visiting, and the objects that he/she has in the inventory. The transitions between states are determined by the actions that the player can perform, the preconditions of these actions, and their effects. More precisely, for each game G will result in a transition system $\langle Q^G, \sum^G, \rightarrow^G, AP^G, P^G, Q_0^G \rangle$, where:

- States in Q^G will be identified with sets of atomic propositions containing a single scene proposition (the scene where the player is situated), an arbitrary number of flag propositions (the flags that are active in the state), and an arbitrary number of object propositions (the objects in the player's inventory).
- Labels in Σ^G will represent the actions that the player can perform to change the state of the game. These labels can have the following forms: (i) T*s_i*, with *i* a positive integer, and with intended meaning "exit number *i* of the scene *s* has been traversed" (*traversing* action), (ii) G*o* with intended meaning "the object *o* has been grabbed" (*grab* action), (iii) U*o* with intended meaning "the object *o* has been used" (*use* action), (iv) UW*o_o′* with intended meaning "the object *o* has been used with the object *o′*" (*use-with* action), (v) GT*o_ch* with intended meaning "the object *o* has been given to the character *ch*" (*give-to* action), and (vi) C*p* with *p* a conversation path and with intended meaning "conversation path *p* has been followed" (*talking* action).
- The definition of the transition relation $\rightarrow^G$ is based on a straightforward formalization of the intended semantics of ⟨e-Adventure⟩ games outlined in Section 3. Transitions departing from a state will be determined by the actions whose preconditions hold in such a state. In turn, each transition will lead to the state which comes from applying the action's effects. The complete formalization can also be consulted in the Appendix.
- AP^G is the set of scene, object, and flag propositions described before.
- The assignment function P^G is straightforward, since the definition of the states carries the associated propositions implicit.
- Finally, the set of initial states Q_0^G contains a single state q_0^G that includes propositions for the scene marked in the game as the initial one, for each object initially contained in the inventory, and for each flag initially active.

Table 3
Examples of macros in the ⟨e-Adventure⟩ assertion language

Definition	Intended meaning
DEF EF(φ) = E(true U φ)	Standard derived temporal connectives defined in Table 2
DEF AF(φ) = A(true U φ)	
DEF EG(φ) = not AF(not φ)	
DEF AG(φ) = not EF(not φ)	
DEF after-eventually(φ,ψ) = AG(φ->EF(ψ))	If in any game point φ holds, it is possible to continue playing until reaching another point where ψ holds
DEF after-always(φ,ψ) = AG(φ-> AF(ψ))	If in any game point φ holds, any possible way of continuing playing will lead to a point where ψ holds
DEF before(φ,ψ) = not E(not ψ U φ)	If a point where φ holds is reached, it was done by previously visiting a point making ψ hold

In Fig. 9 we sketch the transition system for the simplified game used in the case-study. As stated in Section 4, notice that we have explicitly used a reduced version of the real game in order to allow its different aspects to fit in the limited space of this paper. A more realistic game can have thousands and even millions of states (as is the case with the full example on which the simplified version for the case-study is based).

5.4. Generating the verification model for the model checker

In the ⟨e-Adventure⟩ framework, verification models are encoded in the NuSMV specification language. In this encoding, states will be represented by the variables $\texttt{Oo}_1, \ldots, \texttt{Oo}_m, \texttt{Ff}_1, \ldots, \texttt{Ff}_n, \texttt{Scene}$ where there is a boolean variable $\texttt{Oo}_i$ associated with each object proposition, another boolean variable $\texttt{Ff}_j$ for each flag proposition, and a variable `Scene` for the scene proposition (the type of this variable will be an enumeration $s_1 \ldots s_r$ of all the possible scenes). Therefore, the set of object (respectively, of flag and scene) propositions included in the state will be encoded by the values assigned to these variables. If the value of $\texttt{Oo}_i$ (respectively, of $\texttt{Ff}_j$) is true, the state will include the proposition Oo_i (respectively, Ff_j); if the value of `Scene` is `s`, the state will include the proposition S*s*. In addition, transitions will be labeled by an input variable `Act`, representing the current action chosen by the user. Its type is an enumeration of the transition labels induced by the game, enriched with an additional qualifier which will determine the *alternative* for the action. For instance, for each label T*s_i* we will consider enumerate values `Ts_i_j`, which will indicate that exit *i* was traversed in scene *s* in virtue of the alternative number *j* for traversing such an exit. Similarly, the suffix *j* in G*o_j*, U*o_j*, UW*o_o′_j*, GT*o_ch_j*, and C*p_j* will indicate the alternative to perform the corresponding action.

The encoding itself is based on a suitable projection of the transition relation introduced in the previous section in each variable state. This encoding technique is similar to the one followed during the logical design of a sequential circuit, where suitable expressions are *independently* and *consistently* provided for each state and output variable in terms of the previous values of the state and the input variables [24]. The input variable `Act` will stay unassigned, meaning that the user is free to choose any of the possible actions in the context of the game. For the rest of the variables, sets

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

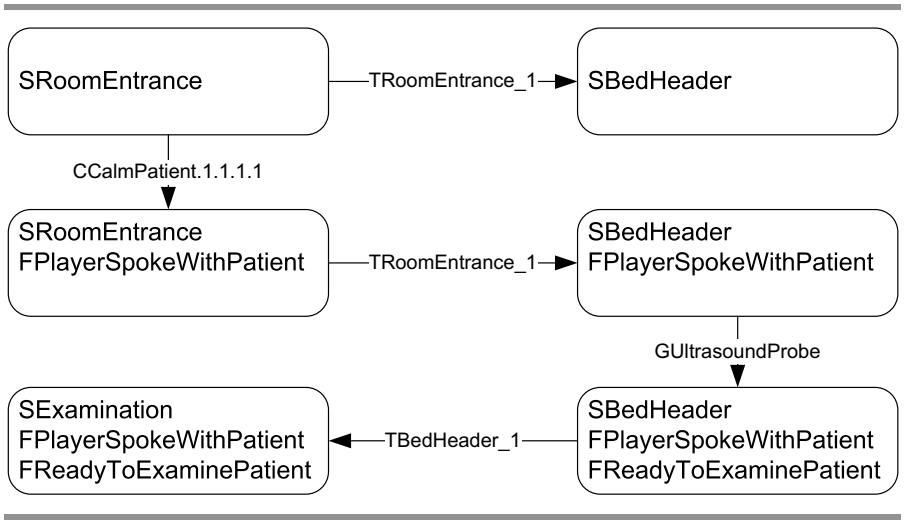

Fig. 9. Transition system for the game in the case-study.

of assignments are provided to determine their `init` and `next` values:

- The `init` values will be derived from the initial state of the game. Indeed, each $\texttt{Oo}_i$ (respectively $\texttt{Ff}_j$) variable will have `true` assigned when the corresponding propositions are included in such a state, and `false` otherwise. `Scene` will be initialized to the initial scene indicated in such a state.
- The assignments of `next` values consider each type of variable individually, as well as those relevant actions that can change their values. For each flag f_i a set $\text{RA}^G(f_i)$ formed by those actions whose effects contain f_i is considered during the generation process. For each object o_i, two different sets will be considered: one with all the *grab* actions that can insert the object in the inventory -$\text{IA}^G(o_i)$, and another one with all the *use*, *use-with* and *give-to* actions that can erase the object from this inventory -$\text{EA}^G(o_i)$. Finally, for each possible value s_i of the scene we will consider the set $\text{MA}^G(s_i)$ of all those actions that can shift the action to scene s_i. The computation of the next values also needs to consider the executability conditions $\text{EC}^G(a)$ of each user's action a. This condition is an expression that is true if and only if action a has been actually executed in the last transition. Its actual encoding, which we will omit for the sake of conciseness, states that an action was executed if its precondition allowed it and the user actually selected it. Finally, the actual assignments for the NuSMV model are determined in terms of the relevant action sets and their applicability conditions. These are detailed in Table 4.
- The properties to check must also be added to the resulting encodings. For this purpose, properties are expanded to raw CTL, then translated into the NuSMV CTL assertion language and the result added as a `CTLSPEC` clause.

For the case-study introduced in Section 4, the resulting NuSMV encoding is shown in Fig. 10. Following the guidelines in Section 5.4, the encoding first defines the state variables that correspond to the scene (line 3), the flags (lines 4–6), and the objects that the player can grab (line 7). The input variable `Act` that corresponds to player's inputs is defined in lines 9–10. This variable takes values in an enumeration of all the possible actions that the player can make in the game. In this case, he/she can traverse two exits (i.e. `T_BedHeader_1_1` and `T_RoomEntrance_1_1`), grab one object (i.e. `G_UltrasoundProbe_1`), and keep up a conversation (i.e. `C_CalmPatient_1_1_1_1_1`). Lines 12-17 define the expressions

Table 4
Characterization of the next expressions

Variable	Expression to be assigned
`next(Scene)`	*case* $\bigvee_{a\in \text{MA}^G(s_1)} \text{EC}^G(a) : s_1;$ ... $\bigvee_{a\in \text{MA}^G(s_n)} \text{EC}^G(a) : s_n;$ $1 : \textit{Scene};$ *esac*
$\texttt{next}(\texttt{F}f_i)$	$\left(\bigvee_{a\in \text{RA}^G(f_i)} \text{EC}^G(a)\right) \vee \texttt{F}f_i$
$\texttt{next}(\texttt{O}o_i)$	$\left(\bigvee_{a\in \text{IA}^G(o_i)} \text{EC}^G(a)\right) \vee \left(\texttt{O}o_i \wedge \neg \bigvee_{a\in \text{EA}^G(o_i)} \text{EC}^G(a)\right)$

$\neg$, $\wedge$, and $\vee$ are translated in NuSMV by !, &, and | respectively.

that will determine the applicability of the previous actions at every state of the checking process. For instance, lines 12–13 establish that the player can traverse exit *T_BedHeader_1* if he/she selects that action (i.e. `T_BedHeader_1_1`) and the conditions specified by the game to traverse it hold (in this case the player must be in the scene *BedHeader* and the flag *ReadyToExaminePatient* must be active). The next elements in the encoding are the initialization of the variables. Lines 20–22 initialize the variables according to the description of the game. For instance, the variable Scene takes the value of the scene marked as *start* in the description of the game, *RoomEntrance* in this case. Lines 25–26 also perform an initialization of variables but, in this case, the values assigned come from the adaptation of the game. Lines 28–36 perform the calculus of the next values of the different state variables according to the expressions in Table 4. Finally, line 38 contains the property to verify, which was described at the end of Section 5.2, appropriately expanded and encoded in the NuSMV format.

5.5. Verifying properties and generating counterexamples

Once the verification model has been generated, this model is fed to the model checker. As mentioned before, currently we are using NuSMV, although it could be substituted by another model checker if required. The model checker will inform where the properties to verify hold, or, otherwise, will generate a counterexample trace. This trace will illustrate the reason why the property fails in terms of a sequence of states.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

```
MODULE main
VAR
   S_Scene : { BedHeader, RoomEntrance, Examination };
   F_ReadyToExaminePatient : boolean;
   F_UltrasoundNotAvailable : boolean;
   F_PlayerSpokeWithPatient : boolean;
   O_UltrasoundProbe : boolean;
IVAR
   Act :  { T_BedHeader_1_1, T_RoomEntrance_1_1,
            G_UltrasoundProbe_1, C_CalmPatient_1_1_1_1_1 };

DEFINE EC_T_BedHeader_1_1 := (Act = T_BedHeader_1_1) & (S_Scene = BedHeader &
   F_ReadyToExaminePatient);
DEFINE EC_T_RoomEntrance_1_1 := (Act = T_RoomEntrance_1_1) & (S_Scene = RoomEntrance);
DEFINE EC_G_UltrasoundProbe_1 := (Act = G_UltrasoundProbe_1) & (S_Scene = BedHeader &
   !F_UltrasoundNotAvailable & F_PlayerSpokeWithPatient);
DEFINE EC_C_CalmPatient_1_1_1_1_1 := (Act = C_CalmPatient_1_1_1_1_1) &
   (S_Scene = RoomEntrance);
ASSIGN
   init(S_Scene) := RoomEntrance;
   init(F_PlayerSpokeWithPatient) := 0;
   init(O_UltrasoundProbe) := 0;

-- From the game adaptation
   init(F_ReadyToExaminePatient) := 1;
   init(F_UltrasoundNotAvailable) := 1;

   next(S_Scene) := case
      EC_T_BedHeader_1_1    : Examination;
      EC_T_RoomEntrance_1_1 : BedHeader;
      1                     : S_Scene;
   esac;
   next(F_ReadyToExaminePatient) := (EC_G_UltrasoundProbe_1) | F_ReadyToExaminePatient;
   next(F_UltrasoundNotAvailable) := F_UltrasoundNotAvailable;
   next(F_PlayerSpokeWithPatient) := (EC_C_CalmPatient_1_1_1_1_1) | F_PlayerSpokeWithPatient;
   next(O_UltrasoundProbe) := (EC_G_UltrasoundProbe_1) | O_UltrasoundProbe;

CTLSPEC !E[ !F_PlayerSpokeWithPatient U (S_Scene = Examination)]
```

Fig. 10. NuSMV encoding for the case-study.

