Orthogonal Decomposition of Finite-State Behavior Models As a Basis for Determining Components in Software Architectures

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

Abstract

We present a theoretical foundation for determining the components of a software's architecture from its finite-state behavior model. We introduce the notion of orthogonal decomposition of a finite-state model and show that the components in such a decomposition correspond to the components of a suitable architecture for the software. This gives us a systematic method for partitioning the operations in the software into the components. We give examples to show that our method succeeds even when an intuitive approach in identifying the components fails.

Keywords: Software architecture, Behavior Model.

1. Introduction

The term "component" is used informally in a wide variety of meanings in Software Architecture, Software Design, and Component-Based software development. The lack of a formal definition of a component in software architecture, in particular, prevents us developing a systematic method for determining the components and the architecture of a software from, say, its functional requirements. The informal definitions used in various architecture description languages (such as WRIGHT [1], ACME [2], RAPIDE [6, 7] and others) merely describe some features of a component (such as ports) but do not define their roles in the total computation in a way that tells us what can or cannot be considered a component for a given software. Selecting a suitable set of components involve partitioning the basic operations (functions) of the software into disjoint groups in such a way that simplifies the coordination (communications and control) among the groups, with the operations within each group tightly coupled as a semantic unit. Interestingly, much attention has been paid in the literature on how to describe a software architecture, including its components and the communications, using graphical notations and other programming-language like formalisms but there is virtually no work on how can one proceed systematically to determine the components themselves. A similar remark holds for the class-hierarchy design for a software. There are a wide variety of notations in UML [8], for example, to describe a class-hierarchy, but there is very little known [3, 4] about how to determine a good (or optimal, in some sense) class-hierarchy for a given software. We address here the problem of determining the components in a software's architecture from its finite-state behavior model.

The architecture of a software gives a more abstract and also a more complete view of the software than a class-hierarchy design; the latter supports, and is influenced by, the architecture instead of the other way. A software's requirement can be specified effectively by a finite-state behavior model, which shows the valid operation-sequences, and the use-relationship between the data-items and the basic operations (functions) in the software, which shows the inputs and outputs of each operation. We can determine a proper class-hierarchy based on the finite-state model and the use-relationship [3, 4] by organizing the data-items and operations in various classes in a way that fits the use-relationship. The class-hierarchy (together with the use-relationship) gives the static-part of the software's design and the finite-state model gives the dynamic part. The class-hierarchy does not, however, give a full representation of the software. It does not show how many objects are there of any given type (class) and when they are created (or destroyed); also, it may not show how two objects that are unrelated via the class-subclass relationship interact with each other, unless there is an aggregate class containing both of them. One employs use-cases (scenarios) to augment a class-hierarchy in order to describe the interactions among two or more such objects. The use-cases describes an operation sequence in a much more low-level details and can be included in the behavior model of software's architecture.

2. A Simple Architectural Model

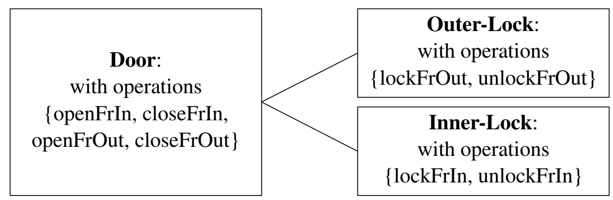
For our purpose, it suffices to view an architecture as an undirected graph G consisting of a collection of nodes (components) and a collection of undirected links (connectors) among pairs of nodes, with the following meanings. Each node x provides a set of services or functionalities $F(x)$ in response to events in the external environment, where the sets $F(x)$ are mutually disjoint ($F(x) \cap F(y) = \emptyset$ for $x \neq y$). This means only one component reacts to any event in the environment; this is the case if we assume sufficiently fine granularity of the events. For each node x, we also have a finite-state behavior model $M(x)$ which describes the valid sequences of operations in $F(x)$. We assume that all finite-state models are deterministic and are in minimum form (for this all states are regarded as final-states, [5]). We use an event-name to refer also to the function (operation) that responds to that event. The link (x, y) between two nodes x and y indicates that they can exchange information between them, which consists of the states of the nodes, i.e., the states of $M(x)$ and $M(y)$. We model such information from y to x as conditions or guards on the transitions in $M(x)$. For most part, the graph G will be connected, but this is not necessary as will be seen. (We do not consider "internal" events or, equivalently, function-calls as part of such information flow between nodes; for our purpose, we can model them just as any other external events. We also do not consider parallel operations or multiple events at any given time point.)

A guard for a transition in $M(x)$ may depend on the event. If we let $N(x) = \{y\text{: } (x, y) \text{ is a link in } G\}$ denote the neighbors of x, then a guard can be an arbitrary boolean expression involving the states of $M(y)$, $y \in N(x)$. The guard for the transition in $M(x)$ for a given state-event combination means that the transition can take place only if the states in $M(y)$ for the various neighbors y of x satisfy the boolean expression in the guard. The graph G gives the architecture's static part and the finite-state models $M(x)$ taken together gives its dynamic part.

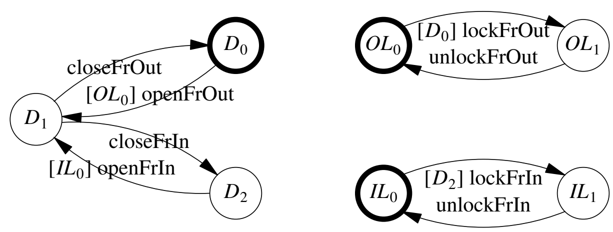
We can assume that the nodes inform their neighbors about their current states using a shared state-vector SV, where $SV(x)$ = the current state of $M(x)$. Only node x has the write-access to $SV(x)$ and each $y \in N(x)$ has the read-access to $SV(x)$. Following each transition in $M(x)$, x updates $SV(x)$, instead of having to inform its neighbors individually the current state of $M(x)$. Since $F(x)$'s are disjoint, only one $SV(x)$ is modified at any time.

