A Formal Model of Use-Cases and Its Application in Generating

A Hierarchical Class-Structure

Sukhamay Kundu
Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803
Email: kundu@csc.lsu.edu

Arnab Ganguly
Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803
Email: agangu4@lsu.edu

Abstract—Creating an object-oriented design from user requirements, given as a set of use-cases, means deriving a detailed class structure that can support an implementation of those requirements. We introduce here the Augmented Finite-State (AFS) model for a set of use-cases. An AFS model of a single use-case \( U \) incorporates the inputs, outputs, and operations for each interaction in \( U \), including the “internal” dataflows among those interactions. The AFS model for a set of use-cases \( \mathcal{U} \) combines the AFS models of individual use-cases \( U_j \in \mathcal{U} \) to account for common interactions among \( U_j \)'s and the control-flows among \( U_j \)'s. After we decompose the combined model into a unique set of disjoint Maximal Linear Segments (MLSs), we derive one class from each MLS and finally create the class-hierarchy based on the next-relationship among the MLSs. One advantage of our approach over those based on the concept-analysis is that the AFS model gives a simple controller for the call-sequences of the class-methods corresponding to each \( U_j \).

Keywords—Augmented finite-state model; class hierarchy; object-oriented design; refactoring; use-case model.

I. INTRODUCTION

Software design is an essential part of any software development effort. For an object-oriented software, the design consists of the classes (their attributes, methods, and method-parameters) and the relationships among those classes, which includes the class-hierarchy and other associations. The design gives a global view of the functionalities and structure of the software, and plays a critical role in understanding, implementation, and analysis of the software.

Semi-automated generation of UML-models and class-diagrams from natural language description of requirements are discussed in [1][2]. Automated generation of UML-models are discussed in [3][4]. These works are based on Natural Language Processing, and rely on use-cases defined using a semi-formal syntaxes and semantics. Cockburn [5] argued that there is no formal syntax or semantics for writing use-cases. Roussev [6] uses an informal notion of “balance” of objects involved in a use-case, and assumes the use-cases are given in terms of pre-conditions, post-conditions and invariants on the objects involved in the use-cases. The identification of objects is a key missing step in [6]; also, the notion of “balance” of objects has a basic flaw because “information” do not behave like the physical quantities force, energy, and mass, and we don’t have a principle like the “conservation of energy” for “information”.

II. A FORMAL DEFINITION OF USE-CASE

Jacobson [11] defines a use-case as a sequence (chain) of interactions \( U = \{t_1, t_2, \ldots, t_n\} \), which provides the user a useful service, i.e., corresponds to a complete high-level functional requirement. If \( U_1 \) and \( U_2 \) are two use-cases, then clearly \( U_1 U_2 \) is also a use-case. Henceforth, a use-case \( U \) will mean an elementary use-case, which cannot be decomposed into a sequence of two or more disjoint smaller use-cases.

A. Interaction \( t_j \)

Formally, an interaction \( t_j = (in_j, op_j, out_j) \) is a triplet, where \( in_j = in(t_j) \) is a set of input data-items, \( op_j = op(t_j) \) is an operation, and \( out_j = out(t_j) \) is a set of output data-items.
The inputs \( in_j \) consist of two disjoint parts: user-inputs \( in_j^u \) and other "external" (with respect to \( U \)) inputs \( in_j^x \). The term "interaction" means \( in_j^x \neq \emptyset \); however, one possible exception to this is that the last interaction \( t_n \) in \( U \) may have \( in_n^x = \emptyset \) (cf. Theorem 1). An operation \( op_j \) may have multiple parts and a user may provide different parts of \( in_j^x \) at different stages of \( op_j \), with \( in_j \) and \( out_j \) interleaved. The inputs to \( op_j \) may also include zero or more additional "internal" data-items \( d_i \) that are generated by previous \( op_j \)’s, \( i < j \) and \( t_i \in U \). We refer to such \( d_i \)’s as dataflows. The outputs \( out_j \) also consists of two parts: user-outputs \( out_j^u \) and external-outputs \( out_j^x \); because parts of \( out_j^x \) may be saved for use in other use-cases and hence included in \( out_j^x \), we may have \( out_j^x \cap out_j^u \neq \emptyset \). Clearly, \( in_j = in_j^u \cup in_j^x \) and \( out_j = out_j^u \cup out_j^x \). Henceforth, we use \( t_j \) and \( op_j \) interchangeably when no confusion is likely. Figure 1 shows the structure of an interaction \( t_j \). We can regard \( op_j \) as a function of \( in_j \) and zero or more \( d_i \), \( i < j \), i.e., \( in(op_j) = \bigcup_{i<j} in_i \). In contrast, \( in(t_j) = in_j \) and thus \( t_j \) is not exactly the same as \( op_j \). The dataflow \( d_j \) is not determined by \( op_j \), but by \( t_k, k > j \) in \( U \); also, \( d_j \) need not be a subset of \( out_j^x \).

(3) \( d_i \cap d_j = \emptyset \) for \( i \neq j \). Any part of \( d_i \) can be made available to each \( op_j, j > i \), as needed.

(4) Each \((t_j, t_{j+1})\)-pair has an associated transition-condition \( c_{j,j+1} \) which needs to be satisfied after \( op_j \) is completed in order for \( op_{j+1} \) to start; \( c_{j,j+1} = true \) means the condition is trivially satisfied. The condition \( c_{j,j+1} \) does not depend on \( in_k, k > j \), and may depend only on parts of \( in(op_j) \). All non-trivial conditions \( c_{i,i+1}, i \leq j \), contribute directly or indirectly to the pre-condition for \( op_{j+1} \). We assume for now that \( c_{j,j+1} \) is evaluated by \( op_j \).

