IDEOSY

An Ideographic and Interactive Program Description System[†]

Alessandro Giacalone, Martin C. Rinard, and Thomas W. Doeppner Jr.

Department of Computer Science
Brown University
Providence, Rhode Island 02912

1. Introduction

IDEOSY is an experiment in the use of a formal semantics as the basis for a programming system and in the use of an *ideographic* language as the primary means of user-computer communication. The important characteristics of our system are that it uses an ideographic syntax, has a syntax-directed editor, supports the definition of various equivalence properties and the proofs of such equivalence, and has an interpreter. It currently runs on Apollo workstations and on VAXes running Berkeley UNIX[‡] using any of a variety of high-resolution color displays.

Our formalism is based on Milner's Calculus of Communicating Systems (*CCS*) [1]. We have found CCS to be a convenient formalism for describing programs and have even used it for describing the UNIX operating system [2]. Its algebraic properties are very useful for building descriptions out of components and for proving the equivalence of descriptions. Since CCS is an operational semantics, we may directly interpret descriptions written in CCS.

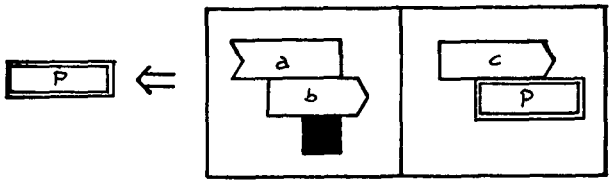
The idea behind using an ideographic interface such as ours is that a graphically suggestive language will aid the process of translating one's intuitive idea of a program's structure into a formal description. Our language, IDCCS (for *Ideographic Calculus of Communicating Systems*), was described in a previous paper [3]. It uses ideographs (pictures) to represent the various elements and operators of CCS. This orientation, i.e. the use of ideographs instead of words, is the cornerstone of our system. It allows rapid interaction between the user and the system, as the user may quickly manipulate descriptions by selecting ideographs representing its components. This ability to work with a formal description graphically is exploited not only in the editor but in the proof-aider and interpreter as well.

In the next few sections we first introduce the editor and the language. We discuss the concept of equivalence in CCS and then cover the support for formal reasoning in IDEOSY. This is followed by a discussion about the use of our interpreter. Finally, we discuss current limitations, extensions and future developments.

2. The Editor

We describe the editor of IDEOSY with the help of an example, trying as much as possible to show what the actual screen looks like. The editor can be used in two basic ways: to construct an expression in a top-down fashion, i.e. starting from scratch, and to modify an existing expression.

We illustrate the top-down part first by showing how we insert the definition of the agent named P below into the system and, in particular, how the defining behavior expression is constructed. The expression defines the behavior of P as that of an agent which can either execute a sequence of two communication actions (named *a* and *b*) and then terminate or execute an action named *c* and then reproduce its behavior.



The symbol

stands for an agent identifier. Juxtaposed boxes define, in the syntax of IDCCS, an "exclusive or" of the alternatives enclosed in each box. The symbols

[†]This work was partially supported by the National Science Foundation under grant MCS 8121806 and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA order No. 4786.

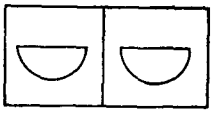
[‡]UNIX is a trademark of Bell Laboratories.

are ideographs for input and output actions, respectively. Actions arranged diagonally from upper left to lower right define a sequential execution of those actions. Finally, the symbol:

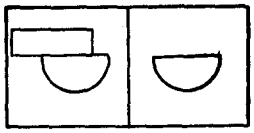
is an ideograph for a "null agent", i.e. an agent that does not perform any action, but represents the termination of an agent.

