Extending the Kouretes Statechart Editor for Generic Agent Behavior Development

Georgios Papadimitriou, Nikolaos I. Spanoudakis, Michail G. Lagoudakis

Technical University of Crete, Chania, Crete, Greece
{gpapadimitriou, nispanoudakis, lagoudakis}@isc.tuc.gr

Abstract. The development of high-level behavior for autonomous robots is a time-consuming task even for experts. The Kouretes Statechart Editor (KSE) is a Computer-Aided Software Engineering (CASE) tool, which allows to easily specify a desired robot behavior as a statechart model utilizing a variety of base robot functionalities (vision, localization, locomotion, motion skills, communication) for the Monas robotic software architecture framework. This paper presents an extension to KSE, which allows defining generic agent behaviors using automatic framework-independent code generation, as long as the underlying framework is written in C++. This way a user can program physical (robots) or software agents that can be executed on any platform using any compatible software framework. This paper demonstrates the transparent use of the extended KSE in the SimSpark 3D soccer simulation and the Wumpus world.

Keywords: agents, robotics, model-based development, CASE

1 Introduction

The Kouretes Statechart Editor (KSE) [5,7] is a graphical Computer-Aided Software Engineering (CASE) tool, which allows a software developer to define the behavior of an agent (software or physical) using a graphical model and automatic code generation. The graphical model is a statechart, a platform-independent model, modeling the control of the different agent capabilities. A complete statechart is automatically converted by KSE to source code, which can be compiled, linked with the base capabilities, and executed on the target platform. KSE follows ASEME, a model-driven software engineering methodology from the Agent-Oriented Software Engineering (AOSE) domain [6]. An advantage of KSE is the analysis tool, which enables the user to abstractly and compactly define the desired behavior using liveness formulas. A small set of liveness formulas can lead to a large statechart model (design phase), therefore the user can save a lot of time by seeding the design through the analysis tool.

In our recent work, we used KSE extensively for developing the behavior of the Aldebaran Nao humanoid robots of our team Kouretes [www.kouretes.gr] competing in the RoboCup Standard Platform League (SPL). Besides robotic

behavior development, KSE has also been used for developing the behavior of software agents for the popular Java-based JADE framework [jade.tilab.com].

The motivation behind the work presented in this paper is the need for using a simulator, when modeling a robotic team behavior. Regularly testing new features on the real robots has several shortcomings, i.e. the robots need maintenance after some hours, a number of people are needed in order to set up an experiment with the robotic team in the lab, and the experiments themselves tend to take quite long to setup and demonstrate. Thus, we decided that we needed to use a simulator for modeling and testing team behavior and, when the simulation proved successful, then move to field tests with the real robots. The SimSpark simulation environment, which is also used for the RoboCup 3D soccer simulation league, was the ideal candidate for our goals. However, the SimSpark platform was not compatible with our Monas robotic software architecture framework [4], which we use for deploying behaviors on the Nao robots. Thus, we decided to develop a platform-independent code generation component for the KSE tool. To this end, we added a few platform-specific parameterization features within code generation. By adopting this parametric approach and providing the correct parameters for the underlying platform, we can now use the KSE tool for deploying platform-independent agents on any platform by exporting the generated code directly in the C++ programming language using the specification provided by the parameters.

This paper focuses on describing the extensions of KSE and demonstrates its practicality on two diverse domains: the SimSpark[1] simulation platform for 3D robot soccer and the Wumpus World[2] simulator, a classical testbed for prototyping Artificial Intelligence programs. In the rest of the paper, after examining the background technologies in Section 2, we present the main features of KSE, focusing on new additions and extensions, in Section 3. Subsequently, we present the results of our application and evaluation on the two aforementioned domains in Section 4. Finally, we outline our findings and future work in Section 5.

2 Background

ASEME [6] is an Agent-Oriented Software Engineering Methodology that supports a modular agent design approach and introduces the concepts of intra- and inter- agent control. The former defines the agents behavior by coordinating the different modules that implement its own capabilities, while the latter defines the protocols that govern the coordination of the society of the agents. ASEME follows the Model-Driven Engineering (MDE) paradigm, where software development is driven by instantiating the appropriate models at each phase.

