Structuring Software Functional Requirements For Automated Design And Verification

Sukhamay Kundu
Computer Sc. Dept, Louisiana State University
Baton Rouge, LA 70803
(kundu@csc.lsu.edu)

Abstract

We propose a new domain model for representing software functional requirements that can help the subsequent phases of design, development, and testing. The model consists of a use-relationship between the operations and the data-items and a finite-state model for the valid sequences of those operations. We show how this model can help us in the analysis and structuring of the requirements via the identification of submodels, which in turn can simplify the design and other phases. We also give an algorithm to compute the use-relationship for an object-oriented source-code as a reengineering step for requirements verification. We give several detailed examples.

1. Introduction

It is well-known that a proper formulation of the requirements plays a critical role for a software project's success and the final shape of the software [1, 6]. However, we do not yet have a proper method for describing requirements although many languages have been proposed for this purpose [3, 5]; we also lack proper methods for analyzing requirements. Since the software requirements have multiple facets due to different viewpoints of customers, users, and developers, we may need different languages to address them and also need different methods for their analyses. In particular, they should allow us to express the requirements in a form that is suitable for automation of the design phase as much as possible and that allows us to determine the level of compliance between a software and a given set of requirements.

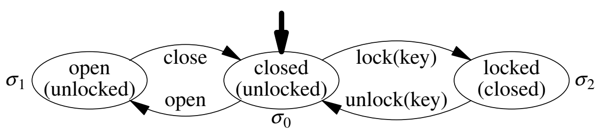
From a user's viewpoint, the software requirements specify the operational or functional part of the software. It gives the high-level queries (operations) for which the user needs answers (resp., responses) from the software based on the data which the user makes available to the software for that purpose. A good method for describing the functional requirements and their analyses can be very helpful in the design, implementation, and testing of a software because of their direct impact, in spite of the fact that functional requirements do not address certain other requirements like real-time constraints and safely features. We focus here on the methods for specifying functional requirements. The formal language Z [7] is too complex for most users for this purpose. It requires us to specify pre- and post-conditions for each function in a precise and exact form, which are needed for formal correctness-proof of a given design, but as such they do not help us in determining the software architecture or the software design. Moreover, since we cannot algorithmically determine the pre- and post-conditions from the source-code of a function and nor can we algorithmically verify that a source-code is in agreement with the given post-conditions, assuming that the pre-conditions hold, this limits the use of Z in practice.

What we need is a more light-weight method for specifying functional requirements that can be easily used and that can help the software design phase in a significant way. We present such a method based on: (1) a use-relationship $U(F, V)$ between the major operations (functions) F corresponding to the user-queries or user-actions and the data-items V which affect their interactions, and (2) a finite-state model $M(F)$ which shows the valid sequences of the operations F. We can think of $U(F, V)$ as a static model of the problem domain and $M(F)$ as a dynamic model. We also show how $M(F)$ and $U(F, V)$ can be used together in structuring the requirements by identifying subsystems, which allow us to create a design for the total system from those for the subsystems.

Since the requirements and design represent certain high-level features ("what") of a software, we should be able to extract the relevant information by filtering out from the source-code the low-level algorithmic operations oriented ("how") information. Note that the pre/post-conditions of a function that are used in Z cannot be determined in this way from a source-code. A simple form of the "what" information is the use-relationship between a function and the various data-items, disregarding how and when a data-item is used. This suggests the use of $U(F, V)$ in the functional requirement specification, which shows for each function $f \in F$ the read/write uses of various data-items (variables and other special constants). We write $V(f)$ for the data-items used by f. If f has parameters, then $V(f)$ gives the set of all data-items that might be used by f for some combination of values for its parameters. We exclude from $U(F, V)$ the low-level functions that may be introduced in the implementation and we also exclude the low-level data-items which represent results of intermediate computations by those functions. For our purpose, V consists of the data-items which are related to the interactions among the functions F and as we will see a proper choice of V is important in submodel identification and hence the design process. In [4], we had shown how to determine an appropriate class-hierarchy for an object-oriented design from $U(F, V)$.

2. A Simple Design Problem and Its Requirements

Suppose we want to design a software for operating a door with a lock using the operations F = {open, close, lock, unlock}. Fig. 1(i) shows the finite-state model $M(F)$. For example, the door cannot be locked unless it is closed and it cannot be opened unless it is unlocked. The parameter "key" in the lock/unlock operations indicates that these operations require a key. Assume that a door is delivered by the manufacturer in the closed and unlocked position for safety, and thus we take the state σ_0 = "closed (unlocked)" in Fig. 1(i) as the start-state of $M(F)$. We assume here that a door can be unlocked from either side irrespective of the side from which it is locked. We remove this restriction later in Section 3 where we consider a more complex design problem.



(i) The finite-state model $M(F)$; start-state = σ_0.

Use-relationship $U(F, V)$	v_1: openClosed-Status	v_2: locked-Status	v_3: orig-Key
f_1: Open()	x(rw)	x(r)	
f_2: Close()	x(rw)		
f_3: Lock(key)	x(r)	x(rw)	x(r)
f_4: Unlock(key)		x(rw)	x(r)

(ii) Here, $Val(v_1)$ = {open, closed}, $Val(v_2)$ = {locked, unlocked}, v_3 is a constant, r = read, and w = write.

