Abstract

Computers provide three essential services: Data transformation, data storage, and data communication. Transformation and storage have long been well entrenched in algorithmic and data modelling languages. Unaccountably, communication is only seen as an I/O operation without explicit language support. At long last, this deficiency has now been remedied with the new Data - Context - Interaction (DCI) paradigm.

Alan Kay introduced the notion of object orientation in the early seventies. He regarded an object as a virtual computer and an object system as thousands of computers hooked together by a very fast network. The locus of DCI is in the DCI Context where we find the specification of object system behavior. The Context declares a network of communicating objects where the nodes are called Roles and the edges Connectors. The system behavior algorithm is composed from RoleMethods; methods that are associated with the Roles.

This study is about an execution model for DCI. An important aspect is the handling of DCI Contexts at runtime; where they are stored and how they are accessed. DCI in general and the DCI Context in particular make data communication a first class citizen of computer programming.
1 Introduction

NOTE 1: Readers not familiar with hardware and assembly programming may find it useful to glance at the appendixes first.

Note 2: Readers familiar with the DCI paradigm may jump directly to Section 2 (p. 4): The Plain Old Java Object (POJO) Execution Model

Computer systems can essentially provide three services: Data transformation, data storage, and data communication. Algorithmic languages such as FORTRAN and Algol are data transformation languages. Data modeling languages such as SQL and NIAM support data storage and retrieval.

Communication has long been a first class citizen in the hardware world; the bus centered IBM PC came in 1981. The software world has been slow in following suit. Communication has remained an I/O operation outside the scope of all major programming languages.

The idea of communicating objects could have lifted communication to become a first class citizen of software. Alan Kay, who coined the term object orientation, regarded an object system as a network of communicating computers:

In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing “computer stuff” into things each less strong than the whole--like data structures, procedures, and functions which are the usual paraphernalia of programming languages--each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network. [Kay-93]

The Smalltalk programming language is a class-based language where a class specifies one of Kay’s virtual computers. A class defines what an instance does with an incoming message; it says nothing about the message sender or the enclosing network of communicating objects. The roots of this deficiency can be found in a faulty assumption: If a class is programmed to make its instances “do the right thing”; these instances will behave appropriately when they are combined into systems of communicating objects. This assumption fails when “the right thing” depends on context so that a class cannot be properly understood when seen in isolation.

Another problem with class-based programming is that it ignores the well known axiom that “the value of a system is greater than the sum of its parts”. The added value is caused by a particular organization of the parts into a coherent structure, their context. The most important part of an object system is communication, i.e., what happens in the space between its objects.

True object oriented systems can be described in some modeling languages such as OOram [OORAM-92] and UML [UML-09]. Since Kay’s notion of an object is recursive; an object can be a member of an enclosing network as well as being the container of an inner network.

Figure 1 illustrates how a simple example of a network of communicating objects accomplish a simple task. Consider a person, Joe Smith, who wants to transfer some funds from one of his bank accounts to another. He goes to his bank and requests that it shall effectuate the transfer. Joe gives the transfer order to the bank, the bank instructs Account 1234 to do the transfer, this account deducts the amount from its balance before it instructs Account 5432 to deposit the amount. This way of looking at a computer application has two advantages: It is easy to understand for the bank customer and it can be a high level picture of the computer program.
Class based programming languages cannot be used to specify a system of communicating objects since it can only describe one object (class) at the time. This fundamental deficiency is remedied in the Data-Context-Interaction (DCI) paradigm by going outside the limitations of the class concept.

DCI can be studied from different viewpoints. When DCI is seen from the end user, we find use cases and user mental models. When DCI is seen from the application programmer’s point of view, we also see the DCI paradigm with its three perspectives of Data, Context, and Interaction. See for example the Wikipedia article Data, Context, and Interaction [DCI-Wiki] for a definition, The DCI Architecture: A New Vision of Object-Oriented Programming [ReeCon-09] for an overview, and The Common Sense of Object Oriented Programming [Ree-09] for details and commented examples.

This study is written for the systems programmer who wants to create a DCI infrastructure. The object of the study is to gain an understanding of the realization of DCI in a computer. There are many possible ways of doing this, most of them require compromises caused by the nature of an available programming language and its runtime system. We here present a clean execution model that implementers can use as a reference.