[1] SimSpark is a generic simulator for various multi-agent environments. It supports developing physical simulations for AI and robotics research with an open-source application framework. [simspark.sourceforge.net]

[2] The Wumpus World simulator is a simple C++ framework for simulating the Wumpus World described in Russell and Norvig's classic text book "Artificial Intelligence: A Modern Approach". [www.eecs.wsu.edu/~holder/courses/AI/wumpus]

The transition from one development phase to another is assisted by automatic model transformation leading from requirements to computer programs. The ASEME platform-independent model, which is the output of the design phase, is a statechart [2], and is referred to as the Intra-Agent Control (IAC) model.

KSE adopts the ASEME methodology and assists the developer from the analysis phase to the design and code generation phases. More specifically, KSE supports (a) the automatic generation of the initial abstract statechart model using compact liveness formulas, (b) the graphical editing of the statechart model and the addition of the required transition expressions, and (c) the automatic source code generation for compilation and execution on the robot. KSE has been developed using the Eclipse Modeling Project [www.eclipse.org/modeling] technologies and has been integrated with the Monas software architecture [4], which provides the base functionalities.

Monas modules focus on specific functionalities (vision, motion, etc.) and each one of them is executed independently at any desired frequency completing a series of activities at each execution. Statechart modules are executed using a generic multi-threaded statechart engine, which was built on top of existing open-source projects, provided the required concurrency, and met the real-time requirements of the activities on robots. Code generation in KSE initially targeted specifically the Monas framework.

3 The KSE Generic C++ Generator

The KSE Generic C++ Generator (referred to as *GGenerator* in the rest of the paper) is a tool that extends KSE in order to give the developer the ability to create behaviors for agents, which are automatically transformed to generated C++ code ready to be compiled with the underlying software architecture (assumed to be also in C++) without any additional dependencies. Thus, GGenerator enables the user to create agent behaviors for different platforms and different environments, with the only prerequisite that the underlying software framework accepts C++ code.

We achieve this (a) by enhancing the statechart engine used in KSE with a generic blackboard [3] interface, (b) with a new generic grammar for specifying transition expressions, and, (c) by creating a new source code generator for the extended tool.

The blackboard interface is generated automatically by GGenerator during the source code generation phase and can target any C++ framework, according to its initialization and the parameters given. This interface is used to *bind* our statechart model with any user-specified framework. This way the user is not concerned with model-framework compatibility issues, because the created blackboard interface serves as a middleware between the statechart model and the target environment and handles their communication. This interface, along with our generator, is integrated in KSE.

The initialization of the blackboard interface depends on the *properties files*, which are defined by the user only once during the early stages of the be-

havior development process. These files are named `include_classes.txt` and `instances.txt`. Both files are given in plain text format, they are parsed by the KSE GGenerator tool, and provide the required information for generating and setting up correctly a blackboard interface that targets a specific framework. In these files we add functions, variables, and header files needed for the communication between our statechart model and the target environment. The former file lists the C++ header files we need to include, while the latter lists the functions and the variable instances we need to access.

The creation of the properties files requires some knowledge about the framework we target. Prior to generating our blackboard interface, we have to point out the framework's modules and structures that supply the information needed for correct communication between our statechart model and the environment. In most cases, the information needed for creating an agent behavior includes the updated state of the target environment or an updated set of readings from the agent's sensors, without being limited to these examples, since we can always accept any kind of information the framework provides. The properties files are easily created, once the required information has been identified and located in the target framework. In the next section we give two examples of how to create the properties files for two different platforms.

We also allow the user to register user-defined variables in the same blackboard interface using the graphical editing interface of KSE. These variables can be of any type, can be updated in any node of the statechart, and can be used in transition expressions anywhere in the statechart, along with any framework-provided variables. This ability allows the behavior module itself to maintain its own internal memory.

Finally, to execute the generated behavior on the target framework, we need to register our generic blackboard interface along with our statechart model for execution on the target framework. This is the last stage of our behavior development process and needs to be done only once for a target framework; any number of generated behaviors can be tested afterwards on that framework.

