Journal of the Learning Sciences

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/hlns20

Using Learning Analytics to Understand the Learning Pathways of Novice Programmers

Matthew Berland a, Taylor Martin b, Tom Benton c, Carmen Petrick Smith d & Don Davis e

a Department of Curriculum and Instruction, University of Wisconsin-Madison
b Department of Instructional Technology and Learning Sciences, Utah State University
c Department of Curriculum and Instruction, University of Texas at Austin
d Department of Education, University of Vermont
e Department of Interdisciplinary Learning and Teaching, University of Texas at San Antonio

Accepted author version posted online: 05 Sep 2013. Published online: 29 Oct 2013.


To link to this article: http://dx.doi.org/10.1080/10508406.2013.836655

PLEASE SCROLL DOWN FOR ARTICLE

Taylor & Francis makes every effort to ensure the accuracy of all the information (the “Content”) contained in the publications on our platform.
Using Learning Analytics to Understand the Learning Pathways of Novice Programmers

Matthew Berland  
Department of Curriculum and Instruction  
University of Wisconsin–Madison

Taylor Martin  
Department of Instructional Technology and Learning Sciences  
Utah State University

Tom Benton  
Department of Curriculum and Instruction  
University of Texas at Austin

Carmen Petrick Smith  
Department of Education  
University of Vermont

Don Davis  
Department of Interdisciplinary Learning and Teaching  
University of Texas at San Antonio

Many have suggested that tinkering plays a critical role in novices learning to program, and recent work in learning analytics (Baker & Yacef, 2009; Blikstein, 2011) allows us to describe new relationships in the process. Using learning analytics, we explore how students progress from exploration, through tinkering, to refinement, a pathway that we term EXTIRE. The work contributes to learning sciences by: showing empirical support for previously theorized processes; identifying a role of tinkering in novices’ learning; and presenting a data-driven approach to creating
process descriptions. Furthermore, our findings illuminate how tinkering can be a valuable approach for novices.

Although many agree that computational literacy is of vital importance to our country, it is an “amorphous concept” (Haigh, 1985, p. 161; see also diSessa, 2000). One important aspect of computational literacy is the ability to flexibly use computers as a tool based on solid understandings of basic concepts integral to programming, including “iteration, recursion, systematic procedures, and modularization” (Haigh, 1985, p. 165; cf. Resnick & Wilensky, 1998; Wing, 2008). Also, students’ ability for self-expression with computers represents another significant concern. Papert (1980) argued that computational literacy is analogous to print literacy, in that print literacy does not stop at reading—a print-literate individual can express himself or herself in print. Similarly, computational literacy includes the ability to express oneself with computational tools and methods such as programming computers.

In this article, we use educational data mining and learning analytics (henceforth, EDM) to better understand the mechanisms of how students begin to learn to program and work through creative processes with computation. In particular, we focus on how students’ initial tinkering with code—a process variously championed and maligned—shapes how they learn to build working programs. This work is framed as an EDM study because recent work in learning analytics has made it possible to look at the effect of many miniscule changes (a hallmark of tinkering) across many students, and recent work in EDM has made it possible to look at the relationship of those many tiny changes to the aggregate work of the class and more traditional summative measures.

**LEARNING TO PROGRAM**

Despite its importance, relatively little is known about the process of how people begin to learn to program (Baldwin & Kuljis, 2001). Research into the thoughts and strategies novices use when developing programs can help researchers gain traction in researching the learning processes involved.

This should not be understood to signify that there is generally a lack of research examining novice programmers; on the contrary, there is a long history of examining novice programmers (e.g., Soloway & Spohrer, 1989). Rather, often owing to practical considerations, some aspects of novice programmers’ learning trajectories have received relatively little or no attention. Among those studies conducted, many have focused on the error-prone and inefficient strategies of novice programmers relative to experts (Soloway & Ehrlich, 1984; Weiser &
Shertz, 1983). These studies have been primarily conducted in traditional programming environments, typically with classes of students working independently at desktop computers using a standard programming language such as Pascal or BASIC (e.g., Putnam, Sleeman, Baxter, & Kuspa, 1989).

The findings of these studies suggest that novices tend to have an inadequate mental model of how computers operate (Bonar & Soloway, 1983; Robins, Rountree, & Rountree, 2003; Rogalski & Samurçay, 1990; Winslow, 1996). Such novices often rely on general instead of problem-specific strategies (Winslow, 1996). Also, they fail to apply knowledge and strategies they already possess in the appropriate situations (Winslow, 1996), even when they can articulate the steps that the computer should take to solve a problem (Bonar & Soloway, 1983). Furthermore, in contrast to more proficient programmers, they often approach problems using bottom-up instead of top-down strategies (Winslow, 1996), and their code is often inefficient (Helminen, Ihantola, Karavirta, & Malmi, 2012).

ALTERNATIVE PROGRAMMING ENVIRONMENTS

Researchers have noted that the programming language itself may significantly impact students’ ability to learn programming concepts, successfully program, and transfer skills (Cunniff, Taylor, & Black, 1989; Hundhausen, Farley, & Brown, 2009; Stefik, Siebert, Stefik, & Slattery, 2011). Following this idea, many alternative programming environments have been designed with the goal of providing tools and applications that help novices learn to program by better engaging them with programming (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), creating physical artifacts (cf. Eisenberg, Elumeze, MacFerrin, & Buechley, 2009), and minimizing frustration by reducing the possibility of difficult-to-interpret error messages (Maloney et al., 2010). For instance, researchers have found that tangible or embodied interfaces can promote children’s engagement with programming (Buechley, Eisenberg, Catchen, & Crockett, 2008; Kafai, Fields, & Searle, 2012). In the UbiPlay environment, children program pressure-sensitive tiles on a playground in order to provide interactive stories and games (Mattila & Vääätänen, 2006). Other common tangible environments include Topobo blocks, for constructing mobile (creature) creations (Raffle, Parkes, & Ishii, 2004); and the LilyPad Arduino, a programmable, often wearable, microcontroller (Buechley et al., 2008). Similar efforts aim to make the world more programmable with adaptive, malleable materials such as shape-changing, flexible displays that can serve as interfaces (Coelho et al., 2009; Coelho & Zigelbaum, 2011).