Example 1. Figs. 1(i)-(ii) show an architecture for a door with a two-sided lock. We assume that initially the door is closed from outside and unlocked from both sides. The door can be locked from outside only if it is closed from outside, and likewise it can be locked from inside if it is closed from inside. The guard $[D_0]$ in the transition for lockFrOut-event in $M(OL)$ in Fig. 1(ii) and the guard $[D_2]$ in the transition for lockFrIn-event in $M(IL)$ in Fig. 1(ii) indicate these constraints. The same information is represented more directly in the complete model $M(DL)$ for the door with two-sided lock shown in Fig. 1(iii), which do not use guards. We also assume that the door can be closed (and hence locked) only from one side at a time, and it can be unlocked only from the side it is locked; see the transitions for unlockFrOut and unlockFrIn in $M(DL)$. Finally, the door can be opened from a given side only if it is closed but unlocked from that side. This is captured by the guards $[OL_0]$ and $[IL_0]$ in $M(D)$ in Fig. 1(ii). There is no link between the components Outer-Lock and Inner-Lock in Fig. 1(i) because none of the guards in any of $M(OL)$ and $M(IL)$ involves a state of the other. Once the door is open, it can be closed from either side (imagine the person operating on the door can move in and out of the room when the door is open).

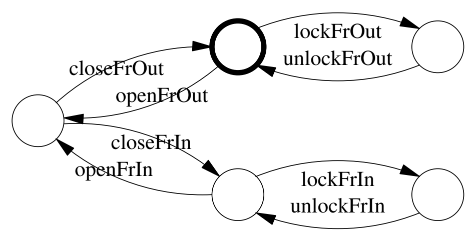
One reason for using distinct names for the outside and the inside operations on the door is that they may involve different details of pushing/sliding the door in different ways; also the operations lockFrOut and unlockFrOut may involve a parameter for the key of the lock that must be matched for these operations to be successful while the operations lockFrIn and unlockFrIn may involve a parameter to indicate, say, push botton or remote control operation, without a need for the key matching. □



(i) An architecture with three nodes and two links.



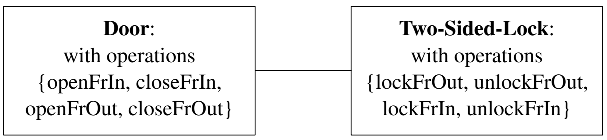
(ii) The finite-state models $M(D)$ for door, $M(OL)$ for outer-lock, and $M(IL)$ for inner-lock; bold circles indicate start-states.



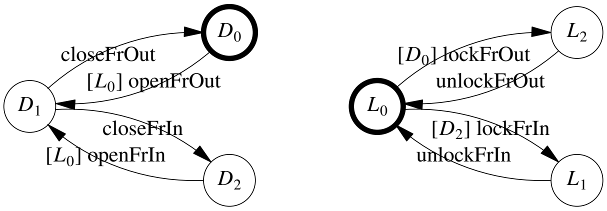
(iii) The complete finite-state model $M(DL)$ for the door with two-sided lock; the bold circle indicates the start-sate.

Figure 1. An example architecture for a door with sided lock.

Example 2. Fig. 2(i) shows an alternate architecture for the door with two-sided lock. The guards $[OL_0]$ and $[IL_0]$ in various transitions in $M(D)$ in Fig. 1(ii) are now replaced by $[L_0]$ because the state L_0 (= "unlocked") in the finite-state model $M(L)$ for the two-sided lock in Fig. 2(ii) corresponds to the (and-)combination of states OL_0 in $M(OL)$ and IL_0 in $M(IL)$. Likewise, the state L_2 in $M(L)$ corresponds to the state combination (OL_1, IL_0) in Fig. 1(ii) and the state L_1 to (OL_0, IL_1). If we decide to decompose the node Two-Sided-Lock into two parts as in Fig. 1 and hide this from the node Door, then we need to have an interface in Two-Sided-Lock for updating the state information in SV properly for use by $M(D)$. □



(i) An architecture with two nodes and one link.



(ii) The finite-state models $M(D)$ for door and $M(L)$ for two-sided-lock.

Figure 2. An alternate architecture for the door with two-sided lock.

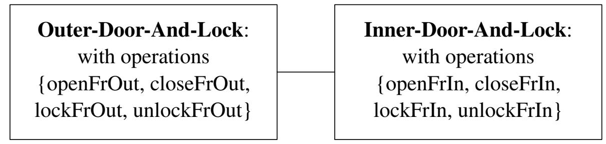
Example 3. Fig. 3 shows yet another architecture for the door with two-sided lock having two nodes. We consider this to be simplest of the three architectures in terms of the dependence among the nodes (fewest number of guards and fewest number of conditions in them) although one may intuitively feel that Fig. 2 is the best architecture because of the door and the lock being modeled separately in keeping with our usual perception of the physical reality. The finite-state models for the two nodes in Figs. 2 and 3 have the same number of states and transitions. □

The above examples show that there can be many ways of forming the components in an architecture for a given set of operations, including the number of components.