As already mentioned, within the KSE tool, the definition of a statechart can be assisted by the use of liveness formulas as a seed. The *liveness formula* [8] is a process model that connects activities using the Gaia operators. Briefly, `A.B` means that activity `B` is executed after activity `A`, `A*` means that `A` is executed zero or more times, `A+` means that `A` is executed one or more times, `A~` means that `A` is executed indefinitely (it resumes as soon as it finishes), `A|B` means that either `A` or `B` is executed exclusively, `A||B` means that `A` and `B` are executed concurrently, and `[A]` means that `A` is optional. Liveness formulas are automatically converted to a base graphical statechart model.

Every statechart model we create to describe an agent behavior must contain transition expressions, which are responsible for the correct execution of the statechart. These expressions are inserted manually by the user during behavior specification for controlling the statecharts execution. In transition expressions, the user can optionally define *events* and *conditions*, as well as multiple *actions*, if desired, using any framework-specific or user-defined variables. In any case,

```
transExpression = [ event ] [ "[" condition "]" ] [ "/" actions ]
event           = string
condition       = "(" condition ")" | condition operator condition
                  | operand operator condition | operand operator operand
operator        = ">" | "<" | ">=" | "<=" | "==" | "!=" | "&&" | "||"
operand         = constant | variable
variable        = letter string | letter string "." variable
actions         = timeoutAction | action
timeoutAction   = TimeoutAction "." letter+ "." number
action          = variable actionOp variable | action ";" action
actionOp        = "="
constant        = number | string
number          = digit | digit number
string          = character | character string
character       = letter | digit
letter          = "a" | "A" | "b" | "B" | "c" | "C" | ...
digit           = "0" | "1" | "2" | "3" | ...
```

Fig. 1. Transition expression grammar in EBNF format.

these variables must be registered in the blackboard's interface for transition expressions to work properly. Note that the user is provided with the ability to create any kind of variables he/she finds necessary for the transition expressions during the graphical editing of the statechart model.

Since we modified the statechart engine used to execute KSE-created statecharts by adding a generic blackboard interface, we also needed a new C++ code generator different from the existing one, which was customized for the Monas framework. At this point, we also changed the transition expressions grammar to be equally generic. The new statechart engine supports a more abstract transition expression grammar compared to the one used for the Monas architecture [5]. This means that it should be relatively easy for a user to edit them, even without having complete knowledge of the underlying framework he/she works with. The new grammar is independent of the underlying platform, and uses common syntactic rules for expressions. It is shown in EBNF format in Figure 1. We created the new C++ generator for our extended statechart engine using the Eclipse Modeling Framework (EMF[3]) along with Xpand[4]. We fully integrated our GGenerator extension in KSE, so that the generated behavior can be easily tested on the user-specified framework.

4 Applications

We use GGenerator to create behaviors for agents in two target different frameworks, mainly to demonstrate the platform-independence property. The two C++-based frameworks we use for our examples are: (a) Simple Soccer Agent

[3] The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.

[4] Xpand is a language specialized on code generation based on EMF models.

Table 1. Comparison of the two different target platforms.

SimSpark 3D Soccer Simulator	Wumpus World Simulator
Partially Observable	Fully Observable
Stochastic	Deterministic
Dynamic	Static
Physical representation of the agent	No physical representation of the agent
Uncertainty	No uncertainty
Sequential	Episodic
Continuous	Discrete
Multi-agent	Single-agent
Competitive, Cooperative	non-Competitive, non-Cooperative

framework, which interacts with the Simspark 3D Robot Soccer Simulator, and (b) WumpSim framework, which interacts with the Wumpus World Simulator. The two simulation environments have many differences, shown in Table 1.

4.1 Creating a Behavior for the SimSpark 3D Soccer Simulator

In this example, we use GGenerator to create a behavior for an agent that plays soccer. Our simple behavior can be described as follows: *The agent searches for the ball in the soccer field and, if the ball is found, the robot walks towards the ball and takes it in possession. If the game is over, the robot pauses.* Our purpose is to work only at the high level of agent programming (game strategy) and not at the low level (Walk, Kick, and other actions) or with agent-simulator technical aspects (e.g. communication with the simulator). The underlying framework we chose is based on the software release of the RoboCup team Berlin United - Nao Team Humboldt [1]; it is written in C++ and provides basic activities, such as *Walk*, *Turn Left*, *Scan For Ball* or *Stand Up*. The first step is to create the properties files; these files are shown below. In fact, we just need to include the appropriate header files to gain access to those framework-provided variables that allow the user (and the statechart) to gain information about the game and the perception of the ball, but also request the execution of a motion.