Many environments aim to provide ownership of programming products by making them real products of students’ activity, such as models, stories, or animations. For instance, the NetLogo programming language and framework uses...
agent-based modeling to introduce learners to systems thinking and computational thinking (Stonedahl, Wilkerson-Jerde, & Wilensky, 2009; Wilensky, 1999). In addition, using programming for storytelling and other multimedia purposes can benefit students traditionally underrepresented in science, technology, engineering, and mathematics (Adams, 2007; Forte & Guzdial, 2004; Reynolds & Caperton, 2011). Alice focuses on animation and multimedia (Kelleher, Pausch, & Kiesler, 2007). RiTa focuses on applying computational concepts and tools to analyzing, altering, and generating text (Howe, 2009). Scratch (Resnick et al., 2009) utilizes a design blocks framework to provide a friendly and accessible programming interface for “Scratchers” to create games, animations, stories, music videos, and many other applications that they can then share.

A feature of many of these alternative programming environments is that they are designed to make it possible for relative novices to access and utilize more complex programming concepts more quickly than with traditional programming languages. These environments minimize the possibilities of creating programs that will not run, particularly for simpler syntactic reasons. Instead, when users cannot achieve their goals, they receive feedback quickly, and that feedback makes it easier to trace problems to their cause. This is true of Logo (Papert, 1980) and its descendants NetLogo (Wilensky, 1999), Scratch (Resnick et al., 2009), and Squeak (Ingalls, Kaehler, Maloney, Wallace, & Kay, 1997), as well as rule-based platforms such as AgentSheets (Repenning, Ioannidou, & Zola, 2000).

Although there is less research on novices’ learning in alternative programming environments than in traditional ones, there is evidence that alternative environments support accelerated progress toward core concepts (or “powerful ideas” as per Papert, 1980) in computational literacy. In a study of 80 relatively novice students working on Scratch projects, Maloney, Peppler, Kafai, Resnick, and Rusk (2008) identified several programming concepts common across hundreds of projects. In Year 1 of their study, several concepts were shown to increase (ordered largest to smallest): user interaction, loops, conditional statements, communication and synchronization, Boolean logic, variables, and random number use; all of these are core concepts across the computational literacy literature as well as the computer science literature. In Year 2 of the study, concepts of Boolean logic, variables, and random numbers were used more frequently.

In evaluating the Alice multimedia programming environment, Kelleheer et al. (2007) provided a comparison of two versions of Alice: one optimized for storytelling purposes and one not. The authors examined the use of loops, new method statements, and user time on task. Students working with Storytelling Alice utilized more new methods and spent far more time editing and running their programs, whereas students using generic Alice spent most of their time on less substantive programming activities, primarily changing appearance features of their programs (e.g., constructing scenery for stories and games).
TINKERING

A common goal among the design of these alternative programming environments has been to support tinkering, an exploratory activity many researchers claim helps novices learn to program (Hancock, 2003). Though definitions of tinkering vary, there is a variety of evidence that certain benefits persist across the definitions of tinkering. This suggests that a more operationalizable definition of tinkering may lead to a better understanding of whether and how tinkering supports student learning. Among identified definitions, tinkering is described as follows:

1. A non-goal-oriented (or atheoretical) exploration of a problem space (Petre & Blackwell, 2007)
2. Creating a working version of a product/program without necessarily understanding all of the things done to make it work (H Hancock, 2003)
3. Playful experimentation (Beckwith et al., 2006)
4. A process of testing minor changes (Brandt, Guo, Lewenstein, Dontcheva, & Klemmer, 2009)
5. A process of trial and error (Dorn & Guzdial, 2010; Law, 1998)
6. Bricolage: a process described by Turkle and Papert (1990) as a conversation between programmer and program, navigation through missteps, and planning little more than a step ahead—a description that aligns strongly with Clegg and Kolodner’s (2007) description of bricolage in scientific inquiry as direct interactions with objects, rather than more distant, planning-oriented approaches
7. Fussing: a process of minor change that leads to improved products, arising from serendipitous variability in students’ actions and ideas while engaged with a learning activity (Martin, 2009)
8. Just-in-time activity (Petre & Blackwell, 2007; Suchman, 1987; Turkle & Papert, 1990), whereby just-in-time research (such as Wikipedia searches) among programmers has also been noted as an aspect of tinkering (Brandt et al., 2009; Dorn & Guzdial, 2010)

These different definitions of tinkering appear to be divergent because they describe two distinct aspects of the word: Tinkering describes both (a) an orientation and (b) a set of activities. The tinkering orientation is essentially atheoretic—tinkering, bricolage, and playful experimentation describe the act of either having or needing no plan in the process of creating or modifying a computer program. The set of activities described as tinkering include trial and error, messing around or fussing, finding and using feedback mechanisms (such as testing), or combinations of those activities.
How Tinkering Could Impact Novices’ Learning

Some researchers suggest that the tinkering orientation and associated activities are helpful for novices learning to program, whereas others claim that they are less beneficial. We discuss both views here.

The iterative and variable nature of tinkering activity can help novices get started in programming. The bottom-up or guess-and-check processes (Burke & Kafai, 2010; Resnick et al., 2009) involved in tinkering support an iterative process of the discovery of key elements in a programming language or environment (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Turkle & Papert, 1990). Researchers have found that novice programming students who tinker extensively and systematically (not haphazardly) experience greater learning gains (Perkins et al., 1986). They have found that upon reaching an impasse, such students repeatedly change and run their code to find a solution, while utilizing planning skills and learning more about programming, whereas others tend to simply quit or proceed haphazardly. This relates significantly to examinations of mathematics learning, whereby researchers have found that similar iterative processes incorporating increased variability in students’ mathematical problem-solving attempts can improve learning (e.g., Martin, 2008; Siegler, 2007). For example, when students attempt new types of fraction problems, variability in their actions with manipulatives and in their mathematical ideas can help learners “unstick” themselves from unhelpful, nonoptimal, and “buggy” patterns of activity while providing learners with a better overview of more useful patterns of activity (Martin, 2009).

In addition to facilitating learners’ discovery or construction of knowledge, tinkering may support learners in other novel ways. For example, tinkering may support novices’ learning trajectories by giving them the opportunity to be successful with programming while developing the knowledge and planning skills that they will need to become better programmers (Hancock, 2001). Also, the playfulness involved in tinkering may allow students to do more advanced programming sooner, particularly in alternative programming environments. This is apparent in younger students, who, while tinkering in programming activities, do not consider themselves to be programming but rather to be playing (Petre & Blackwell, 2007). In research on alternative programming environments such as Scratch and in tangible environments such as e-textiles (e.g., Buechley et al., 2008), qualitative results frequently indicate that students creating advanced programs do not identify the core of their activity as programming but rather focus more on the creative aspects of their activity (Maloney et al., 2008).

