Proceedings of ICAD2000
First International Conference on Axiomatic Design
Cambridge, MA – June 21-23, 2000
ICAD006

HIERARCHICAL STATE DECOMPOSITION FOR THE DESIGN OF PLC SOFTWARE BY APPLYING AXIOMATIC DESIGN

Matthias Schreyer
iems@ust.hk

Mitchell M. Tseng
tseng@ust.hk

Department of Industrial Engineering and Engineering Management
The Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong

ABSTRACT

Axiomatic design is applied with state transition representation to software design for Programmable Logic Controllers (PLC). By comparing several approaches of applying the principles of axiomatic design in order to generate an equivalent description in state transition representation, we conclude that hierarchical statecharts are an important description of the decomposition and zigzagging procedure at conceptual design stage. This notation helps to identify the goal states and other important design parameters, such as time delays, input signals from sensors, and shared memory data. Since statecharts may provide inherent uncoupled designs we introduce a modified design table for state transition description. The state transition table combines the state transition model with the matrix notation and maps the input conditions to output actions for each state. The table can then easily be employed for implementing the PLC program code. This is particularly important in large-scale PLC controlled systems where a substantial number of control engineers have to collaborate by exchanging the input/output specification among the partial systems. An example is prepared to illustrate the proposed design method.

Keywords: axiomatic design, programmable logic controller, statecharts

1 INTRODUCTION

1.1 RELATED RESEARCH

Automation systems like manufacturing systems or transportation systems require concurrent design of control hardware as well as control software. Suh has proposed [1999] a modular system architecture that is derived from the transformation of functional requirements into the design parameter and may indicate the operational sequence of the hardware/software modules in the system. The system architecture reflects the hierarchical decision making process and denotes the way of how to satisfy a selected set of functional requirements. The operational sequence of system modules is assumed to be static which might be valid in an ideal world. Automation systems, however, exhibit reactive and dynamic behavior where the operational sequence cannot be completely predicted. For instance, a general characteristic of flexible automation systems is sharing of resources, which means that components are mutually dependent. System components have to communicate by sending request and acknowledge signals in order to provide a correct behavior. This negotiation process can cause variations of the operational sequence depending on particular system conditions.

Hintersteiner and Tate [1998] have defined a system design template denoted as a generic matrix for designing manufacturing processes and control. In accordance with the functional requirements, the sequence of design decisions is following: (1) process functions, (2) transport functions, (3) supervisory control, (4) controller and computer hardware, and finally (5) the integrating framework. Decomposing a particular process or transport module into a lower level will also expose a similar design sequence: first the system hardware (actuator, transmission, sensor), secondly the control algorithm, and the implementation of the controller hardware at last. Similarly, Hintersteiner and Nain [1999] have further elaborated the system architecture templates in order to decompose the control functions into so-called command and control algorithm (CCA).

1.2 DESIGN OF PLC SOFTWARE

The design of PLC programs is a complex and error-prone task. Until today, relay ladder logic (RLL) is predominantly employed to program PLCs and to implement the desired behavior in automation systems though a couple of interchangeable programming languages have been accepted in the international standard [IEC 61131-3, 1993]. RLL is advantageous because it is a graphical notation derived from electric circuit diagrams and is, therefore, easy to understand for engineers and electricians. However, RLL is well known for its fundamental misconception with respect to the structure of data and programs in modern programming languages. Consequently, large-scale PLC programs become tedious to trace, modify and debug.

PLC programs are inherently coupled regarding sharing of data and exclusive system states. The latter aspect indicates that certain system states interfere and interlock with other states and prevent them from execution. Since the procedural nature of an

RLL program can only reflect a small fraction of states combinations and their interference, an RLL program is initially often inconsistent and incorrect and requires a significant effort to achieve a certain level of robustness for system installation and operating.