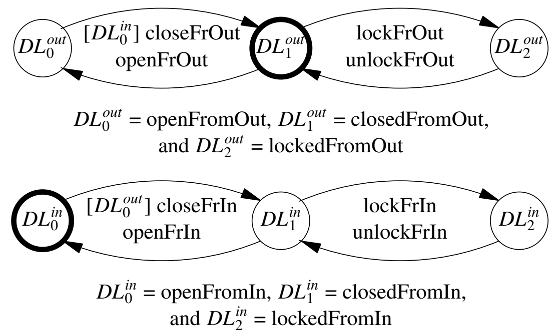
2.1. Role of event-names

In programming languages, the scope and meaning (polymorphism) of variable and function names play a significant role. This in turn have had much impact on software design and development. We show that event-names have a similar important role in software architecture.

It is clear from the finite-state model in Fig. 1(iii) that if we were to use the same name "lock" for the events lockFrOut and lockFrIn, there would be no confusion because the states (contexts) in which they are applicable distinguishes them. In the software implementation, we



(i) Another architecture with two nodes and one link.

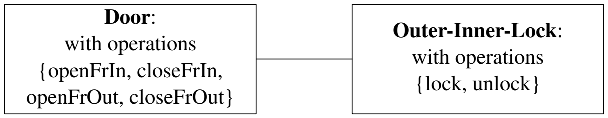


(ii) The finite-state models $M(DL^{out})$ for outer-door-and-lock and $M(DL^{in})$ for inner-door-and-lock.

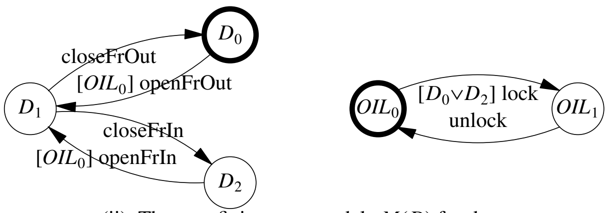
Figure 3. Yet another architecture for the door with two-sided lock.

can use the state information to distinguish between the two versions of lock-event and use the appropriate function and parameters to process each version of the lock-event. For the same reason, we could use the same name "unlock" for the events unlockFrOut and unlockFrIn. Fig. 4(iii) shows the new finite-state model for the door with two-sided lock and Fig. 4(i) shows the simplified form of the architecture in Fig. 1(i). Note the guard $[D_0 \vee D_2]$ for the lock-event in the finite-state model $M(OIL)$ for the two-sided lock in Fig. 4(ii). If the condition D_0 holds then we use lockFrOut operation for the lock-event, and if the condition D_2 holds then we use lockFrIn operation for the same lock-event; since the door-state cannot change from D_0 or from D_2 until the next unlock event, the same guard conditions can be used to decide whether to use unlockFrOut operation or unlockFrIn operation for the unlock-event. The architecture in Fig. 2(i) too reduces to that in Fig. 4(i) because of this name change, with the finite-state model $M(L)$ in Fig. 2(ii) reducing to $M(OIL)$ in Fig. 4(ii), when minimized; the state OIL_0 in $M(OIL)$ corresponds to the state L_0 in $M(L)$ and the state OIL_1 in $M(OIL)$ corresponds to the set of states $\{L_1, L_2\}$ in $M(L)$.

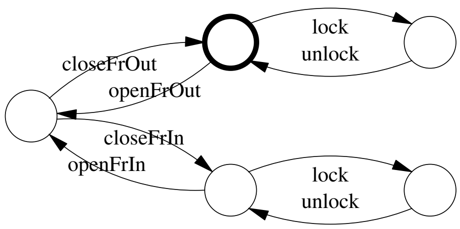
We remark that we cannot additionally use the same name "open" for the events openFrIn and openFrOut in Fig. 4 (or in either of Figs. 1 and 2). This is partly because the states D_0 and D_2 in $M(D)$ would now become equivalent and that would prevent proper interpretation of lock-event as lockFrOut or lockFrIn. Also, in Fig. 4(iii) the start-state and the state directly below it, i.e., the state resulting from closeFrIn-event would become equivalent



(i) A simplified form of the architecture in Fig. 1(i).



(ii) The new finite-state models $M(D)$ for door and $M(OIL)$ for outer-inner-lock.



(iii) The finite-state model for the door with two-sided lock without from-out and from-in distinctions for lock/unlock events.

Figure 4. A new architecture for the door with two-sided lock based on a slightly different finite-state model.

and thus it would no longer properly represent the door with two-sided lock. This shows the importance of naming events properly to distinguish different situations.

If we, however, keep the distinct names lockFrIn and lockFrOut, then we could use the same name open for openfrIn and openFrOut and also use the same name unlock for unlockFrIn and unlockFrOut. The model $M(DL)$ in Fig. 1(iii) with the name changes would continue to represent the door with two-sided lock properly, and all three models $M(D)$, $M(OL)$, and $M(IL)$ in Fig. 1(ii) with the name changes work properly for the architecture in Fig. 1(i), except that we need to relax the disjointness of $F(x)$'s. A different situation arises for the architecture in Fig. 2. The states D_0 and D_2 in Fig. 2(ii) now become equivalent and must be merged, but then whenever lockFrIn is applicable in $M(L)$ we have lockFrOut is also applicable, which was never meant to be the case. Thus, this architecture is no longer valid (the theory of orthogonal decomposition presented in Section 3 also give us the same result). The architecture in Fig. 3(i), however, remains valid with these name changes, including the models in Fig. 3(ii). Since $M(DL^{out})$ and $M(DL^{in})$ cannot be in the states DL_1^{out} and DL_1^{in}, respectively, at the same time, it is possible to select the appropriate operation openFrIn or openFrOut in response to open-event.

2.2. Poor choice of event-names

We now give another example of a poor choice of event names that makes the finite-state model for the door with two-sided lock more complex and, more importantly, the new model does not allow any decomposition of the events (functions) to form a non-trivial architecture with two or more components.

