17/3/98

CHAPTER 4

Building Correct Systems.

Summary. Refining correct solutions, components, an integrated design and test strategy, a simple refinement example, testing refinements, conclusions.

The approach that we have taken so far has been illustrated with simple, small examples of systems. In reality systems are large and complex. How can the ideas be translated to a more realistic scale? The question is not easily answered and no software development method has really proved successful in this respect. One of the problems is that existing techniques tend to be programmer centred or at best analyst centred rather than oriented around the process of establishing requirements and transforming these in a consistent way into a complete design and test process. Creating a process that can adapt to possible changes in the business and its requirements is also hard to achieve. So the problem of building correct systems is made more difficult. Attempts have been made to introduce a more formal approach to the refinement of a specification or a design into a more complete representation of the system, ultimately leading to an implementation. Wirth [83] and Parnas [84] discusses refinement as a principle. Jones [17] looks at the issue of reification in the context of the transformation of a formal specification in VDM into a more concrete representation. Morgan [85] also introduces a formal mechanism for developing specifications in Z. These approaches are essentially procedures for transforming abstract notations, with or without possible machine support, and sit uncomfortably with the preferences of many software designers for a less symbolic and more diagrammatic way of describing systems.

In this chapter we will try to maintain the link with the client view and at the same time recognise that systems are large and complex, need to be designed in stages and, from time to time, redesigned. We will also look closely at the issue of components. It has been suggested in many places that a *software components approach* to design would overcome some of the real quality problems that afflict the field. Taking inspiration from the practice of many other types of engineering it is believed that if we could develop libraries of robust and well tested software components and build complete systems by composing these together in suitable ways to form complete solutions we will have taken a major step to solving the software crisis. Furthermore, if the requirements change then it may be possible to either add new components or replace some with others to swiftly achieve a re-engineered system. So far attempts at doing this have been failures. The idea of software reuse that lies at the heart of this approach is often used as a motivation for object oriented languages and methods. However, reusing classes and objects from a library or from other systems will not necessarily result in an improvement over current practice because of the extra testing that is required. The widespread belief that well tested classes and objects from other applications will not require retesting is a fallacy, all that is done is that the testing effort has been redistributed not reduced. Furthermore, software designers tend to “interfere” with their “solutions”. In other words, unlike other types of engineering, there is no fabrication phase whereby the design is handed over to a technical team for turning into a concrete artefact. Up to a few minutes before the software is installed and goes “live” the programmer can make changes to it. This has been the cause of many problems in the past and is inevitably the result of either failing to capture the requirements properly or failing to understand them or just building and testing the system poorly.

4.1 Refining correct solutions.

Let us recall what a correct system solution is:

a correct model of the requirements + a complete implementation

+ a complete and successful test strategy

We identify the client's needs through their perception of their business processes and the integration of these into enterprise solutions. There are a number of ways in which this process could be scaled up to more complex systems. We could begin with a very top level view of the business and a possible solution which would emphasise the way in which the system was constructed from several smaller systems which communicate with each other in particular ways. These system components would then be further decomposed until a point was reached where the components were sufficiently small they could be analysed and built using the techniques described earlier. In this approach what is important is that the integration of the components, that is the *communication model*, is precise and understood, is preserved during the design process and tested thoroughly throughout. The integration testing of any system is vital and can be an area where massive effort has to be applied. The framework for this approach should also allow for the inevitable changes that will occur to the requirements during the course of the project. Although requirements change there will usually have to be a point where they are frozen to prevent *requirements creep* and to enable a solution to be built. What is important is that the specification and the design are fully defined in such a way that later changes might be made and verified easily.

An alternative strategy might be to look at the available components and to see if these could be composed in some way to construct a possible solution. The problem then being that the verification of the constructed solution against a, possibly rather vague requirement, is a major task. This bottom up strategy seems to be the main approach to implementing systems using an object oriented programming language where the integration and control of the components is distributed throughout the system.

The approach discussed here will incorporate both strategies. It will involve the top down decomposition based approach integrated into a refinement oriented method that allows the test strategy to develop with the requirement specification model and which is able to utilise the combination of trusted components into a complete and fully tested solution. We now turn our attention to trying to identify what a *component* might be.

4.2. Components.

What is a component? Is it a procedure, a module, a function, an object, a class or what?

How will we describe the detailed behaviour of the components? How can we put components together so that the resulting system has a predictable and desirable behaviour? Is it the case that

poor quality components will lead to a poor quality system? These are some of the questions that we will be trying to address in this chapter.

A component is some *coherent* part of the system which has some intrinsic *common property*, something which has an *identifiable* and *consistent behaviour* or *structure* which makes it meaningful to treat it as an entity in some sense. The problem with this definition is that it is too general and there can be many ways in which it could be satisfied. We will try to address the issue through our approach to systems using computational models. So we make the following tentative definition.

Definition 4.2.1 A *component* for a system is a business process function or a stream (or enterprise) X-machine

This brings with it a recognition that a component must have:

1. An interface comprising of a definition of what inputs can be applied to the component and what outputs it can be produce.
2. An internal control structure which determines how the component is organised dynamically.
3. An internal memory store which can be updated as the component operates.
4. An explicit functional description which defines what the component does under all possible circumstances.

These ingredients are the classic elements of a stream X-machine, the internal control structure can be of a trivial nature in some cases whereby they can be regarded as basic processing functions. Later on we will consider the issue of a *configurable* component which will offer a convenient basis for system development.

Thus the process of decomposing an enterprise model into components is one of breaking it up into communicating networks of stream X-machines, essentially a top down approach. We could build a system from components by integrating into a coherent X-machine a collection of simpler X-machines, this might be regarded as a bottom up approach. From the point of view of building correct solutions it is then possible to distribute some of the verification and testing effort. This means that we might be looking at determining whether the components were correct at a different time, perhaps in a different place, to the verification of the complete system. Thus we could separate the testing of the components from the testing of their integration in a systematic way. This, of course, happens already, to some extent, systems are often built in parts, the parts tested and then put together. The integration of the parts being tested separately. When this is done it is quite likely that the description of the functionality of the components is incomplete, the way they interact and interface with the rest of the system can be poorly understood and so much of the testing effort may well be misplaced. The goal of a library of well tested, tried and trusted components that can be easily integrated into complete

solutions that need a minimal amount of testing is not going to be achieved under these conditions.

There is another general problem. We noted, above, that there were two broad approaches to designing systems - top down and bottom up. Life is never that simple, however, and we need to examine how a model is built and revised through an iterative process which could, like some theories of evolution, involve major generational mutations and changes as we traverse the development cycles.