We write \( I_U^U = \bigcup in_j^u \) and \( I_U^x = \bigcup in_j^x \), where the unions are taken over all \( j \), and finally \( I_U^U = I_U^U \cup I_U^x \). Similarly, we write \( O_U^U = \bigcup out_j^u \), \( O_U^x = \bigcup out_j^x \), \( O_U = O_U^U \cup O_U^x \), \( D_U = \{ d_j \} \) and \( C_U \{ c_{j,j+1} : 1 \leq j < n \} \). The entities in \( I_U^U \), \( O_U \), and \( D_U \) are the names of data-items and not any specific values for them. (An instance of a use-case \( U \), with concrete values for the data-items in \( I_U \) and hence concrete values for the data-items in \( D_U \cup O_U \), is called a scenario.)

If the use-case \( U \) is used after the use-case \( U \) and we need to use parts of \( in_j^u \) at \( t_j \in U \) as parts of \( in_j^x \) at \( t_j \in U \), then we can include those parts of \( in_j^u \) into \( out_j^u \) and those parts of \( out_j^u \) can now become a part of \( in_j^u \). This avoids having to provide the common parts of \( in_j^u \) and \( in_j^u \) more than once.

A proper choice of the individual interactions \( t_j \) in modeling a use-case \( U \) is a non-trivial task. A simpler or smaller \( t_j \) can help to reduce errors in determining \( in(op_j) \), \( out(op_j) \), and \( c_{j,j+1} \), but it can also introduce unnecessary details in the design of a class-structure for \( U \). A complex or larger \( t_j \) can, on the other hand, prevent sharing interactions between different use-cases. These issues are described next.

C. Merging Interactions

If we merge \( t_j, t_{j+1} \in U \) into a single interaction \( t_{j,j+1} \), then we have \( in_j^u \cup in_{j+1}^u = in_j^u \cup in_{j+1}^u \) and similarly for \( out_j^u \cup out_{j+1}^u \), and \( out_j^x \cup out_{j+1}^x \). In general, \( d_j \cup d_{j+1} \subseteq d_j \cup d_{j+1} \), with \( t_{j,j+1} \) hiding dataflows from \( t_j \) to \( t_{j+1} \) (making them internal to \( op_{j,j+1} \)); in the extreme case, we may have \( d_j \cup d_{j+1} = d_j \cup d_{j+1} \). Also, \( op_{j,j+1} = op_j \cup op_{j+1} \), the composition (roughly speaking in view of \( c_{j,j+1} \) of \( op_j \) and \( op_{j+1} \), in that order. The new interaction sequence \( t_1, t_2, \ldots, t_j, t_{j-1}, t_{j,j+1}, t_{j+1}, \ldots, t_n \) has less information about \( op_{j,j+1} \) (equivalently, \( t_{j,j+1} \)) because we know less about which parts of \( in_{j,j+1} \) are used by which parts of \( op_{j,j+1} \) to produce which parts of \( out_{j,j+1} \) and \( d_{j,j+1} \). There is, however, no change in \( I_U^U, I_U^x, O_U^U, O_U^x \).

Except for the loss of some information as noted above, it is safe to merge \( t_j \) and \( t_{j+1} \) when \( c_{j,j+1} = true \). In this case, each use-case \( U \) containing \( t_j \) will also contain \( t_{j+1} \) and thus we can replace \( t_j \) and \( t_{j+1} \) by \( t_{j,j+1} \) in each \( U \).

The merging of \( t_j \) and \( t_{j+1} \) has no impact on the condition from \( t_{j-1} \) to \( t_{j,j+1} \), i.e., \( c_{j-1, (j,j+1)} = c_{j-1,j} \), which will be evaluated by \( op_{j-1} \). However, determining the condition \( c_{(j,j+1), j+2} \) from \( t_{j,j+1} \) to \( t_{j+2} \) might pose a problem as shown below. Consider the situation on the left-side in Figure 2. The part \( c_{j+1}: "x + z \neq 0" \) in \( c_{(j,j+1), j+2} \) means \( c_{j+1} \land c_{j+1} \), shown on the right-side in Figure 2 is the result of "pushing down" the condition \( c_{j+1}: "x \neq 0" \) through \( op_{j+1} \), which gives \( x_{new} = x_{old} - z \), i.e., \( x_{new} = x_{new} + z \), and thus \( c_{j+1}: "x \neq 0" \) becomes \( c_{j+1} : "x_{new} + z \neq 0" \), because \( c_{j+1} \). But a difficulty arises if we replace \( x = x - z \) in \( op_{j+1} \) by \( y = x^2 - z^2 \) because we cannot express now \( x_{old} \) in
terms of $x_{\text{new}}$. We can, however, add the computation "$x_{\text{old}} = x$" at the end of $op_j$ and call it $op'_{j+1}$, let $op_{j+1} = op'_{j+1} \circ op_{j+1}$ and $c_{(j+1),j+2}^x: (x_{\text{old}} \neq 0) \land (y > x)$. But such tricks do not always suffice as would be the case if "$x = x − z$" in $op_{j+1}$ is replaced by "$x = x − z/x$"; $op_{j+1}$ now needs $x \neq 0$ and thus we cannot afford to do the test "$x_{\text{old}} \neq 0$" after $op_{j+1}$.