`include_classes.txt`	`instances.txt`
`BallPercept.h`	`BallPercept theBallPercept;`
`MotionRequest.h`	`MotionRequest theMotionRequest;`
`SimsparkGameInfo.h`	`SimsparkGameInfo gameState;`

The next step is to use the KSE graphical interface to create our behavior. We start by providing the liveness formulas that describe our agent behavior abstractly:

```
LogicalAgent  =  Init . ( Play | NoPlay )+
Play          =  [ StandUp ] . ( PlayBall | ScanBall )
PlayBall      =  Turn | Walk
```

The first formula indicates that our behavior (LogicalAgent) will execute *Init* (for initialization of the player) and then will choose one or more times between *Play* or *NoPlay* exclusively (depending on the current game state). The second formula suggests that our behavior may execute *StandUp* (if needed) and then will choose between *PlayBall* or *ScanBall* exclusively (depending on whether the ball is visible or not). Finally, the third formula indicates that our agent will choose between *Turn* or *Walk* exclusively (depending on where the ball is seen). As soon as we provide the KSE GGenerator with the liveness formulas, the initial statechart model is generated and the user has to associate it to a source code repository that provides the code for the basic agent activities (*Walk*, *Scan*, etc.). If the framework does not provide some activity, our tool generates the corresponding skeleton code and the user is asked to provide the corresponding C++ code using the built-in editor in KSE. Note that the abstract behavior specification of the liveness formula specifies what activities are included in the desired behavior, but gives no information on when execution switches from one activity to another. It is the execution of transition expressions that makes this switching between activities possible.

After the statechart is created, we use the KSE tool to add the necessary variables we are about to use in the transition expressions. In our case these are: `myInertialSens`, `ballFound`, and `horizontalAngle`. The first variable is of type `class` and holds information about the posture of the robot. The second one is of type `bool` and is true, if the ball is seen by the robot and false otherwise. The last one is of type `double` and holds information about the horizontal angle between the robot and the ball. These are user-defined variables, which are updated from the framework-provided variables inside the statechart, and are used in our transition expressions for controlling the statechart execution. They are added in the blackboard interface using the editing tool provided by KSE.

The next step is to fill in the transition expressions, according to the grammar in Figure 1, using the KSE editor. For example, the transition expression:

```
[ (ballfound == true) && (horizontalAngle < 0.1) ]
```

states the conditions (ball is visible straight-ahead) under which the robot will execute the *Walk* activity within the *PlayBall* state. The final statechart model with all the transition expressions is shown in Figure 2.

The last step of our procedure is to use GGenerator to produce the C++ code for this statechart model and transfer the generated files to the source folder of the framework, so that it is compiled and used in the target framework.

To execute the generated agent behavior we have to instantiate it within the target framework. This is a framework-specific step and depends on the requirements set by the framework itself. In the present example, it amounts to simply adding a couple of C++ lines in the configuration files of the framework to register the generated behavior as a new module within the framework. This is the only handwritten C++ code during our behavior generation procedure and is done only once, since any updates to the statechart simply modify the behavior, which remains registered in the framework. In Figure 3 we show three screenshots from the execution of the LogicalAgent behavior. Initially, the agent