4.3 An integrated design and test refinement strategy.

A proposed integrated approach is illustrated in Figure 4.1. In this scheme we develop the initial formal specification from the requirements specification. Outline proof obligations and testing strategies are defined at a high level of abstraction based on the requirements analysis and a hazard analysis where appropriate. It may be possible at this stage to construct outline proofs and to generate outline test scripts based on the detail available at that level of the specification. The refinement process applied to the specification will extend the functionality or reduce the abstraction of the model and provide further opportunities for the description of verification conditions and test strategies at this level of detail. If we can identify the refinement transformations in a formal way we could attempt to refine the proofs and test scripts from the higher level to save repeating these activities where this is not needed. Thus we can transform the model in a controlled and logical way until we reach an implementation, dragging proofs and tests sets with the design. The picture in Figure 4.1 provides a simplified schematic of what might be possible. First the study of the intended application area would include some appropriate hazard analysis, an identification of potential hazards that the system may endure and an initial set of conditions and properties that the system must satisfy. This is a complex and difficult task which cannot not be automated easily. It is beyond the scope of this book but it is included to emphasise the point that there may be overriding conditions imposed on the development of the system which would influence the verification and testing that is to be carried out. For example, if the system was specified as an X-machine it may be possible to identify states which should not be entered under certain conditions of input and memory. These forbidden state/input/memory combinations should be *ensured* by making the design compliant with these requirements.

A word about the idea of formal verification and proof obligations is now pertinent. Each machine defines a computational function which transforms some set of sequences of inputs into corresponding sequences of outputs subject to the initial memory value. As we refine and develop the machine model this function may change. It may have its domain (that is its scope) extended to provide greater functionality in the system. It may be redefined so that it is defined in terms of a related, but not necessarily identical, domain of inputs. During the process of refinement we may reduce the level of abstraction as it relates to the definition of the input set, the output set or the memory. We may increase the number of states in order to control these lower level data structures.

The mathematical properties that the computational function of a solution X-machine must specify will be called *proof obligations*. Demonstrating that a given X-machine satisfies a set of proof obligations is the subject of *formal verification* or *model checking*. This is not the major emphasis of this work, however, we have tried to recognise the fact that the proof obligations of a series of refinements of a machine will be related. Thus our philosophy of computational modelling allows for the precise analysis of the computational functions of the set of refined machines thus providing a foundation for future work on refining formal verification proofs in tandem with the refinement of the model. The benefit is that we would only have to verify or check that part of the model that has changed and this will ease the burden of having to prove everything again.

Our main emphasis is refining test sets. Here there are definite savings to be made. As we will see in Chapter 8 the theory allows us to construct test sets in parallel with the refinement process and to distribute the testing into smaller chunks with a major cost and time saving. Since we are also interested in a genuine component based approach to design we can also fully utilise the prior testing of these components safe in the knowledge that the results from this will be precisely related to our overall testing strategy.

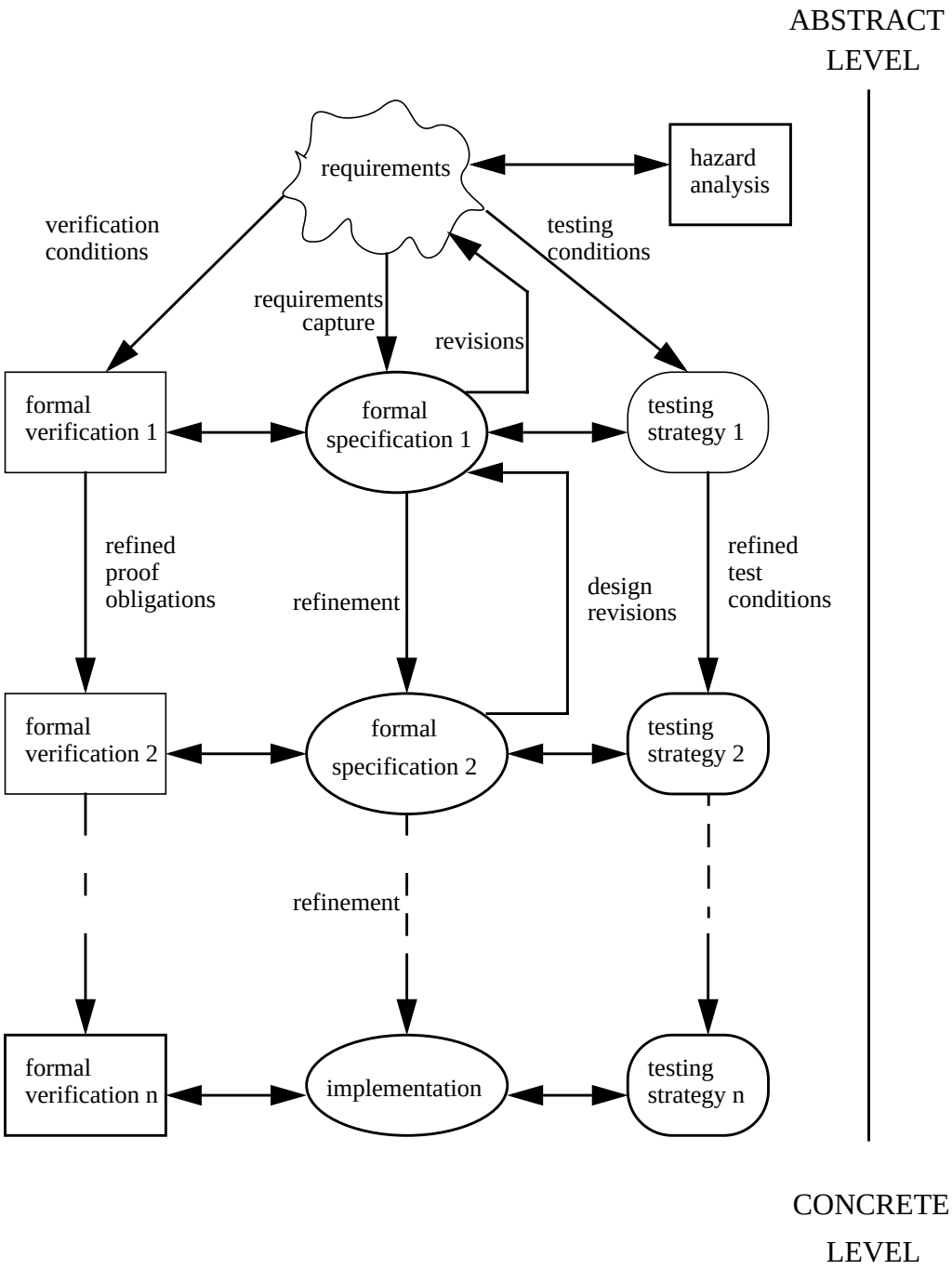


Figure 4.1. An integrated approach to the design, verification and testing of systems.