Suppose we use the same name "open" for both openFrOut and openFrIn and the same name "close" for both closeFrOut and closeFrIn in the model $M(DL)$ in Fig. 1(iii). This gives us non-determinism for the close-event at the leftmost state; the two transitions from this state for the close-event result in two different states. Fig. 5 shows the proper finite-state model for the door with two-sided lock in this case; it is more complex than the model $M(DL)$ in Fig. 1(iii) with one more state and two more transitions. (Note that this is same as the deterministic model obtained from the non-deterministic $M(DL)$ by the subset construction in [5].) We need to keep distinct names for the from-out and from-in versions of the lock/unlock events to avoid additional non-determinism. The theory of orthogonal decomposition of finite-state models given in Section 3 shows that the model in Fig. 5 has no orthogonal decomposition and hence no non-trivial architecture with two or more components (unless we generalize our notion of architecture).

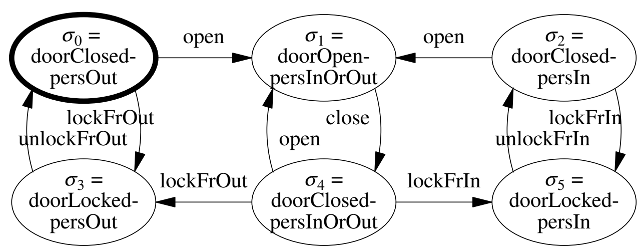


Figure 5. The finite-state model for the door with two-sided lock and the same name for two forms of open/close.

3. Orthogonal Decomposition

After we briefly review some basic terminology, we define the notion of orthogonal decomposition of a finite-state model M into a set of submodels or projections of M. We then show that each of the architectures given in Figs. 1-4 corresponds to an orthogonal decomposition of the finite-state model $M(DL)$ for the door with two-sided lock. Finally, we give a necessary and sufficient condition for a collection of projections of M to form an orthogonal decomposition.

Given a finite-state model M and a partition of the events (operations) in M into a family of disjoint subsets $F(x)$, the corresponding architecture (if any) is completely determined. We first determine each $M(x)$, including the guards for its transitions, from $F(x)$ and M via the projection operation defined below. The architecture is then obtained by simply defining one node x for each $M(x)$ and defining a connecting line between x and y if some transition in one of $M(x)$ and $M(y)$ uses the states of the other in its guard. (We can use a directed link $x \rightarrow y$ to indicate more specifically that some transition in $M(x)$ uses the states of $M(y)$ in its guard, and similarly for the directed link $y \rightarrow x$.) In particular, if the finite-state behavior model M of a software does not have an orthogonal decomposition, then the software has no non-trivial architecture with two or more components.

3.1. Some basic notations

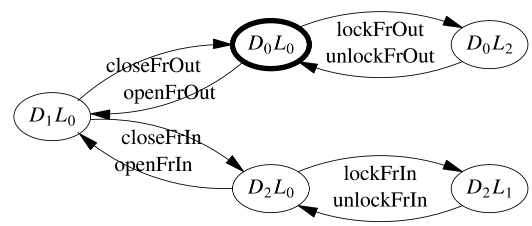
Let $\Sigma = \Sigma(M)$ denote the set of states of a finite-state model M and $E = E(M)$ the set of events in M, i.e., the labels of the transitions in M. We denote the transition from a state $\sigma_i \in \Sigma$ to the state $\sigma_j \in \Sigma$ for an event $e \in E$ by $t(\sigma_i, e, \sigma_j)$ or simply by $t(\sigma_i, e)$; if the event e is understood from the context, we also denote the transition simply by $t(\sigma_i, \sigma_j)$. In order to relate a transition $t(\sigma_i, e)$ in M with other finite-state models M', we can add a guard $[\sigma'_j]$, $\sigma'_j \in \Sigma(M')$, to the label e for the transition. This indicates that the transition at state σ_i in M for the event e is applicable only if M' is in state σ'_j. The guard $[\neg\sigma'_j]$ would mean on the other hand that M' must be in a state other than σ'_j for that transition to be applicable. We also allow arbitrary boolean combination of states from the same or different models to form a guard. Note that a boolean term of the form $\sigma'_j \wedge \sigma'_k$ in a guard is simplified to F (false) if σ'_j and σ'_k are different states in the same M'; however, we cannot simplify $\sigma'_j \wedge \sigma''_k$ if the states σ'_j and σ''_k come from two different finite-state models M' and M''. If the guard for a transition simplifies to F, then this amounts to not having that transition at all; on the other hand, if the guard simplifies to T, then this amounts to not having the guard. We write a guarded transition as $t(\sigma_i, [g]e, \sigma_j)$ or simply as $t(\sigma_i, [g]e)$, where g denotes the guard-condition. Since we consider only deterministic finite-state models, two guarded transitions $t(\sigma_i, [g]e)$ and $t(\sigma_i, [g']e)$ for the same event e at a given state σ_i implies that $g \wedge g' = F$. Also, we can replace two transitions $t(\sigma_i, [g]e, \sigma_j)$ and $t(\sigma_i, [g']e, \sigma_j)$ between the same pair of states σ_i and σ_j for an event e by the single transition $t(\sigma_i, [g \vee g']e, \sigma_j)$). For minimizing a finite-state model with guards on its transitions, we regard guard-event combinations as distinct events and all states as final.

It is convenient to assume initially that there are no guards in M or any of its projections as defined below. Since the guards are artifacts due to the presence of other finite-state models, we describe the method of assigning guards to the transitions in a projection separately.