To define our expression, we inform the editor that we wish to define a process and provide a name for it (in this case P). At this point, the editor displays the symbol below:

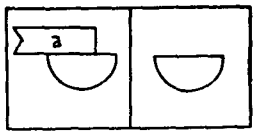
The "bowl" symbol is the *start* symbol in the IDCCS grammar as well as the non-terminal symbol (and ideograph) for a process. An expression is constructed by repeatedly replacing a non-terminal symbol with the right-hand side of a production of the IDCCS grammar. We specify which production we wish to apply by selecting ("picking") it from a menu of ideographs, which varies according to the type of non-terminal selected for replacement. Assuming that the "exclusive or" ideograph is selected, the picture below is displayed.



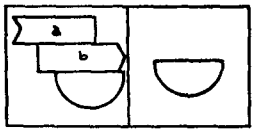
We must now "expand" the two bowl symbols in each alternative of the "or". Assume we expand the one on the left first. Since we want to produce a sequential expression, we select the corresponding ideograph and obtain the picture:



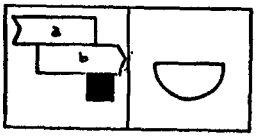
On the left we now have a construction formed by the ideograph for an *action* followed by the bowl. Assuming that we expand the action symbol first, we have to choose among the types of action (input or output). By choosing an input action and naming it a, we obtain the picture:



By repeating the procedure above for the bowl symbol appended to the input action just created, we can create an output action named b and obtain the picture:



At this point, we replace the bowl symbol with the symbol for a null process and obtain:



By proceeding analogously with the right side of the "or" we obtain the expression we wanted.

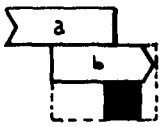
Besides the top-down expansion of non-terminals, it is possible to replace any construct that appears in an expression (i.e. any terminal, non-terminal, sub-expression, or the whole expression) with a construct of the same syntactic type. Summarizing, to replace one construct with another, one

(1) selects the construct to be replaced from the expression on the screen (possibly the whole expression);

(2) picks the *replace* command (it is a button always on the screen);

(3) identifies the new construct (the system will produce an error message if the new construct is not of the correct type).

The construct to be replaced (step (1)) is selected by picking any point within the smallest rectangle that encloses it (an operation called *range picking*). For example, in the expression below:



the dashed rectangle defines the area within which we must pick in order to select the sub-expression:

Terminal symbols (which are enclosed in rectangles) are a particular case and define a range on their own (so that in the picture above we would have to pick within the rectangle but outside the terminals, otherwise one of the terminals would be selected).

The replacing construct is selected (step (3)) by either picking it from an expression on the screen (by range picking) or selecting a definition among those

known by the system.

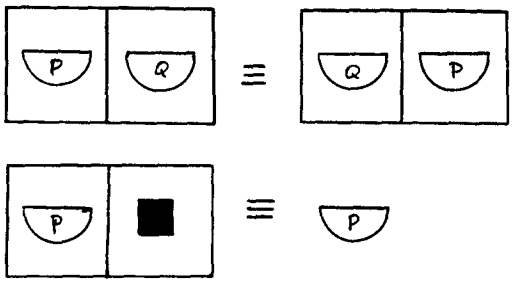
The rules that define which construct is range-picked in each case are non-ambiguous but not always immediate. However, the flexibility introduced by *replace* surpasses its drawbacks: for example, the "deletion" of a construct from an expression is a particular case of replacement and is actually implemented as a "derived operator" of *replace* (note that the system will not allow the deletion of a construct from an expression if the resulting expression is structurally incorrect).

3. CCS Equivalence Relations

In CCS, a number of equivalence relations are defined on behavior expressions: *identity* ($=$), *direct equivalence* ($\equiv$), *strong congruence* ($\sim$), *observation equivalence* ($\approx$) and *failure equivalence* ($\simeq$). The equivalence relations differ in how refined a partition they determine on the set of behavior expressions: stronger relations determine more refined partitions and smaller equivalence classes, thus imposing stronger conditions for two behavior expressions to be considered equivalent. The relations were listed above in decreasing order of "strength", in the sense that each relation *implies* all the following ones:

$$= \subset \equiv \subset \sim \subset \approx \subset \simeq$$

Some equivalence relations are *congruences*, i.e. are such that when two expressions are equivalent (with respect to that equivalence) then each expression can be replaced by the other in any operator context (typically, as a part of a more complex expression) without altering the meaning of the the entire expression. This is not true of all the equivalence relations defined in CCS. Those which *are* congruences are particularly useful. They allow the definition of "equations" (more properly, *equational laws*) between classes of processes (represented by behavior expressions). Examples of (simple) equational laws, expressed in IDCCS syntax, are:



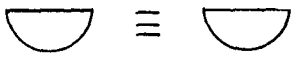
which define, respectively, the commutativity of the "or" operator and the possibility of eliminating null operands from an "or".

The equations can be used to apply semantics-preserving syntactic transformations to behavior expressions, for example to simplify an expression, or, more generally, to put an expression in a form which is "useful" for further reasoning. In the following, we refer to such equations simply as "rules". In IDCCS syntax rules are pairs of expressions which (in general) include non-terminal symbols of the grammar (possibly named) and therefore identify "classes" (not to be confused with *equivalence* classes) of expressions.

4. Support for Formal Reasoning

Rules such as those listed above constitute the basic knowledge IDEOSY has about CCS semantics: the system represents the equivalence relations defined on CCS behavior expressions as a list of rules appropriate for each equivalence relation. The system can apply the rules it knows and thus perform all the repetitive, and usually quite complicated, work involved in reasoning about CCS definitions. The user is then free to concentrate on more "strategic" decisions. Moreover, the system can support the user's decisions by detecting which of the rules it knows are relevant, i.e. could be applied in each case. A few basic rules are "hardwired" in IDEOSY. Any other rule which is found to be generally useful can be inserted in the system at any time.

A rule is inserted by using the editor of the previous section. For rules, the syntax that guides the editor is that of an equation, i.e. two expressions related by some equivalence symbol. For example, if one wanted to insert one of the rules shown above, the first picture displayed after the user picked the desired type of equivalence would be:

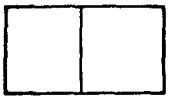


With this, the "editor" is telling the user (1) which type of equivalence the property being defined refers to, and (2) that the user has to define the two expressions.

When the editor is used to edit a rule, the user is allowed to name non-terminals. For example, in the two examples of rules shown above the commutativity of the "or" operator and the elimination of null operands from or's were defined by appropriately naming by P and Q the non-terminal symbols for processes (the bowls). The system interprets non-terminals in rules as "syntactical variables", i.e. as representatives of all the expressions they can generate via the applications of the productions of the grammar.

Once a rule is acquired, it is associated by the system with all of the terminal symbols that appear in it. Thus in the example above, the rule is associated with the symbol:

(being a property of the "or" operator), while in the case of the "*NIL*-elimination" rule mentioned above, the rule is associated with both the symbols




When the user wants to transform an expression, he or she can obtain a list of the rules applicable to sub-expressions by just picking a (terminal) symbol from the expression. The system scans the set of rules associated with the symbol and attempts to unify the expression on the left side of each rule with the expression on the screen. The terminal symbol originally picked determines the context (i.e. the sub-expression) with which the unification is attempted. During the unification, the syntactical variables present in the right side of the rule are bound to sub-expressions of the expression on the screen. Each rule that unifies successfully is then displayed on the screen as an ideograph on the menu. At this point the user can select a rule to apply. The system will replace the left side of the rule with the right side in the expression on the screen using the bindings established during the unification to determine the new expression.