4.4 A first simple refinement.

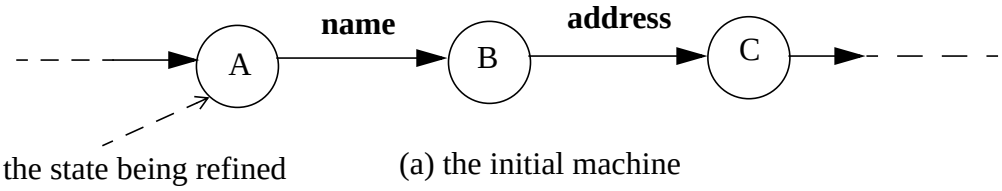
Our refinement strategy is to take an existing stream X-machine and to alter it in some useful way to produce a new machine that includes the relevant behaviour of the original machine together with some desired further properties. Thus we might have a machine which is presented in a very abstract way with little detail of some of the basic functionality or structure

and create a new machine, based on the old one, but with more detail exposed. By doing this in a coherent and consistent way we can then utilise some of the testing strategies that have been developed and produce an integrated design approach that allows testing to be refined alongside the specification. This will lead to a number of savings in the complexity of the testing process and to considerable reductions in effort.

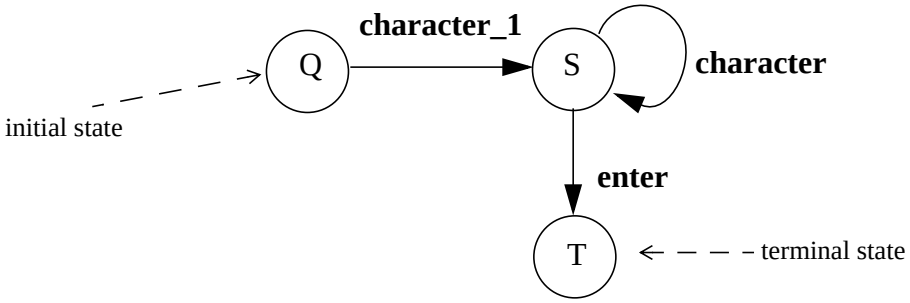
Now we consider the ways in which we can refine machine models. Let us start with a stream X-machine model and look at a transition and its associated basic function. The trigger for this transition will be a set of input-memory pairs. Suppose that we wish to replace this transition with a new stream X-machine. This could be the case if the input trigger to the original transition was of a compound type and we wished to decompose it into a sequence of simpler type values. In Figure 4.2(a) we start with a simple machine that captures the **name** input (a sequence of characters) and stores this in some parameter slot of the memory and then captures the **address** input (also a sequence of characters) and stores this in another part of the memory. To refine the **name** function to a lower level of detail we have to replace it with a machine that takes as inputs characters and allows for a sequence to be built up incrementally. This could be the machine in Figure 4.2 (b), which we call a *refining machine for state A* in the original machine. The final stage is to insert or substitute the second machine into an appropriate place in the original machine to obtain a *refined machine* as in Figure 4.2 (d). This is done by identifying or merging states as shown in Figure 4.2 (c).

Now, the problems that we might find include upsetting the semantics of the original machine in undesirable ways and introducing non-determinism. In the original machine, for example, the user types in a name, leaves the state A and then types in an address. There is no means of getting back to state A directly after the name has been typed in, to type in another name, for example. If there was a transition from T to Q in the refining machine then there would be a transition from B(T) to A(Q) in the refined machine which could mean that the user could type in a name and then type in another before going on to the address. This disturbs the semantics of the machine in an unsatisfactory way. Another reason for avoiding transitions leaving from terminal states in refining machines is that we may have the possibility that doing this sort of operation could disturb the deterministic nature of the original machine. For example, if the refining machine included a transition from state T back to state Q then it is possible that there was a non-deterministic situation in the state B(T) caused by the domains of the functions on the transitions leaving that state intersecting in a non-trivial way. To avoid this in general we only allow refining machines to have a single terminal state (with no transitions leaving it).

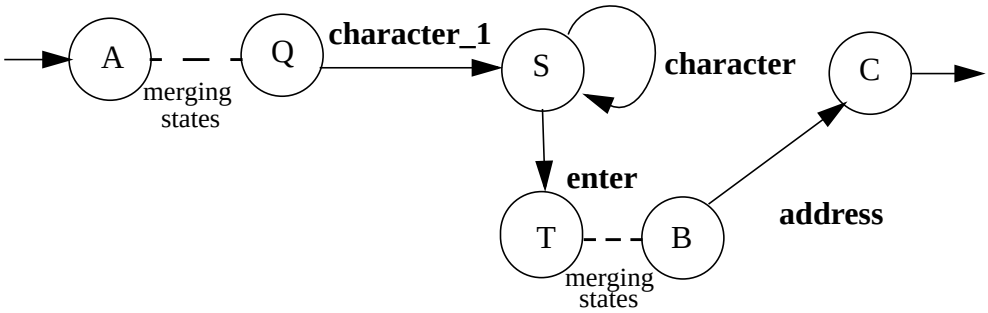
The construction described could be repeated with other transitions from the same or other states. These also could cause problems if done in an incorrect way. Thus we require that when several states are being refined that the refining machines do not share states.



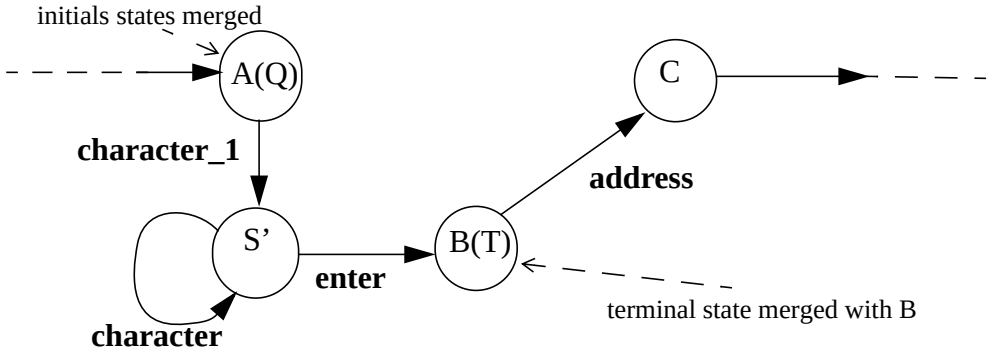
(a) the initial machine



(b) refining the transition **name** into a refining machine.



(c) machines about to be combined and states about to be merged.



(d) a refined version of the original machine.

Figure 4.2. A simple refinement of a stream X-machine.

Once a refinement has been carried out we want to ensure that the original machine and the refined machine behave in a suitably related way. So, for example, a path in the original machine which is traversed when a suitable sequence of inputs is applied should correspond to a suitable path in the refined machine. Naturally, the input sets may have changed and this may mean that the path in the refined machine is longer and its output sequence may not exactly match that of the original machine. This means that we have to define a clear relationship (function) between the input alphabets of the two machines, between the output alphabets of

the two machine and between their memories. Then the paths can be compared in a meaningful way as illustrated, in a general sense, in Figure 4.3.