Turning back to the case-study, the simple game focuses on the need for having a proper communication with the patient. For this purpose, the player cannot examine the patient without previously speaking with him/her. In the specification of the game (see Fig. 4), this is assured with a sequence of conditions: to examine the patient (i.e. scene *Examination*), the player has to grab the ultrasound device (i.e. flag *ReadyToExaminePatient* that is activated by the corresponding *grab* action); in order to do so, the player needs to speak with the patient before (i.e. flag *PlayerSpokeWithPatient* that becomes active after keeping the conversation *CalmPatient*). Thus, in the usual execution of the game, this constraint holds, and the model output is positive as seen in Fig. 11a.

However, adaptations of the game allow some of the flags involved to be changed without performing the triggering actions. This is the case for the flag *ReadyToExaminePatient*, which is activated in the adaptation rule described in Fig. 6. When checking the corresponding encoding in NuSMV (see Fig. 10), it turns out that the player is able to begin the examination without a previous conversation with the patient. Fig. 11b shows this situation. From the initial state (i.e. `State: 1.1` of the model checker), the user can traverse the door (i.e. `Act = T_RoomEntrance_1_1`) to go the scene corresponding to the bed header (i.e. `State: 1.2` of the model checker). After reaching `State: 1.2`, the player just needs to traverse the last door (i.e. `Act = T_BedHeader_1_1`) to begin the examination. Given the adapted values of the flags, all the traversing actions are allowed. Thus, the specified constraint can be violated due to the adaptation of the game.

In the trace of Fig. 11b, it must be noted that states of the model checker are characterized for the whole set of variables in the model and their specific values. For instance, `State: 1.1` is not just characterized by the scene *RoomEntrance*, but also by the values of the flags and the objects in the inventory.

As the traces in Fig. 11 illustrate, the outcome of a model checker like NuSMV is not well-suited to analyze the game for at least two reasons. First, the explanation is given in terms of the transition system for the game and CTL primitives. This output is likely going to be useless for game writers in order to verify and fix the game. The second reason is the verbosity of the textual output. The model checker describes the violations of properties as sequences of states (characterized by their variables and the other expressions in the encodings) and the values that the input variable `Act` takes in the transitions. Given that the steps in the trace where the player makes the unexpected actions that break the property are not known, the writers may need to examine every single step and variable in the counterexample trace. For these reasons, the ⟨e-Adventure⟩ framework for verification needs further processing of these outputs to facilitate their interpretation.

5.6. Generating the animation

In this case, explanations about problems with the properties should also follow the principles of the domain-specific language approach. That is, they must be provided in a language close to the language that is common in game writing, regardless of the underlying use of a model checker. Hence, the counterexamples provided by the model checker are not directly shown to game writers. Instead, the framework for the verification of properties in ⟨e-Adventure⟩ plays an animation where writers can see a sequence of the player's interactions with other components of the game. This animation shows the trace as it can be played in the en-

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

(a)

```
NuSMV > read_model -i eAdventure.txt
NuSMV > go
NuSMV > check_ctlspec
-- specification !E [ !F_PlayerSpokeWithPatient U
        S_Scene = Examination ]   is true
```

(b)

```
NuSMV > read_model -i eAdventure.txt
NuSMV > go
NuSMV > check_ctlspec
-- specification !E [ !F_PlayerSpokeWithPatient U
        S_Scene = Examination ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  S_Scene = RoomEntrance
  F_ReadyToExaminePatient = 1
  F_UltrasoundNotAvailable = 1
  F_PlayerSpokeWithPatient = 0
  O_UltrasoundProbe = 0
  EC_T_BedHeader_1_1 = 0
  EC_T_PatientRoom_1_1 = 1
  EC_G_UltrasoundProbe_1 = 0
  EC_C_CalmPatient_1 = 0
-> Input: 1.2 <-
  Act = T_RoomEntrance_1_1
-> State: 1.2 <-
  S_Scene = BedHeader
  EC_T_BedHeader_1_1 = 1
  EC_T_RoomEntrance_1_1 = 0
  EC_G_UltrasoundProbe_1 = 0
  EC_C_CalmPatient_1 = 0
-> Input: 1.3 <-
  Act = T_BedHeader_1_1
-> State: 1.3 <-
  S_Scene = Examination
```

Fig. 11. Sample outputs from the model checker: (a) a trace where the property holds; (b) a trace with a counter-example.

gine. In this way, the writers do not see the evolution of the transition system and the CTL formulae, but how the player acts to violate their design. As mentioned in Section 5.1, the functionality alluded is obtained by means of the *Animation Generator*, which transforms a trace of the model checker in an *animation description*, and the ⟨*e-Adventure*⟩ *Robot*, which plays that description.

Turning back to the case-study, the property focuses on the fact that the player cannot examine the patient without previously speaking with him/her. The results for this property, both for the adapted and non-adapted game, have already been shown in Fig. 11. The sequence of actions appearing in the counterexample of Fig. 11b (i.e. `T_RoomEntrance_1_1` and `T_BedHeader_1_1`) corresponding to the violation of the property can be parsed to generate a sequence of actions suitable for the ⟨e-Adventure⟩ robot. The XML sequence of actions appears in Fig. 12. The ⟨e-Adventure⟩ robot can take this as an input and generate an animation displaying the sequence as seen in Fig. 13.

6. Related work

Model-checking has a long tradition in verifying software systems with respect to their specifications, leading to the so-called *software model-checking* approach. This approach has paid attention to both general and domain-specific contexts. In this section we briefly examine some of these efforts both at the specification and programming level. We also examine some efforts oriented to facilitate the specification of the properties to check, since it is a key aspect to ensure the adoption of these technologies.

```
<actions>
  <sequence>
    <action type="exit" idTarget="BedHeader"/>
    <action type="exit" idTarget="Examination"/>
  </sequence>
</actions>
```

Fig. 12. Sequence of actions for the ⟨e-Adventure⟩ robot corresponding to the violation of property "`before(SExamination, FPlayerSpokeWithPatient)`".

6.1. Model-checking at the design level

Model-checking has been used to check properties of the software systems at the *design* and the *specification* levels. Since at these levels systems can be described omitting low-level details, many times the resulting descriptions are amenable to supporting automatic extraction of verification models. In [9] model-checking is used to check the properties of high-level system specifications written in RSML, the requirement state machine language, a state machine language based on the *statechart* formalism [27]. This language can be fully translated into SMV [34] in order to verify the properties expressed in CTL. An alternative translation of statecharts into SMV is reported in [14], where the emphasis is put on preserving the hierarchical structure of the formalism in the resulting SMV models, using the SMV modularization facilities. In [36], SPIN [30] is used as the model-checking framework, as well as in [32], where the focus is put on UML statecharts. The work in [25] proposes two different translations of UML activity diagrams. These translations map activity diagrams in finite-state machines, which are subsequently mapped into the NuSMV language.

