

# ION: A Universal Graphical Input/Output System Designer

Shawn Sullivan  
MIT Media Lab  
20 Ames Street  
Cambridge, MA, USA, 02139  
1-205-401-4243  
[stsully@mit.edu](mailto:stsully@mit.edu)

Ted Selker  
Associate Professor, MIT Media Lab  
20 Ames Street  
Cambridge, MA, USA, 02139  
617-253-6968  
[selker@media.mit.edu](mailto:selker@media.mit.edu)

## ABSTRACT

In this paper, we describe a new software design program, called the Input/Output Nexus (ION) which allows users to graphically create, control, and observe most types of systems which receive some sort of input and generate output. ION uses an architecture which has been shown to be very effective in processing input in previous research as the basis of the user-configurable system. Within this very flexible architecture, users are free to create a wide variety of systems to their specifications using an intelligent interface which builds systems according to the details inputted by a user in the simple graphical interface.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *computer-aided software design, modules and interfaces, object-oriented design methods, user interfaces.*

## General Terms

Design.

## Keywords

Architecture, attention score, automatic code generation, design pattern, input/output system, numeric data, non-numeric data.

## 1. INTRODUCTION

Many computer systems are charged with monitoring data input, evaluating that data, and generating various forms of output based on that input. In most instances, substantially different or even entirely new computer systems must be built to conform to each individual case; although these systems behave similarly with a common purpose, they are individually developed due to small differences in input handling and output generation. We therefore

introduce a development environment for creating such systems called the Input/Output Nexus (ION). ION provides a graphical environment in which users may create, develop, and manage input/output systems of various kinds without writing any code. The ION interface allows users to easily construct, modify, and observe their systems by automatically creating their system based on simple interactions and a small number of user-inputted system characteristics. ION's intelligent interface permits rapid, simple creation of many types of input/output systems through an intuitive GUI. ION facilitates quick, easy, effective system design, requiring little or no surrender of power by the system designer due to automation. With ION, input/output systems can be built not in weeks, but hours.

## 2. MOTIVATION

After developing many systems in the Context Aware Computing Group at the MIT Media Lab, we began to notice much of our development time had been devoted to developing very functionally similar systems individually. Typically, these systems contain some sensors or other forms of input, a central device which receives and interprets data from the sensors, and a few output devices controlled by the central device. These basic three sections of the system may be software or hardware, simple or complex, but ultimately with many of the systems we were building were just systems to observe something and react to it. Realizing this common functionality, we can build a system which abstracts this functionality and allows the developer to only concern himself with how input is received and used, and what sort of output to generate, rather than having to build an entire system from scratch.

A previous research project, CarCoach, provided the basis for the overall design of ION [3]. CarCoach monitored several sensors in a car in real time as the car was driven, and provided feedback to the driver based on the quality of their driving. Here we see the three basic divisions described previously: sensors, some central monitor, and output. CarCoach was successful in efficiently communicating and interacting with several external devices in real time; however, development of CarCoach took several months due to the complexity and sheer magnitude of work required to get it operational. The system worked well, but took a substantial amount of time to build. We wanted a way to make other systems similar to CarCoach as effective as CarCoach was, but in less time.

Analysis of CarCoach's success was reveals it stems from its effective five-step process for handling input and generating output. These five steps represent the basic actions which must be taken by a system to effectively process multiple inputs to output and are

- 1) Receive input data
- 2) Normalize input data
- 3) Compare input data
- 4) Determine overall state implied by input data
- 5) Generate appropriate output

By creating one program which encapsulated the functionality of these steps, we realized, we could create systems similar to CarCoach in basic functionality but drastically different in many other ways quickly and easily by configuring a few aspects of the system, such as where to look for input and how to normalize it. Further, configuring these aspects could be done without writing any code, as long as a few basic functionalities, such as communication protocols, were included in ION. We wanted ION to serve as a fill-in-the-blank input/output system generator, and using these five steps, we saw that we could make ION just that.