Section 2 (p. 4): The Plain Old Java Object (POJO) Execution Model
GOJO - Good Old Java Objects with their stack and heap

Section 3 (p. 5): The DCI Execution Model
Programming with Contexts and Roles. Program execution with DCI is different from any other execution model. The DCI Contexts live on the stack and their Roles are somewhat similar to dynamically scoped variables.

Section 4 (p. 8): The DCI Separation of Concern: Three Constraints
Three important constraints on the execution model given in the previous sections.

Section 5 (p. 12): Conclusion
Communication is now a first class citizen of computer programming.

Section 6 (p. 13): References
For completeness, we have included older memory management systems and find that they are inadequate for object oriented program execution. Readers unfamiliar with computer architecture and assembly programming may find these appendixes helpful for understanding the main sections (section 2 and section 3).

Appendix 1 (p. 14): Static Memory Management
Binary code, assembly, FORTRAN-like languages.

Appendix 2 (p. 16): Stack Based memory Management
Algol etc.
2 The Plain Old Java Object (POJO) Execution Model

In POJO systems, memory is divided into two parts: A *heap* and a *stack*. The *heap* is a part of the memory that is reserved for objects. Objects represent system state.

Another part of the memory is reserved for the *stack*. The stack is LIFO list of activation records with one activation record for each method activation. (see Appendix 2 (p. 16): Stack Based memory Management).

2.1 System state: *What the system IS*

When a class is instantiated, the new object is placed in an available part of the heap. The block of memory that represents the object includes a pointer to the object’s class and a slot for each instance variable. Every object has a unique and immutable identity. The memory management system maintains a dictionary that binds object IDs to the location of the object on the heap. (Conceptually, that is. There are many different implementations.) An object stays on the heap until it is either deleted programatically or, more common these days, when it is no longer accessible and is removed automatically by a garbage collector. The memory management system is part of the runtime system, often called the *virtual machine (VM)*.

2.2 System behavior: *What the system DOES*

System behavior is controlled through the stack. Individual object behavior is triggered when an object receives a message. The currently running method is interrupted and a new activation record is pushed onto the stack:

1) A method in the sender object creates a message. This message includes the sender object ID, the receiver object ID, the message name, and possible message arguments.

2) The VM adds an activation record on the top of the stack as illustrated in figure 2.

3) The VM follows the class pointer in the receiver object to find its message dictionary. Using the message name as a key, it looks up the corresponding method in the receiving object. The lookup is repeated along the superclass chain if necessary. (A *doesNotUnderstand*-exception is raised on failure). A link to the current method together and the program counter in the activation record are initialized.

4) The execution of the selected method is triggered. A method may change the object’s state. It can also send messages and the process is repeated from point 1) above.

Figure 2 illustrates the runtime memory location of variables that are visible from a binary GOJO instance method during its execution. There is an activation record for the current method. The stack grows and shrinks on top of this frame as the method is executed. The activation record includes a link to the receiver object and slots for the method’s temporary variables.\(^1\)

\(^1\)actual arguments and local variables.
The method source code namespace includes identifiers for the temporary variables, the instance variables of the receiver object, and other variables\(^1\). The compiler transforms the source code into a binary method as a sequence of instructions (byte codes) to the VM. Some instructions move values between the top of the stack and various memory locations, some request message sends, and some do other operations.

### 3 The DCI Execution Model

The DCI execution model builds on the POJO model. The handling of system state is as described in section 2.1 above. The system behavior is as described in section 2.2 with the addition of a new kind of variable: the Role. We will see how this gives rise to an extended runtime model and a corresponding change to the code that is input to the compiler.

#### 3.1 Programmer’s view of DCI

A programmer needs to make many decisions in order to generalize the execution model shown in figure 1 (p. 3) and fill it with details. The Delta project\(^{[Holbæk-Hanssen-75]}\) defined a very powerful notion of system that can be used to build the programmer’s mental model:

A system is a part of the world which we choose to regard as a whole, separated from the rest of the world during some period of consideration, a whole which we choose to consider as containing a collection of components, each characterized by a selected set of associated data items and patterns, and by actions which may involve itself and other components.