While these approaches are based on some kind of domain-specific language, those languages are oriented to specify high-level (design, analysis, etc.) views of a system that subsequently must be mapped into a final implementation. In this way, they do not prevent properties from being violated in those implementations, since additional bugs can be introduced during the detailed design

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

1. Initial scene *RoomEntrance*

2. Changing position to *BedHeader* to perform examination

3. Completing the examination

Fig. 13. Animation in the ⟨e-Adventure⟩ robot of the violation of property "`before(SExamination,FPlayerSpokeWithPatient)`".

and the implementation processes. This is avoided in ⟨e-Adventure⟩ by adopting a domain-specific generative approach where the final running videogames are automatically generated from the high-level specification, that is, the ⟨e-Adventure⟩ document that describes the game.

6.2. Model-checking for general-purpose programming languages

There are also several efforts in software model-checking dealing with general-purpose programming languages that directly operate on the state space of the programs. In order to deal with the undecidability issues, some approaches are focused on subsets of the source language. For instance, the system described in [5] deals with C boolean programs (i.e. programs including only boolean variables) and can solve the reachability problem for any sentence of the program using symbolic model-checking techniques.

Other systems sacrifice completeness in order to deal with the whole language. An example is the system proposed in [42], which works with the whole C/C++ language. Properties to verify are included as asserts in those code points where they must hold. The states of the transition system related with the program are stored in a hash table optimized with suitable compression techniques. The checking process can finish with all the properties verified, by finding some violation, or by running out of memory. Other more sophisticated proposals are focused on extracting models from the programs and expressing these models in a suitable language for a model checker. Some approaches rely on predefined translation schemas. The cost to pay is to deal only with a subset of the source languages. An example is JPF, the Java PathFinder described in [29,54], which automatically translates a non-trivial subset of the Java language into PROMELA, the specification language of SPIN.

Other approaches introduce auxiliary specifications to guide the extraction process for each particular program. In [31] a verification model's skeleton is automatically extracted by considering only the source program's control structures. Next, this skeleton is populated with relevant actions, defined in a lookup table that specifies how constructs in the source program are translated into the verification model. Finally, the populated skeleton is used to fill a model template, which declares the required variables, and which must also be manually provided. The result is a working verification model written in PROMELA. In [33] a similar solution is proposed, but model extraction is firstly articulated by slicing the source program, written in C, into those parts relevant for checking the properties. The slicing process uses a simple enumeration of relevant state variables and subroutines. A set of translation patterns are then used to translate the sliced program into a suitable verification model, expressed in the formalism of the Murφ model checker [19]. Bandera [17], a sophisticated framework for extracting verification models from Java programs, uses the properties to be proven in order to automatically slice the program. The sliced programs are then abstracted by using an *abstraction specification* written in BASL, the Bandera abstraction specification language. Specifications in BASL consist of abstract domains, functions mapping concrete Java values in abstract domains, and abstract operations for *each* concrete method in the Java classes. The resulting models can be further translated into the specification languages of different model checkers (e.g. SMV, SPIN, etc.).

While all these approaches focus on general-purpose programming languages, our work is organized around a domain-specific solution. This allows us to provide a complete method for the automatic extraction of verification models from the game descriptions. More important, it also allows the active participation of game writers as well as developers during the verification process, since properties can be stated in domain-specific terms (that is, flags, objects and scenes, instead of sentences and variables in general-purpose programming languages).

6.3. Model-checking in specific domains

The use of model-checking to verify software in specific domains is closer in spirit to the work described in the present paper. In [50] model-checking is proposed to verify properties of hypertext systems. For this purpose hypertexts are interpreted as a kind of finite transition system called *links automata*. These automata are used to check properties expressed in HTL*, a language based on the branching-time temporal logic CTL* [23]. The work in [46,47] is intended to model check web applications. They propose the automatic extraction of a finite-state model of a web application as the result of the automatic analysis of web sites. This model is then implemented using NuSMV. The approach presents some difficulties with server-side and client-side dynamic facilities (i.e. the presence of scripting code). In [21] the focus is shifted to UML-based models of web applications using the domain-specific UML profile proposed in [15]. In [3,4] an experience in the e-learning domain is reported, where SPIN is used to ensure the preservation of properties formulated on automatically devised curricula when they are adapted by the learners.

Regarding the use of model-checking in combination with domain-specific languages, the methods used are similar to those de-

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

scribed in the present paper: to define a translation which allows for the interpretation of each description in the language as a finite transition system. In [28] SPIN is used to check properties of specifications in the Executive Support Language (ESL), a domain-specific language for the construction of control systems for autonomous robots and space crafts. The work in [10] presents the domain-specific language Teapot for developing cache coherence protocols, and a verification approach based on its translation to the Murφ checker. In [43] SPIN is used together with the web-services flow language (WSFL), a domain-specific language to orchestrate web-services. Similar approaches are used in [55] (work which involves the MAP dialogue protocol language for multi-agent systems supported by web-services), in [56] (work centered on the web-service choreography description language WS-CDL), and in [44] (work which uses the BPEL4WS language and verification models based on timed automata checked using the UPPAL model checker).

6.4. Facilitating the specification of properties

An important aspect of our work is to promote the active involvement of game writers and developers, who are not necessarily experts in model-checking technologies, in the whole verification process. For this purpose, it is necessary to provide some means of making model-checking techniques more usable and accessible for those non-experts. This subject has also been addressed by several works centered on providing easier-to-use formalisms to express properties. In LUSTRE [26], a programming language oriented to the development of reactive systems, it is possible to express safety properties in a lineal temporal logic (LTL; see [23]) and to use the procedural abstraction mechanisms of the language to define new derived operators. This resembles the macro facility included in the ⟨e-Adventure⟩ property language. The works in [6,20] propose graphical notations for LTL formulae. Other approaches, based on controlled English-like grammars, are proposed in [18,22]. In [16] the specification language of the Bandera system is described. This language supports the use of the common patterns used in temporal logics-based specifications identified in [22]. The Bandera system also supports the inclusion of new user-defined patterns, although the extension mechanisms are not described in the cited works.

In our proposal, the use of a simple macro-definition mechanism in the specification language, together with the ⟨e-Adventure⟩ robot, allows experts in temporal logics, developers, and game writers to effectively collaborate for model-checking conversational videogames.