### 3. DESIGN

#### Design Overview

ION has two major divisions, each with a graphical interface for user interaction and information display. These divisions are the design module, which allows system designers to build and configure systems, and the execution module, which runs the systems designed in the design module. All of ION is built around encapsulating the five steps stated previously in Section 2 and allowing users to simply interact and observe the components of the system associated with each of the steps, to be explained shortly. Before explaining the design of these two modules, however, an overview of the basic abstractions made by ION is useful.

#### ION Abstractions

To encapsulate the five steps listed in Section 2, ION abstracts each of them into its own representation. These names of these representations and what they represent are

- 1) Source – an input device and its connection to the system
- 2) Analyzer – a data normalizer uniquely associated with a particular Source
- 3) Monitor – a central collector of normalized data from Analyzers and overall decision maker
- 4) State – a representation of the overall decision made by the Monitor based on the input
- 5) Response – a single form of output associated with a State

Data flows from input to output in numerical order as presented here. Each of these five abstractions plays an important role in defining the system design.

#### *The Source Abstraction*

The Source abstraction represents one input device as well as how it is connected to the rest of the system. The primary function of the Source in the system is to specify how to communicate with an input device, not to actually configure the input device itself. Currently, TCP [6], UDP [7], IP [5], and File I/O are supported methods of communication. Specifying the communication details of each Source is simple: the system designer simply specifies the port, IP, and/or file location as appropriate. The designer is not required to actually code any of the communication herself; ION automatically knows how to correctly communicate once provided with the location of the input device's interface to the rest of the system. Thus, for all communication protocols ION understands, no code is required to connect a device to a system created in ION using a known communication protocol. Further, adding additional communication protocols is simple. Once the protocol is coded once in ION's source code, it can be used as often as the user likes simply by entering the pertinent communication information, such as port numbers or other addresses associated with communication to a particular input. Sources are also capable of generating random input within configurable bounds for testing or other purposes.

Presently, Sources are only capable of receiving numeric input; that is, input whose information is wholly represented by the digital number it is, not some additional interpretation of the binary data such as ASCII. This functionality allows for a large majority of potential inputs to be received through the Source because all data, even from non-numeric devices, are represented numerically (digitally) in a computer. Thus, all numeric input and input easily translated to a numeric value, such as whether a switch is "up" or "down" or what color something is, can be handled by a Source. Other data, however, such as text input from a file, cannot be easily understood numerically, though addition of an input format which allows a user to configure a translation for complex non-numeric data into something numeric and therefore usable by a Source could allow these inputs to be handled as well. Throughout the rest of the paper, "numeric input" shall refer to input whose significance is the digital input itself, not a further interpretation of that number, while "non-numeric input" shall refer to input whose significance lies in an interpretation of the binary input, such as ASCII.

#### *The Analyzer Abstraction*

Sources specify how to communicate with input devices, but they do not prepare their data for use by the rest of ION. Because inputs can theoretically have any value, and input devices may generate data whose ranges are unrelated to each other, the data from them must be normalized to enable accurate and simple comparison. The details of normalization are left to the designer, as it should be, since the designer, not ION, is designing the system. ION enables the designer to configure how data from each input is normalized through the Analyzer abstraction. Each Source is uniquely associated with an Analyzer, which specifies how data from that source will be normalized. Data is normalized with a mathematical function, which must be of the form  $y=f(x)$ , where  $x$  is the data from the Source, and  $y$  is the calculated normalized data. The system designer may choose a function from a list of common functions, linear, exponential, logarithmic, and sigmoid, each fully scalable and shiftable, or enter an arbitrary function of  $x$ . The sigmoid function  $1/(1+e^{-x})$  may seem unusual to include

with the other three much more common functions as standard, but previous work, especially on CarCoach, has shown that it is excellent when shifted and scaled at normalizing data which is significant above or below (if flipped over the  $x$ -axis) a threshold and insignificant otherwise.

