A Class Grammar for General Games

Cameron Browne

Queensland University of Technology,
Gardens Point, Brisbane, 4000, Australia
c.browne@qut.edu.au

Abstract. While there exist a variety of *game description languages* (GDLs) for modelling various classes of games, these are aimed at game *playing* rather than the more particular needs of game *design*. This paper describes a new approach to general game modelling that arose from this need. A *class grammar* is automatically generated from a given library of source code, from the constructors and associated parameters found along its class hierarchy, to give a context-free grammar that provides access to the underlying code while hiding its implementation details.

Keywords: Class Grammar, Game Description Language (GDL), General Game Playing (GGP), Game Design, Ludii

1 Introduction

There currently exist a number of software systems for modelling and playing various types of games, including deterministic perfect information games [1], combinatorial games [2], puzzle games [3], strategy games [4], card games [5], video games [6], even complete logical game worlds [7], to name but a few. Each system defines games using a custom *game description language* (GDL), primarily for the *playing* of games. In this paper, we examine such GDLs from the perspective of *designing* games, and propose a new approach that might obviate the need to write a specific GDL for each different type of game.

I introduce the notion of a *class grammar*, which is a formal grammar derived directly from the class hierarchy of the underlying source code. The class grammar is the visible tip of the iceberg of code underneath; it provides a clean, simple interface to the underlying code that offers full functionality, while hiding the implementation details. This approach is described in the context of a new general game system called Ludii, and has potential benefits not only for game design but also for the modelling and playing of games.

The following sections compare some GDLs from a design perspective, describe the syntax, operation and implementation of the class grammar, and give some formatting guidelines for programmers for producing a cleaner grammar.

2 Game Description Languages for Game Design

The tasks of game *playing* and game *design*, although closely linked, have different needs. Game playing focusses primarily on the correctness of the underlying

models and the efficiency of their implementation, while game design involves additional aspects, such as the ease with which game descriptions can be modelled and manipulated by the designer, the expressiveness of the GDL, and how readily the design process can be automated.

Kernighan and Pike list four principles of good software design: *simplicity*, *clarity*, *generality* and *automation* [8]. I propose a similar set of properties that a GDL should possess, in order to be effective for the purpose of game design:

1. *Simplicity:* Game descriptions should be simple to write and modify.
2. *Clarity:* Game descriptions should be readily comprehensible.
3. *Generality:* The GDL should support a wide range of games.
4. *Extensibility:* The GDL should be easy to extend to support new concepts.
5. *Evolvability:* Game descriptions should combine to produce mostly valid (i.e. playable) children with characteristics of their parents.

The ideal GDL, from a design perspective, would allow the designer to quickly prototype new ideas for equipment, mechanisms and complete games, be easily extended as required, and easily automated for the purposes of play-testing, evaluation, optimisation of rules and equipment, and even self-guided game design. Further, the ideal GDL should be hierarchical in nature, with useful game-related concepts called *ludemes* [9] chunked into convenient building blocks, to be easily tried in combination with other rules and equipment in other contexts.

The following subsections briefly examine some individual GDLs, and their suitability for game design, with these points in mind. Note that the focus here is on abstract and board game design, rather than video game design.

2.1 Zillions Rules File

Zillions Rules File (ZRF) is the proprietary game description format for Zillions of Games, a commercial program for modelling and playing Chess-like (and similar) games and puzzles [10]. Appendix A shows Tic-Tac-Toe described in ZRF, by way of example.

ZRF is a scripting language, much like a C macro, which utilises a library of pre-defined keywords for defining equipment, piece movement, and so on. It is highly structured and excellent for modelling Chess-like games, with an in-built AI that can provide a surprisingly responsive and tricky opponent for Chess variants. The syntax is reasonably straightforward and extensible for those familiar with functional programming languages.