D. Decomposing an Interaction

If we can decompose an $op_j$ into a chain of suboperations ($op_{j+1}, op_{j+2}, \ldots, op_{j+m}$), $m \geq 2$, then should we replace $t_j$ in $U$ by the chain of interactions $(t_{j+1}, t_{j+2}, \ldots, t_m)$, where $op_{j+p}$ corresponds to $t_{j+p}$? If we did, then we will have $c_{j+1,j+1} \equiv c_{j+1,j} \land c_{(j+1),j+2} \equiv \text{true}$, $1 \leq p < m$. This implies that it is safe to merge $t_{j,p}$'s and hence the decomposition is unnecessary. Note that because $c_{j+1,j+2} \equiv \text{true}$, there is no use-case $U'$ that includes $t_{j,p}$ but not $t_{j,p+1}$.

E. Deleting an Interaction

In general, the deletion of a $t_j \in U$ may not give a valid use-case $(t_1, t_2, \ldots, t_j, t_{j+1}, t_{j+2}, \ldots, t_n)$ for example, if $t_k$, $k > j$, requires $d_j$ generated by $op_j$ then removal of $t_j$ makes $op_k$ inapplicable; hence $t_k$ needs to be removed. This may, in turn, require other $t_m$, $m > k$, to be removed and so on. On the other hand, if $k > j$ is the smallest index such that $t_k$ requires $d_j$ then $(t_1, t_2, \ldots, t_{j−1}, t_{j+1}, t_{j+2}, \ldots, t_k, t_{k−1})$ may not be a valid use-case because the output of $t_k$ may involve a prompt to the user to provide an input ($in_k^e \neq \emptyset$). The same argument shows that deletion of $t_k−1$ may create a problem, and so on. A similar argument shows that an initial part of a use-case may not be a valid use-case. Likewise, a tail part $(t_k+1, t_{k+2}, \ldots, t_n)$ of a use-case may not be a valid use-case because the output of $t_k$ may involve a prompt to the user and that prompt $t_k+1$ becomes meaningless.

III. AUGMENTED FINITE STATE (AFS) MODEL

The AFS model of a set of use-cases $U$, denoted by $AFS(U)$, combines the notions of finite state machines, flowcharts, and Dataflow Diagrams (DFDs, which can be regarded as high-level dataflow-abstractions of flowcharts). As a finite-state machine, each state $s_j$ in $AFS(U)$ corresponds to an interaction $t_j$ in a use-case in $U$. Each transition $(s_j, s_k)$ corresponds to the next-interaction $t_k$ of $t_j$ in a use-case that contains $t_j$, and associated with the transition $(s_j, s_k)$ we have the corresponding condition $c_{j,k}$. Clearly, $c_{j,k}$ is independent of the use-case in which $t_k$ is the next interaction after $t_j$, and it can be likened to a branching-condition in a flowchart. We also have the dataflows $d_j$ between interactions or states. As before, we consider a state $s_j = t_j$ to be synonymous with the operation $op_j$ associated with $t_j$. If $|U| = 1$, then $AFS(U)$ takes the form of a single chain. The condition $c_{j,k}$ associated with transition $(s_j, s_k)$ must be satisfied for the transition to take place. For two transitions $(s_j, s'_k)$ and $(s_j, s''_k)$, $s_k \neq s'_k$, the conditions $c_{j,k}$ and $c_{j,k'}$ must be disjoint, i.e., $c_{j,k} \land c_{j,k'} = \emptyset$. Unlike a flowchart, an $AFS(U)$ by definition does not have a cycle and this prevents cyclic data-dependencies. The usual use-dependencies among data-items in assignments and other computations in a flowchart is replaced in $AFS(U)$ by the higher-level abstractions inputs, outputs, and dataflows associated with a state.

We formally define $AFS(U) = (S, S_0, S_{\text{final}}, C, D, I^u, I^e, O^u, O^e, \tau, \delta, \phi^u, \psi^u, \phi^e, \psi^e)$, where

1) $S \neq \emptyset$ is a set of states and $S_0 \in S$ is the start-state; each state $s_j$ is reachable from $s_0$ by a sequence of interactions and has an associated operation $op_j$.  

Figure 1. Illustration of a simple case of $c_{(j+1),j+2}^x$. 

Even if we could define a suitable $c_{(j+1),j+2}$, the creation of $t_{j+1}$ would prevent sharing just one of $t_j$ and $t_{j+1}$ with another use-case $U'$. This can be a good reason not to create $t_{j+1}$. On the other hand, if we have a use-case $U' = \{t_1, t_2, \ldots, t_{j+1}, t_{j+2}, \ldots, t_n\}$ containing both $t_j$ and $t_{j+1}$ but a different $t_{j+2}$, with $c_{j+1,j+2} = y \leq x = c_{j+1,j+2}$ for the pair $(t_{j+1}, t_{j+2})$, then after we create $t_{j+1} \in U'$ we will have $c_{(j+1),j+2} = (x + z \neq 0) \land (y \leq x)$ in $U'$. Obviously, we cannot merge $t_j$ and $t_{j+1}$, $k > j + 1$, and keep $U$ acyclic.

Theorem 1. For a use-case $U = \{t_1, t_2, \ldots, t_n\}$, there is no loss of generality to assume that each $in_j \neq \emptyset \text{ for } j < n$ if some use-cases $U'$ share $t_k \in U$ exactly upto $t_j$. 