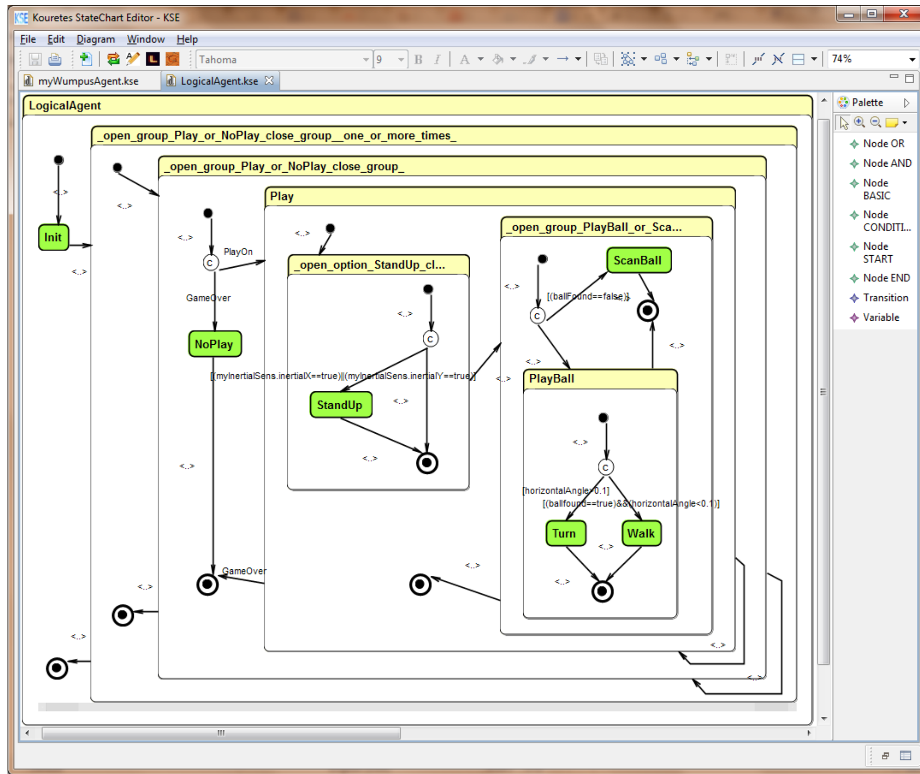

Fig. 2. The complete LogicalAgent statechart within the KSE graphical environment.

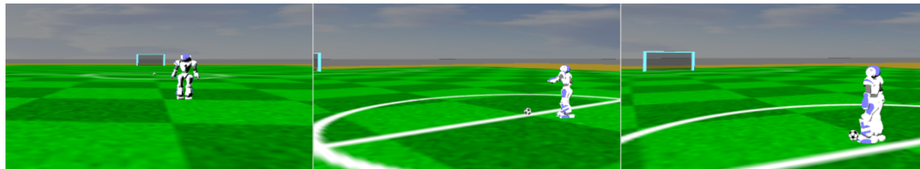

Fig. 3. Three successive screenshots from the execution of the LogicalAgent.

searches for the ball (left), when it finds the ball it walks towards it (middle), and, finally it takes possession of the ball (right).

4.2 Creating a behavior for the Wumpus World Simulator

In this example, we create a behavior for the Wumpus world: *the agent tries to find the gold in the maze and escape alive.* Our goal is not to solve the maze, but to simply demonstrate how easy it is to create a behavior for this environment using GGenerator. To connect our statechart to the Wumpus World platform we first define the properties files, as shown below.

include_classes.txt	instances.txt
WumpusWorld.h Percept.h Action.h	WumpusWorld ww;

Then, we create the liveness formulas that describe the behavior abstractly:

```
Wumpie  =  init . Sense+
Sense   =  percept . ( Play | NoPlay )
Play    =  forward | turnLeft | turnRight | grab | shoot | climb
```

The first formula indicates that our behavior (*Wumpie*) will execute *Init* and then will sense the world and act one or more times (*Sense* capability). The second formula suggests that our behavior will execute *percept* and then choose between *Play* or *NoPlay* exclusively. Finally, the third formula extends the *Play* capability and indicates that our agent will choose one of the moves *forward*, *turnLeft*, *turnRight*, *grab*, *shoot*, *climb* (depending on the information gathered so far). From these liveness formulas, the initial statechart model is generated and the user has to associate it to a source code repository that provides the code for basic agent activities (*forward*, *percept*, *init*, etc.).

In the next step, we use the KSE tool to add the necessary variables we are about to use in the transition expressions of the statechart. In this case these are: `glitter`, `stench`, `breeze`, `turnedRight`, `turnedLeft`, `scream`, `gold`, `arrow`, `posX`, `posY`. These are user-defined variables that are registered in the blackboard during the code generation. Once we add the transition expressions according to the grammar rules, we are ready to generate code for our behavior. Once we register the statechart in the Wumpus framework again by adding a couple of C++ lines in the `Wumpsim` class, we can execute and test our behavior. Figure 4 shows the final statechart for Wumpie.