However, games become harder to describe, and the AI less effective, the further they diverge from a Chess-like basis, e.g. the AI is effectively random for connection games, and some implementation choices, such as the lack of integer state variables and 2D-only graphics, further limit the generality of the system.[1] Zillions of Games has a strong following among game design hobbyists, but has had very little academic application [11].

[1] The 3D connection game Akron took hundreds of man-hours to model in ZRF.

2.2 Stanford GDL

The Stanford Logic Group's Game Description Language (S-GDL) [12], designed for their associated General Game Player (GGP) [1], is *the* standard GDL for academic research.[2] It is a low-level language that describes games in terms of simple, general instructions that update the game state using first order logic. This approach allows reasonable generality at the expense of clarity, and tends to be somewhat verbose. For example, the S-GDL description of Tic-Tac-Toe, listed in Appendix B, uses 384 tokens, compared to the 89 used by ZRF.

S-GDL is problematic in terms of game design. Game descriptions can be time consuming to write and debug, and difficult to decipher for those unused to first order logic. The equipment and rules are typically interconnected to such an extent that any change to any aspect of the game would require significant rewriting. For example, one of the simplest choices that a game designer might want to experiment with is board size, but changing simply the board size from 3×3 to 4×4 in the Tic-Tac-Toe example would require modifying many lines of code and adding several more.

Extending S-GDL involves defining new versions of the grammar with the appropriate additions and dedicated implementations to support them. For example, GDL-II supports imperfect information games [13], rtGDL supports real-time play [14], and rtGDL-II supports both [14].

In terms of evolvability, games described in S-GDL lack high-level conceptual structure, so it is unlikely that ludemes will pass intact from parents to offspring. In fact, S-GDL descriptions tend to be so finely crafted that any random mutation or crossover is unlikely to yield a playable result. S-GDL, to my knowledge, has not been used except for playing known games, and in academic circles.

2.3 Ludi GDL

Ludi is a software system written for modelling, playing, evaluating and evolving combinatorial games [2]. The associated Ludi Game Description Language (L-GDL) describes games as high-level hierarchical structures of ludemes in a LISP-like format, and was developed with game design squarely in mind.

Complete games can be written and tested within minutes (sometime seconds), and the format proved ideal for evolving games using a *genetic programming* (GP) approach [15]. Game descriptions are easy to comprehend even by lay readers, with the exception of certain pre-defined keywords that require documentation, and are easily modified. For example, changing the board size in the L-GDL Tic-Tac-Toe example shown in Appendix C simply involves changing the board size parameter from `(size 3 3)` to `(size 4 4)`.

Ludi was successful as a proof-of-concept in producing the world's first computer-designed games to be commercially published [16], but only supported a small range of combinatorial games and suffered from over-specialisation, with

[2] The acronym "GDL" in the literature typically refers to this particular language, but it is disambiguated here as "S-GDL" to avoid confusion.

a strong preference for N-in-a-row games. Lack of extensibility meant that any rule or equipment outside the scope of the language would require both the language and the program to be modified, highlighting a drawback of the standard approach of separating the language from the implementation. LUDI has not been publicly released or used outside the study for which it was developed.

2.4 LUDII Class Grammar

LUDII[3] is a complete *general game system* (GGS) [17] that builds on the principles pioneered in LUDI, but extends them to improve the key issues of generality and extensibility. This is achieved primarily through the class grammar that constitutes its GDL. The class grammar is automatically generated from the LUDII source code library, and game descriptions expressed in the grammar are automatically instantiated back into the corresponding library code for compilation, giving a guaranteed 1:1 mapping between the source code and the grammar.

Schaul *et al.* point out that: *any programming language constitutes a game description language, as would a universal Turing machine* [18, p.12]. LUDII achieves this, to some extent, by effectively making the programming language (Java) the game description language; it can theoretically support any game that can be programmed in Java to implement its minimal API (described in Section 4.3). The programmer is free to implement whatever rule, equipment or behaviour they want, however they want, while the user only sees the simplified view of the constructor in the grammar and not the implementation details.