As mentioned above coupling cannot be avoided completely in software systems since it is necessary to communicate data. Therefore, our research investigates design methods from the field of computer science and engineering design theory which are potentially suitable for a systematic approach. The systematic approach should support a sequential and logical decision making process, by reducing the manifold interrelations of a complex design to an ordered, traceable and repeatable design procedure. Being aware of the systematic and formal nature of axiomatic design [Suh, 1999], we have discovered that state transition notations are suitable tools since they offer similarly an immediate cognitive mapping of functional requirements to design parameters. In addition, the hierarchical and modular character of statecharts [Harel, 1987] supports a structured top-down design process.

To this end, we propose an extension of the axiomatic design with state transition representations. Mapping this representation into a state transition design table eases the refining of the input/output pattern of each state and identifies potential interferences between states and process modules.

2 PLC CONTROLLED AUTOMATION SYSTEMS

PLC controlled automation systems belong to the category of real-time, reactive and discrete event dynamic systems since they have to respond to external or internal stimuli within a finite and specified time interval [Kopetz, 1997]. Such dynamic systems are regarded as concatenations of events that occur at discrete instants of time and cause a change (transition) of the system state. The PLC and the controlled automation systems are traditionally considered as two separated systems that interact via sensors and actuators, indicating two unidirectional information-flows. The PLC has to be designed and programmed in such a way that the controlled automation system performs the intended behavior.

A PLC (Figure 1) is a simple computing device that consists of a programmable memory for internal storage for instructions, input and output channels and a scanning logic to read the inputs and write the outputs periodically. The input channels connect the incoming signals from sensors and the output channels link the outgoing signals to the actuators or other peripheral devices. The PLC may exchange data with parallel control nodes over a network. Commands can be retrieved from supervisory control instances and status information could be sent back to a higher-level controller. The PLC operates according to cycles of operations consisting of three phases: (a) polling the input channels, (b) computing the outputs according to the program instructions, and (c) updating the outputs in the output channels. The scan time of each cycle is normally fixed.

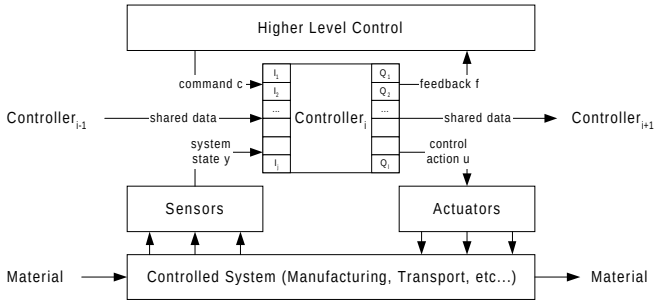


Figure 1. Controller model.

3 APPLYING AXIOMATIC DESIGN TO DESIGNING PLC SOFTWARE

3.1 BASIC CONCEPTS OF AXIOMATIC DESIGN

Axiomatic design theory includes the following basic concepts [Suh, 1999]:

- Domain framework comprising four domains, the customer requirements (CR), functional requirements (FR), design parameters (DP), and process variables (PV);
- Hierarchical decomposition of each domain;
- Mapping through a *zigzagging* procedure between the domains creating structural congruent decomposition trees in each domain and supporting a concurrent engineering method;
- Design axioms postulating (1) to maintain the independence of the functional requirements and (2) minimizing the information content in order to achieve a good design solution.

The fundamental idea of the domain concept is to let the designer distinguish between *what* is going to be achieved in the left-hand side domain and *how* the "whats" are going to be achieved in the right-hand side domain. This is performed top-down by zigzagging between the domains.

In principal, axiomatic design could be applied to any design problem, in mechanical design, system design, software design, and even in the design of business plans or organizations. In a specific technical design context the domains may have different interpretations.

In PLC software design, we propose the following interpretation: the CR domain represents the user requirements and attributes. The FR domain denotes the desired functionality or activities as a composite of elementary control actions. The DP domain expresses the behavior of the system or subsystems in terms of states. The PV domain takes the particular implementation by assigning concrete input/output values to the state variables into account (Table 1).