Others have suggested that tinkering is not helpful for novices. For example, Yeshno and Ben-Ari (2001) characterized the practice of bricolage as aimless. Others have gone further in indicating that tinkering may in fact be detrimental to students’ learning. In a study of debugging by novice and expert computer
scientists, Law (1998, p. 331) characterized tinkering as an “ad hoc trial and error approach” that frequently created more errors than it solved. This was both supported and contradicted by Perkins et al. (1986), who found that haphazard tinkering negatively impacted students’ progress, whereas cautious tinkering proved beneficial.

In summary, it appears that tinkering may be motivating and may allow students to learn more complex concepts quickly, but it may not be well adapted to traditional classroom computer science. However, this statement can be made only tentatively because of the relative lack of systematic work on the subject.

Defining Tinkering for This Study

Across myriad definitions, tinkering appears to be an authentic process by which both novices and experts program and learn to program, and a plurality of the studies cited previously suggest that tinkering is a process or practice with benefits or promise. Given the variance among definitions of tinkering and theories regarding how it could help learning, our goal in this study was to use exploratory data mining methods with log data of students’ programming to potentially refine ideas about tinkering. Thus, we identified a working definition from key features we thought to be common to many of these definitions: Tinkering generally involves exploration, it is a playful activity involving just-in-time planning, it often precedes the ability to articulate the reasons an action produced the desired result, and it produces feedback from the environment.

LEARNING ANALYTICS

Though the literature shows some evidence of positive effects of tinkering, the mechanics of when, how, and why students tinker remain elusive. This may be due to the fact that students use tinkering as a just-in-time response, and, thus, it may happen quickly and with great frequency. Those features make it difficult to capture with traditional quantitative or qualitative methods. Although previously conducted qualitative studies can provide rich descriptions of learning processes at a fine level of granularity, it is difficult to generalize these results. Standard quantitative methods do not support the examination of data at a sufficiently fine granularity. For these reasons, within this study, we used learning analytics and data mining methodologies.

There is a growing body of literature focusing on the use of computational tools, such as data mining, visualization, and numerical modeling, to describe student learning (Baker & Yacef, 2009). Continual advancements in processing power have showcased the potential of data mining, which Rubenking (2001)
described as “the process of automatically extracting useful information and relationships from immense quantities of data” (p. 86). Two primary elements of this approach are relatively new and novel in educational research. First, data mining allows the researcher to study larger data sets in greater depth. Second, data mining can utilize a variety of analytical techniques to find unknown but existing patterns in data rather than proceeding from a query initiated by a traditionally testable hypothesis. On the larger systemic scale, data mining can inform decision-making processes regarding curriculum, potentially providing differently useful answers than standard statistical analyses. For example, using fluid flow and diffusion models, Marder and Bansal (2009) were able to predictively describe the trajectory of standardized test scores relative to dropout probability in high school.

The combined use of data mining and associated computational techniques to study learning processes, referred to as learning analytics, provides significant support in furthering educational research. Data gathered during interstitial periods of learning are copious thanks to many new digital educational applications, and this opens the door to studying how students learn complex content in open-ended environments (Blikstein, 2011; Sherin, 2002). Mining such data may provide new and unique insights into how people learn, which may further inform teaching best practices (Bransford, Brown, & Cocking, 2000). In particular, it may expose otherwise hidden patterns in learning processes and allow the testing of large numbers of hypotheses.

RESEARCH QUESTIONS

In this study, we collected data from 53 female high school students learning to program using IPRO (described in “Programming Environment”). These students were relatively novice programmers. They worked in teams, with each student programming her own virtual soccer-playing robot (a bot). During the course of a 90-min IPRO session, teams iteratively programmed and observed their bots in practice settings and scrimmages against opponents. Observing their robots’ actions allowed students to see their code in action, recognize bugs, and refine their programs.

We used log data from each student over the course of the session to examine the following questions:

1. How, in the aggregate, does students’ programming activity change over time?
2. What does this activity reveal about tinkering processes?
3. How do these changes relate to the quality of the programs students are writing?
METHODS

Participants

Fifty-three high school girls participated. They were all enrolled in a game development summer camp intended to dispel myths about computer science and develop interest in the field. During the weeklong camp, students stayed on campus, collaborated on programming games, and met professional computer scientists. Our research was conducted on the morning of the first day of camp to minimize any potential confounding effects from other game development or programming activities. Participants applied to attend the camp, and camp personnel made acceptance decisions on the basis of performance in science, technology, engineering, and mathematics courses; an application essay; and an academic letter of recommendation. All camp attendees were invited to participate in our study. Although all attendees completed the IPRO programming tasks as part of their scheduled camp activity, only 53 of the 60 attendees returned parent consent forms enabling their data to be examined for the study. Attendees received no compensation for participating.

Participants’ mean age was 16.5 years (SD = 0.6). Eighteen of the girls had recently completed their second year of high school, whereas 35 had just completed their third year. On a survey given before they began IPRO activities, students typically reported a medium level of prior experience and comfort with programming. Measured on a scale from 0 to 10, students’ mean level of comfort with programming was 6.02 (SD = 0.32) and experience with programming was 5.24 (SD = 0.35), which, although already at a medium level, did allow room for growth in the course of the session (Martin, Berland, Benton, & Smith, 2013).

Materials

Programming Environment. During the study, participants used IPRO, a programming game for iOS devices, such as the iPhone and iPod Touch. IPRO, an abbreviation of both “iPod robotics” and “I (can) Program,” is an alternative programming environment in which students use handheld devices to program virtual robots to play soccer. Designed to promote computational thinking in an accessible, engaging way, IPRO challenges students to program virtual soccer players (or bots) using a simple visual programming language designed by the first author. The IPRO programming environment is covered in more depth in Berland, Martin, and Benton (2010), but it can be described succinctly as an iOS app in which each individual student programs her own bot to exist in a shared virtual online space (similar to a massively multiplayer online game). IPRO has two core “screens”—a programming space and a virtual soccer field where students’ bots enact their programs (see Figures 1 and 2). The students work to build a bot in concert with their team of fellow students (usually 4–8 students per team). Students walk around
with their iPod, write new code, debug their bot with friends or alone, and engage in matches between teams. IPRO also uses a customized logging scheme meant to simplify learning analytics and EDM. The authors designed and deployed the environment, data logging system, methods, and analyses; all were built for this study.