A special method of normalizing data found to be an effective capability of ION results in normalized data which can be described as an attention score. While it is perfectly acceptable and effective create systems which perform basic normalization of data, simpler and perhaps more powerful systems may be created by normalizing data such that it is an attention score. Normalized data may be treated as an attention score for the input when all input data are normalized to fall strictly and completely within the same range over expected possible raw input values and when greater values of the normalized data directly indicate more significance. For example, if all expected raw input is normalized to fall within the range of 0 to 100, and a normalized value of any input greater than that of any other input indicates that the first input is more important in the system, then the normalized inputs may be treated as an attention score. In this case, all inputs can be compared directly by comparing their attention scores, simplifying the overall system as well as analysis of system performance. It is important to note that designers who wish to create attention scores must ensure that they normalize data appropriately, as not all inputs may simply correspond linearly to an attention score. For example, if one were to create a system monitoring a combustible to ensure it does not catch fire, very little attention may need to be given to the temperature up to, say, ninety percent of the combustion temperature, but above that threshold, a great amount of attention may be required, suggesting a nonlinear function such as an exponential function may be appropriate for generating the attention score.

It is also important to note that the attention scores only represent one possible design pattern for using ION, not a built-in feature or imposed design. There are many other, perhaps more useful, methods of designing systems in ION. Attention scores are mentioned in particular because past experience has shown them to be both useful and simplifying to the system.

### *The Monitor Abstraction*

The Monitor abstraction is the core of ION. It collects the normalized data from the Analyzers and makes decisions and conclusions about what the data means. The monitor queries each Analyzer in the system and obtains normalized data, never directly interacting with a Source to ensure all data has been properly normalized. The frequency of the queries to Analyzers can be changed from the default value of 2 times per second by designers using the graphical interface, and for most systems should be able to go at least 10 times per second. Once the Monitor collects the data from the Analyzers, it begins using the data to determine what it means. The Monitor can operate in two modes, rule-based and support vector machine [8].

Rule-based mode uses a very simple forward-chaining rule-based system<sup>1</sup> to perform checks on the data, especially good for small,

<sup>1</sup> The rule-based system used is extremely simple, with the only possible predicates consisting of mathematical comparisons of normalized input data to other normalized data or constants. As such debate of differing types of rule based systems is irrelevant, as it is not possible to have rules depend on each

uncomplicated systems. When a rule is fired, it activates a State, to be described shortly, which essentially means the system has found a classification for the data. Rules are fired with a first applicable conflict resolution strategy, though unlike usual rule based systems, all rules which are satisfied are fired, not just one of them. This fact means that multiple States can be active simultaneously, which may or may not be a good thing, depending on whether or not the system was designed that way. The onus is on the system designer to properly create an effective rule-based system in ION.

For larger, more complex systems, manually configuring rules for different scenarios is impractical if not impossible. When a rule-based system is not desired, the Monitor can also use a support vector machine to make decisions. The support vector machine learns what sort of data corresponds to what States because the system designer provides it information with which it can make these correspondences. Essentially, the support vector machine automatically generates rules, but to do this it must be trained. The system designer must provide a data file to ION including what States correspond to the data to train the support vector machine. It is highly advisable to make training the support vector machine the last action in designing the system prior to testing, because any design changes require re-training the support vector machine, which may mean obtaining new data. Currently, system designers must provide their own data obtained independently from ION and tell ION to what State the segments of the data correspond; however, adding a data collection feature the can be turned on to train ION live is certainly possible.

Either data classification method ultimately produces a (generally) small set of States which correspond to the normalized data as defined by either the rules or support vector machine, as appropriate.

### *The State Abstraction*

The State abstraction represents a classification of the input by the Monitor. As designed, States represent the overall environment as determined by the Monitor. For example, a weather monitor system may have states like Cloudy, Clear, Rain, Windy, and Thunderstorm. Note that here, as just mentioned, depending on the design, it is possible for some of these states to occur together. It is possible that Rain and Thunderstorm may be active together, or Thunderstorm may include Rain as part of its description or properties. Again, the system designer controls the semantics.

States can also represent more or less than a general situation based on the data, once again depending on how the designer uses them. They may represent nothing more than a group of associated responses to particular data, or they may themselves be the important conclusion, rather than generated output, if the system is used only for data classification. Another important note about States is that unless some form of output is always wanted every time data is sampled, which can lead to significant lag if the duration of output is greater than the input sampling rate, a "normal" State with no associated Responses (output) is recommended, rather than simply not having a State for "normal" data ranges for robustness.