LUDII has been designed with game design in mind. It is currently under development, but the aim is to provide a solid, general framework that supports as wide a range of games as possible, allowing scope for ever increasing functionality as classes in its source code library are subclassed and extended over time.

2.5 Comparison

Figure 1 shows a graphical comparison between these four GDLs, based on the five key design properties. The values shown are subjective estimates only, and are intended to highlight the relevant strengths and weaknesses of each GDL for the purpose of game design.

Simplicity is estimated by the number of tokens required to define games, on average, and the ease with which game descriptions can be modified. Clarity is estimated by the degree to which game descriptions would be self-explanatory to lay readers. Generality is based on the estimated percentage of games listed in the BoardGameGeek (BGG) online database[4] that it would be feasible to describe. Extensibility is estimated as the ease with which the language can be extended to incorporate new rules, behaviours, equipment, etc. Evolvability is estimated as the likelihood with which randomly mutating and crossing-over game descriptions will produce playable children that resemble their parents.

[3] LUDII is named after its predecessor LUDI but improves on it in most respects.
[4] The BGG database now lists over 80,000 games: https://www.boardgamegeek.com

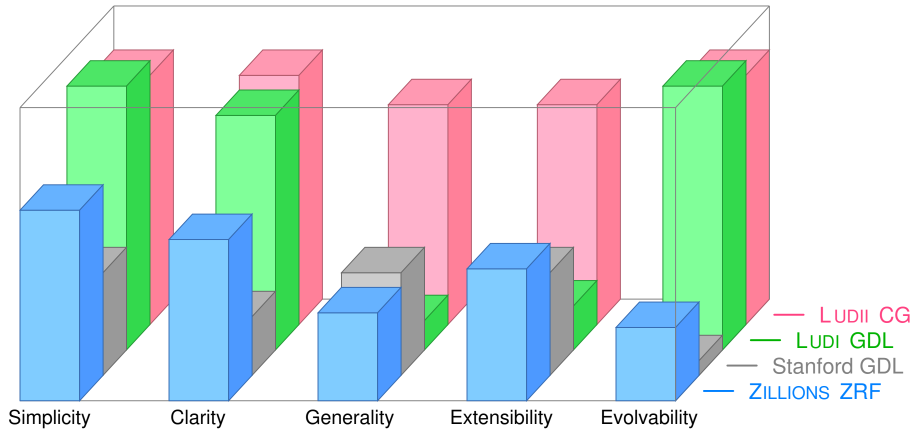

Fig. 1. Comparison of key aspects of GDLs from a design perspective.

ZRF is characterised by reasonable simplicity, clarity and extensibility. S-GDL has reasonable generality and extensibility, but poor evolvability. L-GDL has excellent simplicity, clarity and evolvability, but poor generality and extensibility. The class grammar mechanism devised for Ludii was designed to address the relative shortcomings of existing methods, and produce an approach for computer-assisted and fully automated game design that performs well across all five key design criteria. The following sections describe this approach in detail.

3 Class Grammar

The class grammar is set of *production rules* in which sequences of *symbols* on the RHS are assigned to a *nonterminal symbol* on the LHS, very much like an Extended Backus-Naur Form (EBNF) grammar. It is intrinsically bound to the underlying code library, but is a *context-free grammar* that is self-contained and can be used without knowledge of the underlying code.

3.1 Context

The class grammar involves two main automated parsing steps:

1. *Forwards:* From source code to grammar.
2. *Backwards:* From grammar expressions back to specified source code.

The backwards step is similar in principle to existing approaches for using grammars to generate code. These include C++ code generators [19, 20], Java code generators [21], *parser generators* such as ANTLR [23], and Translational BNF (TBNF) [22], in which code actions are embedded in the grammar.

The difference is that these approaches all involve a grammar maintained separately by the user or system, whereas the class grammar's forwards step makes

it self-generating. The resulting grammar could be described as a *domain-specific language* (DSL) [23, 24], although the potential generality and extensibility of the approach would make this something of a misnomer.