In IPRO, the students’ bots compete on an $8 \times 13$ hexagonal grid with goals at the top and bottom of the screen. The field, with a game in progress, appears in Figure 1.

The IPRO programming language is a signal-based functional visual programming language; it consists of a programming interface and a library of programming “primitives” (basic programming elements, informally called blocks in IPRO). The primitives include Boolean logic operators (e.g., “logical or”); actions (e.g., moving, turning); and sensors to detect the ball, opponents, or a goal. Appendix A shows how to construct a very simple program in IPRO, and Appendix B lists the IPRO primitives. Players can combine primitives to create bots that will, ideally, navigate the game space and push a soccer ball toward the appropriate goal. IPRO includes a variety of logical primitives: a conditional
FIGURE 2  A simple IPRO program. A conditional block first calls the IB (or “left ball”) sensor. If it finds the ball, it moves to the front left; if it does not find the ball, it turns to the right (color figure available online).

IF/ELSE block as well as OR, AND, NOT, greater than, and less than primitives (the latter two referencing distance on the field of play).

These primitives combine to create an IPRO program that consists of a set of nested conditional statements that govern the behaviors of a single soccer-playing bot. The statements typically require sensors to determine the locations of the ball, goals, and other players and move the bot accordingly. The program is repeatedly executed until the match reaches the previously determined end point. A simple IPRO program appears in Figure 2.

There is no limit to the nested complexity of an IPRO program, but the design of the programming language makes syntax errors impossible. Like Scratch (Resnick et al., 2009), blocks only fit together in syntactically valid ways. This lack of syntax errors is an important design factor in presenting an accessible, non-intimidating programming experience. IPRO is a very fast visual programming language: A novice can make a basic working program in less than 5 min. Further detail regarding the IPRO design is provided by Berland, Martin, Benton, and Petrick (2011), and the IPRO program is freely available in the iOS App Store.
Using IPRO on a mobile device (in this case, the iPod Touch) provides two affordances that are relatively new to programming. First, students are able to mimic the actions of the bots by moving around in physical space and by using their physical intuition to inform programming choices (Petrick, Berland, & Martin, 2011). Second, mobility contradicts a common perception of programming as solitary and nonsocial (Cheryan, Plaut, Davies, & Steele, 2009); while using IPRO on a mobile device, students move around and share their work with their peers in order to collaborate, consult, or simply show off.

A student can execute a program in one of three modes. In Solo play, the student’s bot and a ball are each randomly placed on an empty playing field, and the program is then run continuously. This provides an environment well suited for basic program debugging. In Scrimmage mode, a student’s bot is placed on the field alongside other online student bots downloaded from the IPRO server, and each bot executes its program in turn. This mode is useful for testing how a bot reacts to and behaves in the presence of teammates and opponents. Match play is similar to Scrimmage mode but runs a fixed number of turns. Whereas bots compete endlessly in Scrimmage mode, the game ends after a total number of turns in Match play. For example, after 500 total turns (program executions) the game is over, and a winner is determined.

Procedure

The study was conducted during a 2-hr session the morning of the first day of camp following an introduction to the camp conducted by its organizers. The session began with an attitudes presurvey and a pretest and ended with equivalent postsurveys and posttests. We do not report on the surveys or the tests here, as they do not pertain to the research questions in this article.

Activities. Prior to the session, we randomly assigned students to teams of three or four. The floor of the large room where the study took place was covered in tarps decorated like the IPRO field, with a hexagonal grid and goals at each end. A deflated soccer ball was placed on each tarp for students to push from one hexagon to the next as robots do in the IPRO environment. As students entered the room, they sat around the edges of the tarps. After all students had gathered around the tarps, the session’s activities began. Table 1 displays the three main types of activity during the IPRO session.

In Activity 1, a researcher welcomed the students and described the structure of the IPRO session. A large screen displayed a series of basic programs, and the researcher led the students in physically acting out the IPRO bot’s moves using the tarp and the deflated soccer ball (see Figure 3). Then each student was provided with an iPod Touch with IPRO installed. Each participant was given a unique login
TABLE 1
Activities in the IPRO Programming Session

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time (min)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity 1</td>
<td>7</td>
<td>Instruction and guided programming</td>
</tr>
<tr>
<td>Activity 2</td>
<td>19</td>
<td>Individual/team programming in Solo mode</td>
</tr>
<tr>
<td>Activity 3</td>
<td>29</td>
<td>Scrimmage and Match play</td>
</tr>
</tbody>
</table>

FIGURE 3 On a classroom tarp replicating the IPRO soccer field, a student considers whether she has found the ball (currently behind her) by acting out the RB (“right ball”) sensor (color figure available online).

ID and team assignment. A researcher then demonstrated how to create a program and how to access Solo play mode.

In Activity 2, students collaborated in teams of three and four while working on individual iPods to program virtual soccer players in preparation for four-on-four matches. During this activity, researchers’ assistance was limited to technical support issues (e.g., reconnecting to the wireless network). For questions about creating programs, researchers responded by encouraging students to act out the robot’s moves using the grid and soccer ball. Throughout this and subsequent activities, students frequently used the physical grid and ball on the floor to support their own visualization of programming ideas as well as to communicate such ideas to others.

In Activity 3, students alternated between Match play and programming. They competed in team matches in an informal tournament. The matches were projected
on a large screen for all to watch, allowing students to see the results of their programming on their team’s soccer match performance. Following each match, the teams returned to programming in preparation for another match. Students did not engage in a predetermined number of matches but rather iterated through matches and programming until the end of the session. Teams could refine their programs as long as they liked between matches, and when two teams were ready a member of the research team would start their match.

Following the IPRO session, students continued other camp-related activities such as listening to guest speakers and programming in other contexts; however, this was not part of our study.

