A Visual Language for Representing and Explaining Strategies in Game Theory*

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Eric Walkingshaw
Oregon State University
walkiner@eecs.oregonstate.edu

Abstract

We present a visual language for strategies in game theory, which has potential applications in economics, social sciences, and in general science education. This language facilitates explanations of strategies by visually representing the interaction of players' strategies with game execution. We have utilized the cognitive dimensions framework in the design phase and recognized the need for a new cognitive dimension of "traceability" that considers how well a language can represent the execution of a program. We consider how traceability interacts with other cognitive dimensions and demonstrate its use in analyzing existing languages. We conclude that the design of a visual representation for execution traces should be an integral part of the design of visual languages because understanding a program is often tightly coupled to its execution.

1 Introduction

Game theory has had a broad impact on the scientific world, with applications in economics, computer science, and many social sciences. From this breadth arises a need for representing and explaining concepts in game theory to an equally broad audience.

But in addition to scientists and other experts that apply game theory in one way or another as part of their job, the general public needs to be educated as well. In general, it is becoming more and more important to educate the general public about scientific findings and to increase scientific literacy. Broad public support for science and research, which is needed to justify the research (and its funding), can be achieved only by establishing a basic understanding of the scientific principles. How the lack of scientific literacy can negatively impact scientific research and education can be observed today in the United States in sometimes bizarre discussions about issues like the teaching of evolution or funding for stem cell research.

Why does game theory deserve public attention? In addition to offering a mathematically grounded way to strategize in particular situations, game theory has had a huge impact providing descriptive explanations of naturally occurring phenomena. One of the best examples is insight into why humans and animals cooperate and how such cooperation has evolved, described in Robert Axelrod's seminal book *The Evolution of Cooperation* [2]. The book focuses on one of the most important and well-known games in game theory: the Prisoner's Dilemma. In addition to cooperation, the iterated form of this simple game has been used to describe situations from the use of performance enhancing drugs in sports [12] to the nuclear arms race of the Cold War era [16].

Therefore, a notation that can help experts and lay people alike with describing and understanding games and strategies should not only be regarded as the basis for potential programming and simulation tools, but also as a language/medium to communicate and explain scientific results to a broad audience.

Although several visual notations already exist for concisely and clearly defining different types of games, the existing representations of strategies for iterated games are pseudo-code, inflexible and unscalable tables, and unstructured text, which leaves open the question of a visually appealing notation that can be used to explain game strategies and how they work.

In game theory, a game is a formal representation of a situation in which players interact and attempt to maximize their own return. Players interact by making discrete moves and a player's return is quantified by a payoff value. Players are only concerned with maximizing their own payoff. An iterated game simply has the same set of players play the same game repeatedly with the payoffs of each iteration accumulating.

The Prisoner's Dilemma is a game in which each player must choose to either "cooperate" or "defect". Defection yields the higher payoff regardless of the other player's choice, but what makes the game interesting is that if both players cooperate they will do better than if they both defect. In 1980 Axelrod held an Iterated Prisoner's Dilemma tournament. Game theorists from around the world submitted strategies to the competition. The winning strategy was "Tit for Tat", submitted by Anatol Rapaport. Tit for Tat was much simpler than many of its opponents, it cooperates on the first move and thereafter plays the last move played by

*This work is partially supported by the National Science Foundation under the grant CCF-0741584.

its opponent. Thus, if an opponent always cooperates, Tit for Tat will always cooperate, but if an opponent defects, Tit for Tat will retaliate by defecting on the next turn. The surprising success of such a simple strategy turned out to be a breakthrough in the study of cooperative behavior [2].

A formal definition of the Prisoner's Dilemma is shown on the right in a notation known as "normal form". While normal form can technically be used to represent any game in game theory [16], its use is usually restricted to games with two players, each of whom must select one move from a finite list of moves without knowledge of the other player's move. The payoffs for each player are given by a pair of numbers in the cell indexed by each player's move. In this game, for example, if "Me" chooses to cooperate (indicated by the "C") and "Opponent" chooses to defect ("D"), the relevant cell contains the pair $0,3$ indicating a payoff of 0 for "Me" and a payoff of 3 for "Opponent".