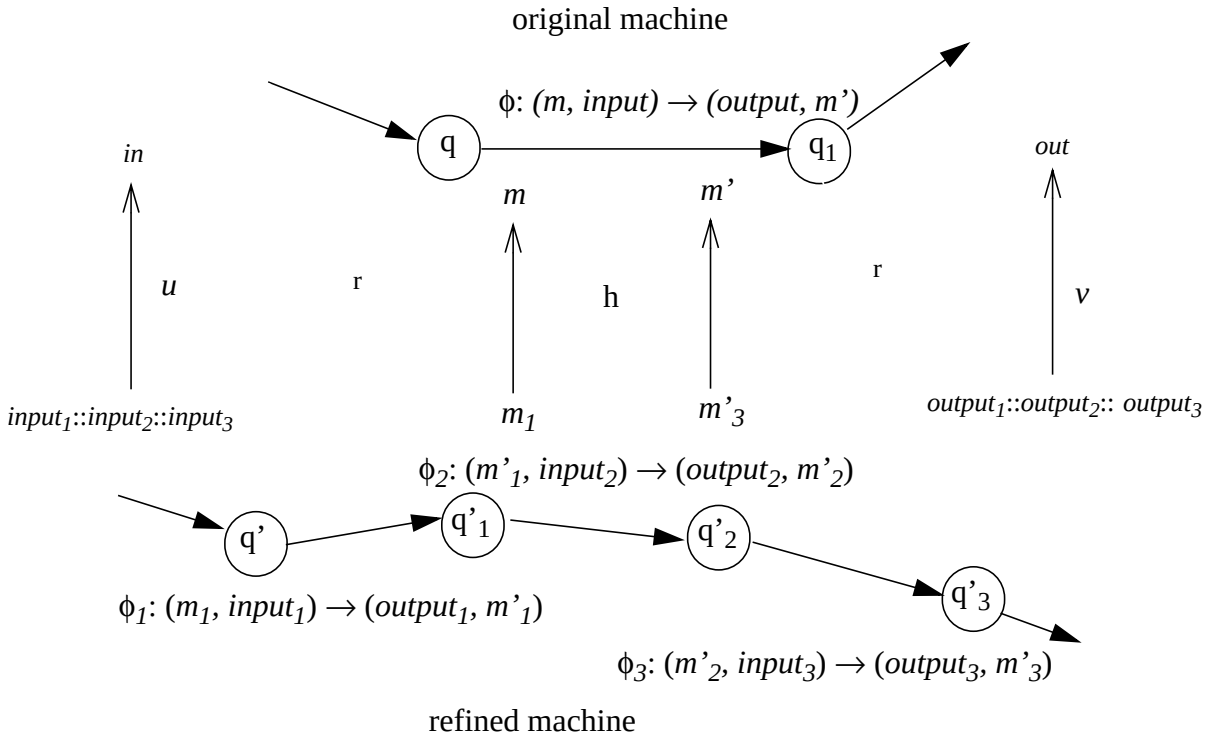


Figure 4.3 Comparing the behaviour of the refined machine at a state with that of the original machine.

Suppose that we have a Stream X-machine with state set *Q*, and input set *Input*, output set *Output*, memory set *Memory* and basic functions Φ. Let $q \in Q$ and suppose ϕ is a transition that leaves the state *q*. We wish to refine this transition from *q*. The refining machine for this will have a state set R_q, and input set *Input'*, output set *Output'*, memory set *Memory'* and basic functions Φ_q. The input set *Input'* is likely to include sequences of inputs that correspond to single elements of *Input.*

The relationship between the two input sets will then be described by a partial function

u : seq(*Input'*) → *Input.*

In a similar way the sequences of outputs of the refining machine may relate to a single output of the old machine so there needs to be a partial function v : seq(*Output'*) → *Output*.

There is a partial function h : *Memory'* → *Memory.*

We call the functions *u, v* and h the *translation functions of the refinement.*

For the case where there is only one transition leaving q in the original machine we can proceed as follows. The refining machine will have one terminal state with no transitions leaving it. The set of states of the refining machine is denoted by R_q. The terminal state is t_q.

Now we paste the refining machine into place by removing the arc ϕ in the original machine and fit the refining machine in its place. The new machine will have a set of states that comprises all of the states of the original machine except q together with the new states of the refining machine with t_q identified with the target state of the ϕ transition and q identified with the initial state of the refining machine.

We define a function $r : Q \rightarrow \mathrm{K}$ to make these relationships explicit. K, the set of *key states*, are a copy of the states of Q and the function r maps Q onto them. In some cases we will have K = Q, as in the following examples.

The situation is slightly different if the state q has several transitions leaving it. In this case we must make sure that our refining machine has as many terminal states as there are transitions leaving q. Furthermore, none of these terminal states can have transitions leaving them. In this case the pasting of the refining machine into the original machine must be done in such a way that the transitions from the state q in the first machine are replicated by transitions in the refining machine and the merged machine is such that the corresponding terminal states in the refining machine are combined with the relevant states in the original. The next example illustrates this.

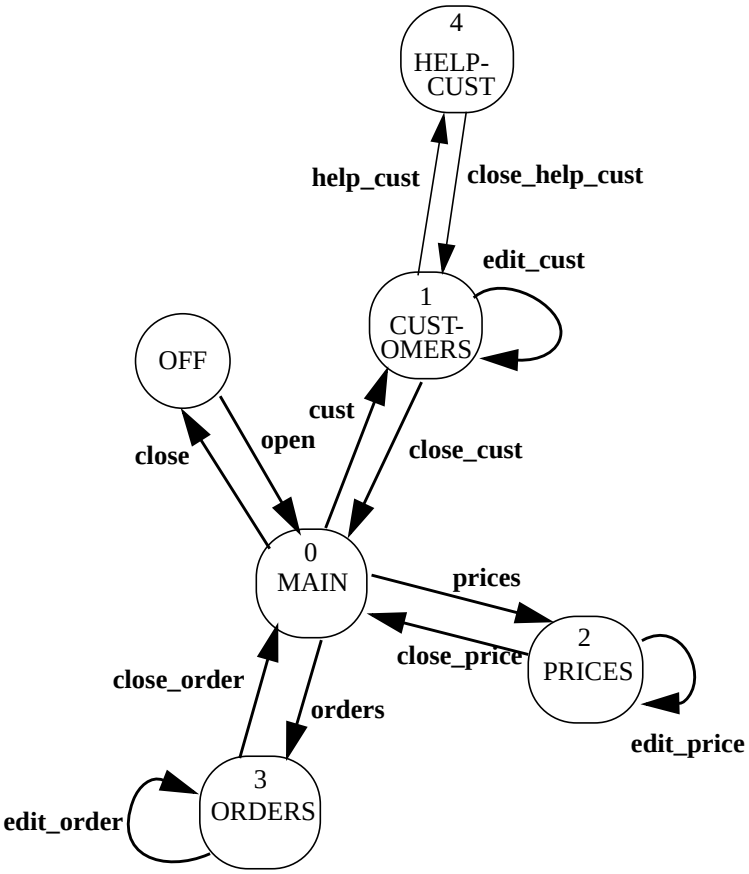


Figure 4.4 A top level view of the Estimator system.