3.2 Syntax

The basic syntax of the class grammar is as follows:

```
<class> ::= { (class [{<arg>}]) | <subClass> | terminal }
```

where:

`<class>`	denotes a LHS symbol that maps to a class in the code library.
`(class [{<arg>}])`	denotes a `class` constructor and its arguments.
`<subClass>`	denotes a subclass derived from `class`.
`terminal`	denotes a terminal symbol (fundamental data type or `enum`).
`[...]`	denotes an optional item.
`{...}`	denotes a collection of one or more items.
`|`	denotes a choice between options in the RHS sequence.

Class names typically start with an uppercase character, but are converted to lowercase in the grammar for readability, convenience, and in keeping with the traditional form of EBNF style grammars. Appendix E shows a sample of the grammar generated from the Ludii code library.

3.3 Forward Mechanism (Generation)

The forward step of converting source code to grammar involves recursively parsing the code library from a specified root class (`Game` in this case) downwards, storing a new symbol for each new class encountered. A chain of dependency is then created from the root class, linking the arguments of each visited constructor by data type, until terminal symbols are reached. Fundamental data types and `enum`s constitute terminals, while all other user-defined classes constitute *non-terminals*.

The grammar is then generated with each class name forming the LHS symbol of a production rule, whose RHS is a sequence of constructors that instantiate that class (or subclasses derived from it) and their parameters. For example, the following abstract base class with no constructors:

```
public abstract class Start { ... }
```

and its two derived subclasses:

```
public class Place extends Start
{
```

```
    public Place(final String what, final int where)
}

public class Store extends Start
{
    public Store(final int who, final String what, final int count)
}
```

generate the following production rules:

```
<start> ::= <place> | <store>
<place> ::= (place (what String) (where int))
<store> ::= (store (who int) (what String) (count int))
```

The result is a summary of the class hierarchy, based on constructors and parameters, that offers full functionality while hiding the implementation details.

3.4 Backward Mechanism (Instantiation)

Each individual game is described as a *symbolic expression* (s-expression) compatible with the grammar. For example, Appendix D shows Tic-Tac-Toe described in the LUDII class grammar.

Game descriptions are parsed in a *top-down* manner [24, p. 225], with each `(class ...)` instance matched with its generating constructor, and parameters recursively instantiated as required. The calling app can then use the `JavaCompiler` and associated classes from the `javax.tools` library to compile the assembled code and produce an executable version of the game.

To maximise extensibility, the game author can append their own custom Java code to the end of the game description file, and call its constructors from within the description as per any other constructor defined in the grammar. This makes the approach quite extensible without the need to modify or recompile the underlying code library, with the caveat that the author of such appended code would need to be familiar with Java and would probably have to develop it outside a Java development environment.

4 Implementation

This section describes some relevant implementation details.

4.1 Programming Language

Java was chosen for the class grammar code base due to its ease of use, portability (it runs on any device and operating system with the appropriate Java virtual machine) and speed (it performs as well as equivalent C++ code, to within a few percent, using current compilers). Further, Java's `reflection` library is ideal for

extracting relevant class information from the code base, and its `javax.tools` compilation tools are ideal for the run-time compilation of reconstructed classes.

4.2 Algorithm

The algorithm for generating the class grammar is summarised as follows:

```
public void generate(final String rootPath)
{
    setPredefinedSymbols();
    findSymbols(rootPath);
    scopeSymbols();
    expandRHSs();
    removeSuperfluousSubclasses();
    collapseSimilarConstructors();
    prioritiseOrder();
    trimRules();
}
```

Firstly, `SetPredefinedSymbols()` creates predefined symbols for fundamental Java data types such as `int`, `float`, `double`, `boolean`, `String`, `Object`, and so on. `findSymbols()` then recursively finds additional symbols corresponding to user-defined classes and `enum`s from the specified root. These are then *minimally scoped* to disambiguate symbols with identical names, by prepending superclass names as required. For example, multiple occurrences of class `or` might be scoped to `start.or`, `move.or`, `end.or`, etc.