---

other.

## The Response Abstraction

The Response abstraction represents a single piece of output generated by the system. Each State may be associated with one or more Responses, thus generating a corresponding number of outputs. When a State is activated, it triggers all of its associated Responses sequentially. Responses are very similar to Sources in that they consist mainly of specifying parameters of a communication protocol. Responses may currently be made by sending messages through TCP, UDP, or IP, writing to a file, printing text to the screen, and playing sound files. Like the Source class, adding Response types requires the one-time writing of a piece of code, and may then be used without writing any code. However, unlike Sources, non-numeric output has some support built in, since arbitrary text can be written to files or printed to screen, and messages sent via TCP, UDP, or IP can be specified, in the content of the Response by specifying text as output.

## The Design Module

The design module is the portion of ION which allows system designers to create and configure the various aspects of the system they are designing. It consists of a graphical interface, used to allow designers to specify the components of their system, and a back end, responsible for creating an XML [1] representation of the system as designed by the user.

### Graphical Interface

In order to design systems with ION, the designer must be provided with an interface. We chose a graphical interface because of its simplicity of use and clarity of content. The interface shows the designer the system as a whole as well as the details of one of the abstractions in the system. Figure 1 shows a sample screenshot of the design module, with the system on the left and the characteristics of one of the Analyzers in the system on the right.

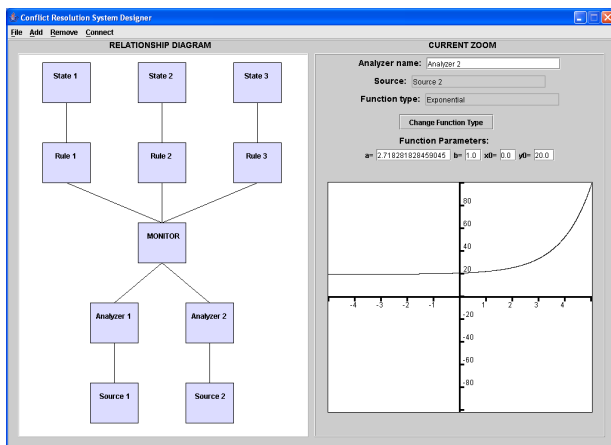


Figure 1. Sample Screenshot from Design Module

As shown in the figure, the left half, which scales with size, clearly shows the topography of the system. Sources at the bottom represent input, and data flows upward through Analyzers to the Monitor at the center, which here uses Rules to determine the States, pictured at the top of the screen. The right half displays the associated details of one component of the system on the left. This half of the display allows users to configure the individual

components of the system, as well as helping the user further understand the details of the system they are creating. The interface provides overall control over the actions of the back end, which creates the system based on the user-inputted data in this interface.

### Back End

Using the basic abstractions described in Section 3.2, system designers can create systems in ION's design module through the graphical interface, which in turn controls the back end of the module. The back end of the design module is relatively simple, consisting mainly of just creating the system the user is designing. When the user creates the system using the graphical interface, the design module maintains a copy of that system in state. Each of the abstractions added to the system by the designer is modeled as an object in memory. As the abstractions are modified in the graphical interface, so too are the objects representing them in memory by the back end modified. Thus, by interacting with the graphical interface, users can manipulate the structure of the system they are designing without the need to write any code, because the interface is capable of constructing the system with only a little direction about the specifics of the system from the user. To access the design later, the user must save the design for later use by the execution module. When the designer saves the design, the representation of the system stored in memory is translated into an XML representation of the system able to be loaded and understood by ION for later use.

## The Execution Module

Once the user has saved the system design into XML, ION's execution model can run the system. The execution module is responsible for actually causing the various parts of the system to interact with each other as described by the system design.