Proof. If $in_j = \emptyset$ and $j < n$, then $c_{j+1,j}$ depends only on $\bigcup_{j' < j} in_{j'}$ and thus we can create the shortened use-case $U'$ by merging $t_j$ into $t_{j+1}$ as follows. We let $c_{j−1,j+1} = c_{j−1,j} \land lift(c_{j+1,j+1})$, where $lift(c_{j+1,j+1})$ is the result of "lifting up" $c_{j+1,j+1}$ through $op_j$, and replace $op_{j+1}$ by $op'_{j+1} = op_j \circ op_{j+1}$. Note that $c_{j−1,j+1}$ can be evaluated by $op_{j+1}$. If there is another use-case $U'' = \{t_1, t_2, \ldots, t_{j−1}, t_{j+1}, \ldots, t_n\}$, which is identical to $U$ upto $t_j$, then we can likewise create the shortened use-case $U''$ by merging $t_j$ into $t_{j+1}$, with $op'_{j+1}$ replaced by $op''_{j+1} = op_j \circ op_{j+1}$ and letting $c''_{j−1,j+1} = c_{j−1,j} \land lift(c_{j+1,j+1})$, where $c_{j−1,j+1}$ is the condition for the pair $(t_j, t_{j+1})$. Note that $c_{j−1,j} \land c_{j−1,j+1} = c_{j−1,j} \land lift(c_{j+1,j+1}) = c_{j−1,j} \land lift(false) = c_{j−1,j} \land false = false$, as desired. The shortened use-cases $U'$ and $U''$ now share only upto $t_j$. □

Two remarks are due here. First, merging $t_j$ with $t_{j+1}$ to avoid $in_j \neq \emptyset$ does not cost us in terms of its effect on the class design. The methods for $op'_{j+1}$ and $op''_{j+1}$ in the classes for $U'$ and $U''$ will now have some commonalities because $op_j$ is a part of both $op'_{j+1}$ and $op''_{j+1}$. However, we can refactor the common part, if needed, to a parent class. Second, we do not merge $t_j$ into $t_{j−1}$ in the proof of Theorem 1 because if there is an use-case $U''$ that is identical to $U$ only upto $t_{j−1}$ then the merging would create an overloaded $t_{j−1}$ in terms of outputs and the operation $op_{j−1,j} = op_{j−1} \circ op_j$, and this can cause problems with the condition for $(t_{j−1,j}, t''_{j−1,j})$-pair for $U''$. If there is no $U''$, we could form $t_{j−1,j}$ to eliminate $t_j$. 

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4
2) \( S_{\text{final}} \subseteq S \) is the subset of final (terminal) states, from which there are no transitions.
3) \( C = \{c_{i,j} \mid \text{conditions associated with transitions } (s_i, s_j) \} = \bigcup C_{U_i}, \) union over the use-cases \( U \in U \).
4) \( D = \{d_i \mid \text{the internal dataflow item generated by } op_i \text{ at } s_i \} = \bigcup D_{U_i}, \) union over the use-cases \( U \in U \).
5) \( I^{u} = \bigcup I^{T}_u, \ I^{e} = \bigcup I^{T}_e, \ O^{u} = \bigcup O^{T}_u \) and \( O^{e} = \bigcup O^{T}_e, \) each union over the use-cases \( U \in U \).
6) \( \tau : S \times C \to S \) is the transition function.
7) \( \delta : D \to S \times P^+(S) \) is the function representing the dataflows, where \( P^+(S) = \{\text{the set of non-empty subsets of } S, \delta(d_j) = (s_i, S_j) = (\delta_1(d_j), \delta_2(d_j)) \} \) means \( S_j = \{s_k \mid s_k \text{ uses } d_j \text{ generated at } s_j \}, \) and \( d_i \neq d_j \) means \( s_i = \delta_1(d_i) \neq \delta_1(d_j) = s_j. \)
8) \( \phi^u : I^{u} \to P^+(S) \) is the user-input function; \( \phi^u(u') = \{s_k \mid s_k \text{ requires user-input } u' \}. \)
9) \( \phi^e : I^{e} \to P^+(S) \) is the external-input function; \( \phi^e(e') = \{s_k \mid s_k \text{ requires external-input } u' \}. \)
10) \( \psi^u : O^{u} \to P^+(S) \) is the user-output function.
11) \( \psi^e : O^{e} \to P^+(S) \) is the external-output function.
12) For each \( s_i \notin S_{\text{final}}, \) the conditions \( c_{i,j} \) are mutually disjoint (i.e., \( c_{i,j} \land c_{i,k} = \text{false} \) for \( j \neq k \)). We may also assume that \( \forall c_{i,j} = \text{true} \) because otherwise we can add a "graceful" exit-transition to a new final state \( s'_{\text{final}} \) with the transition-condition \( c_{i,j} \).
13) For any \( s_i, s_j \in \phi^u(u') \) for an input-user \( u' \), neither \( s_i \) nor \( s_j \) is reachable from the other. Similarly for \( \phi^e \) and \( \psi^e. \) (But this is not required for \( \phi^u. )

In what follows, we consider only the special case where \( AFS(U) \) has a tree-structure. Recall that not all paths in a flowchart, even in absence of cycles, may represent a valid execution-path; likewise, if the transitions in \( AFS(U) \) form a general acyclic digraph, then we may have paths from the start-state \( s_0 \) to a final-state that do not represent a valid use-case and this can severely compromise the derivation of a suitable class-structure from \( AFS(U). \) In the case of a tree-structured \( AFS(U), \) each path from the start-state \( s_0 \) to a final-state represents a valid use-case in \( AFS(U). \)

\( AFS(U) \) helps us to see the relationships among the use-cases \( U \) in terms of their shared interactions. In particular, it helps us to identify inconsistencies inconsistent order of operations, invalid dataflow dependencies, and missing inputs/outputs for the use-cases \( U. \) One must, indeed, resolve all inconsistencies before attempting to create a class-structure from \( U \) for the desired software.