Consider the Estimator example from Chapter 2 illustrated in Figure 4.4 and the process of introducing the CUSTOMERS screen. At the top level we have a state with a rather general and at present incompletely defined function, **edit_cust** defined at the state CUSTOMERS. The purpose of the refinement is to introduce the detailed description of this function and to relate it to a slight simplification in Figure 4.5 of the CUSTOMERS screen derived in Chapter 2. It is a simple matter to alter things to include the full details of the original screen. We have removed some of the input details to make the later analysis clearer. So we will build a refining machine for CUSTOMERS and then paste it into the original machine to produce a refined machine.

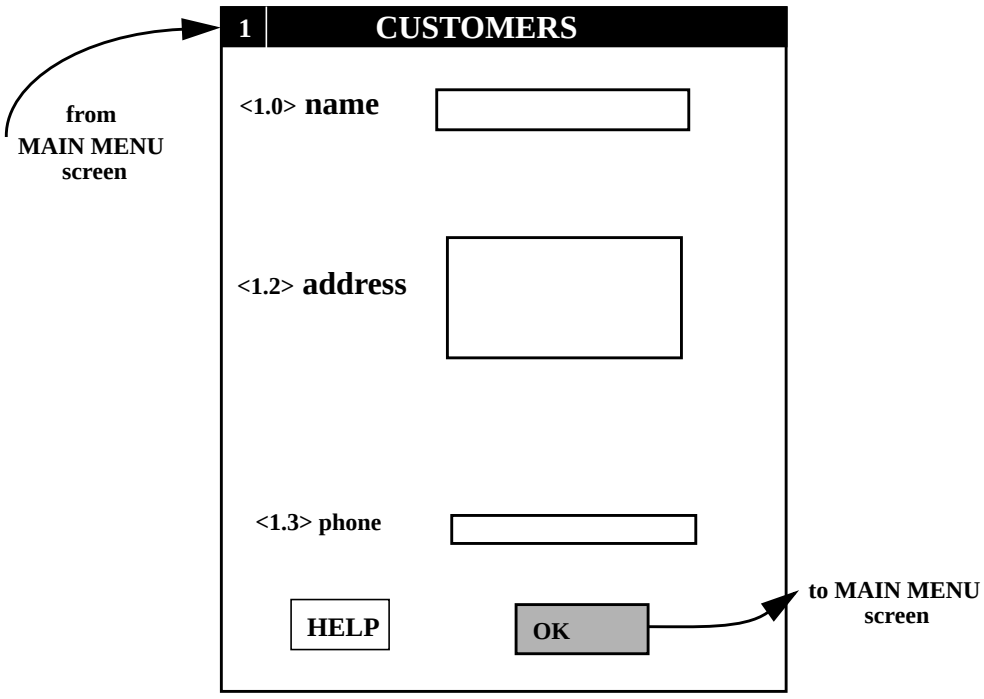


Figure 4.5 The simplified CUSTOMERS edit screen.

The approach is to use a type of refinement that is discussed, from a detailed theoretical point of view, in Chapter 6.

The state in the original machine that we are refining is CUSTOMERS. We need to construct a refining machine and then relate or insert it into the original machine to obtain the refinement. This process requires us to define a number of functions that will indicate how the states in the machines are related and functions to translate between the old and new inputs and output sets.

The set of key states in this case is the original state set - {OFF, MAIN, CUSTOMERS, ORDERS, PRICES, HELP_CUST}.

We also let *Input = {1, 2, 3, off, on, close_1, close_2, close_3, cust_details, edit_price, edit_order, help, close_h}*, as before, and introduce the new input events associated with a simplified version of the CUSTOMERS screen, *Input' = {1, 2, 3, off, on, close_1, close_2, close_3, name, address, phone, OK, edit_price, edit_order, help, close_h}*.

The next stage is to relate these two input alphabets using a function defined on certain sequences of the CUSTOMERS screen. The former input that causes the **edit_cust** function to

operate, *cust_details,* has been replaced by the inputs *name, address, phone and OK* in the new screen.

So we put u: seq(*Input'*) $\rightarrow$ *Input* by defining:

u(*name*::*address*::*phone*::*OK*) = *cust_details.*

$u(1) = 1$, $u(2) = 2$,...u(*close_3*) = *close_3*, u(*price_details*) = *price_details*, ... u(*close_h*) = *close_h*.

A similar function is needed to relate the outputs of the old, unrefined, machine and the new. We must identify the output sets for the two machines, first.

Let *Output* = {*screen_1, screen_2, screen_3, screen_0, screen_1', screen_2', screen_3', screen_0, clear, screen_help*}

So *screen_1* consists of the customer screen with the slots blank, *screen_1'* is the same screen with all the values in the slots.

Also, *Output'* = {*screen_1, screen_2, screen_3, screen_0, screen_1_name, screen_1_address, screen_1_phone, screen_1_OK, screen_2', screen_3', screen_0, clear, screen_help*}

Here *screen_1_name* is the *screen_1* with the value of the input name inserted in the name slot, *screen_1_address* is the customer screen with values in the slots for name and address, etc..

Now define v: seq(*Output'*) $\rightarrow$ *Output* by:

v(*screen_1_name* :: *screen_1_address* :: *screen_1_phone* :: *screen_1_OK*) = *screen_1'*

v(*screen_1*)= *screen_1*, v(*screen_2*) = *screen_2*, ..., v(*screen_help*) = *screen_help*.

Then *Memory* = *CUSTOMER_STORE* $\times$ *PRICE_STORE* $\times$ *ORDER_STORE*

and

CUSTOMER_STORE = seq*(CHAR),*

PRICE_STORE = seq(*CHAR),*

ORDER_STORE = seq(*CHAR),*

whereas the new machine memory is:

$Memory' = CUSTOMER\_NAME \times CUSTOMER\_ADDRESS \times CUSTOMER\_PHONE \times PRICE\_STORE \times ORDER\_STORE$

with

$CUSTOMER\_NAME \times CUSTOMER\_ADDRESS \times CUSTOMER\_PHONE = seq(CHAR) \times seq(CHAR) \times seq(CHAR)$

We have to relate the memory sets of the two machines so we define h: $Memory' \rightarrow Memory$ by $h(a, b, c, d, e) = (a :: b :: c, d, e)$ where $a \in CUSTOMER\_NAME$, $b \in CUSTOMER\_ADDRESS$, $c \in CUSTOMER\_PHONE$ and $a :: b :: c \in CUSTOMER\_STORE$, $d \in PRICE\_STORE$, $e \in ORDER\_STORE$.

The functions are defined as follows:

cust((-, -, -),*cust*) = (*screen_1*, (-, -, -));
close_cust((*cust_details*, -, -), *close_cust*) = (*screen_0*, (*cust_details*, -, -));
edit_cust((-, -, -), *cust_details*) = (*screen_1'*, (*cust_details*, -, -))
open((-, -, -), *open*) = (*screen_0*,(-, -, -));
prices((-, -, -) , *prices*) = (*screen_2*, (-, -, -));
etc.