Figure 1. The finite-state model and the use-relationship for a door with lock.

Fig. 1(ii) shows the use-relationship $U(F, V)$. Note that $V = \{v_1, v_2, v_3\}$ does not include data-items which are specific to the individual operations. For example, the open-operation may use other data-items (e.g., whether the door is a sliding-door or a regular-door and whether it opens to the left or right) that can affect the detailed algorithms of the open-operation. But we do not include such data-items in V because they do not affect the overall software design in regard to the interactions among F. The data-item v_3 (a constant) is included here only because of its relationship to the parameter "key" in lock/unlock functions; its deletion will not affect the submodels or the class-hierarchy design. Note that the variable V must be able to distinguish the states in the finite-state model. In Fig. 1, $\{v_1, v_2\}$ serves this purpose; for example, the state σ_1 = "open" corresponds to $\{v_1$ = open, v_2 = unlocked$\}$. The variables involved in representing the start-state must be initialized before the system can be used; In Fig. 1, v_3 = origKey also needs to be initialized. The use-relationship gives a simple (partial) model of the software's working environment by showing some of the inputs and outputs of each operation in F. A state change occurs when some data-items in V are modified by a function in F.

2.1. Importance of choosing appropriate functions F

We should not combine two or more functions in the requirements into a single function by introducing parameters as it may adversely affect the design phase. Such decisions should be delayed to the implementation phase or the late stages of design. For example, suppose we combine f_1 = Open() and f_2 = Close() into a single function f_{12} = OpenClose(oneOrTwo) using the parameter "oneOrTwo", where $f_{12}(1)$ corresponds to f_1 and $f_{12}(2)$ to f_2. This causes the removal of f_2-row from $U(F, V)$ and relabeling of f_1-row by f_{12}, and hence some loss of information that prevents us from obtaining the best class-hierarchy shown in Fig. 3(iii) based on a factoring of both f_1 and f_3 (Figs. 3(i)-(ii) use factoring of only one of them).

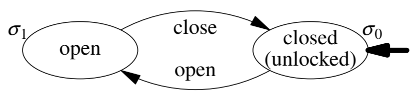
2.2. Importance of choosing appropriate data-items V

In general, it is better not to reduce the number of data-items in V at the outset; the method in [4] can show which data-items in V can be merged into a single one. If we merge v_1 and v_2 in Fig. 1(ii) into a variable v_{12}, with values $Val(v_{12})$ = {"open", "locked", "unlocked"}, then although v_{12} can represent the states in $M(F)$ it causes some loss of information in $U(F, V)$; all functions in F now have the same entry "x(rw)" in the v_{12}-column making the rows for f_1 and f_2 identical and likewise for f_3 and f_4. This means we no longer have the two submodels of Fig. 1(i) shown in Section 2.3 and we cannot derive the best class-hierarchy shown in Fig. 3(iii).

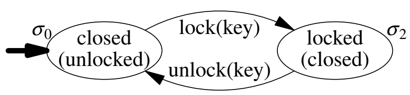
2.3. Structuring functional requirements

A good functional requirement specification method should allows us to determine the subsystems of a large and complex specification so that we can combine the designs for the subsystems to obtain a design of the total system. We show how this can be done with our specification method using the example in Fig. 1. (See Section 3 for a more complex design problem.)

Figs. 2(i)-(ii) show two submodels M_1 and M_2 of the finite-state model in Fig. 1(i). Fig. 2(i) gives the valid sequences of operations for the door part without the lock, with the only operations F_1 = {open, close} $\subset F$. Likewise, Fig. 2(ii) shows the submodel for the lock part without the door, with the only operations F_2 = {lock(key), unlock(key)} $\subset F$. We can think of Fig. 1(i) as a combination of these two submodels. Note that if we were to let σ_1 be the start-state in Fig. 1(i), then σ_1 would also be the start-state in Fig. 2(i) but the start-state in Fig. 2(ii) will remain σ_0 as before. This is because in Fig. 1(i) we can reach the states $\Sigma_2 = \{\sigma_0, \sigma_2\}$ of M_2 from σ_1 only via σ_0.



(i) Finite-state model M_1 for a door without a lock part.



(ii) Finite-state model M_2 for a lock without a door part.

Figure 2. Two submodels of the finite-state model in in Fig. 1(i); in each submodel, start-state = σ_0.

The use-relationship for the submodel M_1 in Fig. 2(i) is $U_1(F_1, V_1) \subset U(F, V)$, where we take $V_1 = \{v_1\}$ because neither of f_1 and f_2 writes to v_2 or v_3. Indeed, we do not need v_2 or v_3 when we consider the functional requirements of a door without lock in isolation, but when we consider M_1 as a submodel of a door with lock we cannot ignore the role of v_2 to the extent that an implementation of the functions F_1 without using v_2 cannot be used without modification in an implementation of F. The use-relationship for the submodel M_2 in Fig. 2(ii) is $U_2(F_2, V_2) \subset U(F, V)$, where $V_2 = \{v_2, v_3\}$. We remark in passing that the merging of f_1 and f_2 into f_{12} does not affect the above submodels.