Def. 1. Given a proper subset $E' \subset E = M(E)$, we define the *projection* $M' = M(E')$ of M on E' to be the submodel of M when we restrict the events to E'. □

We obtain M' by first replacing each event $e \in E - E'$ in M by the empty-event, meaning that the state-change in the corresponding transitions in M take place without any event. This gives a non-deterministic finite-state model. We then convert it to a deterministic model (using the subset construction [5]) and then minimize the model by merging equivalent states [5], if any. Each state $\sigma'_i \in \Sigma' = \Sigma(M')$ corresponds to a subset $S(\sigma'_i)$ of the states $\Sigma(M)$.

Example 4. Fig. 6(i) shows the finite-state model $M(DL)$ in Fig. 1(iii) renamed as M_{DL} and with its states named in a particular way that will become clear shortly. For E' = {openFrOut, closeFrOut, openFrIn, closeFrIn}, the projections $M_{DL}(E')$ and $M_{DL}(E - E')$ equal the finite-state models $M(D)$ and $M(L)$ in Fig. 2(ii), respectively. The mapping of the states of the projections to the subsets of states of M_{DL} is shown in Fig. 6(ii). A state in Fig. 6(i) is named as $D_i L_j$ if it belongs to $S(D_i) \cap S(L_j)$. For E'' = {openFrOut, closeFrOut, lockFrOut, unlockFrOut}, we get the projections $M_{DL}(E'') = M(DL^{out})$ and $M_{DL}(E - E'') = M(DL^{in})$ in Fig. 3(ii); the corresponding mapping of the states of the projection is shown in Fig. 6(iii). □



(i) The finite-state model M_{DL} with its states renamed to relate them to the projections in Figs. 2(ii).

$S(D_0) = \{D_0L_0, D_0L_2\}$	$S(L_0) = \{D_0L_0, D_1L_0, D_2L_0\}$
$S(D_1) = \{D_1L_0\}$	$S(L_1) = \{D_2L_1\}$
$S(D_2) = \{D_2L_0, D_2L_1\}$	$S(L_2) = \{D_0L_2\}$

(ii) The mapping of the states of the projections $M(D)$ and $M(L)$ in Fig. 2(ii) to the subsets of states in (i).

$S(DL_0^{out}) = \{D_1L_0, D_2L_0, D_2L_1\}$	$S(DL_0^{in}) = \{D_0L_0, D_0L_2, D_1L_0\}$
$S(DL_1^{out}) = \{D_0L_0\}$	$S(DL_1^{in}) = \{D_2L_0\}$
$S(DL_2^{out}) = \{D_0L_2\}$	$S(DL_2^{in}) = \{D_2L_1\}$

(iii) The mapping of the states of the projections $M(DL^{out})$ and $M(DL^{in})$ in Fig. 3(ii) to the subsets of states in (i).

Figure 6. The finite-state model M_{DL} with its states renamed to relate them to the projections in Figs. 2(ii).

3.2. Adding guards to a projection

A component in an architecture must be as much as possible independent of the other components. In particular, it must not make any change of state in its finite-state model (i.e., must not do any processing) on recognizing that an event is not part of its event-set. The notion of projection captures this view. On the other hand, a component needs to be responsive in some way to its environment, and we capture this dependence in the form of guards on the transitions of its finite-state model. Let $M_1 = M(E_1)$ and $M_2 = M(E_2)$ be the projections of $M(E)$ for two disjoint event-subsets E_1 and E_2 of E. We add guards to the transitions in M_1 in terms of the states of M_2 as follows. For a transition $t(\sigma_i^{(1)}, e_1, \sigma_j^{(1)})$ in M_1, find the states $S_{12}(\sigma_i^{(1)}, e^{(1)}) = \{\sigma_k^{(2)} \in \Sigma(M_2)$: $S(\sigma_k^{(2)})$ intersects both the sets $S(\sigma_i^{(1)})$ and $S(\sigma_j^{(1)})\}$. Then, the guard on $t(\sigma_i^{(1)}, e_1, \sigma_j^{(1)})$ is the or-combination of the states $\sigma_k^{(2)} \in S_{12}(\sigma_i^{(1)}, e^{(1)})$. Similarly, we add guards to the transitions in M_2 in terms of the states of M_1, and we say that M_1 and M_2 are *mutually conditioned*.

Example 5. In Fig. 2(ii), all the guards in $M(D)$ and $M(L)$ are obtained as described above. In particular, the guard $[L_0]$ also applies to the events closeFrOut and closeFrIn at state D_1 in $M(D)$, but we do not show them to keep the diagram uncluttered; likewise, the guard $[D_0]$ applies to unlockFrOut-event in $M(L)$ and the guard $[D_2]$ to unlockFrIn-event, but we do not show for the same reason. Note that when $M(D)$ is in state D_1, $M(L)$ can only be in state L_0. For example, immediately after $M(D)$ arrives at D_1, $M(L)$ is in state L_0 as indicated by the guards for openFrOut and openFrIn. But the guards for the transitions at the state L_0 show that $M(L)$ cannot change its state while $M(D)$ is in state D_1. Thus, the only possible change of state at this point is in $M(D)$ and from D_1 to D_0 or D_2, at which point $M(L)$ may change its state from L_0. In Fig. 3(ii), the guards in $M(DL^{out})$ and $M(DL^{in})$ are once again obtained by the method described above and as before we do not show some of the guards to keep the diagram uncluttered. (We remark that the presence of the guards that are not shown in Fig. 2(ii) do not affect the cartesian products $M(D) \times M(L)$ defined below, which is another reason for not showing them. The same remark holds for the guards not shown in Fig. 3(ii) and the cartesian product $M(DL^{out}) \times M(DL^{in})$.) □