The inputs like *cust_details* are being treated as a high level abstraction. In the refined machine such inputs are decomposed into lower level abstractions such as *name*, *address*, etc.. These can, at a later refinement stage, be refined into strings of a given type.

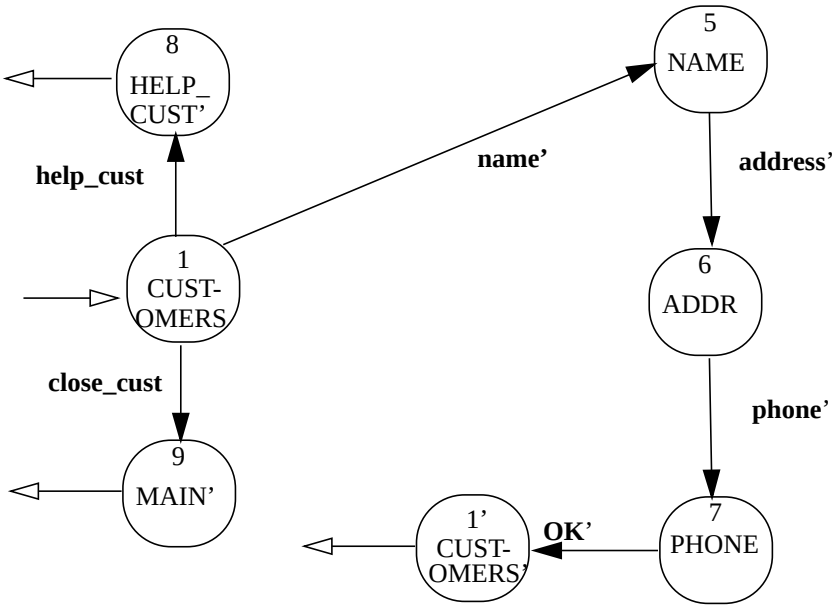


Figure 4.6 A refining machine for CUSTOMERS.

In Figure 4.6 the new functions are:

name'((-, -, - , -, -), *name*) = (*screen_1_name*, (*name*, -, -, -, -));

address'((*name*, -, - , -, -), *address*) = (*screen_1_address*, (*name*, *address*, -, -, -));

phone'((*name*, *address*, - , -, -), *phone*) = (*screen_1_phone*, (*name*, *address*, *phone*, -, -));

OK'((*name*, *address*, *phone*, -, -), *OK*) = (*screen_1_OK*, (*name*, *address*, *phone*, -, -));

etc.

We are restricting the user somewhat into typing in the customer details in a fixed order namely *name*, *address*, *phone* and *OK*. This is simply for the sake of keeping the model simple.

Here $R_{CUSTOMERS}$ = {CUSTOMERS, NAME, ADDRESS, PHONE, MAIN', HELP_CUST', CUSTOMERS'},

$T_{CUSTOMERS}$ = {CUSTOMERS', MAIN', HELP', PHONE},

K = {MAIN, PRICES, ORDERS, CUSTOMERS, HELP_CUST, OFF}, i.e. the set of states of the initial machine.

L= {MAIN', PRICES', ORDERS', CUSTOMERS', HELP_CUST', OFF'}, i.e. all states are primed.

r is the identity.

d(q) = q' for all states of K.

The initial memory of the refining machine is arbitrary in cases where the refined state is not the initial state.

Then *M* '(CUSTOMERS) = (*Input'*, *Output'*, $R_{CUSTOMERS}$, *Memory'*, $\Phi_{CUSTOMERS}$', $F_{CUSTOMERS}$, *r*(CUSTOMERS), $m_{CUSTOMERS}$', $T_{CUSTOMERS}$) is a refining machine for CUSTOMERS and the third machine, in Figure 4.7 is a state refinement of the first.

To check this consider the following operation:

edit_cust((-, -, -), *cust_details*) = (*screen_1'*, (*cust_details*, -, -))

and compare it with the refining machine path:

name'; **address'**; **phone'**; **OK'**

which transforms ((-, -, - , -, -), *name*) into (*screen_1_phone*, (*name*, *address*, *phone*, -, -))

First consider an input *cust_details* applied to the original machine in state CUSTOMERS, with the memory value (-, -, -). Now *cust_details* = *u*(*name*::*address*::*phone*::*OK*) and (-, -, -) =

h(-, -, -, -, -). Also *screen_1' = v*(*screen_1_name* :: *screen_1_address* :: *screen_1_phone* :: *screen_1_OK*).

Therefore applying first the input sequence *name*::*address*::*phone*::*OK* to the refining machine, *M '*(CUSTOMERS) will yield the output - memory pair:

(*screen_1_name* :: *screen_1_address* :: *screen_1_phone* :: *screen_1_OK*, (*name*, *address*, *phone*, -, -))

which can be mapped onto:

(*screen_1'*, (*name*::*address*::*phone*, -, -) by applying *v* to the first component (the output) and h to the second (the memory). Thus the desired result is achieved.