### Graphical Interface

Though the execution module has a graphical interface, this interface is fairly unnecessary once system development and testing is complete. The graphical interface simply has three buttons, "Step", "Run", and "Stop", and a space to display text. The "Step" button causes the Monitor to query each input device only once, obtain the appropriate States, execute the associated Responses, and then stop. The "Run" button causes the Monitor to repeatedly query input devices, to classify the States, and to execute responses until the "Stop" button is pressed. After "Stop" or "Step" is pressed, various data associated with the last query are printed, including the raw input data, normalized input data, and the resulting States corresponding to the normalized data, for diagnostic purposes. In practice, the graphical interface is not needed once the system is complete, and so a version of the execution module can be run which does not have a graphical interface and instead immediately starts the Monitor to collecting and classifying data repeatedly until exited.

### Back End

The execution module's back end is also fairly simple. It begins by loading the XML file representing the design of the system and creating a system based on the contents of that file. This file is the file created and saved by the user in the design module, and fully represents the system they have designed. The execution model builds a copy of the system as objects in memory according to the XML file, and the system can then be run. The execution module

continually loops a procedure which causes the Monitor to collect and interpret data at the rate prescribed by the design until either stopped if it is running with the graphical interface, or when closed if running without the graphical interface.

#### **4.DISCUSSION**

ION has many important features and implications of not only the design of general input/output systems, but also on the engineering process required to create such systems. Because system designers have to write no code, or perhaps very little code to add input/output communication protocols not already present, they can create these systems in far less time and with fewer programming and software engineering skills. The interface is capable of using user input to create systems specified by relatively scant human interaction, instead automatically creating the framework of the system and providing a method for the user to specify the details of the system. This system provides many benefits to the user, but also has a few limitations, due mainly not to restrictiveness but to the need for further research.

#### **Benefits of ION**

ION's most obvious benefit is the elimination of the need for system designers to code systems from scratch. Most forms of conflict resolution systems, resource allocation systems, state monitoring systems, and many other more general types of systems which observe and respond to sensors or other information sources can be created with ION. Such systems can be designed, implemented, and tested in a fraction of the time required to build these systems from the ground up. Rather than coding each portion of the system individually, a system designer only needs to specify where input comes from, how to compare them, and what forms of output to generate.

Another benefit of ION is that it requires less software engineering skill from system designers. Though many professional software engineers are perfectly capable of designing their own systems which work as well or perhaps better than those created with ION, many people who do not have the training or experience to design an effective system can use ION to create systems without having to commit time and money to learning how to design software from the ground up, or outsourcing the system design to professional software engineers. In fact, for novice software engineers, ION can provide an example of a proven software engineering design, the details of which the apprentice designer can manipulate and observe without having to deal with figuring out code and understanding a moderate- to large-size system. For more experienced software engineers, ION provides a system well-suited to quickly creating routine input/output systems, as well as a program effective for rapidly prototyping the design of larger, more complicated systems that may be being designed from scratch.

ION also simplifies testing and debugging of systems. Because ION allows users to create systems without writing code, they can focus on testing the parameters and non-software design of the system they specify, rather than spending time debugging low-level components of the system, such as communication protocols and basic software and class structure. This feature of ION accounts for a great savings of time and money in addition to that conferred by the speed of initial creation and development of the system.

System designers can also quickly modify and extend systems created with ION. System components can be added or removed with a few clicks and parameter configuration. Changing communication protocols to external components can be accomplished simply by telling ION what new protocol and the pertinent details to use, rather than requiring new code to be written. ION also allows the input sampling rate and the method of interpreting input to be set or changed quickly without coding. For example, if a system grows too complex for its original rule-based decision method, ION facilitates a simple transition to a support vector machine, requiring only that ION be told to use a support vector machine and given some training data. With ION, system engineers other than the original designers, as well as nontechnical project members, can see and understand the structure of the system almost instantly through the graphical interface. They can even modify or change the system easily, without having to dig in to the code of the system in order to make simple changes. Systems created with ION, in short, can be readily understood and manipulated swiftly, even by those who have no prior knowledge of the system or even how to write code. ION manages code for the system, automatically constructing systems based on small number of simple user instructions.