Functional Requirements	Design Parameters	Process Variables
Function, Activity	State	State string
FR.x: Provide resource FR.y: Perform action FR.z:...	DP.x: Available State DP.y: Action State DP.z:...	PV.x = [11XX 10XX] PV.y = [111X 11XX] PV.z = ...

Table 1. FRs, DPs, and PVs in PLC software design.

3.2 STATE TRANSITION MODELING

In computer science, state transition models such as finite state machines are commonly used to describe the behavior of reactive and real-time systems such as PLC controlled transport and manufacturing systems. Thereby, the system dynamics is segregated into a number of interrelated states, whereby each state captures a snapshot of the system's behavior. State transition models naturally map the event-driven characteristics of reactive systems, in contrast to the transformational and data-flow modeling methods. State machines are powerful because they provide intuitive graphical representation and they are conducive to build simulation models. Thus, state machines can be analyzed without the necessity of building the real system. Moreover, modeling with state machines can be regarded as a structured design method of PLC programs. A detailed specification of the model can be translated into the PLC control program in relay ladder logic or any other PLC programming language.

Designing with state transition notations means to decompose the system's behavior into a set of states which are bounded by transitions, reflecting the state change (Figure 2). A state corresponds to a *meaningful* system condition, *i.e.* a desired and functional condition. A system state change is caused by internal or external events. External events correspond to state changes of the system to be controlled and recognized by sensor devices, triggering a signal to the controller. State changes of the controller program are internal events and communicated by shared variables. Each event that causes a state change may be accompanied by a certain control action to be performed. In Harels [1987] semantics of *statecharts*, the event/action relationship labels the state transition and occurs at a discrete instant of time (without consuming time), which agrees with the synchrony hypothesis of Berry and Cosserat [1985]. The event/action label reflects the common way to describe the reactive behavior:

Event e.x:If (whenever) an observed input pattern, then Action a.x: perform a set of control actions (output).

A state is, therefore, made of all incoming and outgoing signals that persist over a significant amount of time. The concatenation of input and output signals is regarded as a state pattern or string. When the state pattern changes, a new state is entered. Therefore, we consider the state pattern as a final representation of a state construct in the implementation domain (PV domain). It should be noted that this type of elementary state construct so far does not indicate the sequential order. This might confuse beginners employing the graphical notation of statecharts since a labeled transition carries two elements of information, the event/action pattern and the sequential order (arrow) of states. We shall decouple these elements by strictly assigning the event/action pattern to the state. By doing this, we modify the Mealy-like statecharts into Moore-like statecharts because the output depends only on the current state, and not anymore on both the current state and the transition. Accordingly, we will follow the restrictions of Douglass [1999], who required that states should not overlap and should be disjoint in terms of the events they accept, the activities and control action they perform, and the follow-up transition they take in response to external and internal stimuli. Furthermore, we declare that a system, a part of the system, or a particular object resides in a non-empty set of states at any time. When no action occurs the system is located in the default or initiating state s_0.

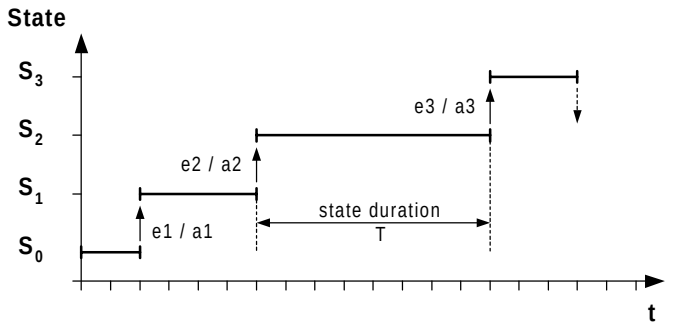


Figure 2. State transitions in a timing diagram.