The identification of a submodel, in general, requires consideration of both the finite-state model M and the use-relationship $U(F, V)$. We state below the formal definition of a submodel. Although the definition looks at first sight a bit too complex, the various conditions in the definition are quite straightforward and it is easy to see why they are needed. The conditions (1)-(2) state the relationship of the submodel M' to the original model M and the condition (3) forces us to avoid over-prarametrized form of a function name in M. (For a function $f(x, y)$, we can consider the functions $f(x, 0)$ and $f(x, 1)$, with different constant values assigned to parameter y, to have different names such as $f_0(x)$ for $f(x, 0)$ and $f_1(x)$ for $f(x, 1)$ for our purpose.) The condition (4) restricts a submodel M' from being unnecessarily too small and thus avoids spurious submodels. Finally, conditions (5) puts a special importance to the start-state σ'_0 of a submodel M' because of its importance as an interface to the outside of M'. It is easy to see that the submodels in Figs. 2(i)-(ii) satisfy the conditions (1)-(6).

Def. 1. We say a subset Σ' of the states Σ of the finite-state model M in a functional requirement specification to form a submodel M' if the following conditions hold:

(1) There is a unique entry-state $\sigma'_0 \in \Sigma'$ such that all paths from the start-state σ_0 of M enter Σ' through σ'_0. The state σ'_0 is taken to be the start-state of M'.

(2) If $\sigma'_i \neq \sigma'_j$ are two states in Σ' and π'_{ij} is a path from σ'_i to σ'_j in M with no repeating nodes, i.e., π'_{ij} does not contain a cycle, then all states and the transitions in the path π'_{ij} are also in M'; in particular, each transition from σ'_i to σ'_j is in M'. (It follows that once we leave Σ' from, say, $\sigma'_i \in \Sigma'$ then we can renter Σ' only through σ'_i; there can be several such σ'_i for an M'.)

(3) If $F' \subset F$ is the set of functions associated with the transitions in M' among its states Σ', then no transition in M which is outside of M' is associated with a function $f' \in F'$.

(4) If $V' \subset V$ is the set of variables that are written by one or more functions in F', then these variables can only be read by the functions in $F - F'$. Moreover, the set of functions F' is maximal in the sense that for no $F'' \supset F'$ we have the same V'.

(5) If a variable $v_i \in V$ plays a role in defining the state σ'_0 in M, then it is written by a function in F' and read by a function in $F - F'$ and vice-versa.

(6) The variables V' should be able to distinguish the states in M', i.e., the finite-state model $M'(F')$ and the subset $U(F', V') \subset U(F, V)$ should form a functional requirement on its own.

We can derive several important properties of the submodels of a finite-state model M from Def. 1. For example, although the start-state σ'_0 of a submodel M' of M may depend on the start-state σ_0 of M, its states Σ' and the transitions among them, the functions F', and the data-items V' do not depend on σ_0. Also, if M'_1 and M'_2 are two submodels of M and neither one is contained in the other, i.e., $\Sigma'_1 \not\subset \Sigma'_2$ and $\Sigma'_2 \not\subset \Sigma'_1$, then there can be at most one common state between Σ'_1 and Σ'_2; moreover, the common state is the start-state of at least one of M'_1 and M'_2. Also, a submodel has at least two states.

Note that any variables in $V - V'$ which play a part in defining a state in Σ' in the context of M can only be used in read-mode by the functions in F'. This is the case for v_2 = lockedStatus in the submodel M_1 in Fig. 2(i) and also for v_1 = openClosedStatus in the submodel M_2 in Fig.

2(ii). Each such function in F' needs to be adapted slightly for use in M. It is for this reason that the open-function in submodel M_1 and the unlock-function in submodel M_2 are slightly modified for use in the various classes `CDoorWithLock`, `CDoor3WithLock`, etc. in the design solutions shown in Fig. 3.)

2.4. Some possible design solutions

Figs. 3(i)-(iv) show four alternative class-hierarchies for an object-oriented implementation of the requirements in Fig. 1 by viewing them in differently. Here, we do not show the types of individual variables and return-values of methods. Also, we use the convention that instance-variable names in a class begin with 'f', the class-names begin with 'C', and the function-names begin with a capital letter. The class-hierarchy in Fig. 3(i) views a door-with-lock as an extension of a door-without-lock (submodel M_1 in Fig. 2(i)). We assume that the constructor of the class `CDoor` initializes `fopenClosedStatus` to "closed" and the constructor of the class `CDoorWithLock` initializes `fLockedStatus` to "unlocked" (and initializes the constant `fOrigKey` with a suitable value supplied by a parameter in the constructor), in keeping with the start-state in Fig. 1(i). In particular, it uses a factoring of Open-function into the part which does not depend on v_2 = lockedStatus and uses that part in the parent class `CDoor`, with the original Open-function which now makes use of CDoor::Open() kept in `DoorWithLock`. See Fig. 4, where each functions returns "true" in case of a proper use and returns "false" otherwise. If we have other data-items about a door that are specific to Open-function (such as whether the door opens to the left or right) then these are included in `CDoor` and the function `CDoor::Open()` will handle them as needed.