With DCI, the programmer chooses to regard the bank part of the fund transfer transaction as a system. He chooses to let this system consist of two interconnected Roles: A FromAccount and a ToAccount. He finally chooses to let the system as a whole be represented by a BankTransferContext; the transfer takes place within this Context.

The user mental model in figure 1 is now refined to become the application programmer’s mental model shown in figure 3:

1) The chosen system is represented by a Context: the BankTransferContext.

---

1. Static, global and other special variables are also visible; they are not discussed in this study.
2) An object in the system’s environment provides a trigger message; this message is called a system operation (or use case, or habit).

3) The system’s Data objects (the accounts) are represented by the Roles they play in this Context: FromAccount, ToAccount. The Context binds these Roles to the appropriate Data objects; here by looking up the account numbers to find the account objects.

4) The vertical timeline below each Role symbol is augmented with a rectangle that represents the method that will be executed. The method is a RoleMethod; it is common to all objects that can play the Role. (Polymorphism is suspended for RoleMethods). Taken together, the RoleMethods specify the Interaction or Algorithm that is executed when the system performs a system operation.

5) RoleMethods executed within the Context do not need the account objects as arguments because the Roles that represent them are visible in the Context.

6) The Data objects are the account objects that play the Roles. (See in figure 4).

Figure 3: Application programmer’s DCI based mental model of the Interaction.

In one of his many clarifying posts on the object-composition@googlegroups.com list, Rickard Öberg wrote:

```
with DCI, the application facade that used to look something like this:
    facade.method(id1,id2,id3,param1,param2)

is now replaced by:
    context<id1,id2,id3>.interaction(param1,param2)
```

A bank offers many different services to their clients, the transfer of funds being one of them. Figure 4 shows how different services are encapsulated in different Contexts. It is only when we look inside a Context that we see that it encapsulates a network of communicating Roles. We have opened the BankTransferContext and see that its Roles are momentarily bound to the bank’s Data objects. We see, for example, that the FromAccount Role happens to be bound to a Data object in the bank that represents Account1234. In addition, the RoleMethod associated with the FromAccount Role is injected into the Account1234 object.
3.2 A runtime view of DCI

A DCI Interaction is triggered by a system operation on a Context object as described in the list on page 4. This message triggers the execution of one of the Context’s instance methods. Figure 5 shows the activation record for this execution colored red and drawn with a heavy outline. This Context activation record stays on the stack until it is popped off at the termination of the trigger method.

The trigger method to the Context first initializes the CurrentContext’s Role bindings and then starts the Interaction with a message to the first Role and thus triggers the first RoleMethod.

RoleMethods are not associated with a particular class so RoleMethods cannot access the instance variable in the receiver object. Instead, RoleMethods address Data objects indirectly through the Role name as illustrated in figure 5. This figure also illustrates that RoleMethods...
access temporary variables in the same way as instance methods. Note that \textit{self} and \textit{super} refer to the current receiver object; i.e., the object that is playing the current Role.

A method that has an activation record above the \textit{CurrentContext} on the stack is executed within this Context. The method can search down the stack to find the top \textit{CurrentContext} and access a Roleplaying object through it. Roles can only be accessed from RoleMethods since Roles are only meaningful within a Context.

Our current DCI metamodel only permits one Context to be active at any time (More about this constraint in section 4.3). This Context is called the \textit{CurrentContext} and can be found by searching the stack down from its top. It is sometimes more convenient to maintain a shadow Context stack; the \textit{CurrentContext} is then found on the top of this stack. Statically typed languages makes it easier to maintain a separate stack for each Context class. The latter solution has been chosen for several of the current DCI implementations.

An important detail is that any object, including a Context object, can play a Role as long as it has the necessary properties. This means that a new Context can be triggered within an executing Interaction.

### 3.3 A compile time view of DCI

The code for a RoleMethod uses Role names as identifiers instead of instance variable names. For example, the RoleMethod shown as a vertical rectangle under the \textit{FromAccount} Role in figure 3 (p. 6) could be:

\begin{verbatim}
FromAccount>>transfer(amount) {
    if (self.balance < amount) self.error ("no funds");
    ToAccount.deposit(amount);
}
\end{verbatim}

Note the \textit{self} variable; it refers to the receiver object, i.e. the object currently playing the \textit{FromAccount} Role.