3.3 DETERMINING THE FRS AND DPS THROUGH HIERARCHICAL STATE DECOMPOSITION

Our focus in this paper is to present the hierarchical state decomposition by applying the basic concepts of axiomatic design. While decomposing the states the designer always takes the desired functions into account. This procedure will graphically be accompanied by the statecharts notation, which augments the flat state machines with the capabilities of nesting states (depth), modular structure (orthogonality), and a communication mechanism (broadcasting). Including the notation of hierarchical states facilitates a top-down design, whereby each functional requirement (FR) can be directly specified by a design parameter (DP) on the equivalent hierarchical level.

When following the mapping procedure from the FR domain to the DP domain, we have to specify what activity (or control action) is to be performed and ask *how* to achieve this action in a particular state or substate (*S.x*). We therefore cast the FRs, the desired activities and control actions, into states and substates.[1]

[1] In the state diagrams throughout this paper, initial (●) and terminal (⊙) pseudo-states are used to denote the starting and final state in a state sequence.

The decomposition follows the zigzagging procedure between the two domains. The hierarchical state set can be visualized as a tree structure, where the leaf-states correspond to the conventional notion of flat state machines (Figure 3).

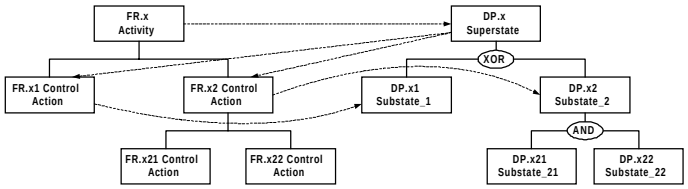


Figure 3. Hierarchical state decomposition.

Setting up the design matrices demonstrates the dependency of the substates according to the type of state decomposition. There are two kinds of state decomposition, the XOR and the AND state decomposition.

3.3.1 XOR State Decomposition

The XOR state decomposition declares that the state has to be in exactly one of the decomposed substates. The XOR decomposition type indicates a decoupled or uncoupled design matrix, according to the sequential order, if any, of the state transitions.

$$\begin{Bmatrix} \text{FR.x1} \\ \text{FR.x2} \end{Bmatrix} = \begin{bmatrix} \text{X} & 0 \\ 0 & \text{X} \end{bmatrix} \begin{Bmatrix} \text{DP.x1} \\ \text{DP.x2} \end{Bmatrix} \quad (1)$$

In case the states take place in a particular sequence, the state design is decoupled. Otherwise, if there is no predetermined sequence, in the sense that the occurrence of a state is independent of a previous state, the design is uncoupled. For example, in Figure 4 the uncoupled case is depicted by state transitions from state *S.x1* to substate *S.x2* and vice versa.

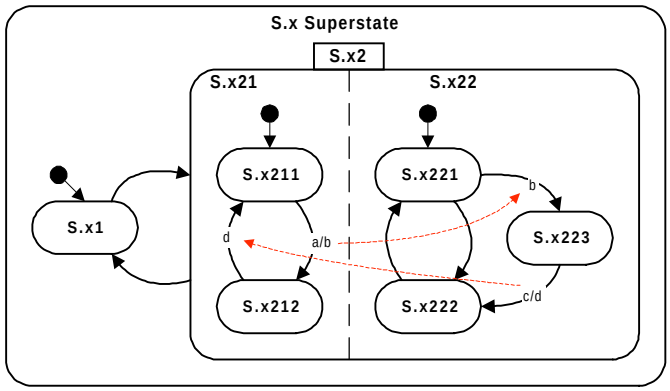


Figure 4. XOR / AND state decompositions.