`expandRHSs()` then creates a production rule for each symbol, with the symbol name as LHS, and expands the RHS to include the constructor(s) for this class and derived subclasses. `removeSuperfluousSubclasses()` removes duplicate occurrences of subclasses in the RHS except for the deepest.

`collapseSimilarConstructors()` combines similar constructor descriptions on the RHS where possible, identifying implicit optional parameters (discussed shortly). `prioritiseOrder()` prioritises package order in depth-first order, and rule order within each package so that base classes come first. `trimRules()` removes unused and empty rules, which might occur in partially implemented code under development.

4.3 Interface

The root `Game` class implements the following minimal API:

```
public void create(final int viewSize);
public void start(final Episode episode);
public List<Turn> actions(final Episode episode);
public Status apply(final Episode episode, final Turn turn);
```

```
public Status playout(final Episode episode);
```

Every game defined in the grammar, when compiled, *must* implement this basic functionality for play. The user therefore defines games in the grammar but executes them through the API. This decouples the grammar from its implementation, from the user's perspective, and makes it context-free. The `playout()` function is for performing optimised playouts, avoiding complete legal move enumerations, for AI implementations such as Monte Carlo tree search (MCTS) [25].

Details regarding the internal game state representation are beyond the scope of this paper, which focusses on the class grammar itself. Suffice it to say that this representation is designed to be general and efficient, but can be subclassed and overridden for the optimisation of individual cases as desired.

4.4 Formatting Guidelines

While the class grammar is conceptually decoupled from its generating code, the programmer can make the grammar cleaner and clearer by following some basic formatting guidelines.

Named Parameters Constructor parameters that are simple (terminal) data types are explicitly labelled in the grammar by their parameter name. This makes the grammar self-documenting to some extent, easier to interpret and reduces ambiguity. For example, this:

```
<what> ::= (what (who int) (where int))
```

is more meaningful to the reader than:

```
<what> ::= (what int int)
```

It is sometimes desirable to *anonymise* named parameters, where this simplifies the grammar and does not create ambiguity; for example, the two parameters in `(add int int)` do not need naming. Such parameters can be explicitly denoted using the custom annotation `@Anon` to override the default behaviour.

Conversely, parameters representing complex (non-terminal) data types are not named in the grammar by default, as the data type itself usually gives enough information to infer the parameter's purpose. However, this behaviour can also be overridden to explicitly name such parameters using the custom annotation `@Name`. Note that parameter naming requires the use of Java version 8 for the relevant `reflection` call, but warrants the move to this version.

Optional Parameters Constructor arguments can be *explicitly* specified as `[optional]` items in the grammar using the custom annotation `@Opt`. For example, the following code:

```
public Board(final Basis basis, @Opt final Modify[] modify)
```

will generate the following rule with an optional parameter:

```
<board> ::= (board <basis> [{<modify>}])
```

Parameters can also be *implicitly* made `[optional]` by providing multiple constructors for a class, such that parameters that occur in one constructor but not another are interpreted as optional. For example, the following pair of constructors would produce the same rule shown above:

```
public Board(final Basis basis)
public Board(final Basis basis, final Modify[] modify)
```

The explicit approach is recommended as it is simpler and less error prone. The implicit approach, although more conceptually elegant, requires care to avoid ambiguous cases, and complicates the initialisation of default values.

Default Values It is useful to set default values for member variables of all classes described in the grammar, in case their corresponding constructor parameters are made optional. However, this is complicated by the fact that we also want to declare them as `final` and make the instantiated objects *immutable* if possible, as per good object oriented design practice [26, pp.73–80].