The above works if our compiler can compile this method within the \textit{BankTransferContext} and generate code that binds the Role name to the bound object. As an example, the \textit{CurrentContext} could be a global variable and the compiler or some kind of macro processor could generate

\begin{verbatim}
FromAccount>>transfer amount) {
    if (self.balance < amount) self.error ("no funds");
    (CurrentContext.getObjectForRole ("ToAccount")).deposit(amount);
}
\end{verbatim}

Or \textit{CurrentContext} could be replaced by code that referenced a ContextStack with a static method in the \textit{BankTransferContext}:

\begin{verbatim}
FromAccount>>transfer amount) {
    if (self.balance < amount) self.error ("no funds");
    (BankTransferContext.getObjectForRole ("ToAccount")).deposit(amount);
}
\end{verbatim}

### 4 The DCI Separation of Concern: Three Constraints

With DCI, a program is decomposed into independent units of code that reflect important concerns that exist right from the end user’s mental model to the innermost parts of the program. It
is an imperative goal of DCI that it shall be possible to reason about the code in a particular unit without being surprised by interference arising from other units. This requirement leads to constraints that must be added to the DCI execution model. We will here discuss three: The need for one-to-one relationship between Role and object in a Context, the need for unique method names within Roles and in Data classes, and the limitation that there can be only one active Context at the time.

4.1 The uniqueness of Roles

In general, there is a many-to-many relationship between Role and Data object. A Role may be played by different objects, even by instances of different classes. Conversely, an object may be capable of playing many different Roles. This flexibility must be constrained during an actual execution. When we reason about how a network of objects interact, we reason about the similar network of Roles. Our reasoning will clearly be invalidated if there is not a one-to-one relationship between Role and object.

An example can illustrate how subtle bugs can arise from object identity problems in very simple Interaction code. The example is the Stars-and-Circles animation discussed in [Ree-09]. The animation shows a background of shapes; each shape representing an object that is an instance of one of the Circle or Star classes selected at random. An execution is shown as a sequence of arrows that are slowly being painted from one shape to another to illustrate the flow of messages through the system.

Figure 6: Two snapshots from the Stars-and-Circles animation.

The result of a correct animation is shown to the left in figure 6. We see that the network is a simple, linear structure consisting of 5 shapes: 1 -> 2 -> 3 -> 4 -> 5. An execution starts with the Context selecting shapes at random to bind them to the five Roles. The animation is then triggered by the Context sending the trigger message `playRole1` to `Shape1`. The corresponding method sets the state of the object playing `Shape1` to 1 and sends the message `playRole2` to the `Shape2`-object. The code is:

```
Shape1>>playRole1 {
    self.setState("1");
    Shape2.rolePlay2;
}
```

Correct Faulty
The animation program displays the message sent from \textit{Shape1} to \textit{Shape2} as an arrow before it triggers the corresponding RoleMethod in \textit{Shape2}. This method is a copy of the above code with the obvious changes. The Interaction code is thus very readable and surely can’t hide any surprises. Right? Wrong! Repeated testing showed that the animation evolved as expected, but on a very few occasions the display looked something like the right-hand picture in figure 6. The visible sequence is here 1 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5.

The problem is a version of what is commonly known as \textit{self schizophrenia}. The Context selected shapes at random to play each of the five Roles. The error was that is so happened that the same object was selected for \textit{Shape2} and for \textit{Shape4} in a particular execution. A Role is an abstraction of object identity, it is a named object. In the faulty example, \textit{Shape2} and \textit{Shape4} happened to be mapped to the same object so its state was first set to 2 and then later reset to 4.

This bug was not visible in the Interaction code, and it was very hard to find by testing. (I happened to have only 50 shapes in the animation and I got a fleeting impression that once in a while there was something wrong.)