3.3.2 AND State Decomposition

The other type describes the AND decomposition of states, *i.e.* an orthogonal state set, declaring that the system or module has to be in all substates of the enclosing super-state. This design may exhibit an uncoupled or coupled design, depending on whether or not the AND states are communicating. At this point, it needs to be clarified that orthogonality has two meanings, independence and concurrency. The states, for instance *S.x21* and *S.x22* in Figure 3 and 4, are executed simultaneously and, to a certain extent, independently. However, states in AND decomposition may communicate through the labeled transitions (event/action statement). The action-statement can cause a state transition in a parallel state. Thus, in case there is no communication at all, the orthogonal state set is independent and, therefore, uncoupled. If AND states are communicating in a unidirectional or bi-directional way, then states are decoupled or coupled, respectively.

$$\begin{Bmatrix} \text{FR.x21} \\ \text{FR.x22} \end{Bmatrix} = \begin{bmatrix} \text{X} & 0/\text{X} \\ 0/\text{X} & \text{X} \end{bmatrix} \begin{Bmatrix} \text{DP.x21} \\ \text{DP.x22} \end{Bmatrix} \quad (2)$$

Originally, orthogonal state regions were introduced into the statechart notation in order to avoid the combinatorial explosion of states in flat state transition models. For instance, if each of three state variables can have three values, the entire state set could consist of 27 states. The usage of orthogonal state regions may visually reduce the number of states to 9. However, by doing this, the designer should not get distracted from the fact that states could still remain coupled. Orthogonality is a feature that helps to impose structure on large and complex state transition models but cannot reduce the coupling *per se*.

3.3.3 Overlapping of States

In fact, there is a third type of state decomposition which is referred as OR decomposition. The OR decomposition indicates an overlapping of states, meaning that a substate resides in more than one superstates. Suppose state *S.x223* in Figure 4 is a substate of both superstates *S.x21* and *S.x22* as depicted in Figure 5. This might be a reasonable design decision because this state is cooperatively used.

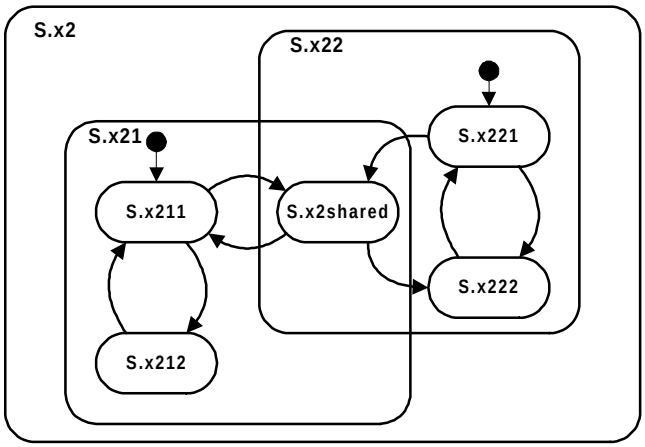


Figure 5. States overlapping.

Harel [1987] argued that there is no deep reason why designer should avoid overlapping of states because it can also reduce the number of states, in a similar way shared methods/procedures are used in modern programming languages. However, we argue that this type should be avoided

since it points to a coupled design; and, indeed, it deteriorates the readability and increases the complexity of the state diagram when used excessively. Once a shared state of two superstates is going to be modified, the change will impact both superstates; therefore:

$$\begin{Bmatrix} FR.x21 \\ FR.x22 \end{Bmatrix} = \begin{bmatrix} X & X \\ X & X \end{bmatrix} \begin{Bmatrix} DP.x21 \\ DP.x22 \end{Bmatrix} \quad (3)$$

In concluding this section, we state that obviously an uncoupled state design should be preferred which means independence of state sequences and non-overlapping of states. Transitions caused by internal or external stimuli should be independent of preceding states. In many cases, some states require a certain history of states; therefore, a sequential execution of states is a common type and cannot be circumvented. In an XOR decomposition we may always find a particular state sequence which indicates a decoupled design. The designer should be careful in using the AND decomposition as it could point to an uncoupled design, if the states are totally independent. Since communication is a necessary element of a controller program, the AND state decomposition could also conceal a coupled design.