Java only allows `final` member variables to be initialised once in the class's execution flow. This is handled in the class grammar by passing parameter values up the `super(...)` constructor chain as appropriate, and instantiating missing values due to optional parameters with their default values in the appropriate constructors. Care must be taken to instantiate the same default values across all constructors for each class, for consistency.

Library Structure The LUDII code library is organised to reflect the underlying class structure, with each Java *package* containing the base class of the same name and immediate subclasses that will create items in the RHS sequence for the corresponding grammar rule. This makes it easier to navigate and maintain the code library using the class grammar as a reference.

Abstract Classes The programmer can influence the format of the generated grammar through judicious use of `abstract` classes. Constructors for `abstract` classes are not shown in the grammar as they cannot be instantiated by the user.

Inner Classes The programmer is free to use inner classes, but these are private to their defining class and so will not appear in the grammar (except for `enums`).

Collections Constructor parameters denoting arrays and Java collections, such as `Lists`, are all represented in the grammar as $\{\ldots\}$, for the sake of brevity. The appropriate data type is reconstructed and populated with the specified items in the backwards (instantiation) step, during code compilation.

4.5 Version Control

As the Ludii code library is a work in progress, and could continue to expand for years to come, *regression testing* is important to guarantee that future additions to the library do not unduly affect existing code.

This will be achieved by maintaining a database of N deterministic playouts for each game described in the grammar, seeding the RNG with a hash code based on the game's (unique) name, and storing the moves thus generated. Any change to the library that makes any known game diverge from its stored playout record will be flagged for investigation.

5 Conclusion

While the class grammar described in this paper is based on the Ludii general game system's source code library, the basic approach – of automatically generating a context-free grammar from a class hierarchy's constructors, then instantiating expressions in that grammar by compiling the appropriately parameterised constructor calls – has general application to any domain for which such a class hierarchy can be defined.

Benefits of the approach for computer-assisted and fully automated game design include: 1) the generality implicit in effectively using the programming language (Java) as the game description language; 2) the extensibility afforded by the ease with which code can be added to the source code library and automatically incorporated into the grammar; and 3) the evolvability of games described in this high-level hierarchical manner. The class grammar is the ideal GDL for Ludii as it develops and expands over the upcoming years.

Acknowledgements. This work was funded by a QUT Vice-Chancellor's Research Fellowship as part of the project *Games Without Frontiers*. Thanks to Stephen Tavener for nudging me towards Java, which proved ideal for this task.

References

1. Genesereth, M., Love, N. and Pell, B.: General Game Playing: Overview of the AAAI Competition. AI Magazine, 26, 62–72 (2005)
2. Browne, C.: Automatic Generation and Evaluation of Recombination Games. Ph.D. Thesis, Faculty of Information Technology, QUT, Brisbane (2008)
3. Shaker, M., Sarhan, M. H., Naameh, O. A., Shaker, N. and Togelius, J.: Generation and Analysis of Physics-Based Puzzle Games. In IEEE Conference on Computational Intelligence in Games (CIG'13), Niagara Falls, 1–8 (2013)