7. Conclusions and future work

In all software development projects, the verification of the validity of the product is a key aspect. Much work has been done on studying good practices that prevent software bugs, including verification of execution properties. However, when dealing with *point and click* adventure videogames, a completely bug-free game is not enough to produce a flawless game. These games execute interactive narratives in which certain actions unlock other actions, with all the interactions being interconnected and sometimes solvable in different ways. This introduces a new complexity in which the description of the game (as opposed to its implementation) can spoil the product by allowing the player to drive the game into an undesired narrative state. In general terms, game development projects tackle this problem by devoting vast amounts of resources to a very thorough testing and quality assurance process, in which several testers play the game trying out every possible action to make sure the game is correctly structured. However, the process is far from perfect: subtle interactions may run through the assessment process undetected; when an error is detected, it is sometimes difficult to ascertain which of the steps actually caused the problem; finally, even if the problem is detected, its correction supposes new requirements stated by the writers and implemented by developers, which can lead to the introduction of new errors.

In this paper, we have introduced a framework for the semi-automatic verification of *point and click* adventure games described with domain-specific languages, using model-checking techniques. Although this process does not fully eliminate the need for beta-testing and quality assurance in this family of videogames (it does not cover aspects such as programming errors, quality of the art assets, or fine-tuning errors in the content), the automatic verification of key runtime properties can significantly reduce the quality assurance effort required by these initiatives.

The framework was implemented around the previously existing ⟨e-Adventure⟩ development platform, where game writers describe the games using the ⟨e-Adventure⟩ language and the ⟨e-Adventure⟩ editor. With this extension, they can also informally state the desirable properties to be checked in the games. Experts in model-checking and temporal logics are able to analyze these properties and to provide suitable operators for them using the macro-definition facility of the ⟨e-Adventure⟩ assertion language. These experts or other developers can use these operators to formalize the intended properties. This can contribute to managing the complexity of the narratives and reducing the additional burden introduced by the adaptation mechanisms supported by ⟨e-Adventure⟩.

In order to develop the verification framework, the ⟨e-Adventure⟩ execution semantics have been abstracted to automatically yield labeled transition systems for each game, which are verified with a model checker. The verification models are automatically derived from the games without further human intervention, and properties are automatically translated from the specifications by expanding the macros and by translating the resulting formulae to the specific formats supported by the model checker. Nevertheless, the process itself is not dependent on specific tools as it can be accommodated by tuning the corresponding transformations. As explained in Section 5.1, if the model checker is changed, only the *Verification Model Generator* and the *Animation Generator* must be modified to consider the new checker. These are the only tools in the ⟨e-Adventure⟩ framework for verification that produce/generate information dependent on the checker. Currently, we are using NuSMV as the model-checking tool of choice. It allows us to deal with state spaces of around 10^{20} states. Despite this huge size, this checker may not be the proper solution for some problems. It may be advisable to use model checkers based on alternative approaches such as the unbounded or partial order reduction model-checking for certain games or properties.

It must be noted that the application of the process described requires a first stage characterized by an intense effort by both developers and experts in model-checking, which is also costly. However, this effort is restricted to the formalization stage, which is related to the game language, not to the individual games. After the initial formalization investment, applying the approach to each individual game has a much lower cost. The writers, even though they may need some help to define properties using the assertion language, can run the model-checking apparatus on each adaptation of the game, analyze the potential violations in game-specific terms using the ⟨e-Adventure⟩ robot, and modify the games to solve these violations. This allows them to gain a great deal of control of the overall verification process.

This paper also explains how the approach was employed in a scenario oriented to train medical doctors in some specific procedures as part of a project jointly developed by the Complutense

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

University of Madrid, the Harvard Medical School and the Massachusetts General Hospital. The simplified case-study has shown the ability of the verification framework to deal with complex temporal properties about the potential ways of playing a game. The semi-automated process facilitates finding errors about these properties and shows examples that may help the author to correct the game.

It should also be noted that the presentation of the entire process in this paper is closely tied to the current implementation of the ⟨e-Adventure⟩ platform. However, the possibility of reducing the quality assurance costs by employing semi-automated property-verification processes should be applicable to other adventure game development platforms, given that the problem tackled is present in all adventure game developments and many of the features of ⟨e-Adventure⟩ on which the solution relies are also present in other frameworks.

As future work, we are planning to provide additional support for the specification of properties, which should contribute to minimizing the need for the collaboration of experts in temporal logics and developers in the customization of the property specification language. Right now, it might be argued that for small projects, the cost involved in the formalization of the properties informally stated by the game writers may not be compensated by savings in quality controls. Ideally, game writers should be able to formalize many of these properties as well as define their own abstractions without external (and costly) assistance. For this purpose, it is necessary to analyze suitable authoring metaphors for temporal properties, which could be further supported by graphical and/or natural language based notations. With the simplification of the verification process and the augmentation of the expressive power of the macro system, it should be possible to adopt this kind of approach to verify arbitrarily complex adventure games with little effort and cost. Regarding performance and flexibility, we are also planning to incorporate other model-checking tools into our framework and enable the automatic selection of the most suitable tool, depending on the nature of the properties to be checked.

Acknowledgements

The Spanish Committees of Science and Innovation and of Industry, Tourism and Commerce (Project Nos. TIN2005-08501-C03-01, TIN2005-08788-C04-01, TSI-020301-2008-19 and TIN2007-68125-C02-01) has partially supported this work, as well as the Regional Government of Madrid (Grant No. 4155/2005), the Complutense University of Madrid (research group 921340, Santander/UCM Project PR24/07 – 15865) and the EU Alfa project CID (II-0511-A). Thanks to Dr. Carl Blesius and Dr. Paul Curier from Harvard Medical School and the Massachusetts General Hospital for their permission to use the CVC-Insertion Protocol game for the case-study.

Appendix. Characterization of the transition relations of ⟨e-Adventure⟩ verification models

In this Appendix we provide a formalization of the transition relations associated with ⟨e-Adventure⟩ verification models. We start by introducing a verification-oriented abstract syntax for ⟨e-Adventure⟩ games. This representation will include the relevant behavioral details for model-checking the games mentioned in Section 5.3, while excluding the irrelevant ones (e.g, presentation, actual conversations, etc.). According to this syntax, a game description will be a set of *information elements*. We distinguish between *game description* and *adaptation description* information elements. Table 5 summarizes the possible game description information elements, and Table 6 the adaptation of description ones. Notice that several of these elements involve some aspects that deserve additional remarks for their understanding:

- Preconditions, which determine when a situation (e.g. performing an action) is possible in the game, are given in the form ⟨*F+*, *F-*⟩, where *F+* is a set of flags that must be active (i.e. with value true) and *F-* a set of flags that must be inactive (i.e. with a false value).