3.4 STATE DIAGRAM AND RELAY LADDER LOGIC

Relay ladder logic (RLL) is predominantly employed in industrial practice to program PLCs. Since the linkage between the semantics of statecharts and relay ladder logic is not as obvious, this paragraph will provide a deeper understanding regarding the correspondence of the two notations.

RLL is a graphical program notation consisting of a set of rungs, vertically arranged and similar in the appearance to a ladder. Each rung represents a logical input/output condition, equivalent to the event/action pair of the labeled transitions in the statecharts notation. In RLL, states are not an explicit construct but each rung with his input/output pattern could be considered as a state. The set of rungs in a ladder diagram is thus a flat state machine. In order to express nested or hierarchical states dedicated Boolean state variables are introduced. However, the programming environment of an RLL editor is not performing a semantic check in terms of states overlapping. A programmer can unintentionally produce coupled designs that are difficult to trace, and modify. A small generic example will illustrate this problem.

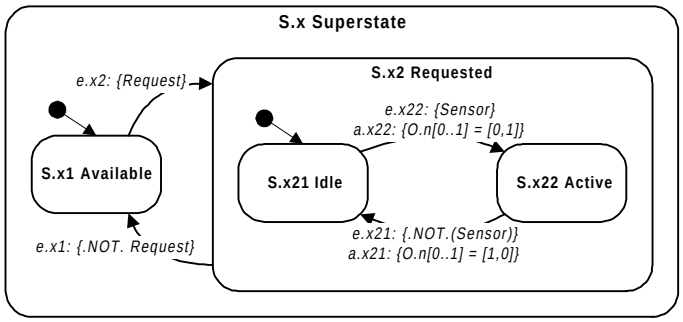


Figure 6. Generic statechart example.

As mentioned in the introduction of this paper, a characteristic of flexible manufacturing systems is the sharing of resources. The usage of shared resources such as the transport system has to be negotiated among requesting participants. The provided services, *i.e.* the control actions, of the shared resource could be encapsulated within a pair of superstates, denoting the availability of the resource at each instant of time. In Figure 6 this type of state pair is called *S.x1 = Available* and *S.x2 = Requested*. Once a shared resource is in the requested state, it cannot serve other requests. Other requests have to wait until the resource becomes available again. The enclosure of states with a superstate in this example is very useful and could be regarded as a kind of protection. It encapsulates a critical uninterruptible state sequences that can only be executed upon the validity of a synchronization signal, which is also comparable to the semaphore construct of Dijkstra [1968].

In the given example the *idle* and *active* state, which performs the actual control actions, are the encapsulated states *S.x21* and *S.x22*. An external sensor signal initiates the *active* state, activating a certain output command. For instance, a green light (O.n[1] = 1) indicates the *active* state, and a yellow light (O.n[0] = 1) the opposite state *idle*.

Figure 7 shows the corresponding implementation of relay ladder logic by using the standard elements. Rung 1 and rung 2 express the states *S.x1* and *S.x2*, respectively. A Boolean state variable B.n[0] is used to represent this state pair, because sequence and hierarchy is not an implicit feature of RLL programs. If a resource request signals occurs bit B.n[0] is latched. After task completion, the bit is unlatched, whereby the completion event (.NOT. Request) is triggered from the final state inside the *active* state *S.x22*. Rung 3 and rung 4 represent the states *idle* and *active* with their accompanied output actions O.n[1] and O.n[0], respectively.