The class-hierarchy in Fig. 3(ii) views a door-with-lock as an extension of a lock-without-door (submodel M_2 in Fig. 2(ii)) and it uses a factoring of Lock-function into the part which does not depend on `fOpenClosedStatus` and puts that part in the class `CLock`.

The class-hierarchy in Fig. 3(iii) views a door-with-lock as a combination of door-without-lock and a lock-without-door based on the two submodels in Fig. 2. If we merge the class `CLock` in Fig. 3(iii) into its child class `CDoor3WithLock` and replace the call to `Clock::Lock(key)` within `CDoor3WithLock::Lock(key)` by its code, then we get the class-hierarchy in Fig. 3(i). The class-hierarchy in Fig. 3(ii) can also be obtained in a similar way. The class-hierarchy in Fig. 3(iv) is a variation of Fig. 3(iii) which does not use the factoring of Open() and Unlock(key); it is inferior to the one in Fig. 3(iii) because the abstract classes `CPartOfLock` and `CPartOfDoor` do not correspond to physical entities in the domain world; they are based on the minimality of the sets of data-items used by Close() and Unlock(key).

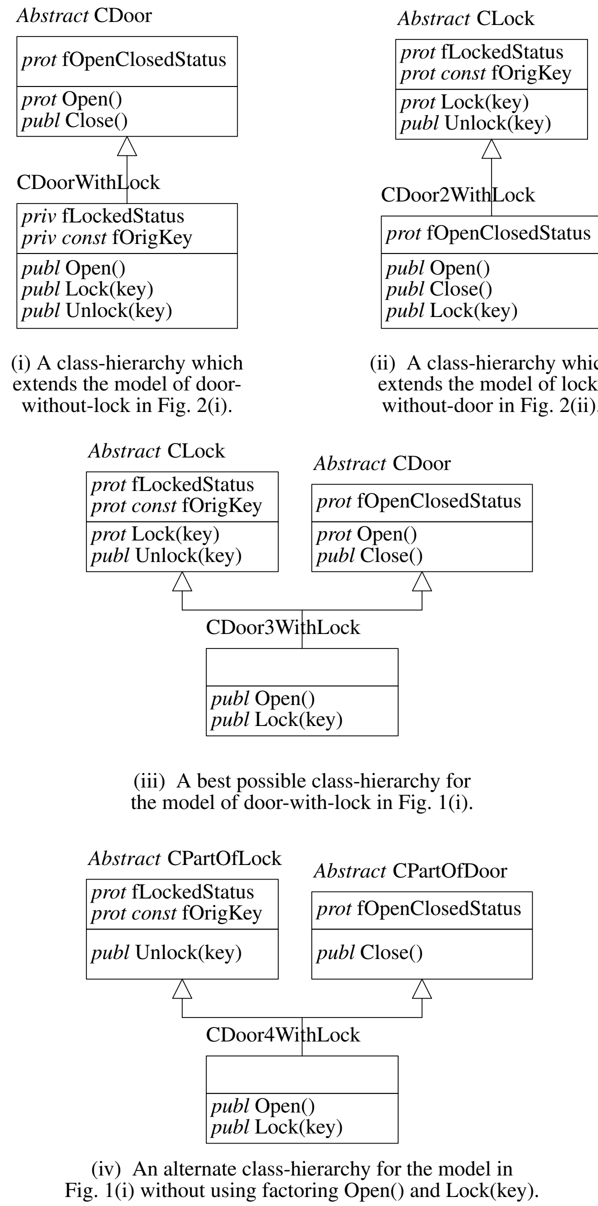


Figure 3. Four class-hierarchies for the functional requirements on a door-with-lock given in Figs. 1(i)-(ii).

The class-hierarchy in Fig. 3(iii) is slightly superior to that in Fig. 3(i) because of the unnecessary access to `fOpenClosedStatus` for the `Unlock`-method in Fig. 3(i); likewise, Fig. 3(ii) suffers from the unnecessary access to `fLockedStatus` and `fOrigKey` for the `Close`-method. Note that if we make `CLock` in Fig. 3(iii) a non-abstract class, then we can implement a door with $n > 1$ locks by adding to `CDoor3WithLock` an instance variable `fAdditionalLocks` which is an array of size $n - 1$ of items of type `Lock`, an instance variable `fKeys` which is an array of size, and the methods

`MultiLock(keys)` and `MultiUnlock(keys)` [4].

```
bool CDoor::Open()
{ if ((closed == fOpenClosedStatus)
      { fOpenClosedStatus = open; return(true); }
  else return(false);
}

bool CDoor::Close()
{ if (open == fOpenClosedStatus)
      { fOpenClosedStatus = closed; return(true); }
  else return(false);
}

bool CDoorWithLock::Open()
{ if (unlocked == fLockedStatus) CDoor::Open();
  else return(false);
}

bool CDoorWithLock::Lock(key)
{ if ((fOrigKey == key) &&
       (unlocked == fLockedStatus) &&
       (closed == fOpenClosedStatus))
      { fLockedStatus = locked; return(true); }
  else return(false);
}

bool CDoorWithLock::Unlock(key)
{ if ((fOrigKey == key) &&
       (locked == fLockedStatus))
      { fLockedStatus = unlocked; return(true); }
  else return(false);
}
```

Figure 4. An implementation of the classes in Fig. 3(i).