Measures

The primary data source was the set of all student programs. Every time a student changed her program, a snapshot was saved of that program at that point in time, which we refer to as a program state (program states $M = 132.39$, $SE = 6.85$). For every change, the time stamp, unique user identifier, and program state were logged. Using these data, we then calculated a set of measures characterizing individual program states and changes between program states. Using a subset of these measures, we used clustering to find categories of program states. We then created a map of change (the TRACKER Path in Figure 4) by examining the likelihood of each program state transitioning to every other program state (visible in Figure 5). The TRACKER Path allowed us to examine our first research question regarding how programming activity changes. Next we examined these changes to investigate our other research questions regarding tinkering and program quality. Guided by the results of our examinations, we propose the Explore, Tinker, Refine (EXTIRE) framework to explain our findings with respect to tinkering and then relate this framework to types of transition activity. Next we examine in aggregate how students’ programming activity in the IPRO session changed over the course of the session and relate that to both the EXTIRE framework and the quality of students’ programs.

In this article, we describe both (a) aggregate measures and (b) features of a program state. We use more traditional statistical approaches to analyze the aggregate measures across the whole classroom, and we use methods from EDM to analyze the relationships between the features. Note that the descriptions of measures and the features are largely the same; this is both for consistency and because the features/measures that we use are the ones we found to be the most salient ones for investigating tinkering.

Measures of Individual Program States

Action. This is the number of action primitives in a program state, as per Appendix B. In our IPRO studies, we have noticed that students tend to include
the action primitives more frequently than other types of primitives in their early programs, so we chose to measure action primitives separately.

Logic. The logic measure is the total number of logic and sensor primitives in a program state, as per Appendix B. Understanding and using Boolean logic is a core, and perhaps the most widely agreed upon, principle of computational thinking (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011).

FIGURE 4 TRACKER Path illustrating the most common paths between clusters of program states (color figure available online).
Unique primitives. This is the number of unique action, logic, and sensor primitives in a program state. This measure provides an estimate of the number of primitives that a student could use in a program state and an estimate of the novelty of that state compared to other students’ program states. An IPRO program state with many different types of primitives is difficult to design, and such a program state requires that more mutually independent program variables be considered.

Length. This is the total number of primitives in a program state, equivalent to lines of code in most other programming languages. The length measure
provides a basic indicator of the effort expended on a program and is useful in evaluating the other measures.

**Coverage.** This is the percentage of possible combinations of sensor inputs for which the program state generates actions; in other words, how likely a bot is to move in a given situation. If a bot moves in every situation, the coverage is 100%. Conversely, a bot that never moves has a coverage of zero. As there are eight sensors—left and right sensors for (a) the ball, (b) an opponent, (c) the robot’s goal, and (d) the opponent’s goal—there are $2^8$ (256) possible sensor combinations. If the soccer robot is not programmed to move when the ball is on its right, then half of the 256 possible combinations leave the robot inactive, yielding coverage of 50%. A limitation of this measure is that coverage is trivially perfect on some simple program states, for example, a novice programmer who has not learned to use sensors may have the bot always move forward, generating a program with 100% coverage. However, as programmers provide different actions for different sensor states, it becomes increasingly difficult to account for all possible inputs. Coverage, therefore, provides a measure of program completeness, though not quality.

**Program quality.** This measure indicates how likely a student’s bot is to win a game. To determine this, we loaded each program state from each student into a simulator that played 250 different games, each lasting 200 turns. In each game, the player and ball began in a set of randomly generated conditions; the metric was stable across 50,000 turns ($250 \times 200$) regardless of the specific starting conditions ($SD < 0.001$). For each game, we counted the total number of goals for and goals against the player’s bot. Each program state was assigned a score that was the average of goals for minus goals against over all games.

Given that these students were new to programming in IPRO, we chose this fairly simple measure of program quality. More practiced programmers might write more advanced programs that our metric would not rate accurately. For example, a program for a purely goalie robot would be more advanced but would receive a relatively low quality score, because, although it prevents goals from being scored, it never scores a goal. However, we saw very few (<0.1%) of those types of program states.

**TRACKER Selection, Clustering, and Tracking of Features**

Data mining is used to identify meaningful patterns within a data set so large or diffuse that standard analysis techniques are not efficient or effective. TRACKER is an EDM tool that we developed to analyze the log data using the features described in the previous section. It is designed to find the most common paths by which students learn to program. The TRACKER process consists of feature
selection, feature clustering, and feature tracking. This section defines, describes, and clarifies these terms and this process, which reflect a common set of practices from EDM (Baker & Yacef, 2009) and other data mining literature (Han, Kamber, & Pei, 2005).

*Feature selection* isolates particular features for inclusion in the analysis. Because our research questions address novices’ process of learning to program, we chose the five features of a program state described previously: number of action primitives, number of logic primitives, number of unique primitives, length, and coverage. These features were selected for the following reasons:

1. The features describe a program state in a way comprehensible to humans. In many data mining projects, features are meaningful only in the statistical sense. In contrast, these features can be described to anyone who can program in IPRO.
2. The features differ significantly across program states and across students.
3. The features are easily collectible first-order data. Each feature is a count.
4. We used measures similar or identical to the aggregate measures described previously.

*Feature clustering* groups program states into statistically generated categories based on the features of those states. Therefore, each program state is a five-dimensional data point, where each dimension corresponds to one of the program features. For this, we used the XMeans clustering algorithm (Pelleg & Moore, 2000). XMeans finds both best fit clusters and the maximally informational number of clusters; it found six clusters. To make the six clusters, the algorithm randomly picked six points near the outer bounds of the graph (centroids) using normalized features as coordinates. Each of our data points closer to Centroid 1 than any other cluster was assigned to Cluster 1; if it was closest to Centroid 2, the point was assigned to Cluster 2, and so on. After each assignment, the centroids were recalculated. When we repeated this process, the cluster assignments eventually converged around centroids that had the minimum average distance to their points. Our data set was small enough that we could closely approximate optimal centroids through brute force iteration.

**RESULTS**

**TRACKER Cluster Results**

After the clusters were determined by XMeans, we then characterized the clusters by picking out the most salient (or variant) relationship between features that appeared in that cluster:
1. The *active cluster* shows a very high ratio of total action primitives relative to length.
2. The *balanced cluster* shows a roughly even match of action primitives and logic primitives.
3. The *compact cluster* shows a high number of action and logic primitives, though the length is near the median.
4. The *logical cluster* is the most complex overall. It is long and shows high numbers of both logic and action, but the coverage is relatively low.
5. The *minimal cluster* describes programs that have very few primitives of any type.
6. The *testbed cluster* describes short, working programs.

The cluster averages are presented in Table 2. The values of each cluster feature measure in this table should be interpreted relative to the other values, as the number of primitives in IPRO bear little relationship to the number of primitives that same logical statement would require in any other programming language.