Other types of systems not mentioned in this paper may be abstracted and automated using ION. Isolating and abstracting core components of other types of systems, such as server/client systems and database management systems, and relating them to the abstractions made by ION can allow these other kinds of systems to be created with ION. For example, a server/client system could be created with ION by specifying the communication protocol to be used, probably TCP, UDP, or IP, for input and output, and creating a rule-based, or perhaps support vector machine in an appropriate scenario, Monitor which responds according to what input is received. A database management system might be implemented by specifying input and output to use various database files and some form of inter-program communication protocol, along with a Monitor using a rule-based system to interact with the database based on the input. Each of these systems use ION in a manner which meshes well with it, but in a fashion ION was not originally intended to be used. ION's flexibility allows it to have many different uses beyond just the originally intended ones. These additional uses are not dubious applications ION to systems it cannot truly represent; rather, they take advantage of the great flexibility of ION's basic function, to receive some form of input and respond to it. By using ION's built in functionalities, as well as adding communication protocols when necessary, system designers can use ION to create most systems which receive some form of input and generate some form of response based on that input. Adding support for non-numeric input, either through providing a method for translation of non-numeric input to numeric input or adding support for non-numeric input to be handled directly, would greatly expand the capabilities and applicability of ION to nearly every system which receives and deals with input. Ingenuity of the system designer can then be applied to creative use of ION and efficient implementation of their system, rather than trying to code systems from the ground up.

Despite these significant benefits deriving from ION's automatic system creation based on a small amount of user input, ION is not restrictive. As long as the appropriate external communication protocols are either already implemented in ION, or can be added

by a software engineer in the one-time addition of code to ION required to add a communication protocol, ION can handle most inputs, and the addition of a method of handling non-numeric input allows virtually all inputs to be supported when appropriate communication and translation protocols are in place. The two methods of interpreting input, rule-based and support vector machine, also are capable of supporting most systems. Users can configure rule-based systems by hand for simple decision making, or can use the support vector machine to handle more complex systems. In the worst case, a system designer may have to create an extensive list of many potentially complicated rules to set up a complex rule-based system when training data is unavailable or not useful for using a support vector machine. However, this inconvenience is certainly offset by the time saved in coding the entire system from scratch. Indeed, it is probable that the user would have to specify the details of the complex rule-based system anyway, only hard coding the rules rather than using the ION design module. ION, therefore, does not impose significant restrictions on input and output forms, or decision making processes for most systems. Some may complain that simply by imposing a uniform overall design of the five previously described abstractions, ION is placing a great restriction on overall design. This is only true insofar as it forces the system designer to specify how input is to be received, how inputs are to be used, and how to generate output or other responses based on the inputs. The system designer must do these things even if ION is not used. While a designer can specify these basic characteristics any way they wish if creating a system by writing code from the ground up, the basic handling of data usage by ION is a logical, simple, flexible one, and one which previous software development has shown to be quite capable of creating a variety of systems.

### **Limitations and Further Work**

ION does, however, have some limitations, some of which have been described earlier. Non-numeric input is currently not supported. However, methods for incorporating non-numeric input have been described. The simplest method would be to incorporate some form of translating non-numeric input into numeric input in some form of an enumeration scheme. This method allows some basic non-numeric input, such as the translation of words or small phrases which compose the whole of the input, but more complex non-numeric input requires further refinement. To incorporate all or a large majority of non-numeric input, a portion of ION must be made able to directly interpret non-numeric data, perhaps through natural language processing or pattern matching. However, ION in its present form allows a very large number of systems to be created, as a great number of software data and most hardware data is purely numeric, and relatively few systems are likely to need direct interpretation of non-numeric data, the notable exceptions being text searching, understanding, and other similar text processing systems.

ION also currently allows only a single Monitor, the central data analyzer, to be present in the system. Situations where multiple Monitors may be beneficial can certainly be conceived. However, in most cases, one can achieve the effect of multiple monitors by layering systems created with ION, using the output of some ION systems as input to others. In fact, the power of layering ION systems provides further interesting research, as does the possible allowing of multiple Monitors in a single system.