IV. METHODOLOGY

Given the AFS-model of a single use-case \( U, \) we use the Class-Creation-Rules below to obtain a class that supports an implementation of \( U. \) These rules can be used also, more generally, for any linear chain of interactions. Initially, the class-methods have no parameters and this has the advantage of a simple control mechanism for executing the methods in a class (see Section V-C). We may later use refactoring to introduce new methods (possibly, with parameters) for common or similar parts of the original class-methods, and replace the common parts in the original methods by calls to the new refactored methods with suitable parameter-values.

Class-Creation-Rules for a single use-case \( U: \)
1. The variables are the internal dataflows, which may include parts of \( I^{u}_U \cup O^{e}_U. \)
2. The class-methods are the operations \( op_j \in U \) or parts of them.

If \( |U| > 1, \) we can first create one class for each use-case in \( U \) and then refactor common class-variables and methods to create the final class-structure. A better method is to build the class-structure directly from the combined model \( AFS(U), \) whose tree-structure directly leads to a tree-structured class-hierarchy, with one class for each maximal linear segment (MLS) of the tree. A linear segment in \( AFS(U) \) is a path \( \pi \) in \( AFS(U), \) where each state in \( \pi \) other than those at the start and end of \( \pi \) has a single child (next) node.

We remark that the notation \( t_j = (in_{j}, op_{j}, out_{j}) \) implies that if \( t_j \in U \) is shared and equals \( t'_j = (in'_{j}, op'_{j}, out'_{j}) \in U'. \) then \( in_{j} = in'_{j} \), \( op_{j} = op'_{j} \), and \( out_{j} = out'_{j}. \) However, the dataflow \( d_j \) from \( op_{j} \) in \( U \) may differ from the dataflow \( d'_j \) from \( op'_{j} \) in \( U'. \) For a tree-structured \( AFS(U), \) with \( U, U' \in U, \) \( t_j = t'_j \) implies \( op_{j} = op'_{j}, \) can compute \( d_j \cup d'_j, \) and thus we can replace both \( d_j \in U \) and \( d'_j \in U' \) by \( d_j \cup d'_j. \) Viewed another way, this simply points out that while the classes for \( U \) and \( U' \) obtained by the Class-Creation-Rules may contain different class variables due to \( d_j \neq d'_j, \) when we merge those classes to create a class-hierarchy the class containing \( op_{j} = op'_{j} \) can include the variables for both \( d_j \) and \( d'_j. \)

We illustrate below our method by deriving a class-structure for a bank’s ATM-system with three high-level functional requirements or use-cases \( U = \{U_1, U_2, U_3\}, \) where \( U_1 = \text{successful withdrawal}, \) \( U_2 = \text{failed withdrawal due to insufficient funds}, \) and \( U_3 = \text{balance enquiry}. \) We first derive a class for each \( U_j \) using the Create-Class-Rules and then show that the class-structure obtained by refactoring these classes can be obtained directly from \( U. \)

A. Informal Description of ATM

A user swipes a debit card in the ATM’s card-slot. The ATM reads the debit card and prompts the user to enter the PIN. We assume for simplicity that no invalid ATM card or PIN is used, and there is no cash dispenser malfunction. The ATM validates the PIN and asks the user to choose one of two displayed options “withdrawal” and “balance-enquiry”. If the user selects withdrawal-option, the ATM calculates and displays the maximum allowable withdrawal amount based on the available ATM cash and debit-card-account-information. Then, the ATM asks the user to enter the withdrawal amount and it reads that amount. Then, either the ATM displays the updates to debit-card-account-info and dispenses the desired cash, or it displays a transaction-fail-message when withdrawal amount is too large. If the user selects balance-enquiry option, the ATM displays the debit-card-account-balance. In each case, the ATM writes a transaction-log for future audit analysis as part of session-closing operation.

B. Formal Description of Use-Case \( U_1 \)

We show below the decomposition of \( U_1 \) into four interactions \( t_1-t_4 \) and also show each \( op_j \) in detail, including its lower level operations. This simplifies the identification of inputs \( in_{j}, \) outputs \( out_{j}, \) dataflow items \( d_j, \) and the transition conditions \( c_{j,j+1} \) for \( U_1, \) which are shown in Tables I and
II. The data-item ATMAndOtherInfo in \(\text{in}_{j}^5\) includes ATM-cash-balance among others. The Transaction-Log-Information (TLI) has many parts, including dateTime of transaction, transaction amount, update of Bank-Debit-card-Account-Detail-Info, etc. Different \(\text{out}_{j}^5\) includes different parts of TLI.

\[\text{U}_1: \text{Successful withdrawal}\]

\(t_1:\) User swipes the debit card in the ATM's card slot and enters PIN when requested. \([\text{op}_1]: \) ATM reads the debit card number, then reads Bank-Debit-Card-PIN Information based on the debit card number, displays "Enter PIN" message, reads the PIN entered and validates it with Bank-Debit-Card-PIN information, reads Bank-Debit-Card-Account-Detail Information for the debit card number, and writes part-1 (debit card number, PIN, and transaction dateTime) of TLI.

\(t_2:\) User sees the displayed transaction options and selects the "withdrawal" option. \([\text{op}_2]: \) ATM displays the transaction-options "balance-enquiry" and "withdrawal", requests the user to select an option, reads the selected option (= "withdrawal" for \(\text{U}_1\)), and writes part-2 (transactionOption = withdrawal) of TLI.