Table 5
Game description information elements

Information element for description	Intended meaning
⟨*next-scene*,*s*,*i*,*ns*,*c*,*e*,*j*⟩	*ns* is a scene or "cutscene" that can be reached from the scene or "cutscene" *s* by traversing exit number *i* of the scene. *c* is the condition that must hold in the current state of the game in order to be able to go through this exit and *e* is the set of flags activated by this action. This tuple denotes the *j*th form of making this traversal
⟨*object*,*s*,*o*,*c*⟩	When condition *c* holds, object *o* is visible in scene *s*
⟨*character*,*s*,*ch*,*c*⟩	When condition *c* holds, character *ch* is visible in scene *s*
⟨*grab*,*o*,*c*,*e*,*j*⟩	The player can grab object *o* when condition *c* holds and object *o* is visible. The result of this action is that the object is included in the inventory and flags in *e* are activated. This is the *j*th way to perform this action
⟨*grab*,*o*,*c*,*e*,*s*,*j*⟩	Its meaning is the same as ⟨*grab*,*o*,*c*,*e*,*j*⟩ proposition but it also changes the current scene to *s*
⟨*use*,*o*,*c*,*e*,*j*⟩	The player can use object *o* when condition *c* holds and object *o* is visible. The result of this action is that flags in *e* are activated. This is the *j*th way to perform this action
⟨*use*,*o*,*c*,*e*,*s*,*j*⟩	Its meaning is the same as ⟨*use*,*o*,*c*,*e*,*j*⟩ proposition but it also changes the current scene to *s*
⟨*use-with*,*o*,*o′*,*c*,*e*,*cons*,*j*⟩	Object *o* can be combined with object *o′* when condition *c* holds, *o* is in the player's inventory, and *o′* is visible. This action activates the flags in *e* and removes from the inventory the objects in the set *cons*. The set *cons* can be only one of two values, the empty set or the set that contains just the object *o*. This is the *j*th way to perform this action
⟨*use-with*,*o*,*o′*,*c*,*e*,*s*,*cons*,*j*⟩	Its meaning is the same as the ⟨*use-with*,*o*,*o′*,*c*,*e*,*cons*,*j*⟩ proposition but it also implies going to scene *s*
⟨*give-to*,*o*,*ch*,*c*,*e*,*cons*,*j*⟩	Object *o* can be given to character *ch* when condition *c* holds, *o* is in the player's inventory, and *ch* is visible. This action activates the flags in *e* and removes from the inventory the objects in set *cons*. Set *cons* can be the empty set or the set that just contains object *o*. This is the *j*th way to perform this action
⟨*give-to*,*o*,*ch*,*c*,*e*,*s*,*cons*,*j*⟩	Its meaning is the same as the ⟨*give-to*,*o*,*ch*,*c*,*e*,*cons*,*j*⟩ proposition but it also changes the current scene to *s*
⟨*conversation*,*ch*,*p*,*c*,*e*,*j*⟩	Conversation *p* can be held with the character *ch* when condition *c* holds and character *ch* is visible. The result of this action is the activation of the flags in *e*. This is the *j*th way to perform this action. Here *p* is in fact the codification in decimal format of a path in a conversation of the original game (see Section 3.1), like *c*23.1.2.1.3.5

Table 6
Adaptation description propositions

Information element for adaptation	Intended meaning
⟨*start*,*s*⟩	Set scene *s* as the initial one for this instance of the game
⟨*in-inventory*,*o*⟩	Include object *o* among those that are initially in the inventory. By default, objects are not initially in the inventory
⟨*active*,*f*⟩	Set flag *f* as initially active. The default initial value for flags is false

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

Table 7
Rules formalizing the transition relation for ⟨e-Adventure⟩ verification models

Rule	Informal name and intended meaning
$\dfrac{\left(\begin{array}{l}\langle \text{next-scene}, s', i, s, c, e, \_\rangle \in R^G \wedge \\ \mathrm{H}(c, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{Ts', i_G} [(q - \{Ss'\}) \cup \{Ss\} \cup \mathrm{P_F}(e)]}$	*Traversing an exit*. The user wants to traverse exit *i*, the current scene contains such an exit, and it is practicable in the current state of the game. The current scene is changed to the next-scene and the flags activated by the action are added to the state
$\dfrac{\left(\begin{array}{l}\langle \text{object}, s', o, c_v\rangle \in R^G \wedge \mathrm{H}(c_v, [q]) \wedge \\ \left(\begin{array}{l}(\langle \text{grab}, o, c_a, e, \_\rangle \in R^G \wedge NS = Ss') \vee \\ (\langle \text{grab}, o, c_a, e, s, \_\rangle \in R^G \wedge NS = Ss)\end{array}\right) \wedge \\ \mathrm{H}(c_a, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{Go_G} [(q - \{Ss'\}) \cup \{NS, Oo\} \cup \mathrm{P_F}(e)]}$	*Grabbing an object*. The user wants to grab an object, which is visible in the current scene. The action is also practicable in the current state. If a scene change is specified, the new scene is set as the current one. Otherwise, the current scene stays unchanged. The object is then included in the player's inventory and the activated flags are also recorded
$\dfrac{\left(\begin{array}{l}\langle \text{object}, s', o, c_v\rangle \in R^G \wedge \mathrm{H}(c_v, [q]) \wedge \\ \left(\begin{array}{l}(\langle \text{use}, o, c_a, e, \_\rangle \in R^G \wedge NS = Ss') \vee \\ (\langle \text{use}, o, c_a, e, s, \_\rangle \in R^G \wedge NS = Ss)\end{array}\right) \wedge \\ \mathrm{H}(c_a, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{Uo_G} [(q - \{Ss'\}) \cup \{NS\} \cup \mathrm{P_F}(e)]}$	*Using an object*. The user wants to use an object, which is visible in the current scene. The action is also practicable in the current state. If a scene change is specified, the new scene is set as the current one. Otherwise, the current scene stays unchanged. The activated flags are recorded
$\dfrac{\left(\begin{array}{l}Oo \in q \wedge \langle \text{object}, s', o', c_v\rangle \in R^G \wedge \mathrm{H}(c_v, [q]) \wedge \\ \left(\begin{array}{l}(\langle \text{use-with}, o, o', c_a, e, o_c, \_\rangle \in R^G \wedge NS = Ss') \vee \\ (\langle \text{use-with}, o, o', c_a, e, s, o_c, \_\rangle \in R^G \wedge NS = Ss)\end{array}\right) \wedge \\ \mathrm{H}(c_a, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{\mathrm{UWo\_o'}_G} [(q - (\{Ss'\} \cup \mathrm{P_O}(o_c))) \cup \{NS\} \cup \mathrm{P_F}(e)]}$	*Using an object with another one*. The first object is in the inventory, the other one is visible in the current scene and the action is practicable. The inventory is modified as indicated, maybe erasing the used object. Also, should it be indicated, the scene is changed in accordance. Finally, the flags activated by the action are recorded
$\dfrac{\left(\begin{array}{l}Oo \in q \wedge \langle \text{character}, s', ch, c_v\rangle \in R^G \wedge \mathrm{H}(c_v, [q]) \wedge \\ \left(\begin{array}{l}(\langle \text{give-to}, o, ch, c_a, e, o_c, \_\rangle \in R^G \wedge NS = Ss') \vee \\ (\langle \text{give-to}, o, ch, c_a, e, s, o_c, \_\rangle \in R^G \wedge NS = Ss)\end{array}\right) \wedge \\ \mathrm{H}(c_a, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{\mathrm{GTo\_ch}_G} [(q - (\{Ss'\} \cup \mathrm{P_O}(o_c))) \cup \{NS\} \cup \mathrm{P_F}(e)]}$	*Giving an object to a character*. The object is in the inventory, the character is visible in the current scene and the action is practicable. The inventory is modified as indicated, and the scene is changed if indicated. The new activated flags are recorded
$\dfrac{\left(\begin{array}{l}\langle \text{character}, s', ch, c_v\rangle \in R^G \wedge \mathrm{H}(c_v, [q]) \wedge \\ \langle \text{conversation}, ch, p, c_a, e, \_\rangle \in R^G \wedge \\ \mathrm{H}(c_a, [q]) \wedge Ss' = \mathrm{S}([q])\end{array}\right)}{[q] \xrightarrow{Cp_G} [q \cup \mathrm{P_F}(e)]}$	*Talking*. The user wants to follow a conversation path. The character, who is able to talk and hold such a conversation, is visible in the current scene, and the conversation is practicable. The flags activated by the conversation are recorded in the new state