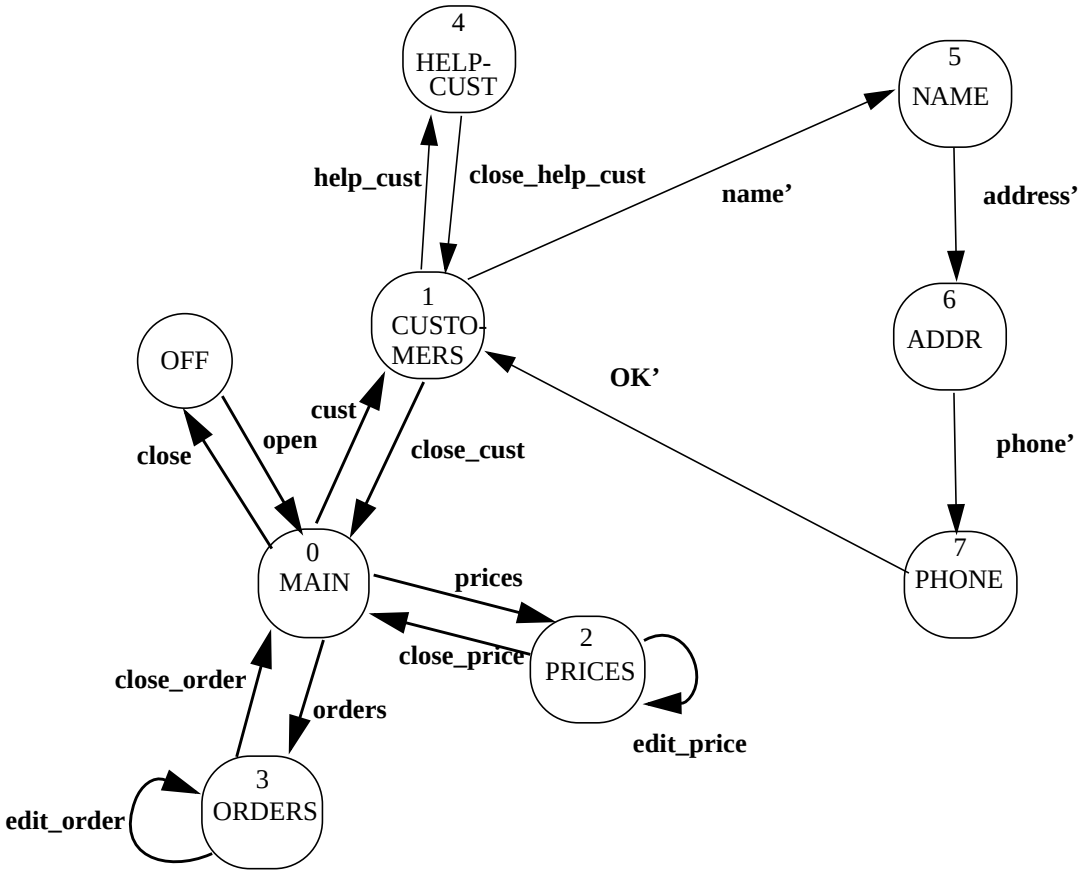


Figure 4.7 A view of the Estimator system with edit_cust refined into a submachine.

What can we say about this refined machine?
Firstly, it has a precise relationship with the original machine in the sense that we can relate the semantics clearly through the covering relations that hold between the input and output alphabets and the memory. The theory in Chapter 6 demonstrates that the state refinement we have

constructed is actually a *functional refinement*. This relates the functions computed by the machines in a transparent manner. If the original machine computes a function f : seq(*Input*) → seq(*Output*) and the refined machine computes the function f' : seq(*Input'*) → seq(*Output'*) then these are related by the diagram:

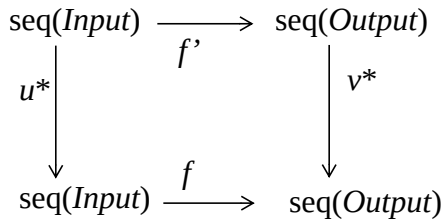


Figure 4.8 Functional refinement.

where u^* is the sequential generalisation of u : seq(*Input'*) → *Input* and v^* is the sequential generalisation of v : seq(*Output'*) → *Output.* The diagram means that for any $x \in$ seq(*Input'*)

$$v^*(f'(x)) = f(u^*(x))$$

Once we have understood how to refine a state and an associated processing function it is an easy matter to consider a more general refinement process where we carry out several such refinements in one go. Naturally, one has to be careful to keep the refining machines separate and to integrate the refinements in a consistent way. A complete definition of how this can be done is covered in Chapter 6.

4.5 Testing refined machines.

There is another benefit when we come to examine the test sets for the refined machine. There are two possible strategies. We could generate the complete stream X-machine after it has been fully refined and then apply the complete functional testing method in one go. This will involve generating a single, and potentially very large, test set as we described in Chapter 3. The alternative approach is to generate test sets for the the refining machines separately, test them and then construct a testing strategy for the integration of these into the initial top level machine. The theory behind this approach is explained in Chapter 8.

The procedure is as follows. First we test an implementation of the refining machine displayed in Figure 4.6. Naturally, we must ensure that the design for test conditions are satisfied. We must also consider how the component is to be integrated into the top level system. It is necessary that the implementation contains some mechanism for indicating to the top level machine whether it is in its terminal state. This is done through the use of *transfer variables*. What we do is define an extra basic function in the refining machine at its terminal state which is a loop transition that, when fired, indicates that the terminal state has been reached. The detailed definition and examples are in Definition 8.2.1 and Figure 8.2. The test set for this part is determined according to the method in Chapter 3. In the general case this loop transition augmentation has to be done in *each* refining machine.

Let us now assume that this has been done and this component (implementation of the refining machine) has passed the tests. We now proceed to testing the complete system. The complete system has been implemented using the components integrated into a complete implementation using the state refinement method. We thus have a potential implementation for the machine in Figure 4.7.

The complete test set that will be applied to this compete implementation is of the form:

$Y_r = Y' \cup U_k$,

where Y is a test set of the original (unrefined machine) and Y' is a set of input sequences in the refined machine that are constructed from Y and U_k is a special set of test inputs known as a k-distinguishing set of the refining set. We will explain how these are constructed in the case of the Estimator example, where MAIN is the initial state.

4.5.1. Step 1 - generating Y:

Y is t(X),

where $X = T(\Phi^k W \cup \Phi^{k-1} W \cup ... \cup W)$

Now, S = {< >, **close**, **orders**, **prices**, **cust**, **cust** :: **help_cust**} is a state cover of the original machine thus $T = S \cup S \bullet \Phi$ hence

$T = \Phi \cup \{\textbf{close}\} \bullet \Phi \cup \{\textbf{orders}\} \bullet \Phi \cup \{\textbf{prices}\} \bullet \Phi \cup \{\textbf{cust}\} \bullet \Phi \cup \{\textbf{cust} :: \textbf{help\_cust}\} \bullet \Phi$

Now let W = {**edit_order**, **edit_price**, **edit_cust**, **open**, **cust**}

If k = 1 then

$X = [\Phi \bullet W] \cup [\{\textbf{close}\} \bullet \Phi \bullet W] \cup [\{\textbf{orders}\} \bullet \Phi \bullet W] \cup [\{\textbf{prices}\} \bullet \Phi \bullet W] \cup$
$[\{\textbf{cust}\} \bullet \Phi \bullet W] \cup [\{\textbf{cust} :: \textbf{help\_cust}\} \bullet \Phi \bullet W] \cup [\Phi^2 \bullet W] \cup [\{\textbf{close}\} \bullet \Phi^2 \bullet W] \cup$
$[\{\textbf{orders}\} \bullet \Phi^2 \bullet W] \cup [\{\textbf{prices}\} \bullet \Phi^2 \bullet W] \cup [\{\textbf{cust}\} \bullet \Phi^2 \bullet W] \cup$
$[\{\textbf{cust} :: \textbf{help\_cust}\}\Phi^2 \bullet W]$

Thus $Y = t$(X) which is obtained, in this case, by replacing the functions by their corresponding input (button press event) so **orders** is translated into orders and so on. In general this may not be the case and the test function will have a more complex definition.

4.5.2. Step 2: refining Y to Y' - this is done by replacing any occurrence of the input cust_details with the sequence name :: address :: phone :: OK

4.5.3. Step 3: constructing the k-distinguishing set.

S = {< >, **close**, **orders**, **prices**, **cust**, **cust** :: **help_cust**} is a state cover of the original machine. If we take k = 1 then (see Chapter 8) we identify all the paths that start at the initial state of the original machine and which comprise either an element from the state cover or an element of the state cover followed by a basic function.