\(t_3:\) User sees the displayed max allowed withdrawal amount and enters the desired withdrawal amount. \([\text{op}_3]: \) ATM calculates and displays the max allowed withdrawal amount based on the ATM-cash-balance (which is part of the external input ATMIdAndOtherInfo) and the debit-card-account-detail information (such as the single-transaction-limit, the daily-withdrawal-limit for the card, today’s-current-total-withdrawal, and the debit-card-account-balance). Then, ATM requests user to enter the desired-withdrawal-amount, reads it, and writes part-3 (max allowed withdrawal amount) of TLI.

\(t_4:\) User takes the dispensed cash. \([\text{op}_4]: \) ATM updates ATM-cash-balance part of ATMIdAndOtherInfo and the Bank-Debit-Card-Account-Detail information (such as today’s-total-withdrawal and debit-card-account-balance), displays the updated debit-card-account-balance, displays "collect-cash" message (which includes the amount withdrawn), dispenses cash for the withdrawal amount, and writes part-4 (updates of ATM-cash-balance, this withdrawal amount, debit-card-account-balance) of TLI.

\[\text{TABLE I. Inputs } \text{in}_{j} \text{ and outputs } \text{out}_{j} \text{ for } \text{U}_1\]

<table>
<thead>
<tr>
<th>(\text{in}_{1})</th>
<th>(\text{out}_{1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{u: Debit-card number, PIN number})</td>
<td>(\text{u: } \text{&quot;Enter PIN&quot; message})</td>
</tr>
<tr>
<td>(\text{c: Bank-Debit-Card-PIN Info, Bank-Debit-Card-Account-Detail Info})</td>
<td>(\text{c: Part-1 of TLI})</td>
</tr>
<tr>
<td>(\text{in}_{2})</td>
<td>(\text{out}_{2})</td>
</tr>
<tr>
<td>(\text{u: Selected Transaction Option})</td>
<td>(\text{u: Transaction-options display, &quot;Select Option&quot; message})</td>
</tr>
<tr>
<td>(\text{c: Bank-Debit-Card-Account-Detail Info})</td>
<td>(\text{c: Part-2 of TLI})</td>
</tr>
<tr>
<td>(\text{in}_{3})</td>
<td>(\text{out}_{3})</td>
</tr>
<tr>
<td>(\text{u: Desired Withdrawal Amount ATM-id-And-Other Info})</td>
<td>(\text{u: Max Allowable Withdrawal Amount display, &quot;Enter Desired Withdrawal Amount&quot; message})</td>
</tr>
<tr>
<td>(\text{c: ATMIdAndOtherInfo, &quot;Collect Cash&quot; message})</td>
<td>(\text{c: Part-3 of TLI})</td>
</tr>
<tr>
<td>(\text{in}_{4})</td>
<td>(\text{out}_{4})</td>
</tr>
<tr>
<td>(\text{u: Cash Collected})</td>
<td>(\text{u: New BankAccountDetailInfo, &quot;Collect Cash&quot; message})</td>
</tr>
<tr>
<td>(\text{c: New BankAccountDetailInfo, ATMIdAndOtherInfo, Part-4 of TLI})</td>
<td>(\text{c: Part-4 of TLI})</td>
</tr>
</tbody>
</table>

In general, the conjunction of \(c_{j,j+1}'s\) for the interaction-sequence of a use-case \(U\) does not give a pre-condition for \(U\) because each \(c_{j,j+1}\) is stated in terms of values of data-items "after" the operation \(\text{op}_j\). For \(U_1\), \(c_{2,3} \wedge c_{3,4}\) does give its pre-condition. We consider \(d_2\) to be an "implicit" dataflow from \(\text{op}_2\) to \(\text{op}_3\) because execution of \(\text{op}_3\) requires \(c_{2,3}\) to be true. Similarly, we consider \(d_3\) to be an implicit dataflow from \(\text{op}_3\) to \(\text{op}_4\). (The controller to drive the execution of the methods in the class for \(U_1\) will use \(d_2\) and \(d_3\); see Section V-C.) See Figure 3, which shows the finite-state machine model and the dataflow model for \(U_1\). There is no dataflow from \(t_1\) to \(t_2\).

\[\text{TABLE II. Conditions } \text{c}_{j,j+1} \text{ and data-items } \text{d}_j \text{ for } \text{U}_1\]

<table>
<thead>
<tr>
<th>(\text{c}_{1,2})</th>
<th>(\text{d}_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{true (no invalid card or PIN)})</td>
<td>(\text{Bank-Debit-Card-Account-Detail Info})</td>
</tr>
<tr>
<td>(\text{&quot;Withdrawal&quot; = Selected Transaction Option})</td>
<td>(\text{Selected Transaction Option})</td>
</tr>
<tr>
<td>(\text{Max Allowable Withdrawal Amount } \geq \text{ Desired Withdrawal Amount})</td>
<td>(\text{Max Allowable Withdrawal Amount, Withdrawal Amount Desired})</td>
</tr>
</tbody>
</table>

Figure 3. FSM (left) and DFD (right) for the use-case \(U_1\)

**C. AFS Model of \(U_1\)**

Figure 4 shows \(\text{AFS}(U_1)\), obtained by combining the DFD and the FSM shown in Figure 3. It also shows the SuccessfulWithdrawal-class obtained from \(\text{AFS}(U_1)\) based on Tables I and II and the Class-Creation-Rules, and it is
suitable for implementing $U_3$. The detailed analysis of each $t_j$ identified several low level functions (methods) for the associated $op_j$, and we have labeled each of those low-level functions as "//op j" in SuccessfulWithdrawal-class. One could merge the functions with the same label "//op j" into a single function, and call it $op_j$. If we let each merged function $op_j$ have parameters corresponding to the dataflows to it and let each $op_j$ call $op_{j+1}$, then we can eliminate all the class variables except $d_1$; even $d_1$ can be eliminated if we use a parameter in $op_2$ and pass $d_1$ for it when called by $op_1$. But we keep the class variables as shown to simplify the design of a single general purpose controller for any sets of use-cases.