We can easily generalize the method for assigning guards given above to the case when the transitions in M themselves have guards. Consider a transition $t(\sigma_i^{(1)}, e_1, \sigma_j^{(1)})$ in $M(E_1)$. This means there are states $\sigma_i' \in S(\sigma_i^{(1)})$ and $\sigma_j' \in S(\sigma_j^{(1)})$ in M and one or more transitions $t(\sigma_i', [g_{ij}]e_1, \sigma_j')$ in M for the same event e_1 connecting these states. We also have a unique state $\sigma_k^{(2)}$ in $M(E_2)$ such that $\sigma_k^{(2)} \in S_{12}(\sigma_i^{(1)}, e^{(1)})$. This is because e_1 is replaced by the empty-event in obtaining M_2; $\sigma_k^{(2)}$ is the unique state such that $S(\sigma_k^{(2)}) \supseteq \{\sigma_i, \sigma_j\}$. The guard for the transition $t(\sigma_i^{(1)}, e_1, \sigma_j^{(1)})$ in $M(E_1)$ now consists of the or-combination of all possible terms of the form $\sigma_k^{(2)} \wedge g_{ij}$, taken over all possible g_{ij} arising from all possible state-pairs (σ_i', σ_j'). If M has no guards, i.e., each $g_{ij} = T$, then $\sigma_k^{(2)} \wedge g_{ij} = \sigma_k^{(2)}$ and this gives the guards described earlier.

It is not hard to see that if $E'' \subset E' \subset E$, then the projection $M'(E'')$ of $M' = M(E')$ gives the projection $M(E'')$ of M, with all its guards correctly determined with respect to $M(E - E'')$. In particular, each and-combination term in the guard for a transition in $M'(E'')$ may look like $\sigma_j'' \wedge \sigma_k' \wedge g$, where σ_j'' is a state of the projection $M'(E' - E'')$ of M', σ_k' is a state of the projection $M(E - E')$ of M, and g is the guard of a transitions in M that contributed to the transition in $M(E'')$. However, in that case the set of states $S(\sigma_j'') \subset \Sigma(M')$ must contain σ_k' and hence we can simplify the guard to simply $[\sigma_k' \wedge g]$, a proper form when we consider $M'(E'')$ as the projection $M(E'')$ of M.

Def. 2. If the finite-state models $M_1 = M(E_1)$ and $M_2 = M(E_2)$ are mutually conditioned, then we say a state $\sigma_i^{(1)}$ in M_1 and a state $\sigma_j^{(2)}$ in M_2 are *compatible* for an event e if one of the following conditions hold:

(1) $e \in E_1$ and $\sigma_j^{(2)} \in S_{12}(\sigma_i^{(1)}, e)$. This means $\sigma_j^{(2)}$ permits the transition at $\sigma_i^{(1)}$ in M_1 for the event e.

(2) $e \in E_2$ and $\sigma_i^{(1)} \in S_{21}(\sigma_j^{(2)}, e)$. This means $\sigma_i^{(1)}$ permits the transition at $\sigma_j^{(2)}$ in M_2 for the event e. □

3.3. Cartesian product and orthogonal decomposition

Let $M_1 = M(E_1)$ and $M_2 = M(E_2)$ be a pair of projections of M for disjoint event-sets E_1 and E_2. The guards in M_1 and M_2 make them mutually conditioned.

Def. 3. Each state of the cartesian product $M_1 \times M_2$ consists of a pair of states of the form $\sigma_{ij} = (\sigma_i^{(1)}, \sigma_j^{(2)})$, where $\sigma_i^{(1)} \in \Sigma(M_1)$ and $\sigma_j^{(2)} \in \Sigma(M_2)$, and the events $E(M_1 \times M_2) = E(M_1) \cup E(M_2)$. If $t(\sigma_i^{(1)}, e_1, \sigma_k^{(1)})$ is a transition in M_1 and the state $\sigma_i^{(1)}$ of M_1 is compatible with the state $\sigma_j^{(2)}$ of M_2 for the event e_1, then we have the transition from σ_{ik} to σ_{jk} in $M_1 \times M_2$ for the event e_1. Likewise, if $t(\sigma_j^{(2)}, e_2, \sigma_k^{(2)})$ is a transition in M_2 and the state $\sigma_i^{(1)}$ in M_1 is comparable with the state $\sigma_j^{(2)}$ in M_2 for the event e_2, then we have the transition from σ_{ij} to σ_{ik} in $M_1 \times M_2$ for the event e_2. The start-state of $M_1 \times M_2$ is the combination of the start-states of M_1 and M_2. □

Remark 3.1. In general, not all states $\sigma_{ij} \in \Sigma(M_1) \times \Sigma(M_2)$ in $M_1 \times M_2$ may be reachable from the start-state and also there may be some equivalent states. This means we may have to minimize $M_1 \times M_2$ as defined above by merging equivalent states after we remove the states that are not reachable from the start-state in order to obtain the final finite-state model for the cartesian product; henceforth, $M_1 \times M_2$ shall denote this minimized form.

Remark 4.2. The guard for a transition t in $M_1 \times M_2$ is the simplified form of the guard in the transition t_1 in M_1 (or the transition t_2 in M_2) that give rise to t and is obtained by the elimination of state symbol from M_2 (resp., M_1) in the guard in transition t_1 (resp., t_2).