Another very interesting avenue for future research is the inclusion of the Open Mind knowledge base [4] or other similar systems which serve to increase the general knowledge and usefulness of computer systems by providing information to computer systems. For example, the Open Mind system can be used in conjunction with two ION systems. The first system could receive various inputs and generate an output, and the second system could use the Open Mind system as an input, along with the output of the first system, to ensure that the output of the first system is consistent with its knowledge base. Such work would require further refinements and additions to ION, such as providing some form of support for non-numerical input and output to interface with the largely text-based Open Mind system, but nonetheless such research seems highly stimulating and exciting.

### **5. RELATED WORK**

ION has a strong background in other research. ION basically takes an effective architecture and provides a simple graphical interface which allows users to easily and reliably use that architecture as a design pattern in the creation of their own systems. This purpose is in the vein of research done in 1996 by Frank Budinsky, Marilyn Finnie, John Vlissides, and Patsy Yu, whose research focused on providing automatic code generation for several design patterns for software engineers and on following more rigorously the traditional descriptions and uses of design patterns [2]. ION departs from their research by removing traditional terminology from the design in an effort to increase intuitiveness and clarity to people other than software engineers, while still attempting to provide a useful and power interface to all users. ION also focuses on a single, through very general, architecture, rather than providing support for a variety of architectures and design patterns.

ION also takes inspiration from the plethora of other currently available programs which abstract more complex software engineering tasks through the use of graphical interfaces or other more intuitive forms than code, such as wizards and WYSIWYG graphical interface creators which have become standard parts of many programs, including not only software development programs but also everything from configuring operating systems to Internet web forms. In fact, clarifying and simplifying control and interaction with a computer and its data has been a theme present in computer science ever since Blaise Pascal added number dials to his adding machine so he didn't have to count gear teeth and rotations.

Finally, and as mentioned earlier, the architecture refined in the CarCoach project, began by Ted Selker and Taly Sharon and refined and completed by Shawn Sullivan at the MIT Media Lab, served as the template upon which ION's architecture was built.

This list of related works is by no means exhaustive of all research which may have played a part in the eventual creation of ION. It is only a list of primary sources used in researching the design and creation of ION. The authors apologize for any omissions and will rectify them upon suitable request.

### **6. CONCLUSION**

Because of its simplicity and the ease with which it allows software engineers to develop a wide variety of systems, ION has the potential for great significance to software engineering. ION

essentially is an automatic code/ data structure generation utility (depending on what one decides to call the XML files representing the created systems) designed specifically to create systems capable of receiving and dealing with input. The exciting field of automatic code generation promises to make software design faster and more accessible to the average person, all while improving the overall quality of software in general. Through observing an effective design solution to a problem and encapsulating its structure into a simple interface, ION provides a novel development tool permitting users to develop many kinds of systems quickly and effectively. Thus far ION has proved stimulating and promising in its scope and usefulness, and further research and use of this young program seems promising.

## 7. REFERENCES

- [1] Bray, T., Paoli, J., Sperberg-McQueen, C. M. Extensible Markup Language (XML) 1.0. *Technical Report REC-xml-19980210, World Wide Web Consortium* (1998)  
<http://www.w3.org/TR/REC-xml>.
- [2] Budinsky, F., Finnie, M., Vlissides, J., and Yu, P. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35, 2 (1996) 151-171.
- [3] Frank, A. J., Selker, T., Sharon, T., and Wagner, L. CarCoach: A Generalized Layered Architecture for Educational Car Systems. *IEEE International Conference on Software, Science, Technology, & Engineering, IEEE 2005* (Feb. 2005) 13-22.
- [4] Lim, G., Lin, T., Mueller, E. T., Perkins, T., Singh, P., and Zhu, W. L. Open Mind Common Sense: Knowledge Acquisition from the General Public. *Lecture Notes in Computer Science, Volume 2519* (Jan 2002), 1223 - 1237R2.
- [5] Postel, J. Internet Protocol (Sep 1981).
- [6] Postel, J. Transmission Control Protocol (Sep 1981).
- [7] Postel, J. User Datagram Protocol (Aug 1980).
- [8] Schölkopf, B., and Smola, A. J. *Learning with Kernels*. MIT Press, Cambridge, MA, 200