D. Merging and Decomposing Interactions in $U_1$

We can merge interactions $t_1$ and $t_2$ or, equivalently, states $s_1$ and $s_2$ in Figure 4 into a single state because $c_{1,2} = true$. This will not adversely affect handling $U_2$ and $U_3$ because both $t_1$ and $t_2$ are common to $U_2$ and $U_3$. We do not merge $s_2$ and $s_3$ because that would prevent sharing $s_2$ (and its associated operations, inputs, and outputs) between $U_1$ and the use-cases $U_2$ and $U_3$. On the other hand, we do not decompose a $t_j, 1 \leq j \leq 4$, into several $t_{j,k}$’s corresponding to the methods in Figure 4 having the label "//op j" because that does not give us a better sharing of $t_{j,k}$’s among the use-cases $U = \{U_1, U_2, U_3\}$, and thus does not help in the design of a class-structure for $U$.

E. Classes and Formal Description of Use-Cases $U_2$ and $U_3$

Shown below are the decompositions of the use-cases $U_2$ and $U_3$ into their component interactions. The use-case $U_2$ shares its first three interactions with $U_1$ and $U_3$ shares its first two interactions with $U_1$. Note that $t_{10}$ in $U_3$ is the same as $t_2$ in $U_1$, even though the value of data-item $in_1 = in_2$ is different in $U_1$ and $U_3$; that difference is reflected in $c_{2,3} \neq c_{10,11}$. The condition $c_{7,8} = \neg c_{3,4}$ gives $c_{7,8} \land c_{8,7} = false$ and $c_{3,4} \land c_{7,8} = true$. The pre-condition for $U_2$ is $c_{2,3} \land c_{3,8}$ and that for $U_3$ is $c_{2,11}$.

$U_2$: Failed withdrawal due to insufficient funds

$t_{j+4}$: Same as $t_j$ in $U_1$ for $1 \leq j \leq 3$.

t_8: User sees “insufficient funds” message. [op8: ATM displays insufficient funds message for the desired withdrawal amount, and writes part-5 (“failed withdrawal”, withdrawalAmount = 0) of TLI]

$U_3$: Balance enquiry

$t_0$: Same as $t_1$ in $U_1$.

t_10: Same as $t_2$ in $U_1$ except that the user selects the “balance-enquiry” option.

t_{11}: User sees account balance information. [op11: ATM displays the account balance and writes part-6 (“balance enquiry”) of TLI]

Table III gives the inputs and outputs for the interactions $t_8$, $t_{10}$ and $t_{11}$ in the use-cases $U_2$ and $U_3$. Table IV gives the conditions for these new interactions.

F. AFS Models for $U_2$ and $U_3$

We do not show the FSM and DFD for $U_2$ and $U_3$, but their AFS models and the corresponding classes are shown in Figures 5 and 6. As in the case of $U_1$, we could merge the functions (methods) with the same label "//op j" in Figures 5 and 6 into a single function and call it $op_j$.

G. Merging of States for $U_2$ and $U_3$

We do not merge $s_2$ with $s_3$ or $s_{11}$ because $c_{2,3}$ and $c_{2,11}$ are disjoint; likewise, we do not merge $s_3$ with $s_4$ or $s_8$.

V. CLASS STRUCTURE AND IMPLEMENTATION

The classes in Figures 4-6 together allow us to implement the ATM described in section IV-A. We get the class-hierarchy shown in Figure 8 when we eliminate the duplicate attributes and methods in these classes using refactoring and combine the classes into a hierarchy. We can also directly get the same class-hierarchy, without creating the classes in Figures 4-6, from the combined AFS model for $U_1-U_3$ shown in Figure 7.

A. Combining AFS Models

We use the following notion of equivalent states to combine two AFS models $M = AFS(U)$ and $M' = AFS(U')$ for the sets of use-cases $U$ and $U'$. Two states $s_j \in M$ and $s'_j \in M'$ are equivalent if all computations along the path $\pi(s_j)$ from the start-state of $M$ upto $s_j$ are identical to those along the path $\pi'(s'_j)$ from the start-state of $M'$ upto $s'_j$ in terms of the

| TABLE III. Inputs and outputs for $U_2$ and $U_3$ that are different from $U_1$ |
|-----------------|-----------------|
| $in_8$ | $\emptyset$ (empty) |
| $out_8$ | "Insufficient Funds" message |
| $in_{10}$ | $u$: Selected Transaction Option |
| $out_{10}$ | $\emptyset$ (empty) |
| $in_{11}$ | $\emptyset$ (empty) |
| $out_{11}$ | $u$: Account Balance e: Part-6 of TLI |

| TABLE IV. Conditions $c_{j,k}$ for $U_2$ and $U_3$ that are different from $U_1$ |
|-----------------|-----------------|
| $c_{7,8} = c_{3,4}$ | Max Allowable Withdrawal |
| $\neg c_{3,4}$ | Amount + Withdrawal Amount Desired |
| $c_{9,10} = c_{1,2}$ | true (no invalid Card) |
| $c_{10,11} = c_{2,11}$ | "Balance Enquiry" = Selected Transaction Option |

Figure 5. AFS model and class for use-case $U_2$
order of computations and the underlying constraints. To be precise, \( s_j \) and \( s'_j \) are equivalent if

1) For \( k < j \), \( s_k \) is equivalent to \( s'_k \).
2) The dataflows to \( s_j \) and \( s'_j \) are identical in terms of the data-sources and the data-items.
3) The operations at \( s_j \) and \( s'_j \) are the same: \( op_j = op'_j \), including the inputs \( in^{u}_i = in^{u}_i \) and \( in^{e}_i = in^{e}_i \) and the outputs \( out^{u}_j = out^{u}_j \) and \( out^{e}_j = out^{e}_j \).
4) The dataflows from \( s_j \) and \( s'_j \) are the same (\( d_j = d'_j \)).
5) The condition \( c_{j-1,j} \) in \( M' \) is the same as \( c'_{j-1,j} \) in \( M \), i.e., \( c_{j-1,j} = c'_{j-1,j} \).