		Opponent C	Opponent D
Me	C	2,2	0,3
Me	D	3,0	1,1

In this paper we introduce an *integrated* notation for defining game strategies and game execution traces that is based on this well-known representation. The emphasis is not just on the visual language for describing strategies, but rather on the design of a notation that allows the combined representation of strategies *and* game execution traces. The ultimate design goal is to obtain a representation that supports explanations of how strategies work. We have utilized the cognitive dimensions framework in the design phase of our notation. Having realized that the motivation for the integrated notation and the explanatory component was only partially covered by existing cognitive dimensions, we have identified a new cognitive dimension of "traceability", which measures the ability of a notation to represent execution and the relationship between a program and its execution.

The remainder of this paper is structured as follows. In Section 2 we introduce the notation for strategies, traces, and their composition. In Section 3 we describe the design process based on cognitive dimensions. This section also shows how some crucial design decisions could not be supported by existing cognitive dimensions and thus provides the motivation for the new cognitive dimension of traceability that is defined and illustrated with further examples in Section 4. We discuss related work in Section 5 and present some conclusions in Section 6.

2 A Notation for Game Theory

The notation we have designed is composed of three distinct but related components. The first is the notation for defining strategies, defined in Section 2.1; the second is the representation of game traces, defined in Section 2.2; and the third is a view which demonstrates how any given game instance within a game trace occurred, given the players' strategies and the game trace, defined in Section 2.3.

2.1 Strategy Notation

Our notation is based on the well-known normal-form game representation that we have briefly shown in the previous section. This representation views a game as a matrix indexed by moves that contains the payoff result for each possible combination of moves in its cells. More formally, if the set M_i represents the set of moves available to player i, the *domain* of an n-player game is given by $D = M_1 \times \ldots \times M_n$, and a game is a mapping $G : D \rightarrow \mathbb{R}^n$ so that the number r_j in the tuple $(r_1, \ldots, r_n) = G(m_1, \ldots, m_n)$ represents the payoff for player j in the game that results when player i plays move m_i. Since we consider only 2-player games, our domain will always be $D = M_1 \times M_2$, and games are given by mappings $G : D \rightarrow \mathbb{R}^2$.

A strategy is defined by a list of rules (given below the dotted line) and a possibly empty horizontal list of initial *move patterns* (given above the dotted line). A rule $L \rightarrow R$ is given by a *matching pattern* (L) and a move pattern (R). Figure 1 demonstrates our visual notation for strategies with a definition of Tit for Tat.

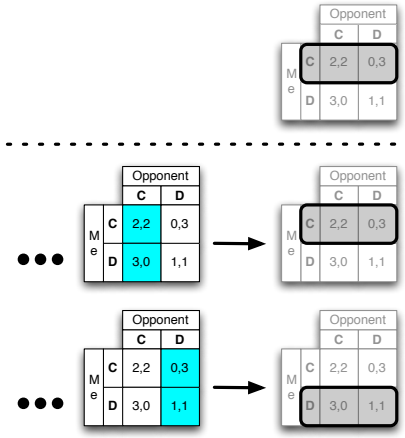


Figure 1. Definition of Tit for Tat

A move pattern is a special case of a *simple pattern*, which is obtained from the normal-form matrix notation by marking a subset of cells in the matrix. Formally, a simple pattern can be represented by a binary partition of the matrix domain D into marked and unmarked cells, and this partition is uniquely determined by the set of marked cells. A move pattern for player 1 is a simple pattern such that the marked set equals $\{m\} \times M_2$ where $m \in M_1$. So, for example, the initial move of Tit for Tat is defined by $C \times \{C,D\} = \{(C,C),(C,D)\}$.

If present, a list of k initial move patterns defines the first k moves of the player that uses this strategy. In the case of Tit for Tat, the first move is always C. Note that the moves

(and patterns) in the strategy are defined from the perspective of the player "Me". A strategy for the player "Opponent" can be obtained from a "Me" strategy (and vice versa) by flipping rows and column markings in all matching and move patterns, as defined by the function *flip*.

$$\begin{aligned} flip(\{m\} \times M) &= M \times \{m\} \\ flip(M \times \{m\}) &= \{m\} \times M \\ flip(L \to R) &= flip(L) \to flip(R) \end{aligned}$$

A *game play* is given by a pair of moves, one for each player. That is, $p \in D = M_1 \times M_2$, and a *game trace* is defined as a sequence of game plays, $p_1 \ldots p_n$.