Fig. 7 shows the cartesian product construction for $M(D)$ and $M(L)$ in Fig. 2(ii), where the shaded states are not reachable from the start-state D_0L_0; the transitions that are eliminated because of the guards are shown here as dashed lines and we do not show the event-labels for these transitions to keep the diagram uncluttered. There are no equivalent states here. This shows that the model $M(DL)$ in Fig. 1(iii), or equivalently, the model M_{DL} in Fig. 6 equals $M(D) \times M(L)$. The presence of guards usually causes fewer states and transitions in the product than otherwise. For example, if we disregard the guards in $M(D)$ and $M(L)$ in Fig. 2(ii), then their cartesian product will have all 3×3 = 9 states and 24 transitions in Fig. 7.

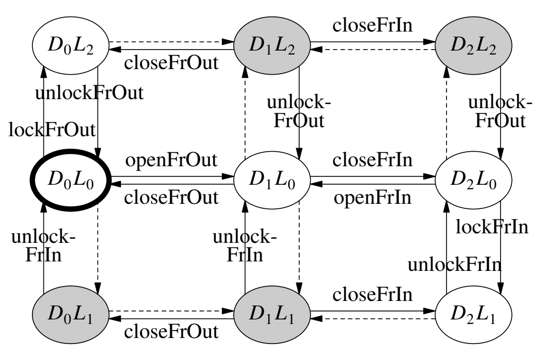


Figure 7. The cartesian product $M(D) \times M(L)$; the shaded states are unreachable from the start-state D_0L_0.

Def. 4. We say a decomposition of the events E of a finite-state model M into two disjoint subsets $E = E_1 \cup E_2$ is *orthogonal* if $M = M(E_1) \times M(E_2)$, i.e., we can recover M from the projections $M(E_1)$ and $M(E_2)$. We say each of $M(E_1)$ and $M(E_2)$ is an orthogonal projection of M. □

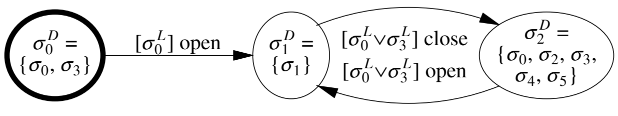
We emphasize that the orthogonality of two complementary projections $M(E_1)$ and $M(E_2)$ is in no way dependent on the start-state. It depends solely on the relative role of transitions for events in E_1 with respect to those for events in E_2. The orthogonality of a projection $M(E_1)$ implies the orthogonality of the complementary projection $M(E_2)$. If we make an orthogonal decomposition of $M_2 = M(E_2)$ into two or more components, then each state of M_2 can be regarded as an and-combination of the states of its components and therefore each occurrence of a state $\sigma_k^{(2)}$ of M_2 in a guard of a transition in M_1 can now be replaced by the and-combination of corresponding states of the components of M_2.

Example 6. Each of the pairs of projections in Fig. 2(ii) and in Fig. 3(ii) gives an orthogonal decomposition of $M(DL)$ shown in Fig. 1(iii) and also in Fig. 6. Likewise, the pair of projections in Fig. 4(ii) give an orthogonal decomposition of the slight variation of $M(DL)$ shown in Fig. 4(iii). On the other hand, $M(OL)$ and $M(IL)$ in Fig. 1(ii) form an orthogonal decomposition of $M(OIL)$ in Fig. 2(ii). Thus, the three finite state models in Fig. 1(ii) form an orthogonal decomposition of $M(DL)$. □

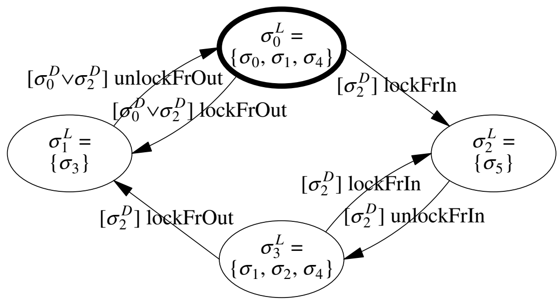
It is clear that if a partition $E = E_1 \cup E_2 \cup \cdots \cup E_n$ into disjoint subsets gives an orthogonal decomposition of M, i.e., $M = M(E_1) \times M(E_2) \times \cdots \times M(E_n)$ and we group $M(E_i)$'s in disjoint subsets, then the family of cartesian products, one for each group, together form an orthogonal decomposition of M.

Example 7. Let M_0 denote the finite-state model in Fig. 5. Figs. 8(i)-(ii) show the projection M_0^D of M_0 for the door operations E_D = {open, close} and the projection M_0^L for the two-sided lock operations E_L = {lockFrOut, unlockFrOut, lockFrIn, unlockFrIn}. In each state in M_0^D and M_0^L, we show the associated subset of $\Sigma(M_0)$. It is easy to see that $\mathrm{M}_0^D \times M_0^L \neq M_0$ and thus M_0^D and M_0^L do not form an orthogonal decomposition of M_0. Here, the problem arises due to the transitions to and from σ_3^L for the events unlockFrIn and lockFrOut, respectively, each having the guard $[\sigma_2^D]$, where σ_2^D = door closed (from inside or outside). This allows the state (σ_2^D, σ_3^L) in $M_0^D \times M_0^L$, which can be reached from the start-state (σ_0^D, σ_0^L), to have an incoming transition for unlockFrIn-event and an outgoing transition for lockFrOut-event. However, lockFrOut-event cannot immediately follow unlockFrIn-event in M_0. Likewise, the state (σ_2^D, σ_0^L) in $M_0^D \times M_0^L$ has an incoming transition for unlockFrOut-event and an outgoing transition for lockFrIn-event. □