Over time, there is a many-to-many relationship between Role and Class. A class may realize the requirements of many Roles. Conversely, the requirements of a Role may be realized by many classes. But we need to be more restricted:

\begin{quote}
\textit{Role-to-object binding can only be done by the Context.}
\textit{A Role must uniquely identify an object;}
\textit{no two Roles can be bound to the same object at the same time.}
\end{quote}

\section*{4. 2 The uniqueness of RoleMethod names}

Roles are properties of Contexts and are named within a Context namespace. Roles declared within different Contexts can, therefore, have the same name without conflict.

RoleMethods are declared within methodful Roles and Methodful Roles act as namespaces for their RoleMethods. RoleMethods declared within different Roles can, therefore, have the same name without conflict. This appears to be safe, but the injection of RoleMethods into Data objects can give rise to unpleasant surprises. Figure 7 is taken from [Ree-09] and shows an overview of DCI where RoleMethods are injected into the appropriate classes.
A Data class (with its superclasses) forms a namespace for method names. We see from figure 7 that RoleMethods are injected into Data classes and thus become part of the namespaces of these classes. In addition, different Roles may inject their RoleMethods into the same classes. In some implementations, RoleMethods are injected directly into the object that plays the Role at runtime. All this leads to the following constraint:

*The name of a RoleMethod must be unique within the Role and also within all objects that may play this Role.*

4.3 The uniqueness of the CurrentContext

Figure 5 illustrates a stack with more than one Context activation record. We compare this figure with the dynamic scoping example in figure 9 (p. 16) and see that Contexts could be accessed using dynamic scoping, but the fact that we can do it doesn’t mean that we should do it. DCI is about powerful and simple mental models for both code writer and code reader. Wikipedia warns that dynamic scoping can be dangerous because it can lead to unreadable code. Multiple active Contexts are, therefore, not permitted in DCI.

*Only one Context can be active at the time.*
5 Conclusion

The DCI execution model extends the POJO model with a capability for controlling the communication between named objects within a network of communicating objects. With DCI, we see the basic capability of computing in a three-layered architecture:

1) *Data storage* is done in an object’s state variables. An object’s apparent state can be derived, i.e., computed from other state.

2) *Data transformation* is done in an object’s methods.

3) *Data communication* is done by message interaction within a network of connected objects.

Data communication is promoted to join data transformation and data storage to become a first class citizen of computing and its associated programming languages.
# References

| [DCI-Wiki] | *Data, Context, and Interaction*;  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kay-93]</td>
<td>Alan Kay: The Early History of Smalltalk; ACM SIGPLAN Notices archive; 28, 3 (March 1993); pp 69 - 95</td>
</tr>
<tr>
<td>[Reed-05]</td>
<td>David Reed: <em>Organization of Programming Languages</em>. Creighton University, 2005</td>
</tr>
</tbody>
</table>
http://www.artima.com/articles/dci_vision.html |
| [Ree-09] | Reenskaug T.; [WEB PAGE]  
http://www.omg.org/spec/UML/2.2/Superstructure/PDF/ |
| [Wikipeadia] | [WEB PAGE]  
The work is released under CC-BY-SA. See  
http://creativecommons.org/licenses/by-sa/3.0/ |
Appendix 1: Static Memory Management

Figure 8 shows a very simple binary, sequential computer that will serve as an example to illustrate basic memory management. Its memory has 256 locations with addresses 0..255. The instruction format consists of two bytes, the first is the operation code, the last is a memory address. This imaginary computer has the operation repertoire as shown in Table 1.

Figure 8: A simple, stored program computer.

Table 1: Instruction repertoire.

<table>
<thead>
<tr>
<th>operation code</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HALT</td>
</tr>
<tr>
<td>1</td>
<td>LOAD a value from memory to the accumulator</td>
</tr>
<tr>
<td>2</td>
<td>STORE a value from the accumulator to memory</td>
</tr>
<tr>
<td>3</td>
<td>ADD a value to the contents of the accumulator</td>
</tr>
<tr>
<td>4</td>
<td>JUMP. Value is address of next instruction</td>
</tr>
</tbody>
</table>

6.1 Binary code

We write code directly for the hardware, no intermediary. A hardware controller feeds program operations one by one into the Operation register where they are executed.

As an example, put the value 3 into memory slot 100 and 4 into 101 (manually, from the computer console). Then execute the following program.