Besides performing all the transformation work, IDEOSY also keeps track of the user's activity in transforming expressions (i.e. in applying transformation rules) and displays on the screen the "history" of such activity in the form of a tree (the *proof tree*). Each node of the tree represents a stage in the proof reached through the transformation of an expression. Every time the user applies a new rule, the system creates a new node which is a child of the current node, and labels the arc between the two nodes with the identifier of the type of equivalence used in the transformation. An arc labeled by an equivalence symbol means that the nodes at the two extremes of the arc are equivalent with respect to that equivalence (and thus with respect to all weaker equivalences). Every node also has associated with it all the information necessary to reactivate the stage it represents. The nodes of the displayed tree are thus also ideographs that allow the user to move around in the proof by retrieving any previous stage.

5. The Interpreter

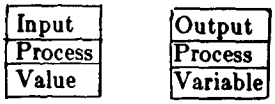
The last part of IDEOSY we describe is an environment in which CCS expressions are interpreted in an operational sense, i.e. are considered as defining actual executing modules. The interpreter subsystem consists of two main components: a *CCS interpreter* and a *user interface*. The interpreter simulates the concurrent execution of "processes" defined by CCS behavior expressions; in particular, it considers the *composition* operator of CCS as defining an execution environment in which communication channels are set up between complementary ports of processes. The user interface, based on the IDCCS graphical syntax, allows a user to interact (to "communicate") with the executing agents: the user is considered an *observer*, in the CCS sense, of the agents. In addition, the execution can be stopped at desired points and the "state" of the executing system can be displayed.

Beside being a tool for executing CCS specifications in IDEOSY, the interpreter was designed and developed for the purpose of investigating possible implementations of any language based on a port-oriented communication scheme. For this reason, the interpreter is conceived as an "abstract machine" whose structure is defined by the data-structures used in the interpreter and whose machine language is defined as a (very small) set of basic operations in terms of which the whole interpreter is implemented. These basic operations, which essentially update entries in tables, can be summarized as follows. A communication between two processes is implemented by evaluating an expression (the one being sent), updating the local environment of the receiving process and, finally, updating the "instruction pointers" of the two processes. The CCS *composition* operator is interpreted as a "create-process" primitive which provokes the allocation of new process descriptors. Private communication channels are allocated by the interpreter to implement the CCS *restriction* operator (an operator used to restrict the visibility of ports).

The user can start the interpreter at almost any point within an IDEOSY session. It will interpret the *current* expression (roughly, the expression currently displayed). Usually, the current expression is a composition of processes. The user at the "console" of the system is viewed as a process with two ports named *user-in* and *user-out*. The two ports can be used to communicate with the executing processes (i.e. to *observe* them). To do this, those processes that need to communicate with the user must have corresponding ports named *user-in* and *user-out*:

- *user-in* is used to input values from the user.
- *user-out* is used to output values to the user.

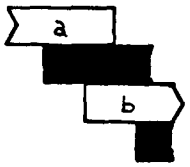
On the screen, two fields identify the two ports, as in the picture below.



The *user-in* and *user-out* actions are treated in an asymmetrical way by the interpreter. When a process tries to execute a *user-in* (*user-out*) action, the interpreter lets the user decide whether or not to "accept" the communication (communications are synchronized in CCS). A message appears on the *user-in* (*user-out*) field on the screen saying which process is trying to execute an input (output) action from (to) the user. At this point, the user

can provide a value (accept the value being output) or refuse the communication. In the second case, the action is put back in the corresponding queue in the action table and the execution proceeds with another action.

The user thus can control the execution of processes by either providing (authorizing) or not providing (not authorizing) input (output) to (from) the processes. Another tool the user can use to control the execution is a special type of action called a *checkpoint*, represented in IDCCS syntax by a black rectangle, as in the expression below.



A checkpoint is a signal for the interpreter that it must display the state of the system. Checkpoints are treated like any other action internally and inserted in a special entry always present in the action table. When the scheduler selects that entry (and when there is a checkpoint action in the corresponding queue) an IDCCS expression which represents the system at that moment is constructed and displayed. Checkpoints can be inserted in expressions anywhere an action is legal, so that the user can make the system stop and display its state at strategic instants (e.g., to check which alternative of an "or" has been selected in non-deterministic programs, right before a composition is executed to follow the creation of processes, etc.).