2.5. Keeping $U(F, V)$ uncluttered

The users involved in developing functional requirements for a software are typically in a position to specify the finite-state model M and the use-relationship $U(F, V)$ in a reasonable form and when the designers begin to analyze the requirements and identify submodels both M and $U(F, V)$ may be revized. At this stage, we do not need any detailed knowledge about the algorithms to be used in implementing the operations in F. The use-relationship can be also easily updated when we refine the existing operations or the data-items or both. The same is true when we add new operations and new relevant data-items. In order to simplify the identification of submodels, it is important however that we keep only the basic operations in F and eliminate other operations which are simple combinations of them. For example, consider the function f_5 = ExtendedOpen(key), which first unlocks a locked door and then opens it. It corresponds to the transition from σ_2 to σ_1 in Fig. 1(i), i.e., apply f_4 = Unlock(key) first and if it succeeds then apply f_1 = Open(); we write $f_5 = f_4 \bullet f_1$. The row for f_5 in the new $U(F, V)$ is the "union" of the rows for f_4 and f_1 in Fig. 1(ii), i.e., identical to the row for f_3 = Lock() except that the entry "x(r)" in v_1-column is replaced by "x(rw)". Note that $f_6 = f_4 \bullet f_2$ would have the same row as f_5. But, most importantly, if we include f_5 in $U(F, V)$, then we can identify only the submodel M_1 in Fig, 2 from the new $U(F, V)$ and based on it we can derive only the class-hierarchy in Fig. 3(i), with f_5 included in the class `CDoorWithLock`.

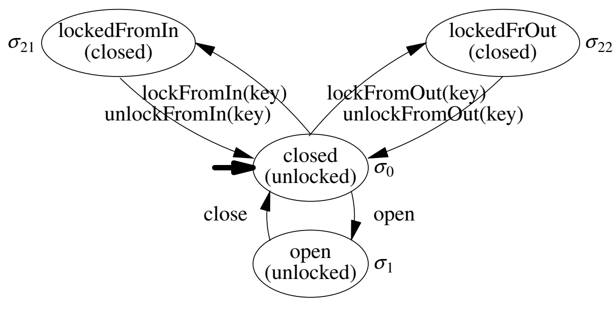
Note that the class-hierarchy in Fig. 3(iii) can accommodate f_5 directly in the class `Door3WithLock`; f_5 can also be accommodated similarly in the other class-hierarchies in Figs. 3(ii) and (iv). The new Fig. 3(iii) remains the best class-hierarchy and, interestingly, the algorithm in [4] will give this class-hierarchy from the new $U(F, V)$.

3. A More Complex Design Problem

We now modify the design problem in Section 2 so that the door can be unlocked only from the side it is locked and one should not be able to lock the door simultaneously from both sides; assume that the same key is used for locking and unlocking from either side. A typical bedroom-door fits this requirement whereas a front-door in a house fits the requirements considered in Section 2. Figs. 5(i) shows the finite-state model; we can regard the state σ_2 in Fig, 1(i) being split here into the two states σ_{21} and σ_{22}. Fig. 5(ii) shows the use-relationship $U(F, V)$, where the rows are arranged to facilitate the visualization of the functions and variables in the six submodels shown below. More details of the submodels are given in Table 1. The submodel M_{31} fits a bath-room door and the submodel M_{32} a storage-room door. Note that $U(f_{32}, v_{21})$ = x(r) allows f_{32} to check that the door is unlocked from inside before it is locked from outside, and a similar remark holds for $U(f_{31}, v_{22})$ = x(r); cf. condition (5) in Def. 1.

- M_1: Door without lock
- M_{21} Lock without door and lock/unlock operations only from inside
- M_{22} Lock without door and lock/unlock operations only from outside
- M_2 Lock without door and lock/unlock operations from both inside and outside
- M_{31} Door with lock and lock/unlock operations only from inside
- M_{32} Door with lock and lock/unlock operations only from outside

We show two alternative choices of V and the corresponding $U(F, V)$ in Figs. 6(i)-(ii). They give rise to different submodels and class-hierarchies and require different implementation for one or more functions. In Fig. 6(i), the two values in $Val(v_4)$ = {inside, outside} are relevant only when v_2 = "locked", and they distinguish the state σ_{21} from σ_{22}; on the other hand, for v_2 = "unlocked", the two values in $Val(v_1)$ = {open, closed} distinguish the states σ_0 and σ_1. Thus, v_2 and v_4 together distinguish the states in Fig. 5(i). Here, f_3 = Lock(key, inOrOut) combines f_{31} and f_{32} in Fig. 5(i) using the additional parameter "inOrOut", with Lock(key, "in") giving f_{31} and Lock(key, "out") giving f_{32}; a similar remark holds for f_4 = Unlock(key, inOrOut). The use-relationship in Fig. 6(i)



(i) The finite-state model *M(F)*; start-state = σ_0.