**TRACKER Path**

*Feature Track Model Discovery* refers to the process for using the results of the cluster analysis to determine whether there were likely and common sequences of a program state moved from one cluster to another over time (i.e., a track). To illustrate our process, we describe the discovery process for one possible track. Consider that every time a student started a program, she erased her entire program (at, say, State Number 96) and built a program with complex logic (by State Number 127); we would describe this track as \( \text{cluster}_{\text{minimal}} \rightarrow \text{cluster}_{\text{logical}} \). This signifies that her program code was nonexistent at Program State 97, but by Program State 127 she had added several logic primitives.\(^1\) We describe this

<table>
<thead>
<tr>
<th>Cluster Name</th>
<th>No. of Logic</th>
<th>No. of Action</th>
<th>No. Unique</th>
<th>Coverage</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>99.0</td>
<td>4.2</td>
</tr>
<tr>
<td>Balanced</td>
<td>2.5</td>
<td>1.9</td>
<td>2.3</td>
<td>100.0</td>
<td>7.2</td>
</tr>
<tr>
<td>Compact</td>
<td>5.5</td>
<td>2.9</td>
<td>4.1</td>
<td>99.8</td>
<td>14.6</td>
</tr>
<tr>
<td>Logical</td>
<td>6.3</td>
<td>3.9</td>
<td>4.7</td>
<td>59.3</td>
<td>17.2</td>
</tr>
<tr>
<td>Minimal</td>
<td>0.4</td>
<td>0.5</td>
<td>0.4</td>
<td>0.2</td>
<td>1.8</td>
</tr>
<tr>
<td>Testbed</td>
<td>0.9</td>
<td>1.0</td>
<td>0.9</td>
<td>98.7</td>
<td>3.5</td>
</tr>
</tbody>
</table>

\(^1\)Note that although empty program states are in \( \text{cluster}_{\text{minimal}} \), they only constitute a negligible percentage (<0.1%) of the data.
track in terms of transitions. In this example, we see the transition $\text{cluster}_{\text{minimal}} \rightarrow \text{cluster}_{\text{logical}}$. Next, say we find another student who followed the transition $\text{cluster}_{\text{minimal}} \rightarrow \text{cluster}_{\text{active}}$. That is, her program states were fairly trivial at Program State 22, but by Program State 41 she had added several action primitives, moving to the active cluster. At this point, our transition table would contain $\text{cluster}_{\text{minimal}} \rightarrow \text{cluster}_{\text{logical}}, 50\%$, and $\text{cluster}_{\text{minimal}} \rightarrow \text{cluster}_{\text{active}}, 50\%$, as it is equally likely that $\text{cluster}_{\text{minimal}}$ transitioned to $\text{cluster}_{\text{logical}}$ or $\text{cluster}_{\text{active}}$. Repeating this process across all transitions made by all students, we computed the most likely feature track model.

Figure 4’s TRACKER Path presents the most likely transitions between clusters aggregated over all students (as described previously). Only those paths with a greater than 10% probability of occurrence are presented. That is, a transition from active to balanced is shown on Figure 4 only if at least a tenth of the active program states transitioned to balanced. These percentages are shown to the right (or under) the relevant transitions. The paths in Figure 4 only show the transitions between clusters and do not include transitions that result in a program state staying in the same cluster.

Figure 4 can be read by starting at re/start and following the arrows. re/start is the state when there are no primitives on the IPRO board—all students begin there, and many wipe all of their code and start over several times. From re/start, most students transition to a minimal program state, and from minimal, most students transition to either balanced or active program states. This means that the most common changes students make to program states with few primitives in them is adding more action primitives or adding both more action and more logic primitives. From this point, the next transitions could be to program states in new clusters or back to program states in clusters already visited. For example, if a student’s program was in the active cluster, the next cluster might be balanced (if she adds logic primitives to her program) or it might be minimal again (if she removes primitives). Given that transitions may cycle, an arbitrarily large number of paths could be followed.

The EXTIRE Framework

Examining the TRACKER Path led us to propose the EXTIRE framework. Essentially, the framework aims to characterize types of transition activity in order to refine the theory of tinkering. The nomenclature was selected as it seemed to best represent aspects of tinkering identified in the literature as well as succinctly provide meaningful correlate descriptions for trends observed in cluster type transitions, cluster type frequency by time, and program quality over time.

Figure 5 shows how we classify different cluster type transitions. We characterized exploring activity as the area of actions in the TRACKER Path in which students are most frequently transitioning back and forth from re/start to minimal
and *active* program states. When students are *exploring*, they are working on learning global goals for IPRO activity. We characterized *tinkering* activity as the area in the TRACKER Path in which students’ program states most frequently transition among *active, balanced, testbed, and compact* program states. This change represents a shift from attempting to determine global goals for IPRO activity to targeting local goals while establishing building blocks for programs. We characterized *refining* activity as the area in the TRACKER Path in which students’ program states transition back and forth primarily between *balanced* and *logical* program states. This activity represents attempting to achieve global goals using the basic building blocks from tinkering activity and includes debugging to fix specific errors in an existing program state.

Although a student’s program edits may transition frequently, this does not indicate that the student is frequently transitioning phases. Students can transition hundreds of times over the course of an hour, and each one does not represent a perspectival shift. Any single program edit may yield any transition type, but we use the term *phase* to represent an aggregation of many transition types for that one student’s activities. Similarly, by aggregating all students’ program edits, we can understand the group’s progression of EXTIRE phases over time.

*Aggregating EXTIRE Phases Over Time.* An important next goal was to determine how markers of EXTIRE phases (*explore, tinker, and refine*) changed over time at the class level. Figure 6 presents a graph of the frequency of each program state cluster—*logical, compact, testbed, balanced, active,* and *minimal*—for each minute of the IPRO session. Figure 6 begins at Minute 10 when students began programming. We characterized the next 17 min as the *Explore Period* because that is when the *minimal* cluster completes its steady drop in frequency and stabilizes; in the EXTIRE framework, *exploration* is the only phase in which the *minimal* cluster plays a substantive role. The time period boundaries are set at those points yielding the most different cluster characteristics.

In the first identifiable time period the majority of program edits are minimal or active, with increasingly more balanced edits. This highlights that the majority of program edits are transitioning most frequently (58.5%, as per Table 3) among minimal and active clustered program states. Note that this decreases precipitously to 18.7% of transitions in the second period and 14.3% in the third. This disaggregation proved useful in determining additional characteristics of the varied periods.