4. Mahlmann, T., Togelius, J. and Yannakakis, G.N.: Modelling and Evaluation of Complex Scenarios with the Strategy Game Description Language. In IEEE Conference on Computational Intelligence in Games (CIG'11), Seoul, 174–181 (2011)
5. Font, J., Mahlmann, T., Manrique, D. and Togelius, J.: A Card Game Description Language. In European Conference on Applications of Evolutionary Computation, Vienna, 254–263 (2013)
6. Schaul, T.: An Extensible Description Language for Video Games. IEEE Transactions on Computational Intelligence and AI in Games, 6 (4) 325–331 (2014)
7. Kulick, J.: World Description Language - A Logical Language for Agent-Based Systems and Games. Bachelors thesis, Freie Universität Berlin, Fachbereich fü Mathematik und Informatik (2009)
8. Kernighan, B. W. and Pike, R.: The Practice of Programming. Addison-Wesley, Boston (1999)
9. Borvo, A.: Anatomie D'un Jeu de Cartes: L'Aluette ou le Jeu de Vache. Librarie Nantaise Yves Vachon, Nantes (1977)
10. Mallett, J. and Lefler, M.: Zillions of Games: Unlimited Board Games & Puzzles. Online: http://www.zillions-of-games.com (1998)
11. Hom, V. and Marks, J.: Automatic Design of Balanced Board Games. In Artificial Intelligence and Interactive Digital Entertainment Conference (AAIDE'07), Stanford, 25–30 (2007)
12. Love, N., Hinrichs, T. and Genesereth, M.: General Game Playing: Game Description Language Specification. Report LG-2006-01, Stanford Logic Group (2006)
13. Thielscher, M.: A General Game Description Language for Incomplete Information Games. In AAAI Conference on Artificial Intelligence, Atlanta, 994–999 (2010)
14. Kowalski, J. and Kisielewicz, A.: Game Description Language for Real-time Games. In General Intelligence in Game-Playing Agents (GIGA'15), B. Aires, 23–30 (2015)
15. Koza, J. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Massachussetts (1992)
16. Browne, C.: Evolutionary Game Design. Springer, Berlin (2011)
17. Browne, C., Togelius, J. and Sturtevant, N.: Guest Editorial: General Games. IEEE Transactions on Computational Intelligence and AI in Games, 6 (4) 1–3 (2014)
18. Schaul, T., Togelius, J. and Schmidhuber, J.: Measuring Intelligence through Games. Technical Report, arXiv:1109.1314v1 (2011)
19. Hall, P. W.: Parsing with C++ Constructors. ACM SIGPLAN Notices, 28 (4) 67–69 (1993)
20. Conway, D.: Parsing with C++ Classes. ACM SIGPLAN Notices, 29 (1) 46–52 (1994)
21. Pohjalainen, P.: Object-Oriented Language Processing. Modular Programming Languages, Lightfoot D. and Szyperski, C. (eds.), LNCS 4228, Springer, Berlin 104–115 (2006)
22. Mann, P.: A Translational BNF Grammar Notation (TBNF). ACM SIGPLAN Notices, 41 (4) 16–23 (2006)
23. Fowler, M. and Parsons, R.: Domain-Specific Languages. Addison-Wesley, Boston (2011)
24. Ghosh, D.: DLSs in Action. Manning, Stamford (2011)
25. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S.: A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games, 4:1 1–43 (2012)
26. Bloch, J.: Effective Java. Second Edition. Addison-Wesley, Boston (2008)

Appendix A: Tic-Tac-Toe in ZRF

The following code describes Tic-Tac-Toe in the Zillions of Games Zillions Rules File (ZRF) format [10] (88 tokens):

```
(define add-to-empty ((verify empty?) add))
(game
    (title "Tic-Tac-Toe")
    (players X O)
    (turn-order X O)
    (board
        (grid
            (start-rectangle 16 16 112 112)
            (dimensions
                ("top-/middle-/bottom-" (0 112))
                ("left/middle/right" (112 0))
            )
            (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
        )
    )
    (piece (name man) (drops (add-to-empty)))
    (board-setup (X (man off 5)) (O (man off 5)))
    (draw-condition (X O) stalemated)
    (win-condition (X O)
        (or (relative-config man n man n man)
            (relative-config man e man e man)
            (relative-config man ne man ne man)
            (relative-config man nw man nw man)
        )
    )
)
```