One may wonder that since M_0^L cannot be in state σ_3^L while M_0^D is in state σ_0^D, we can replace the guard $[\sigma_0^L]$ in M_0^D for the transition from σ_0^D for open-event by $[\sigma_0^L \vee \sigma_3^L]$, which would allow us to minimize M_0^D by merging the states σ_2^D and σ_0^D, and maybe this will help us to achieve the equality $M_0 = M_0^D \times M_0^L$. However, we actually should not do this because there are many transitions in M_0^L with the guard $[\sigma_2^D]$ and there are other transitions with the guard $[\sigma_0^D \vee \sigma_2^D]$, and the merging of σ_2^D and σ_0^D would create problems for M_0^L. In particular, this would mean at the initial state of $M_0^D \times M_0^L$, the event lockFrIn would be legal but that is not the case in M_0. On the other hand, we cannot not merge the states σ_0^L and σ_3^L to minimize M_0^L further because the transitions for lockFrOut-event has different guards from these states. (If we were to merge the states σ_0^D and σ_2^D in M_0^D, then we could merge the states σ_0^L and σ_3^L in M_0^L as well, but this still would not give $M_0 = M_0^D \times M_0^L$.)



(i) The projection M_0^D of M_0 for the door operations {open, close}.



(ii) The projection M_0^L of M_0 for the two-sided lock operations {lockFrOut, unlockFrOut, lockFrIn, unlockFrIn}.

Figure 8. Projections of the finite-state model in Fig. 5 on two complementary subsets of events.

One can show, in fact, that there is no partition of the event-set of M_0 such that the corresponding projections of M_0 form an orthogonal decomposition of M_0.

3.4. Sufficient conditions for orthogonal decomposition

The first theorem below shows that if a finite-state model M has a certain easily recognizable structure, then there is an orthogonal decomposition of M. The orthogonal decompositions in Figs. 1-4 can be obtained using this theorem. The second theorem gives a necessary and sufficient condition based only the mapping S from the states of the projections of M to the subsets of states of M. It does not require us to compute the guards in the projections or the cartesian product of the projections. We omit the proofs of following two theorems for want of space.

Theorem 1. Suppose we can write the states Σ of a finite-state model M as $\Sigma = \Sigma_1 \cup \Sigma_2$ with the properties:

(i) The intersection $\Sigma_1 \cap \Sigma_2$ consists of a single state σ_0.

(ii) Each transition of M is between two states in Σ_1 or between two states in Σ_2, i.e., the only way to pass from a state in Σ_1 to a state in Σ_2 is via the state σ_0.

(iii) If E_i is the set of events corresponding to the transitions among the states Σ_i, then $E_1 \cap E_2 = \varnothing$. (In particular, there is no transition in M from σ_0 to itself.)

Then, the projections $M_1 = M(E_1)$ and $M_2 = M(E_2)$ give an orthogonal decomposition of M. □

Theorem 2. The projections $M_1 = M(E_1)$ and $M_2 = M(E_2)$ of a finite-state model M for a disjoint partition of its event-set $E = E_1 \cup E_2$ give an orthogonal decomposition of M if and only if for each pair of states $\sigma_i^{(1)}$ in M_1 and $\sigma_j^{(2)}$ in M_2 the intersection $S(\sigma_i^{(1)}) \cap S(\sigma_j^{(2)})$ of the corresponding subsets of states in M has size ≤ 1. □

Example 8. If we form the cartesian product $M_0^D \times M_0^L$ of the projections shown in Fig. 8, then it will be seen that the only states in $M_0^D \times M_0^L$ responsible for the inequality $M_0 \neq M_0^D \times M_0^L$ are (σ_2^D, σ_0^L) and (σ_0^D, σ_3^L). Also, we see from Fig. 8 that $S(\sigma_2^D) \cap S(\sigma_0^L)$ and $S(\sigma_2^D) \cap S(\sigma_3^L)$ are the only intersections with size larger than 1, violating the condition in Theorem 2. The orthogonal decompositions in Figs. 1-4 satisfy the condition in Theorem 2. □

4. Conclusion

We have shown that the orthogonal decompositions of the finite-state behavior model of a software provide a sound basis for defining the components of an architecture for the software. In this approach, the emphasis lies on the operations (functional elements) of the software. The only information shared between the components are their "states" for control purpose and this is determined by the finite-state behavior model itself. This puts the architecture-design task at a higher level than the class-hierarchy or software design task, where the latter involves the use-relationship between the operations and the data-items.

5. Reference

1. R.J. Allen and D. Garlan, A formal basis for architectural connection, *ACM Trans. on Software Engineering and Methodology,* 6(1997), pp. 213-249.
2. D. Garlan, R. Monroe, and D. Wile, ACME: architectural description of component-based systems, *in Foundations of Component-Based Systems* (ed. G. Leavens,‘and M. Sitaraman), Cambridge University Press, 2000, pp. 47-68.
3. S. Kundu, Structuring software functional requirements for automated design and verification, *Proc. COMPSAC-07,* Beijing, July 24-27, 2007.
4. S. Kundu and G. Nigel, A formal approach to designing a class-subclass structure using a partial order on the functions, *Proc. COMPSAC-05,* Edinburgh, Jul 26-28, 2005, pp. 213-220.
5. H.R. Lewis and C.H. Papadimitriou, *Elements of the theory of computation,* Prentice-Hall, 1981.
6. D.C. Luckham and L.M. Augustin, D. Bryan, J.J. Kenney, W. Mann, and J. Vera, Specification and analysis of system architecture using Rapide, *IEEE Trans. on Software Engineering,* 21(1995), pp. 336-355.
7. D.C. Luckham and J. Vera, An event-based architecture definition language, *IEEE Trans. on Software Engineering,* 21(1995), pp. 717-734.
8. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Lang. ref. manual*, Addison-Wesley, 1999.