We characterized the next 25 min as the *Tinker Period* because that period shows the most even distribution of all cluster types but is also marked by a clear increase in both *balanced* to *logical* program states. In this Tinker Period, the number of transitions among balanced, active, compact, and testbed clusters goes from 15.4% to 31.0% and decreases back to 18.1% in Period 3. These are cluster
transitions characteristic of tinkering activity as presented in the TRACKER Path (see Table 3 and Figure 5).

Finally, we characterized the last 13 min as the *Refine Period* because it shows a higher proportion of *logical* program states at a steady rate than the other two phases. This is characterized by developing *logical* programs, which is a primary feature of *refining* activity. Here, the transitions among logical and balanced clusters increase to 38.5% of all transitions (from 27.7% in Period 2 and 6.5% in Period 1).
TABLE 4
Percentage of Program Edits With Low, Average, or High Program Quality Scores Aggregated by Period With Means and Standard Deviations of Each Student’s Best Quality

<table>
<thead>
<tr>
<th>Period</th>
<th>Quality of All Program Edits (%)</th>
<th>Student’s Maximum Quality Program Edit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Average</td>
</tr>
<tr>
<td>Explore</td>
<td>31</td>
<td>59</td>
</tr>
<tr>
<td>Tinker</td>
<td>33</td>
<td>39</td>
</tr>
<tr>
<td>Refine</td>
<td>27</td>
<td>45</td>
</tr>
</tbody>
</table>

**EXTIRE Phases and Program Quality.** With this evidence suggesting that the character of students’ programming activity changed over time, we examine how these changes related to the quality of the program states produced by students in each period.

Table 4 shows how trends in program quality changed over time. Indeed, the number of program quality scores that were low, average, and high differed significantly for each period, $\chi^2(4, N = 53) = 441.35, p < .05$ quality programs per period became significantly better over time ($F = 5.639, p < .01$, partial $\eta^2 = .163$). Students in the Explore Period produced more average or lower quality programs than in the other two periods. The other two periods also showed many low-quality program states, but the number of average and higher quality program states increased.

**DISCUSSION**

Students learned to program in IPRO by working through different phases of activity in sequence. Broadly speaking, first they explored the space of programs, then they tinkered with those programs, and finally they refined their programs. Characterizing their activity at only that level of granularity does a disservice to the variety of approaches and the potential value of enabling those different approaches. Thus, we found the following.

**Research Question 1: How, in the Aggregate, Does Students’ Programming Activity Change Over Time?**

Students learned to program in IPRO: Their programs became more complex, more functional, and of higher quality over time (as suggested by the data reported in Table 4 and Figure 6, and as reported in detail in Martin et al., 2013).
we found here, however, is that the path was circuitous. There is not a clear linear path from worse to better, and the paths taken by different students were different. There was an “epistemological pluralism” (Turkle & Papert, 1990) to their approaches; the students generally wandered through a few relatively similar patterns of activity given the huge space of possible approaches and possible programs. All students had what we characterize as three phases to their learning: exploration, tinkering, and refinement. This EXTIRE framework has the potential to inform how we think about students learning to program by situating their programming activity more broadly. Some instructors of computer programming embrace tinkering, some ignore it, and some try to suppress it; our findings suggest that instructors should more clearly articulate descriptions of tinkering and situate it in the context of exploration and refinement.

Research Question 2: What Does This Activity Reveal About Tinkering Processes?

Tinkering can be characterized by looking at the features of students’ programming work. Students who were tinkering restarted infrequently, used trial and error to successfully modify their programs, and accreted programs together. That is, as shown in Figure 5, there were meaningful sequences and types of activity by which students wrote code. The transitions identified in Figure 5 were a surprise to us: They suggest that beginners can identify local maxima with tinkering but restart if necessary. This may seem obvious post hoc, but there is not prior literature that has suggested it. In part, it was due to the design of IPRO. The rapid feedback of the IPRO environment enabled students to evaluate their programs extremely frequently, and students fussed and experimented frequently. That said, the tenor of student activity changed after some length of tinkering—tinkering was not the final phase of programming even in this short activity. As students better understood the code that they were writing, they began to refine their programs.

Research Question 3: How Do These Changes Relate to the Quality of the Programs Students Are Writing?

Students’ best programs got much better over time, but students were, not surprisingly, still not expert programmers after 90 min of work. The average program quality of student programs grew significantly but modestly. Learning how to program is a much longer process: IPRO is no magic bullet. Recent literature on learning progressions suggests that it can take years, with several intermediate progressions, to develop content area expertise (Shea & Duncan, 2013). That said, our other work with IPRO (e.g., Martin et al., 2013) consistently shows that each student’s best program got better, used more complex logic, and covered more
possibilities as she learned to program. Perhaps more exciting is that our findings suggest that we were able to quantify how students learn with tinkering.

CONCLUSIONS

Though this was only a short study, it provides an opportunity to look closely at the specific processes of how students learn to program. This article does not provide a generalizable model of how all students learn to program but rather explores the ways in which pathways of exploration, tinkering, and refining shape how students learn to program. Furthermore, it provides one map for using EDM and learning analytics for careful process analysis typically only found in more qualitative literature.

EXTIRE and Learning Theory

Depending on the granularity of inspection, the EXTIRE hypothesis is either reasonably novel or further evidence for current understandings of how novices learn to program. For instance, Turkle and Papert (1990) suggested that tinkering is a core method of learning how to program, and female learners often learn to program in a bricoleur style because it matches their existing understandings of the design process. In other words, programming can be a valuable learning tool precisely because it enables tinkering. From a different perspective, Dreyfus and Dreyfus’s (1980) model of skill acquisition also suggests that novice learners “expand the space” so as to gather the bounds of the skill or object to be understood.

In prior work, Berland (2008) and Perkins et al. (1986) suggested that novice programmers tinker primarily to learn. However, such works were largely qualitative, and the findings were mostly exploratory. By comparison, this study provides substantive quantitative data that seem to reflect the same. Moreover, where Soloway and Iyengar (1986) and Klahr and Carver (1988) suggested that debugging is another primary mode, we can further specify the debugging process into “refinement” and provide additional empirical insight into its nature. For example, our further preliminary work suggests that the largest jumps in program quality were at points of removing logic rather than adding it. Though this might contradict intuitions that many nonprogrammers may have about programming, it supports Raymond’s (1999) claim that programmers debug 10 lines of program code for each line they write. It is interesting to see that that process holds up and is learned in the first hour of programming.