Use-Relation-ship $U(F, V)$	v_1: open-Closed-Status	v_{21}: locked-Status-In	v_{22}: locked-Status-Out	v_3: orig-Key
f_{31}	x(r)	x(rw)	x(r)	x(r)
f_{41}		x(rw)		x(r)
f_1: Open	x(rw)	x(r)	x(r)	
f_2: Close	x(rw)			
f_{32}	x(r)	x(r)	x(rw)	x(r)
f_{42}			x(rw)	x(r)

(ii) Here, r = read, w = write, f_{31} = LockFromIn(key), f_{32} = LockFromOut(key), f_{41} = UnlockFromIn(key), and f_{42} = UnlockFromOut(key)

Figure 5. The finite-state model and use-relationship for a door with lock/unlock operations for each side.

Table 1. The submodels for Fig. 5; here, $M_2 = M_{21} \cup M_{22}$, $M_{31} = M_1 \cup M_{21}$, and $M_{32} = M_1 \cup M_{22}$.

Sub-model	States	Functions	Data-items
M_1	$\Sigma_1 = \{\sigma_0, \sigma_1\}$	$F_1 = \{$open, close$\}$	$V_1 = \{v_1\}$
M_{21}	$\Sigma_{21} = \{\sigma_0, \sigma_{21}\}$	$F_{21} = \{f_{31}, f_{41}\}$	$V_{21} = \{v_{21}\}$
M_{22}	$\Sigma_{22} = \{\sigma_0, \sigma_{22}\}$	$F_{22} = \{f_{32}, f_{42}\}$	$V_{22} = \{v_{22}\}$
M_2	$\Sigma_2 = \Sigma_{21} \cup \Sigma_{22}$	$F_2 = F_{21} \cup F_{22}$	$V_2 = V_{21} \cup V_{22}$
M_{31}	$\Sigma_{31} = \Sigma_1 \cup \Sigma_{21}$	$F_{31} = F_1 \cup F_{21}$	$V_{31} = V_1 \cup V_{21}$
M_{32}	$\Sigma_{32} = \Sigma_1 \cup \Sigma_{22}$	$F_{32} = F_1 \cup F_{22}$	$V_{32} = V_1 \cup V_{22}$

allows only the submodels M_1 and M_2; here, M_2 requires $U(F_2, V_2)$ where $F_2 = \{f_3, f_4\}$ and $V_2 = \{v_2, v_3, v_4\}$.

The use-relationship in Fig, 6(ii) uses the same functions as in Fig. 5(i) and it too allows only the submodels M_1 and M_2. But now $F_2 = \{f_{31}, f_{32}, f_{41}, f_{42}\}$ and $V_2 = \{v_2\}$, where $Val(v_2)$ = {lockedFromIn, lockedFromOut, unlocked}. The variables v_1 and v_2 together distinguish all the states in Fig. 5(i). The lock-function f_{31} here reads the value v_2 = "unlocked" and writes the value v_2 = "lockedFromIn" whereas in Fig. 5(i) it reads v_{21} = "unlocked" and writes v_{21} = "locked"; a similar remark holds for f_{32}. The two unlock-functions f_{41} and f_{42} differ also in a similar fashion between here and Fig. 5(i).

Use-relation-ship	v_1: open-Status	v_2: locked-Status	v_3: orig-Key	v_4: locking--Side
f_1: Open	x(rw)	x(r)		
f_2: Close	x(rw)			
f_3	x(r)	x(rw)	x(r)	x(w)
f_4		x(rw)	x(r)	x(r)

(i) f_3 = Lock(key, inOrOut) and f_4 = Unlock(key, inOrOut).

Use-relationship	v_1: open-ClosedStatus	v_2: lockedStatus	v_3: origKey
f_{31}	x(r)	x(rw)	x(r)
f_{41}		x(rw)	x(r)
f_1: Open	x(rw)	x(r)	
f_2: Close	x(rw)		
f_{32}	x(r)	x(rw)	x(r)
f_{42}		x(rw)	x(r)

(ii) It uses the same functions as in Fig. 5(i).

Figure 6. Two alternate use-relationships for Fig. 5(i); they allow only the submodels M_1 and M_2 in Table 1.

Fig. 7 shows the best class-hierarchy for the requirements in Fig. 5, which is similar in spirit to that in Fig. 3(iii). We can think of it as a result of replacing the class `CLock` in Fig. 3(iii) by `CLockFromInOut` and adding the two parent classes `CLockFromIn` and `CLock-FromOut`. It allows us to use different keys for locking from inside and from outside. The function CLockFromInOut::LockFromIn(key) extends the function CLockFromIn::LockFromIn(key) with the additional check that CLockFromOut::fLockedStatusOut = "unlocked". We can also extend this class-hierarchy, which uses only the first four submodels in Table 1, to account for doors with one-sided locks corresponding to the other two submodels.