- Another aspect to consider here is how each action changes the state of the game. Different types of actions have particular effects that are achieved only through them, like adding an object to the inventory or holding a conversation. In addition, all the actions can activate an arbitrary set of flags associated to them. Pay attention to the fact that no action can deactivate flags, although an initial or adapted value of these may be false.
- Finally, some elements include an index *j*, which when appearing is in the last position. It is used to distinguish between alternatives of an action easily, since the ⟨e-Adventure⟩ language does not preclude several declarations of the same action with different preconditions or effects (e.g. traversing a given exit in a scene or combining an object with another one).

Once a suitable verification-oriented abstract syntax for ⟨e-Adventure⟩ games has been established, it is possible to formalize the rules which characterize these games as state-based systems. These rules are shown in Table 7. Notice that to denote states, two notations are used: q indicates that state as a whole while $[q]$ denotes the state whose associated set of propositions is q (i.e. $P^G([q]) = q$ for each state $[q]$). In these rules, $\mathrm{H}(c, q)$ states that condition c holds provided the state q (i.e. $\mathrm{H}(\langle F+, F-\rangle, [q]) \Leftrightarrow (F+ \subseteq q \wedge F- \cap q = \emptyset)$). In its turn, if Φ is a set, $\mathrm{P}_W(\Phi)$ denotes the set resulting from prefixing each element in Φ with the modifier W (e.g. if e is a set of flags, $P_F(e)$ is the set of associated flag propositions). With $\mathrm{S}(q)$ we denote the scene proposition included in state q. Finally, with R^G we denote the abstract representation of game G. Notice also that, according to the rules, if $q_i \xrightarrow{l}{}^{G} q_j$, then the flags in q_i will be contained in the flags in q_j. This formally establishes a monotonic notion of logical truth for flags in ⟨e-Adventure⟩: Actions can activate flags but never deactivate them.

References

[1] S.B. Akers, Binary decision diagrams, IEEE Transactions on Computers C-27 (6) (1978) 516–590.
[2] A. Amory, Building an educational adventure game: theory, design and lessons, Journal of Interactive Learning Research 12 (2–3) (2001) 249–263.
[3] M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, V. Patti, Reasoning-based curriculum sequencing and validation: integration in a service-oriented architecture, in: Second European Conference on Technology Enhanced Learning, Lecture Notes in Computer Science, 4753, Springer, 2007, pp. 426–431.
[4] M. Baldoni, E. Marengo, Curriculum model-checking: declarative representation and verification of properties, in: Second European Conference on Technology Enhanced Learning, Lecture Notes in Computer Science, 4753, Springer, 2007, pp. 432–437.
[5] T. Ball, S.K. Rajamani, Bebop: a symbolic model checker for Boolean programs, in: Proceedings of the 7th International SPIN Workshop, Lecture Notes in Computer Science, 1885, Springer, 2000, pp. 113–130.
[6] A. Browne, Z. Manna, H.B. Sipma, Generalized temporal verification diagrams, in: Proceedings of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, 1026, Springer, 1995, pp. 484–498.
[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model-checking: 10^{20} states and beyond, Informatics Computing 98 (2) (1992) 142–170.
[8] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, N. Sinha, Concurrent software verification with states, events, and deadlocks, Formal Aspects of Computing 17 (4) (2005) 461–483.
[9] W. Chan, R.J. Anderson, P. Beams, S. Burns, F. Modugno, D. Notkin, J.D. Reese, Model-checking large software specifications, IEEE Transactions on Software Engineering 24 (7) (1998) 498–520.
[10] S. Chandra, B. Richards, J.R. Larus, Teapot: a domain-specific language for writing cache coherence protocols, IEEE Transactions on Software Engineering 25 (3) (1999) 317–333.
[11] A. Cimatti, E. Clarker, F. Giunchiglia, M. Roveri, NUSMV: a new symbolic model checker, International Journal on Software Tools for Technology Transfer 2 (4) (2000) 410–425.
[12] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (2) (1986) 244–263.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003