The meaning of a set of rules is defined relative to an existing game trace, and the meaning of a single rule is given by the move pattern on the right. The first rule whose matching pattern matches the current game trace is selected, and its move pattern defines the next move.

Matching patterns, as found on the left side of rules, take one of the following five forms where P_i is the set of cells representing a simple pattern. The conditions under which each pattern matches a game trace $p_1 \ldots p_n$ are shown on the right.

Recent	$\bullet\bullet\bullet P_{n-k} \ldots P_n$	if $\forall i \in \{n-k, \ldots, n\}, p_i \in P_i$
Initial	$P_1 \ldots P_k \bullet\bullet\bullet$	if $\forall i \in \{1, \ldots, k\}, p_i \in P_i$
Always	$P \bullet\bullet\bullet P$	if $\forall i \in \{1, \ldots, n\}, p_i \in P$
Sometimes	$\bullet\bullet\bullet P \bullet\bullet\bullet$	if $\exists i \in \{1, \ldots, n\}, p_i \in P$
Default	(nothing)	always matches

Both rules in the definition of Tit for Tat utilize the *recent* matching pattern—they both consider only the most recent entry in the trace. The definition of Grim Trigger given in Figure 2 is an example of a strategy that utilizes both the *sometimes* pattern and the *default* form.

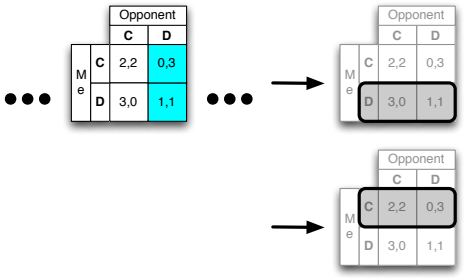
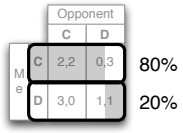


Figure 2. Definition of Grim Trigger

Grim Trigger is like Tit for Tat in that it will continue to cooperate as long as the opponent cooperates, but unlike Tit for Tat, Grim Trigger never forgives. Once an opponent defects, Grim Trigger will defect forever thereafter. Colloquially, the visual definition would read "if the opponent has defected any time in the past, then defect; otherwise, cooperate". Also note that the inclusion of a default matching pattern obviates the need for initial moves.

Strategies also often make use of "mixed strategies", which are strategies in which a move is selected randomly based on some probability distribution. Thus a *mixed pattern* can be used in place of a move pattern anywhere a move pattern would otherwise appear.

A mixed pattern is represented by multiple selected moves, where the shading of each selection is altered to correspond to its likelihood of selection. Annotations describe the distribution more exactly. The example shows a mixed pattern that will cooperate with 80% probability and defect with 20% probability.



Two more strategies to be used in the following discussion are Tit for Two Tats, which is shown in Figure 3 and defects only after two consecutive defects of the opponent, and Opponent Alternator, shown in Figure 4.

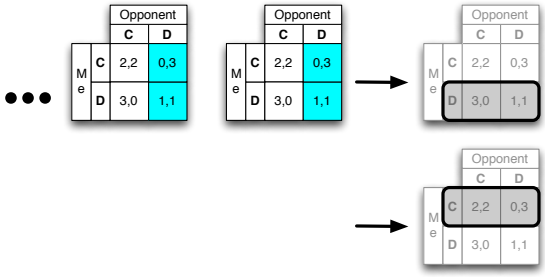


Figure 3. Tit for Two Tats Strategy

The yellow color used in the matching pattern of the Opponent Alternator not only helps to distinguish an "Opponent" from a "Me" strategy, it also supports the integrated display of rules and traces as shown in Section 2.3. Note that since the Opponent Alternator wants to do the opposite of *its own* last move, it is the column that must be colored.

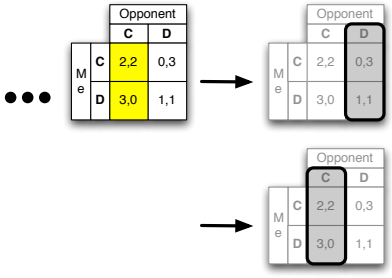


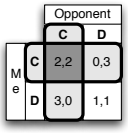
Figure 4. "Opponent" Alternator

2.2 Representing Traces

As defined earlier, a game play is given by a pair of moves. Visually, a game play is represented by the intersection of two move patterns, as shown on the right. The resulting game outcome is not only indicated by the intersecting frames of the move patterns, but also by the darker

cell shading that result from the overlay of the pattern coloring.