Table 2: A small example program. (Format: op/address)

<table>
<thead>
<tr>
<th>operation</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The memory slots from 100 will now contain 3, 4, and 7.
Memory management is done manually when we assign memory slots to instructions and data.

It is a sobering thought that all computers work more or less as described here even if they have a more extensive operation repertoire. Programming languages are devised to make the programmer’s mental model better suited to his or her task. There is no leverage without rigidity. No sophisticated language, metaphor, or paradigm can add to the capabilities of the computer hardware. All they can do is to give the programmer better leverage through reduced capabilities. Assembly code cannot specify logic based on actual memory addresses. FORTRAN does not support self-modifying code.

An argument against DCI, our new programming paradigm, is that “There is nothing new here, I can do the same in Java”. This comparison is irrelevant. Machine code can do all Java does and more, because machine code is what is ultimately executed by the computer. It is better to ask questions about the programmer’s tasks and the kinds of mental models that are more effective for those tasks.

DCI is more restrictive than Java because DCI insists on a stable topology for the runtime communication networks. DCI programs are often longer than Java programs because DCI includes an explicit specification of the network of interacting objects and their runtime interaction.

6.2 Assembly programming

Our program in table 2 can be written as follows:

<table>
<thead>
<tr>
<th>Table 3: An assembly program. (Format: op/variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>STORE</td>
</tr>
<tr>
<td>HALT</td>
</tr>
</tbody>
</table>

We assemble the program into some intermediate code and then use a Loader to load this code into the computer. The Loader recognizes that there are three variables and it assigns memory slots to them. Memory slots 100, 101, and 102, for example. There is a fixed transformation table for operations as shown in table 1.

6.3 FORTRAN programming

Our code expressed in FORTRAN is straightforward:

\[ I = 3 \]
\[ J = 4 \]
\[ K = I + J \]

Memory management in FORTRAN is static, each and every variable has a fixed storage location. (Even local variables in subroutines have their fixed locations, recursion is not supported by FORTRAN.) The FORTRAN compiler transforms the FORTRAN code into something similar to assembly as illustrated in table 2. A Loader then assigns every variable permanently to a memory location and loads the program into memory.

Memory management for FORTRAN is thus static as it is for assembly and binary.
Appendix 2: Stack Based memory Management

Algol 60 and many of its successors are block structured. Blocks and procedures can either be declared within the main program or nested within other procedures as illustrated in figure 9. The difference from static memory management is that there is now a runtime system that dynamically binds variable names to memory locations. Memory is organized in a LIFO list of activation records. The runtime system binds variable names to slots in activation records according to one of two different strategies:

- **Static (lexical) scoping**: non-local variables are bound based on the code structure; search the code lexically up from the current position in the code text to find the first occurrence of the variable.
- **Dynamic scoping**: non-local variables are bound based on the calling sequence; search the stack down from the current activation record to find the first occurrence of the variable.

The stack is used for many other purposes such as evaluating expressions. Our interest here is that it can be used for dynamic scoping:

> With dynamic scope, each identifier has a global stack of bindings. Introducing a local variable with name x pushes a binding onto the global x stack (which may have been empty), which is popped off when the control flow leaves the scope. Evaluating x in any context always yields the top binding. In other words, a global identifier refers to the identifier associated with the most recent environment. Note that this cannot be done at compile time because the binding stack only exists at runtime, which is why this type of scoping is called dynamic scoping. (From [Wikipedia], Scope (programming)).

...As such, dynamic scoping can be dangerous and few modern languages use it.

Figure 9: Static and dynamic scoping. (This figure is copied from [Reed-05]).

```
program MAIN;
 var a : integer;

 procedure P1(x : integer);
 begin
  print x, a;
 end; {of P3}
 begin
  P3;
 end; {of P1}

 procedure P2;
 var a : integer;
 begin
  a := 0;
  P1(a+1);
 end; {of P2}
 begin
  a := 7;
  P2;
 end. {of MAIN}
```

The bottom-right of figure 9 is an upside-down view of the stack while the code shown on the left is executing P3. The code is excessively complicated, but it does illustrate what happens at runtime.