6. Conclusions

IDEOSY has proved to be especially interesting to us because it combines topics of theoretical interest, e.g. the study of the formalisms used for the description of programs, with topics of much more practical interest, e.g. the ergonomics of the user interface. An important test of IDEOSY will be its use in the development of a description of a large program. This test will be made as we continue the work started in [2], this time writing our description in IDCCS rather than CCS and taking advantage of the reasoning facilities provided by IDEOSY to help validate our work. As described in [2], we are investigating extensions to the semantics of CCS. One of these led to the definition of a new formalism [4] based on CCS but slightly different from CCS in the meaning of *ports*. The new formalism is still based on the basic CCS notion of agents that communicate via ports, and its constructs can be represented pictorially by the same syntax (IDCCS) developed for CCS: only the set of equational laws change, and with them the meanings of some of the ideographic constructs.

Although we do not have space here for an extensive treatment of the principles on which the pictorial syntax of IDCCS is based, it should be observed that IDCCS expressions are designed to take advantage of the possibilities offered by high-resolution screens and fast graphics hardware (for example, it is time-consuming to draw them on paper by hand). IDCCS syntax is also designed to generate expressions which are self-explanatory and which keep a recognizable shape no matter how complex they become.

The system we have described is the latest of several versions implemented over the past many months and is currently rather inefficient, due to the fact that it is implemented using a machine-independent graphics package (SGP [5]) which does not take full advantage of the hardware we are using.

We are developing several improvements for IDEOSY. One is to extend the interface so as to allow context-dependent operators. Another, which we are working on currently, is the extension of the flexibility of the user-interface, in particular with regard to the rule-definition language. Currently, the rules that can be inserted in the system can only define properties of binary operators. We are working on a generalization of an IDEOSY interface which will allow the definition of n-ary properties and will include the possibility of defining rules whose applicability depends on syntactical and semantic properties of expressions (e.g., names of ports, equality of expressions, etc.). The problem is syntactical and consists of finding a pictorial representation of indexed families of objects and conditions consistent with the rest of IDCCS and IDEOSY.

Another development of a syntactical/pictorial nature on which we are also working is giving the user the ability to define *derived* operators, i.e. new operators defined in terms of the primitive ones of CCS. The primary difficulty with this addition is finding a method by which the user may define and use new (non-trivial) ideographs without having to resort to two-dimensional parsing.

Other developments we are planning have to do with enhancing the capabilities of the system. These are in two main directions: (1) improvements to proof aiding and insertion of proof-checking capabilities in the environment, and (2) increased capabilities of the interpreter and of the editor. We plan to give the user the capability of defining sequences of rule applications ("strategies") possibly conditioned by the structure of expressions. This will make the user able to define, for example, "simplifiers" tailored on classes of expressions and will save the task of specifying every single step of expression transformations. A proof-checker will eventually be built on top of the machinery currently available to a user and will have as its main functions: (1) automating the more tedious portions of proofs, (2) providing independent verification of steps in manual proofs, and (3) "looking for" applicable strategies and proofs.

7. References

[1] Milner, R.: **A Calculus of Communicating Systems**, Springer-Verlag, Lect. Notes in Comp. Sci. 92, 1980.

[2] Doeppner, T.W., Jr., Giacalone, A.: *A Formal Definition of the UNIX Operating System*, Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983.

[3] Giacalone, A., Kovacs, I.D.: *IDCCS: An Ideographic Syntax for CCS*, Brown University, Dept. of Computer Science, Tech. Rep. CS-83-05, Feb. 83.

[4] Giacalone, A.: *An Approach and Some Experiments Towards the Support of Formal Specifications in Integrated Programming Environments*, Unpublished Ph.D. thesis, Brown University, April 1984.

[5] Foley, J.D., van Dam, A.: **Fundamentals of Interactive Computer Graphics**, Addison Wesley, 1982.