We obtain the same class structure in Figure 8 if we start with the classes in Figures 4-6 and apply refactoring [7].

![Diagram](image_url)

**Figure 7. Combined AFS model for \( U_1-U_3 \).**

**Figure 8. Class-hierarchy from the AFS model in Figure 7.**

It is worth pointing out that if we apply the concept analysis technique [12] to the attributes (variables) and methods of the classes in Figures 4-6, based on the use-relationships between those variables and methods, then we would arrive basically at the same final class structure in Figure 8, except that each of the classes ATMTransaction and Withdrawal will become a chain of simpler classes (involving a partitioning of variables and methods in those classes). We will then simply merge those chains to form the classes ATMTransaction and Withdrawal as given in Figure 8. Note that a major part of building \( AFS(\mathcal{U}) \) involves, via the details of the interactions in the use-case \( \mathcal{U} \), the identification of all class variables and methods, and their use-relationships, and the latter are the inputs to concept analysis. The only part of \( AFS(\mathcal{U}) \) which does not explicitly appear in the class-structure and is not used in concept analysis.
is the transition-conditions; they play, however, a critical role in the correctness of the tree-structure of \( AFS(\mathcal{U}) \), its unique decomposition into maximal linear segments (like \((s_1, s_2)\)), and in developing the controller (see Section V-C) to drive the execution of the methods in the class-structure. In this sense, our approach based on the AFS model is superior to the concept analysis method. After all, there is not much value in a class-structure design unless we clearly understand how to control the calls to its methods.

C. Implementation

Figure 9 shows the execution dependencies among the methods for the AFS model in Figure 7. Here, \( f_j \) represents the group of methods corresponding to \( op_j \) (see Figure 8) for state \( s_j \). A link \((f_i, f_j)\) implies the execution of \( f_j \) follows that of \( f_i \), partly because \( d_i \) generated by \( f_i \) is required by \( f_j \) for its computations or \( d_i \) is needed in determining whether \( f_j \) can execute or not. It is not surprising that the links \((f_i, f_j)\) in Figure 9 parallel the links \((s_i, s_j)\) in Figure 7.

![Controller for executing \( f_j \)'s](image)

**Figure 9.** The execution dependency among methods \( f_j \) corresponding to \( op_j \) in Figure 8; \( d_j \)'s are the dataflows.

There are many ways [13] to implement the dependencies in Figure 9. The simplest and the best solution is a central controller that works as follows. It first calls the root function \( f_1 \), and following the execution of an \( f_j \) it tests the disjoint conditions \( c_{j,k} \) for the "child" functions \( f_k \) of \( f_j \) and calls \( f_k \) if \( c_{j,k} \) is true. Another approach is to start with the root function \( f_1 \) as before but let each \( f_j \) evaluate the conditions \( c_{j,k} \) and call \( f_k \) if \( c_{j,k} \) is true, as its last step. Here, changes in \( \mathcal{U} \) may require small modifications to several \( f_j \)'s, depending on how many classes are affected. In the first approach, the modifications to the central controller can be completely automated. As a third alternative, we can introduce parameters to \( f_j \)'s and let each \( f_j \) store \( d_j \) as a local variable in it and use it as one of the parameters in the call to an \( f_k \). For Figure 9, this means \( f_1 \) executes first and ends with a call to \( f_2 \), with \( d_1 \) as the parameter. Next, \( f_2 \) ends with a call to \( f_3 \) or \( f_11 \), with \( d_1 \) and \( d_2 \) as parameters. Likewise, \( f_3 \) ends with a call to \( f_4 \) with \( d_1, d_2 \), and \( d_3 \) as parameters, etc. This approach requires fewer class-variables, but changes in requirements may cause many changes in the definition and selection of the parameters. This approach clearly gives a poor quality software.

VI. CONCLUSION

We have presented here a systematic, semi-formal method to obtain a hierarchical class-structure, including the attributes and methods for each class in the hierarchy, for an object-oriented design of a software from its requirements given in the form of a set of use-cases. We use two formal models: (1) a detailed model of a use-case in terms of its interactions (operations), which includes the user and external inputs/outputs of each operation and the dataflows to/from it, and (2) an Augmented Finite State (AFS) model for a set of use-cases, which captures shared operations among the use-cases, the points-of-divergence between use-cases and the related control-flow conditions. The AFS model can help to identify missing use-cases, missing interactions in the use-cases, and the potential need to decompose some interactions into simpler ones. If the AFS model has a tree structure, then this directly gives a hierarchical class-structure suitable for an object oriented implementation of the requirements. This means, in principle, one could start from, say, a C-program \( P \) and reverse engineer it to obtain an AFS model for it, and if the AFS model has a tree structure then create a hierarchical class-structure from that AFS model, and finally obtain an object-oriented program \( P' \) with the same functionality as \( P \).

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for some useful comments to improve the presentation.

REFERENCES