4. Computation of Use-Relationship

In an objected-oriented software, we can talk of the use-relationship at different levels, both within a class and between classes. We first show how to compute these two use-relationships at different levels from a given source-code and then how to combine them to obtain the final use-relationship. Our goal is to use it as a reengineering

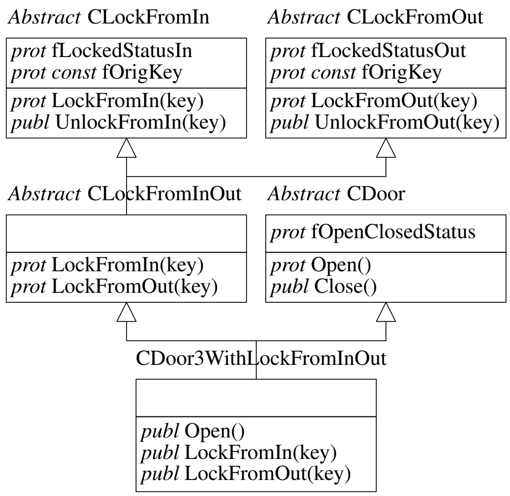


Figure 7. The best class-hierarchy for a door-with-two-sided-lock based on the requirements in Fig. 5.

step for verifying compliance with the original requirement specification. This can be done even when the source-code is only partially developed.

4.1. Use-relationship within a class

Let $F(C)$ = the set of all functions (methods) in a class C and $V(C)$ = the set of all (instance) variables in C. The local variables within a function in $F(C)$ are not included in $V(C)$ but the private variable in C are included in $V(C)$. We can determine $F(C)$ and $V(C)$ by a simple scan of the source-code. Next, we compute the following two direct-use relationships within a class C; the first one is defined on $F(C) \times F(C)$ and the second one on $F(C) \times V(C)$.

(1) The relationship $UFF_D^W(C)$, where $(f, f') \in UFF_D^W(C)$ means $f \in F(C)$ directly calls $f' \in F(C)$.

(2) The relationship $UFV_D^W(C)$, where $(f, v) \in UFV_D^W(C)$ means $f \in F(C)$ directly uses $v \in V(C)$.

We compute both $UFF_D^W(C)$ and $UFV_D^W(C)$ row by row by a simple scan of the source-code of each $f \in F(C)$. The composition of the reflexive-transitive closure of $UFF_D^W(C)$ with $UFV_D^W(C)$ gives both the direct and indirect uses of variables in $V(C)$ by functions in $F(C)$ and hence equals the final use-relationship $UFV^W(C)$ for C.

A more efficient of way of computing $UFV^W(C)$ is as follows. First, we note that if $f_i \in F(C)$ directly or indirectly calls $f_j \in F(C)$ then $V(f_i, C) \subseteq V(f_j, C)$, where $V(f_i, C)$ = the set of variables in $V(C)$ used by f_i. It follows that if the functions $\{f_1, f_2, \cdots, f_k\} \subseteq F(C)$ call each other in a recursive (circular) form, i.e., they form a strong component of the digraph $G(C)$ on the nodes $F(C)$ with the directed links given by $(f, f') \in UFF_D^W(C)$, then $V(f_i, C) = V(f_j, C)$ for $1 \leq i, j \leq k$. This means we need to compute $V(f, C)$ only for one representative f from each strong component of $G(C)$. Second, we reduce $UFF_D^W(C)$ by keeping just the row and column for one representative f for each strong component and eliminating the other rows and columns. Likewise, we reduce $UFV_D^W(C)$ by keeping just the row for the same representative f for each strong component used earlier and eliminating the other rows. Finally, we compute $UFV^W(C)$ by combining the reflexive-transitive closure of the reduced $UFF_D^W(C)$ with the reduced $UFV_D^W(C)$.

4.2. Use-relationship among the classes

We define the interclass use-relationships $UFF_D^I = \{(f, f')$: for some pair of classes $C \neq C'$, $f \in F(C)$ directly calls $f' \in F(C')\}$. Here, f is public or protected but f' is public or protected if C' is a superclass of C and f' is public otherwise. The rows and columns of UFF_D^I consist of all public and protected functions in all classes in the class-hierarchy. The interclass use-relationship UFV_D^I is defined similarly; here, the rows are the same as those of UFF_D^I and the column are all public and protected instance variables of the various classes, excluding the private variables. We now form the composition of the transitive closure of UFF_D^I with $\bigcup\{UFV^W(C)$: for all classes C in the class-hierarchy$\}$. (Formally speaking, we need to extend each $UFV^W(C)$ by adding empty rows for functions f which appear as columns in UFF_D^I but $f \notin F(C)$ and throw away rows for $f \in F(C)$ where f is a private function. Likewise, we need to extend $UFV^W(C)$ for each class by adding empty columns for some of the variables.) This gives the direct or indirect use of variables $v \in V(C')$ in one class C' by a function $f \in F(C)$ in another class $C \neq C'$ having the proper access rights. We let UFV^I be the union of UFV^I and the above composition.

4.3. Final use-relationship $U(F, V)$

This is now given by the union of $UFV^W(C)$ for each non-abstract class C and of UFV^I. We actually need to keep only the rows for public functions in the various classes. The columns are still all instance variables in the class-hierarchy, including the private variables.

Example 1. Consider the class-hierarchy in Fig. 3(i) and the source-code in Fig. 4. For C = `CDoor`, we have $UFF_D^W(C) = \varnothing$ and therefore $UFV^W(C) = UFV_D^W(C) = \{$(CDoor::Open(), v_1), (CDoor::Close(), v_1)$\}$, where v_1 = fOpenClosedStatus. Likewise, for C = `CDoorWith-Lock`, we have $UFF_D^W(C) = \varnothing$ and thus $UFV^W(C) = UFV_D^W(C)$, which is shown in Fig. 8(i). We also have

$UFF_D^I = \{$(CDoorWithLock::Open(), CDoor::Open())$\}$
$UFV_D^I = \{$(CDoorWithLock::Lock(key), v_1)$\}$
$UFV^I = \{$(CDoorWithLock::Open(), v_1),
(CDoorWithLock::Lock(key), v_1)$\}$.