In this example, both players chose to cooperate, resulting in a payoff of 2 for each player.



For representing game traces, we introduce a more abstracted view of a game play. The normal form representation of the game is stripped of headers and labels and represented as a simple grid of potential outcomes. The outcome that was achieved in a particular game play is indicated by filling in the corresponding box in the grid.

Figure 5 shows a trace of iterated Prisoner's Dilemma played Tit for Two Tats against Opponent Alternator. Here we can clearly see the pattern of outcomes that results when these two strategies face each other.

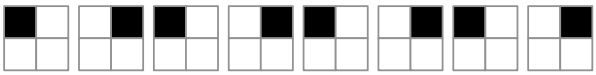


Figure 5. Example Game Trace

A second view of a game trace utilizes colors on a green-red gradient for outcome shading to represent the quality of the payoff from the perspective of either one of the players or for both players combined. The shading color is scaled linearly based on the rank of the outcome so that the best possible outcome from the given perspective will be bright green, the worst possible bright red and intermediate outcomes will be hues of yellow-green, yellow, and orange.

Figure 6 shows the same trace as above, first from the perspective of "Me", playing the Tit for Two Tats strategy and then from the perspective of "Opponent" playing the Alternator strategy. At a glance, it is clear that Alternator is winning.

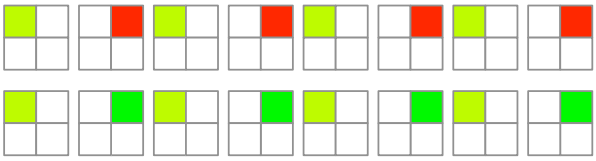


Figure 6. Traces Showing Individual Payoffs

A trace representation can be useful for analyzing the output of two strategies, but for understanding how a trace was generated, we also provide a notation for explaining how a pair of strategies produce a given game play.

2.3 Relating Strategies to Traces

By selecting a specific game play from a high-level view of a trace, we can zoom in to a detailed visual explanation as shown in Figure 7, which shows an explanation of the fourth game play in a trace of Grim Trigger vs. Opponent Alternator. The selected game play is enclosed in a red box with earlier game plays in the trace shown on the same row to the left. The matching rule from the Me strategy is shown above the game trace, and the matching rule from the Opponent strategy is shown below the game trace. Game patterns from the matching rules are shown centered above the matched game plays, and their patterns are mapped onto the trace. The different colors highlight the different places in the trace where the patterns matched.[1]

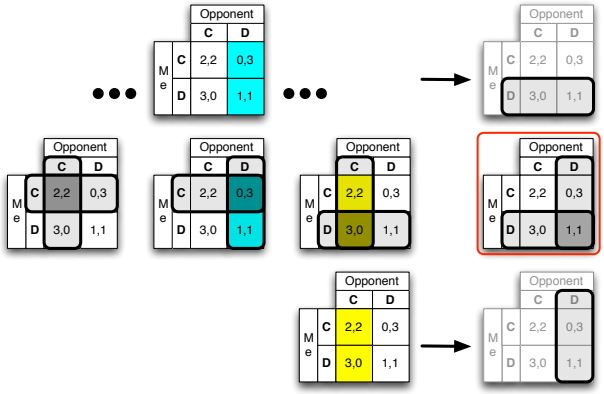


Figure 7. Game Play Explanation

In this way a user can see exactly how a specific outcome occurred given the trace that preceded it. For any previous game play that is relevant to the selection, a matching pattern will appear, and the outcome of that play will fall within the pattern.

A further expanded view (omitted here for lack of space) can be given in which the strategies are given in full, with matching strategies indicated by an asterisk, to allow the user to view the entire strategies of each player in conjunction with the trace.

Also, due to space limitations, we had to focus in this paper on only one game, the Prisoner's Dilemma. Other games can be dealt with similarly. For example, the well-known game of Rock-Paper-Scissors can be inspected at: http://eecs.oregonstate.edu/~erwig/rps.pdf.