[13] E.M. Clarke, O. Grumberg, D.A. Peled, Model-Checking, The MIT Press, Cambridge, MA, USA, 2000.
[14] E.M. Clarke, W. Heinle, Modular translation of statecharts to SMV, Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, 2000.
[15] J. Conallen, Modeling web application architectures with UML, Communications of the ACM 42 (10) (1999) 63–70.
[16] J.C. Corbett, M.B. Dwyer, J. Hatcliff, Expressing checkable properties of dynamic systems: the Bandera specification language, International Journal on Software Tools for Technology Transfer 4 (1) (2002) 34–56.
[17] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, H. Zheng, Bandera: extracting finite-state models from java source code, in: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 4–11th, 2000.
[18] R. Darimont, A. Lamsweerde, Formal refinement patterns for goal-driven requirements elaboration, in: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes, vol. 21 (6), 1996, pp. 179–190.
[19] D.L. Dill, The Murφ verification system, in: Proceedings of the 8th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, 1102, Springer, 1996, pp. 390–393.
[20] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, Y.S. Ramakrishma, A graphical interval logic for specifying concurrent systems, ACM Transactions on Software Engineering and Methodology 3 (2) (1994) 131–165.
[21] F.M. Donini, M. Mongiello, M. Ruta, R. Totaro, A model-checking-based method for verifying web application design, in: Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005), Electronic Notes in Theoretical Computer Science, vol. 151, 2006, pp. 19–32.
[22] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st International Conference on Software Engineering. Los Angeles, CA, May 16–22th, 1999.
[23] E.A. Emerson, Temporal and modal logic, in: J.V. Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics, 1990, pp. 995–1072.
[24] L. Ercegovac, Digital Systems and Hardware/Firmware Algorithms, John Wiley & Sons, USA, 1985.
[25] R. Eshuis, Symbolic model-checking of UML activity diagrams, ACM Transactions on Software Engineering and Methodology 15 (1) (2006) 1–38.
[26] N. Hallbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, Proceedings of the IEEE 79 (9) (1991) 1305–1320.
[27] D. Harel, Statechars: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[28] K. Havelund, M. Lowry, J. Penix, Formal analysis of a space craft controller using SPIN, IEEE Transactions on Software Engineering 27 (8) (1997) 749–765.
[29] K. Havelund, J.U. Skakkebaek, Applying model-checking in java verification, in: Proceedings of the 6th Workshop of the SPIN Verification System, Lecture Notes in Computer Science, 1680, Springer, 1999, pp. 216–231.
[30] G.J. Holzmann, The SPIN Model Checker – Primer and Reference Manual, Addison-Wesley, 2004.
[31] G.J. Holzmann, M.H. Smith, Software model-checking: extracting verification models from source code, Software Testing, Verification and Reliability 11 (2) (2001) 65–79.
[32] D. Latella, I. Majzik, M. Massink, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model checker, Formal Aspects of Computing 11 (6) (1999) 637–664.
[33] D. Lie, A. Chou, D. Engler, D.L. Dill, A simple method for extracting models from protocol code, in: Proceedings of the 28th Annual International Symposium on Computer architecture (ISCA'01), ACM SIGARCH Computer Architecture News, vol. 29 (2), 2001, pp. 192–203.
[34] K.L. McMillan, Symbolic Model-Checking, Kluwer Academic, 1993.
[35] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.
[36] E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in PROMELA/SPIN, in: Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, October, 21–23th, 1998.
[37] P. Moreno-Ger, C. Blesius, P. Currier, J.L. Sierra, B. Fernández-Manjón, Online learning and clinical procedures: rapid development and effective deployment of game-like interactive simulations, Lecture Notes in Computer Science, Transactions on Edutainment I, vol. 5080, 2008, 288–304.
[38] P. Moreno-Ger, D. Burgos, I. Martínez-Ortiz, J.L. Sierra, B. Fernández-Manjón, Educational game design for on-line education, Computers in Human Behaviour (in press). doi:10.1016/j.chb.2008.03.012.
[39] P. Moreno-Ger, I. Martínez-Ortiz, J.L. Sierra, B. Fernández-Manjón, A content-centric development process model, IEEE Computer 41 (3) (2008) 24–30.
[40] P. Moreno-Ger, P. Sancho, I. Martínez-Ortiz, J.L. Sierra, B. Fernández-Manjón, Adaptive units of learning and educational videogames, Journal of Interactive Media in Education 2007/05, 2007a.
[41] P. Moreno-Ger, J.L. Sierra, I. Martínez-Ortiz, B. Fernández-Manjón, A documental approach to adventure game development, Science of Computer Programming 67 (1) (2007) 3–31.
[42] M. Musavathi, D.Y.W. Park, A. Chou, CMC: a pragmatic approach to model-checking real code, in: 5th Symposium on Operating Systems Design and Implementation, USENIX, Boston, MA, December 9–11th, 2002.
[43] S. Nakajima, Model-checking verification for reliable web service, in: Proceedings of the OOPSLA 2002 Workshop on Object-Oriented Web-Services OOWS 2002, Seattle, Washington, USA, November 5th, 2002.
[44] G. Pu, X. Zaho, S. Wang, Z. Qiu, Towards the semantics and verification of BPEL4WS, in: Proceedings of the International Workshop on Formal Aspects of Component Software. Electronics Notes in Theoretical Computer Science, vol. 160, 2005, pp. 33–52.
[45] P. Schnoebelen, The complexity of temporal logic model-checking. Advances in modal logic, in: Papers From 4th International Workshop on Advances in Modal Logic (AiML'2002), September–October 2002, Toulouse, France, vol. 4, 2003, pp. 393–436.
[46] E.D. Sciascio, F.M. Donini, M. Mongiello, G. Piscitelli, Web applications design and maintenance using symbolic model-checking, in: Proceedings of the 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy, March 26–28th, 2003.
[47] E.D. Sciascio, F.M. Donini, M. Mongiello, R. Totaro, D. Castelluccia, Design verification of web applications using symbolic model-checking, in: 5th International Conference on Web Engineering, Lecture Notes in Computer Science, 3579, Springer, 2005, pp. 69–74.
[48] J.L. Sierra, A. Fernández-Valmayor, B. Fernández-Manjón, A document-oriented paradigm for the construction of content-intensive applications, Computer Journal 49 (5) (2006) 562–584.
[49] J.L. Sierra, A. Fernández-Valmayor, B. Fernández-Manjón, From documents to applications using markup languages, IEEE Software 25 (2) (2008) 68–76.
[50] D.P. Stotts, R. Furuta, C.R. Cabarrus, Hyperdocuments as automata: verification of trace-based Browsing properties by model-checking, ACM Transactions on Information Systems 16 (1) (1998) 1–30.
[51] S. Thibault, R. Marlet, C. Consel, Domain-specific languages: from design to implementation application to video device drivers generation, IEEE Transactions on Software Engineering 25 (3) (1999) 363–377.
[52] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, ACM SIGPLAN Notices 35 (6) (2000) 26–36.
[53] R. van Eck, Building artificially intelligent learning games, in: D. Gibson, C. Aldrich, M. Prensky (Eds.), Games and Simulations in Online Learning: Research and Development Frameworks, Information Science Publishing, Hershey, PA, 2007.
[54] W. Visser, K. Havelund, S. Park, Model-checking programs, Automated Software Engineering 10 (2003) 203–232.
[55] C.D. Walton, Model-checking multi-agent web-services, in: First International Semantic Web-Services Symposium, Palo Alto, CA, March 22–24th, 2004.
[56] X. Zaho, H. Yang, Z. Qiu, Towards the formal model and verification of web service choreography description language, in: Proceedings of the 3rd International Workshop on Web-Services and Formal Methods, Lecture Notes in Computer Science, 4184, Springer, 2006, pp. 273–287.

Please cite this article in press as: P. Moreno-Ger et al., Model-checking for adventure videogames, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.08.003