As the students in this study were all female, there is a potential for gender bias in our findings. However, the studies cited here were significantly skewed toward male rather than female participants, as is computer science education in
general. By contrast, in our previous work (Martin et al., 2013) we found no significant gender differences in programming achievement in the same programming environment.

Our findings add to the literature in two ways:

1. By illustrating the importance of tinkering, refining its definition, and providing substantive empirical support for previously theorized processes.
2. By demonstrating how programmers’ first programming activity and concomitant initial construction of programming knowledge can be delineated into three steps: explore, tinker, and refine.

Learning Analytics and Constructionist Learning Environments

Although there is a variety of literature on learning trajectories and on learning analytics, the majority of that research has not been conducted with open-ended or constructionist learning environments. For both practical and theoretical reasons (described by Baker & Yacef, 2009; Koedinger, Cunningham, Skogsholm, & Leber, 2008), the vast majority of the learning analytics literature addresses more constrained learning environments. Practically speaking, the more constrained a data set is, the easier it is to find interpretable and meaningful patterns. The extreme example of this would be a simple two-dimensional data set in which traditional summative statistics can describe specifically why the data do or do not support a very specific hypothesis. As the data set gets more open ended, patterns of learning become increasingly diffuse. For example, it would be difficult to make mechanistic predictions about writing patterns from a set of only 1,000 student essays, as those essays will generally have little in common, whereas if students only have a set of, say, three choices at any point, such as with many intelligent tutors (Koedinger et al., 2008), commonalities will be much easier to identify. Our data suggest that with the appropriate praxis, it will be possible to determine learning pathways in less constrained data sets generated from complex learning environments at a fine granularity and across many students by using methods from EDM.

Because our findings about tinkering build on a substantively larger data set than previously realistic for many researchers and generally support prior theory and research, we believe that this work provides meaningful insight into how people commonly learn to program. However, IPRO is a mobile, social programming environment; it is impossible to have syntax errors in IPRO; and IPRO is designed to be used in more informal settings. Further studies in more traditional settings with more standard programming languages are necessary in order to examine the EXTIRE framework more generally. Also, additional studies with similarly novel programming environments may refine understandings of the EXTIRE framework.
ACKNOWLEDGMENT

Thanks to Chadwick Wood for his work on IPRO, to Gilbert Slade for his help on TRACKER, to Pat Ko for his help on the camp, and to Jay Pfaffman for guidance. Thanks to the Active Learning Lab and the Complex Play Lab for helping refine countless versions of this work. This work was supported by National Science Foundation Grants EEC-1025243 and EEC-0748186. The opinions expressed in this article are ours and do not necessarily represent those of the National Science Foundation.

REFERENCES


APPENDIX A: PROGRAMMING IN IPRO

Program State 1
This is an empty program (named “states”) owned by matthewb.

Program State 1
This is the first menu (the action menu) from which the user (matthew) can add primitives, but no primitives have been added yet.

Program State 2
Matthew has added an IF primitive to the program. By adding a primitive, we have moved from program state 1 to Program state 2.
Matthew has double-tapped on the IF primitive. In this window, the user can create logical statements. This is the **logic window**.

Matthew has double-tapped on the logic window to open the **logic menu**.

Matthew tapped the LR sensor. The sensor reports whether there are enemy robots that can be seen by my front-left sensor. By adding a primitive, he has moved into Program state 3.
Program State 3
By tapping return on the last window, Matthew can now look at the program as a whole. At this point, the program is as follows: “If there is an enemy robot to the front-left of me, then do nothing.”

Program State 3
Matthew double-taps on the screen to bring up the action menu again.

Program State 4
Matthew has now completed a working program by adding the FR action primitive (also moving him into state 4). Now the program is: “If there is an enemy robot to the front-left of me, move forward left, otherwise stay put.”
### APPENDIX B: IPRO KEY

<table>
<thead>
<tr>
<th>ICON</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>![SPECIAL]</td>
<td>The IF block contains logic (double-click to access) and can lead to either actions or other IF blocks.</td>
</tr>
<tr>
<td>![Move front left]</td>
<td>Move front left</td>
</tr>
<tr>
<td>![Move front right]</td>
<td>Move front right</td>
</tr>
<tr>
<td>![Move back left]</td>
<td>Move back left</td>
</tr>
<tr>
<td>![Move back right]</td>
<td>Move back right</td>
</tr>
<tr>
<td>![Turn left]</td>
<td>Turn left</td>
</tr>
<tr>
<td>![Turn right]</td>
<td>Turn right</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ICON</th>
<th>LOGIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>![EQUALS]</td>
<td>EQUALS is true if the two operands are equal. Otherwise, it is false.</td>
</tr>
<tr>
<td>![GREATER THAN]</td>
<td>GREATER THAN is true if the left operand is greater than the right operand. Otherwise, it is false.</td>
</tr>
<tr>
<td>![NOT]</td>
<td>NOT is true if its operand is false. Otherwise, it is false.</td>
</tr>
<tr>
<td>![OR]</td>
<td>OR is true if at least one of its two operands is true. Otherwise, it is false.</td>
</tr>
<tr>
<td>ICON</td>
<td>SENSOR</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td><img src="image1" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image2" alt="Icon" /></td>
<td>Ball is front left</td>
</tr>
<tr>
<td><img src="image3" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image4" alt="Icon" /></td>
<td>Ball is front right</td>
</tr>
<tr>
<td><img src="image5" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image6" alt="Icon" /></td>
<td>Opponent is front right</td>
</tr>
<tr>
<td><img src="image7" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image8" alt="Icon" /></td>
<td>Opponent is front left</td>
</tr>
<tr>
<td><img src="image9" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image10" alt="Icon" /></td>
<td>Target goal is front left</td>
</tr>
<tr>
<td><img src="image11" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image12" alt="Icon" /></td>
<td>Target goal is front right</td>
</tr>
<tr>
<td><img src="image13" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image14" alt="Icon" /></td>
<td>Opponent’s target goal is front left</td>
</tr>
<tr>
<td><img src="image15" alt="Icon" /></td>
<td>SENSOR</td>
</tr>
<tr>
<td><img src="image16" alt="Icon" /></td>
<td>Opponent’s target goal is front right</td>
</tr>
</tbody>
</table>