3 Discussion of Language Design

Throughout the design process we utilized the cognitive dimensions framework supplemented with our new cognitive dimension of traceability (discussed in depth in Section 4). In this section we discuss some of the decisions that were made using these intellectual tools.

Individual cognitive dimensions often conflict with each other. To use them effectively requires identifying the dimensions that are most relevant considering the goals of the

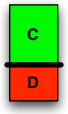
[1] When a Me and an Opponent pattern match and overlap in one game play, the overlapping cells are colored in green.

language, then weighting these more heavily when making decisions involving tradeoffs. Since we believe that traceability greatly impacts understandability, and facilitating strategy explanations was one of our primary goals, traceability emerged as our most heavily weighted dimension.

A central element of both strategy notation and game traces is the normal-form game representation. By basing our language on this ubiquitous and concise notation, we attain a high closeness of mapping with the target domain. Our decision to utilize this notation was made very early on, and many subsequent decisions were centered on conveying various meanings within this framework.

That a game play should be represented by marking the corresponding outcome in a game instance, and that a game trace should be composed of a sequence of such game plays seemed intuitive, and is justified by a high level of role-expressiveness for these aspects of the language. How to represent player moves and matching patterns was less obvious. Since traceability was of such high importance, we knew that we wanted these aspects of our language to integrate well with game traces.

3.1 Design of Move Patterns



Our initial design for player moves is shown at right—a vertical rectangle with a horizontal slider dividing it into two areas. The primary strength of this design is that it concisely represents mixed strategies. Patterns with pure right hand sides would simply have the slider dragged all the way to one side or the other, representing a 100% probability of choosing the given strategy. This syntax could be extended to games with more than two moves by adding more parallel sliders, partitioning the rectangle into the desired number of areas.

We struggled to integrate this notation with program traces, however. We decided that move patterns, like matching patterns, should be represented on a normal form representation of the game itself. This would increase closeness of mapping and consistency, and we hoped it would lead to better traceability.

Our final design, as described in Section 2.1 portrays a move by outlining the corresponding row or column on a normal form game representation. In the definition, we describe a move pattern as a special case of a simple pattern where the marked cells are all in one row or column (depending on the player). We extend the outline to include the corresponding row or column label to make this otherwise hidden dependency more explicit.

When move patterns from each player are combined, we get a game play, and the outcome is defined by the intersection of the two moves. Marking the relevant cells by outlining them (rather than coloring) enables game plays, in turn, to be combined with pattern matches, which are indicated by color. Making sure all of these elements combined well together was critical to attaining a high level of traceability.

We found, however, that just using outlines to indicate moves made the outcomes difficult to detect at a glance, indicating poor perceptual mapping. To counter this impression, we added a light shading to move outlines such that when they are overlaid they produce a darker shading (which we actually exaggerate) to clearly indicate the outcome of the game. Although the shading causes some conflict with pattern matching, the use of a gray shading vs. colorful patterns mitigates this issue, and we considered the tradeoff to be worth the change.

Fortunately, this change also facilitated a solution to the problem of representing mixed strategies, which we had set aside. As described above, mixed strategies are represented by adding additional outlines and adjusting the amount of each that is shaded. When a mixed move pattern is resolved into an actual move (that is, a move is selected from the distribution), the mixed pattern is replaced with a regular move pattern in the trace. This decision violates consistency, which is unfortunate, but not doing so would make game traces much more confusing and cause a huge sacrifice in traceability, which made us accept this deficiency. We also considered using different degrees of shading to represent mixed strategies, but it turned out that differences in shading were too subtle.

Finally, we decided to gray out the game representation that move patterns are defined on. The game beneath a move pattern represents a game that is only a future possibility which has not happened yet and is fundamentally different from realized games in a game trace or in matching patterns (which correspond directly to games in the trace). This decision was also made for consistency reasons—the notation reflects the fact that future games and realized games are semantically different objects.