5 Discussion and Conclusion

In this paper, we presented an extension to KSE that allows to specify statechart-based agent behaviors independently of the underlying software framework, relying on the C++ programming language, as the common code base. The extended KSE tool still supports the C++-based Monas framework for which it was originally customized, but now is open to any other C++-based framework, such as the popular ROS [www.ros.org] framework. Thus, GGenerator increases the added value of the KSE CASE tool. Its main advantages compared to other similar products, noted also in the past [7], include (a) the use of liveness formulas that aid and accelerate the initial statechart generation, and, (b) the option to define the code generation component as an add-on, aiming any target language.

Our future plan, which also motivated this work, is to automatically and massively generate team behaviors for RoboCup using our statechart tools and test them in simulation within an evolutionary framework for discovering suitable

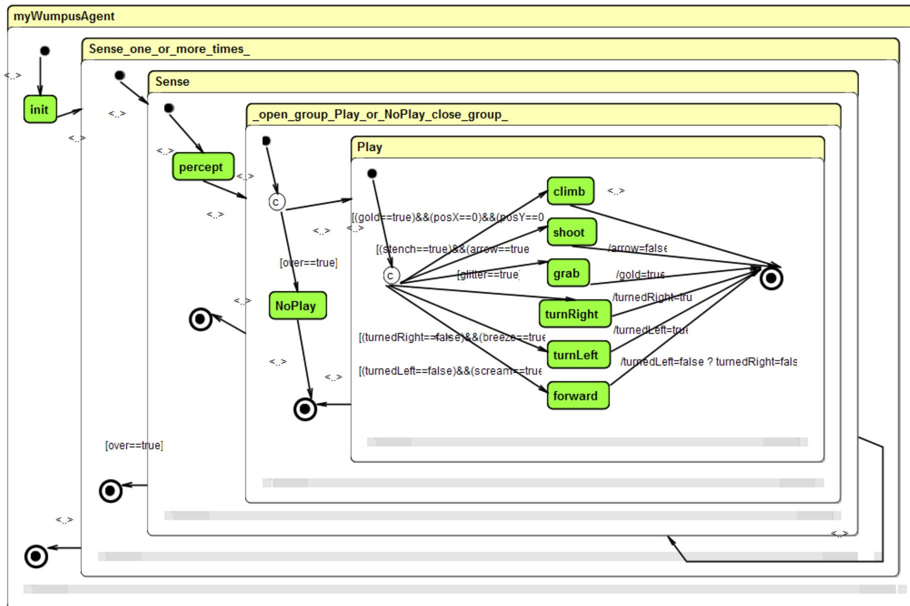

Fig. 4. The complete Wumpie statechart for the Wumpus World.

soccer team behaviors. We also plan to develop additional code generators to support other programming languages, thus expand the range of frameworks compatible with KSE, such as the Python-based Pyro [pyrorobotics.com].

References

1. Hafner, V., Burkhard, H.D., Mellmann, H., Krause, T., Scheunemann, M., Ritter, C.N., Schütte, P.: Berlin United – Nao Team Humboldt 2013. In: RoboCup 2013 Team Description Papers (2013)
2. Harel, D., Naamad, A.: The Statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (1996)
3. Hayes-Roth, B.: A blackboard architecture for control. Artificial Intelligence 26(3), 251–321 (1985)
4. Paraschos, A.: Monas: A Flexible Software Architecture for Robotic Agents. Diploma thesis, Department of ECE, Technical University of Crete, Greece (2010)
5. Paraschos, A., Spanoudakis, N.I., Lagoudakis, M.G.: Model-driven behavior specification for robotic teams. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 171–178 (2012)
6. Spanoudakis, N.I., Moraitis, P.: Using ASEME methodology for model-driven agent systems development. In: Agent-Oriented Software Engineering XI, Lecture Notes in Computer Science, vol. 6788, pp. 106–127. Springer (2011)
7. Topalidou-Kyniazopoulou, A., Spanoudakis, N.I., Lagoudakis, M.G.: A CASE tool for robot behavior development. In: RoboCup 2012: Robot Soccer World Cup XVI, Lecture Notes in Computer Science, vol. 7500, pp. 225–236. Springer (2013)
8. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Autonomous Agents & Multi-Agent Systems 3(3), 285–312 (2000)