P_1 = {< >, **close**, **orders**, **prices**, **cust**, **cust** :: **help_cust**, **close** :: **open**, **orders** :: **edit_order**,
orders :: **close_order**, **prices** :: **close_price**, **prices** :: **edit_price**, **cust** :: **edit_cust**,
cust :: **close_cust**, **cust** :: **help_cust** :: **close_help_cust**}

Also we construct distinguishing sets (Definition 8.1.2) for the refining machines. These are sets of sequences of basic functions that will distinguish the initial state of the refining machine from the rest. In this case they are:

$X_{CUSTOMERS}$ = {**name'**}
for the refining machine of Figure 4.6 and
X_{MAIN} = {**close**}
X_{OFF} = {**open**}
X_{PRICES} = {**edit_price**}
X_{ORDERS} = {**edit_order**}
X_{HELP_CUST} = {**close_help_cust**}
for the others since all the refining machines except one are trivial

Now we can build U_1 .Consider a path from the set P_1 and let q be its end state. The path defines a sequence of inputs s^* from Φ through the application of the fundamental test function for the top level machine. There is a sequence s^{*}' of inputs in the refining machine that are mapped by u onto s^* . We will combine this input sequence with the sequences obtained by applying the test function of the initial state of the refining machine to the distinguishing sets for each state in the original machine to create U_1.

U_1 = { *cust* :: *name* :: *address* :: *phone* :: *OK* :: $t_{CUSTOMERS}$(**name'**),
cust :: *help_cust* :: t_{HELP_CUST}(**close_help_cust**),
prices :: *edit_price* :: t_{PRICES}(**edit_price**),..... }

= *cust* :: *name* :: *address* :: *phone* :: *OK* :: *name*,
cust :: *help_cust* :: *help_cust*,
prices :: *edit_price* :: *edit_price*,..... }.

What we are doing here is exercising paths in the completed system which connect the component machines to the top level machine. In other words we are testing the integration of the component. Hence the path defined by the input sequence (for any initial memory value) *cust* :: *name* :: *address* :: *phone* :: *OK* :: *name* is accessing the component, moving through it to exercise the refined path (function), returning to the main machine and then accessing the component initially again. The other input sequences in U_1 are exercising the rest of the original machine to establish that the CUSTOMERS component has not disturbed the functionality of the rest of the machine. In a more complex refinement these paths would also be exercising the other refinements using other refining machines. So, for example, if we had refined ORDERS and PRICES in a similar way to CUSTOMERS we would have seen sequences representing the access and traversal of these components as well.

From the theory in Chapter 8 we see that the combination of testing the components separately and testing their integration in this way is sufficient to fully test the complete system.

4.6 Components revisited.

We mentioned, earlier, that a component would be regarded as a stream X-machine or a basic process function. In the refinement of the Estimator example we have seen the way in which a stream X-machine, the refining machine was involved in the process of building a more complete solution. Here is an example of a potential component. Suppose that we were constructing the Estimator model and we wished to create the refinement which replaced the high level function carried out by the **edit_cust** operation. We identified that the refining machine needed to have an alphabet that decomposed part of the original alphabet and processing functions that extracted inputs of a suitable type and stored them in a suitable memory structure. The outputs were also related and involved the packaging of the user interface for this refining machine into a suitable screen layout and display. All of this could be *implemented* in a *local* package with the relationships between the original machine and the refining machine described by the translation functions *u, v* and *h*.

A component of this type consists of a stream X-machine with a specified input set, output set, memory set, basic processing functions and the state structure *together* with a well tested implementation. The initial state and the terminal states must also be specified. Such a package could be available in many different programming and hardware description languages. The way that it was integrated into the original machine would be determined by the state being refined and the translation functions. It is a straightforward matter to implement a stream X-machine where the input and output sets are described in conveniently implementable data structures as is the memory. The control structure of the machine provides the overall control architecture of the program and the basic functions are implemented directly from their definitions. A refinement will involve interfacing the refining machine or component to the master program by using the translation functions and replacing the original basic function calls by calls to the component.

Many components are likely to be very similar. We have seen the use of a component which extracted input information and stored it under the label of *name, address, phone*. Thus there were three data slots. Each slot was defined with respect to a specific data type, seq(CHAR). It would be quite feasible to have another component with the same basic structure but which stored the resultant input data under different labels e.g. *reference_number, model_type, price* if the system was a sales record system. Thus the component could be reused if there was a simple way of changing the names of the data slots as well as the names displayed on the user interface screens.

We could take it further and envisage components which allowed us to define the number of slots that were available, the type of each slot and its name. A generic stream X-machine architecture could then be constructed which would amount to a *configurable component*.
In this case the configurable component could be available as a software package with the information presented in a diagram something like the one in Figure 4.9.

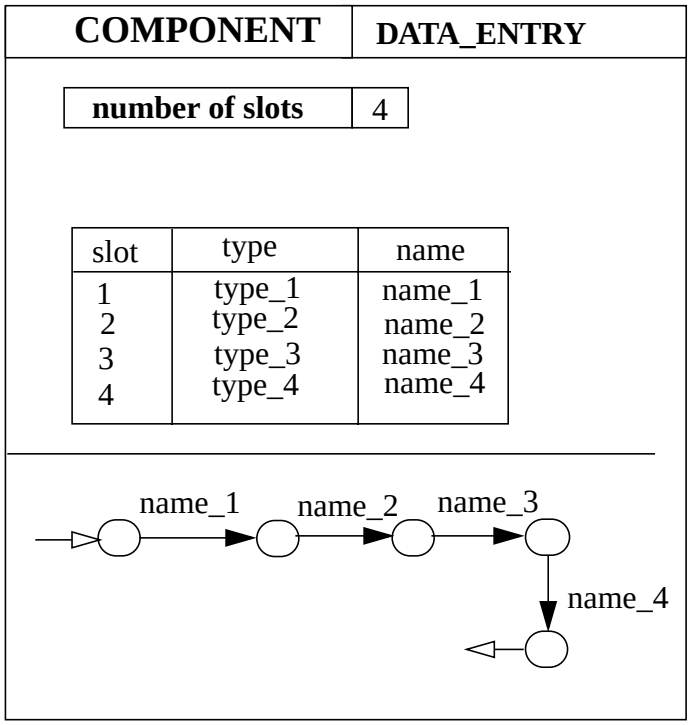


Figure 4.9 A component specification box.

together with definitions of the types and suitable algorithms or functional descriptions of the processing functions. Automatically configured test sets would be available for such components and the use of these would be integrated into the testing strategy of the refined system.

The idea of a configurable component can be taken much further. However, we must not compromise the complete description as a stream X-machine if we are to capitalise on the philosophy of correct system construction and the integrated design and test approach. The use of well tested and easily testable configurable components, which would be available as software modules with stream X-machine descriptions, is a natural direction for software and hardware engineering to take, rather than the seemingly unreliable world of classes and objects and other ill defined “component” approaches.