3.2 Design of Game Traces

Game traces can be viewed at two abstraction levels. The first is at the level that strategies are defined. Here we can view an explanation of the trace relative to the strategies that produced it as described in Section 2.3. Many of the design decisions that were involved in making this level of abstraction work are described in the previous section, but a few others warrant mentioning. First, aligning the matching pattern of the applicable rule with the corresponding game plays in the trace was an easy decision made to maximize role-expressiveness and minimize hidden dependencies. Second, we had to consider a tradeoff between low diffuseness and hidden dependencies in determining whether or not to only show the matching rule of each strategy, or to show the strategy in its entirety with the matching rule indicated separately. Ultimately, we decided that each had a role in the language and decided on the dual explanation and expanded explanation views.

The second trace abstraction level is the high-level view described in Section 2.2. Our goal here was to simplify the notation as much as possible to allow users to focus solely on the game trace. We believe that the simplified game representations used in this view maximize traceability, terseness, and visibility while maintaining reasonable levels of consistency and role expressiveness.

By examining the game trace with minimal distractions, our hope is that users can clearly see how strategies interact with each other by the pattern (or lack thereof) represented in their game trace. The high-level view also includes a way to visualize the success of a strategy or combination of strategies by color-coding the outcome based on the quality of the payoff. We believe that this representation provides excellent perceptual mapping as it makes the relative success indicated by a trace recognizable at a glance.

4 Traceability as a Cognitive Dimension

Traceability measures a notation's ability to represent and to relate to its semantics. A more general name for this aspect would be something like "semantic accountability", but since in many cases the semantics is represented by a trace, we have chosen the more concrete name. The rationale for investigating the traceability of a notation is to get a sense of the overall understandability of the notation and, in particular, of how much program understanding can be supported by the trace notation.

The investigation of traceability requires a definition of the trace notation, which depends on the program notation. This also means that traceability can be investigated only for those notations for which the notion of a trace makes sense, which will be most program notations.

In our domain of iterated games in game theory, a trace is given by a sequence of individual game instances. When considering imperative languages, a trace may be a series of memory fragments. A trace for lambda calculus is given by a sequence of lambda expressions. In each case the sequence of elements in a trace is not arbitrary, but rather the result of an effect caused by part of the program (a pair of rules, a statement from the imperative program, or an application of a lambda abstraction to an expression).

Traceability involves two distinct, but closely related aspects. First, it expresses how well traces can explain the meaning of a program. Second, it reflects how integrated the notations for traces and programs are, and in particular, how well a specific program part can be related to the part(s) of the trace it is affecting.

In this context, we should also mention the distinction between "static" and "dynamic" traces. A static trace represents the time component explicitly, that is, spatially. For example, our notation features static traces, representing the passage of time by moving left to right along the horizontal axis. On the other hand, typical debuggers for imperative languages produce a dynamic trace, where the passage of time is represented by replacing one state with another.

It seems that static traces are prefereable in principle since they reduce the need for memorizing trace elements and can thus reduce the cognitive load on the viewer whereas complex dynamic traces often prompt users to rerun the program because old state can be "forgotten". On the other hand, when the trace notation becomes too complex or too large, a dynamic trace might be required as a modularization device to make the communication of the huge amount of information possible at all.

4.1 Interaction with Existing Cognitive Dimensions

The existing cognitive dimension of closeness of mapping measures how well a program relates to its abstract semantic domain. In contrast, traceability measures how well a program relates to a concrete representation of its semantic effects. Therefore, traceability is similar in nature to closeness of mapping, but uses a trace as a different, more concrete target. Traceability may be considered something of a second-order cognitive dimension since it reflects closeness of mapping between program and execution notation rather than program notation and abstract domain.

The ultimate goal of traceability is to make programs easier to understand. We believe that by visually relating a program to its effects, relationships between a program and its output can be made clearer and easier to understand by people. Other cognitive dimensions have this same motivation. One is visibility, which considers how easily elements of a program's representation can be viewed. Traceability considers how easily a program's relationship to its execution can be viewed. In some sense, traceability is an extension of visibility from static representation to a program's execution.

Traceability is also related to the existing cognitive dimension of progressive evaluation. Progressive evaluation considers whether or not a program can be executed before it is completely written. The intention is to help a user check their program's correctness as they are writing it. Although traceability is primarily concerned with viewing the execution of completed programs, by increasing the visibility of program execution, it may also help increase a user's confidence in program correctness.