Appendix B: Tic-Tac-Toe in the Stanford GDL

The following code describes Tic Tac Toe in the Stanford GDL [12] (384 tokens):

```
(role white)
(role black)
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
```

```
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control white))
(<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b))
    (true (control ?w)))
(<= (legal white noop) (true (control black)))
(<= (legal black noop) (true (control white)))
(<= (next (cell ?m ?n x)) (does white (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= (next (cell ?m ?n o)) (does black (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w))
    (distinct ?w b))
(<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k))
    (true (cell ?m ?n b)) (or (distinct ?m ?j) (distinct ?n ?k)))
(<= (next (control white)) (true (control black)))
(<= (next (control black)) (true (control white)))
(<= (row ?m ?x) (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x) (true (cell 1 ?n ?x))
    (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x) (true (cell 1 1 ?x))
    (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x) (true (cell 1 3 ?x))
    (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?m ?x))
(<= (line ?x) (diagonal ?x))
(<= open (true (cell ?m ?n b))) (<= (goal white 100) (line x))
(<= (goal white 50) (not open) (not (line x)) (not (line o)))
(<= (goal white 0) open (not (line x)))
(<= (goal black 100) (line o))
(<= (goal black 50) (not open) (not (line x)) (not (line o)))
(<= (goal black 0) open (not (line o)))
(<= terminal (line x))
(<= terminal (line o))
(<= terminal (not open))
```

Appendix C: Tic-Tac-Toe in the Ludi GDL

The following code describes Tic-Tac-Toe in the Ludi GDL [2] (29 tokens):

```
(game Tic-Tac-Toe
    (players White Black)
    (board (tiling square i-nbors) (shape square) (size 3 3))
```

```
    (pieces (Piece All (moves
        (move (pre (empty to)) (action (push))))))
    (end (All win (in-a-row 3)))
)
```

Appendix D: Tic-Tac-Toe in Ludii Class Grammar

The following shows Tic-Tac-Toe in the Ludii class grammar (47 tokens):

```
(game "Tic-Tac-Toe"
    (control (player "P1") (player P2") Discrete)
    {
        (board Board (square 3))
        (disc Disc1 (owner P1))
        (disc Disc2 (owner P2))
    }
    (rules
        {
            (store P1 Disc1 (count 5))
            (store P2 Disc2 (count 4))
        }
        (play
            (move
                (from (generate Store Mover))
                (to (generate Board empty))
            )
        )
        (end
            (line (length 3) (dirn Any) (owner Mover))
            (result Mover Win)
        )
    )
)
```

The description `(game "Tic-Tac-Toe")` has the same effect in 2 tokens, due to default parameter values. A full board without a winning line defaults to a `Draw`, after both players are forced to pass in succession.

Appendix E: Sample of the Class Grammar

The following listing shows an incomplete subset of the class grammar generated from the Ludii code library. Rules are grouped by package.

```
<game>       ::= (game (name String) [{<metadata>}]
                       [<control>] [{<equipment>}] [<rules>]
                 )

<metadata>   ::= (String String)

<control>    ::= (control [{<player>}] [<timeType>])
<timeType>   ::= Discrete | Real

<player>     ::= (player [(index int)] (name String))

<equipment>  ::= <component> | <container>

<container>  ::= <board> | <store>

<board>      ::= (board (label String) <basis> [{<modify>}])
<store>      ::= (store (label String) (owner int))

<basis>      ::= <hexHex> | <rect> | <square>
<hexHex>     ::= (hexHex (dim int))
<rect>       ::= (rect (rows int) (cols int))
<square>     ::= (square (dim int))

<component>  ::= <ball> | <disc>

<ball>       ::= (ball (label String) (colour int))
<disc>       ::= (disc (label String) (colour int))

<rules>      ::= (rules [{<start>}] [<play>] [<end>])

<start>      ::= <place> | <store>
<place>      ::= (place <equipment> <site>)
<store>      ::= (store <equipment> <roleType> (count int))

<play>       ::= (play <move.logic>)

<end>        ::= (end <bool> <result>)
<result>     ::= (result <bool> <roleType> <resultType>)

<roleType>   ::= None | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |
                 Any | All | Mover | NonMover | Opposite |
                 Next | Prev | Odd | Even
<resultType> ::= Win | Lose | Draw | Tie | Abort
```