Fig. 8(ii) shows the final $U(F, V)$ obtained and it matches the use-relationship in Fig. 1(ii).

UFV_D^W for the class C = CDoorWithLock	v_2 fLockedStatus	v_3 fOrigKey
Open()	x	
Lock(key)	x	x
Unlock(key)	x	x

(i) The relations UFV_D^W(CDoorWithLock).

$U(F, V)$	fOpen-Closed-Status	fLocked-Status	fOrig-Key
CDoor::Close()	x		
CDoorWithLock:: Open()	x	x	
CDoorWithLock:: Lock()	x	x	x
CDoorWithLock:: Unlock()		x	x

(ii) Computing the final $U(F, V)$ from UFV^I and $UFV^W(C)$ for the various classes C.

Figure 8. Illustration of computing $U(F, V)$ from the source-code in Fig. 4 for the class-hierarchy in Fig. 3(i).

5. Impact of Design Choice On Testing

In Sections 2 and 3, we have shown that the structure of submodels in a given functional requirement specification can be used to design a proper class-hierarchy for an object-oriented implementation. We can use the same submodels that are used in the class-hierarchy design also as a basis for an effective and simple integration testing. It is based on two important properties of the submodels. First, if a state σ' in the global finite-state model M is shared by two or more minimal submodels $M'_1, M'_2, \cdots, M'_k$ of M then no two submodels M'_i and M'_j share a state among them other than σ'. This means we need to test each pair M'_i and M'_j as follows using only two function-applications at a time. For each operation f'_i corresponding to a transition in M'_i to the state σ' and for each operation f'_j corresponding to a transition in M'_j from the state σ', test that the sequence $f'_i \bullet f'_j$ works properly, i.e., the state σ' correctly interfaces f'_i and f'_j. Note that this prevents the combinatorial explosion of integration testing in connection with the state σ' because if $n_i^{to}(\sigma')$ = #(transitions to σ' in M'_i) and $n_i^{from}(\sigma')$ = #(transitions from σ' in M'_i) then the total number of test sequences $f'_i \bullet f'_j$ is $\sum n_i^{to}(\sigma') n_j^{from}(\sigma')$, summed over $1 \le i,\ j \le k$. Also, we can perform the integration testing of each subsystem associated with M'_i as it is developed. Second, if M''_i is a submodel containing M'_i (i.e., M''_i is not a minimal submodel containing the state σ') then for any $j \ne i$, $1 \le j \le k$, either M''_i contains M'_j or σ' is the only state shared between M''_i and M'_j. In this case, we do not need to test the combinations of transitions to σ' in M''_i with the transitions from σ' in M'_j or vice-versa because all those combinations have been already covered during the pairwise integration testing of M'_i and M'_j.

6. Conclusion

We have shown that the use-relationship $U(F, V)$ between the functions F and the data-items V together with the finite-state model $M(F)$ which describes the valid sequences of the operations F provides a simple and powerful scheme for specifying the functional requirements of a software. In particular, we can analyze such a specification to identify the submodels and use them to design the class-hierarchy for a complex specification from those of the submodels and to perform integration testing. This complements our previous work in [4]. Since we can extract the use-relationship $U(F, V)$ from the software source-code as a reverse-engineering step, we can use $U(F, V)$ to verify compliance of a source-code with the given requirement specification.

7. References

1. A. Davis, Software Requirements, Prentice-hall, 1990.
2. R.J. Costello and D.-B. Liu, Metrics for Requirement Engineering, *System Software,* (29), 1995, pp. 39-63.
3. H. Kaiya, H. Hori, M.Saeki, AGORA: attributed goal-Oriented Requirements Analysis Method, *Proc. IEEE Joint International Conf. on Requirements Engineering,* 2002.
4. S. Kundu and N. Gwee, A formal approach to designing a class-subclass structure using a partial order on the functions, *Proc. 29th Annual Intern. Computer Software and Applications Conf., COMPSAC-2005, 2005, pp. 213-220.*
5. J. Mylopoulos, L. Chung, and E. Yu, From object-oriented to goal-oriented requirement analysis, *Comm. of the ACM,* 42(1999), pp. 31-37.
6. C. Shamlin, Defining Software Requirements (2nd ed.), QED Information Sciences, Inc, 1989.
7. G. Smith, The Object-Z Specification language, Kluwer Academic Publishers, 2000.
8. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
9. R. Laney, L. barroca, M. Jackson, and B. Nuseibeh, Composing requirements using problem frames, *Proc. 12th IEEE Intern. Requirements Engineering Conf (RE-04), 2004.*
10. D. Bjørner, Towards posit&prove calculi for requirements engineering and software design, in *From Object-orientation to Formal Methods* (eds. O. Owe, S. Krogdahl, and T. Lyche), 2004 (LNCS# 2635), Springer.