In designing our notation for strategies we recognized a tradeoff between traceability and abstraction. We considered adding an abstraction mechanism to our language that would have allowed other strategies, referenced by name, on the right-hand side of a strategy definition. Besides the usual well-known benefits of abstraction, this change would have also made our language fundamentally more expressive, allowing us to create strategies with more complex matching patterns (for example, multiple existential patterns).

This abstraction mechanism, however, made it much more difficult to illustrate how strategies interact with a game trace. In other words, it negatively affected traceability. Since high traceability was one of our primary design goals, we decided not to adopt this language feature despite its benefits elsewhere.

4.2 Independence of Trace Role and Integration

Traceability as a measure for program understandability depends on how much the trace plays a part in understanding the program and how well the trace can be related to the program. By examining traceability for several existing languages we would like to demonstrate that these two aspect are orthogonal.

In some languages, such as Logo [7] or Alice [1], the trace *is* the program output, that is, it does not need motivation or additional definitions and thus is an integral part of understanding programs. On the other hand, in traditional imperative languages or lambda calculus [3] a trace is only employed to show intermediate results, which means that looking at a trace requires a mode of executing a program that is different from running the program just for its result and that is typically employed only if the result asks for an explanation.

The prominent role of traces in Logo or Alice does not imply a tight integration of traces and program notation, which is obvious since in both cases the program notation is textual, but the traces are pictures or animations. Explicit mechanisms to annotate the program notation and synchronize it with the visual program output are required to enable tracing. In this respect, the situation for Logo and Alice is very similar to that of debugging interfaces of traditional imperative languages.

On the other hand, traces explaining the evaluation of lambda calculus expressions are highly integrated with the program notation since each part of the trace is a (partially evaluated) program (that is, a lambda expression). All that is needed as additional notation is typically the underlining of the subexpression being reduced in the next step.

4.3 Traceability as a Notation Design Tool

To elevate the notion of traceability to a cognitive dimension we should give advice as to what kind of changes in notation could increase traceability [4]. Such guidelines are important if we want to have traceability not only as an evaluative tool, but also as a design toool.

One strategy to improve traceability is to share notation between traces and programs to support the identification of program parts with relevant parts of the trace. Also, aiming for modular program notation allows the isolation of program parts to facilitate the combination with traces.

Moreover, based on the observations in Section 4.2, we believe that observing and making explicit the two aspects of role and integration for a particular language, helps to find ways to improve the language. For example, to increase traceability of Alice we could try to define a notation that lets program (parts) be associated with the animated objects, which would increase the integration aspect tremendously.

5 Related Work

Our notation builds on the normal-form representation of games—a matrix of payoffs indexed by each players' move. While very common, it is not the only well-used game representation.

Games in *extensive form* are represented as trees. Decisions and variables are represented as non-leaf nodes and payoffs as leaves. Moves are represented as edges from a decision node to its children. This representation can more easily represent games in which a single player must make more than one decision, games involving random variables, and games with more than two players. This increased flexibility, however, comes at the expense of terseness.

Multi-agent influence diagrams are high-level graphs showing the relationships between variables and the decisions made by agents in a game [9]. This representation excels at showing relevance in real-world games with many external variables, but usually omits the move and payoff information required to define a game precisely.

Some aspects of our design could be extended to these other representations. Matching patterns of previous payoffs, for example, could work with extensive form games by simply replacing matrices with trees and marking the outcomes by highlighting the corresponding leaves. Move patterns could be represented by highlighting edges from decision nodes; these could be combined to form a game play, represented by a complete path from the root to a leaf node. There are many obstacles that would have to be addressed for such an approach, however. Inheriting from the normal-form representation, our notation assumes *imperfect information*, which means that a player makes his move without knowledge of the other player's move for this game. Extensive form games often lack this assumption, however. Extensive form games also easily accommodate situations in which a player must make more than one decision in a single game, which is not something our current design considers.

Representing strategies is a surprisingly unexplored field. Strategies are typically defined textually or with psuedo-code. One attempt at providing a more user-friendly means to specify strategies for the Iterated Prisoner's Dilemma competition provided tables enumerating every possible combination of moves for the past one to three rounds of play, allowing the user to specify an action for every case [8]. Besides not scaling past a few games of recent history, there are a many common strategies that are simply not expressible in this notation.

The cognitive dimensions framework provides a terminology relating programming language design to cognitive issues in programming [6]. The framework has been built upon and extended many times, for example, [5, 14, 17]. Actively encouraging extensions, Alan Blackwell outlines some guidelines for proposing new dimensions in [4]. We have tried to take these guidelines into account in our definition of traceability in Section 4.

Our visual notation for strategies employs the idea of visual rewrite rules that is part of many visual languages. Two well-known examples are AgentSheets [11] and StageCast creator [13]. In the AgentSheets approach, for instance, a program is given by a set of visual rewrite rules that map one graphic state, represented as a 2D tile, to the next. When a program is run, these rules are applied repeatedly producing an animation. These animations can be considered a form of dynamic trace which helps to visualize and understand the effects of the rules. AgentSheets lacks a mechanism, present in our language, to explicitly relate an element of the current state to the corresponding rules that generated that state, which would be difficult since the state manipulated is more complex than that of game instances. Moreover, AgentSheet rules can refer only to the previous state, which means that actions described by rules cannot be predicated on the history of the state, and existential or forall quantifications cannot be expressed. Since AgentSheets offers many more general-purpose programming elements than our game theory notation, it is not surprising that it is more difficult to achieve the same degree of traceability, an observation that reflects the tradeoff between traceability and abstraction.

Software visualization considers the representation of programs and their execution to facilitate reasoning and understanding throughout the software development cycle [15]. Our new cognitive dimension of traceability extends these considerations to the language design phase. Petre and de Quincey state, in an overview of the field of software visualization, "Currently, it is still arguable that what is visualized is what can be visualized, not necessarily what needs to be visualized" [10]. By addressing traceability in the language design phase, languages can make their programs more amenable to visualization, which will make it easier to increase visualization coverage to those areas that most need it.

6 Conclusions and Future Work

This work is part of our effort to develop a new paradigm of explanation-oriented languages, which are languages whose objective is not only to describe the computation of values, but also to create explanations of how and why those values are obtained. By directing the language designer's attention to the notation of traces and the integration with the program notation the cognitive dimension of traceability advocates the design of notations with explanatory value.

As we have demonstrated, the presented notation for game strategies and traces exhibits a high degree of traceability and can thus serve as an explanatory tool for game theory for a broad audience.

For future work we intend to apply design explanatory notations in other domains of public interest (for example, to explain simple probabilistic reasoning) using the traceability criterion as a design guideline. Moreover, we plan to consider the redesign of existing languages to improve their traceability.

References

[1] Alice 3.0. http://www.alice.org.

[2] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1984.

[3] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North Holland, 1984.

[4] A. Blackwell. Dealing with new Cognitive Dimensions. *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community., University of Hertfordshire*, 2000.

[5] R. Dondero Jr and S. Wiedenbeck. Subsetability as a New Cognitive Dimension? *Proceedings of PPIG 18*, 2006.

[6] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[7] B. Harvey. *Computer Science Logo Style (2nd ed.)*. MIT Press, 1997.

[8] G. Kendall, P. Darwen, and X. Yao. The prisoner's dilemma competition, 2005. http://www.prisoners-dilemma.com.

[9] D. Koller and B. Milch. Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, 45(1):181–221, 2003.

[10] M. Petre and E. de Quincey. A gentle overview of software visualisation. *PPIG Newsletter*, 2006.

[11] A. Repenning and T. Sumner. Agentsheets: a medium for creating domain-oriented visual languages. *Computer*, 28(3):17–25, 1995.

[12] B. Schneier. Drugs: Sports' prisoner's dilemma. *Wired News*, August 2006.

[13] D. Smith, A. Cypher, and L. Tesler. Programming by example: novice programming comes of age. *Communications of the ACM*, 43(3):75–81, 2000.

[14] M. Stacey. Distorting design: unevenness as a cognitive dimension of design tools. *Adjunct Proceedings of HCI*, 95, 1995.

[15] J. Stasko. *Software Visualization: Programming As a Multimedia Experience*. MIT Press, 1998.

[16] P. Straffin. *Game Theory and Strategy*. The Mathematical Association of America, 1993.

[17] S. Yang, M. Burnett, E. Dekoven, and M. Zloof. Representation Design Benchmarks: A Design-Time Aid for VPL Navigable Static Representations. *Journal of Visual Languages and Computing*, 8(5-6):563–599, 1997.