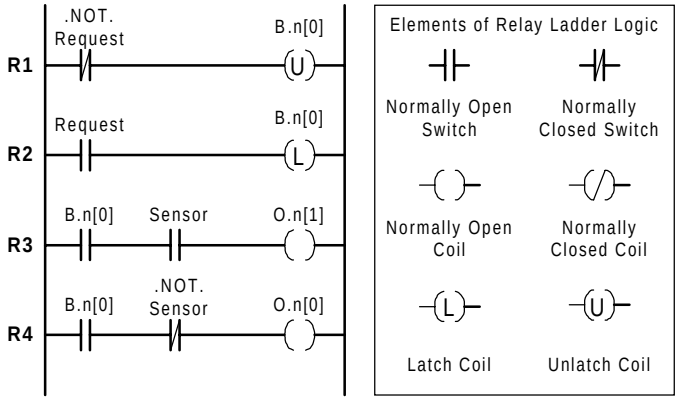


Figure 7. Ladder logic implementation.

As mentioned above, we disallow overlapping of states, which provides a clearer and more rigorous representation of the system states. In ladder logic states overlapping is possible if the designer is careless and has no clear state transition model in mind. According to our knowledge, this is one major reason why RLL programs are error-prone and difficult to trace. For instance, in rung 4 of the given example, the programmer might disregard to check the first state bit (B.n[0]) when this rung occurs at a different position in a much more larger program.

Note that a common industrial application has hundreds or even thousands of rungs, and there is often more than one state bit that has to be evaluated in a rung statement.

In case the state bit is missed, the semantics of the underlying state model becomes different from the intended design. The states *available* and *init* would become overlapped. In case there is no request and no sensor signal, the system would reside in both states at the same time. In accordance with our argumentation in the previous chapter we may say that the design is coupled and there is no semantic check in an RLL program editor that could prevent the designer from doing this. Figure 8 shows the corresponding statechart notation. Note that the encapsulating superstate *requested* becomes obsolete and looses its shielding function. We may also argue that this function is not anymore adequately designed and therefore coupled.

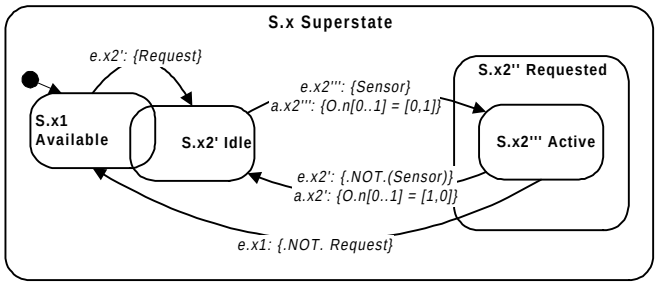


Figure 8. Statechart for coupled design.

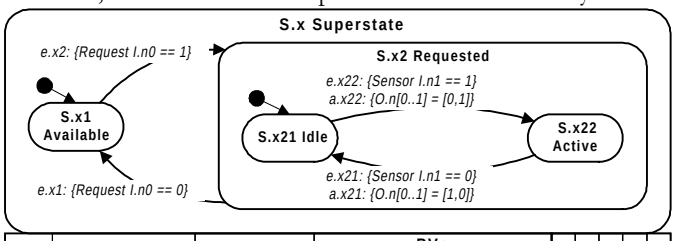
Certainly, this state diagram is a reconstructed example and a cautious designer would not select this design solution. Inversely, we can however conclude that using the statechart in a correct way, *i.e.* applying the AND/XOR state decomposition type, the designer will be prevented from creating this kind of coupling.

3.5 STATE-TRANSITION DESIGN TABLE

Following the hierarchical state decomposition with statecharts apparently results in a de- or uncoupled design. However, the achieved design matrices are uncommon for control engineers. Therefore, in the next step the statechart notation is mapped into state-transition design table. This table combines the functional requirements, design parameters (states), and the process variables as the state pattern at a glance. The table comprises the following information:

- Functionality, provided service/activity (FRs),
- State description (DPs),
- Input-patterns of each state,
- Output-pattern associated with each state amplifying actuators,
- Temporal boundaries, if any, of each state,
- Allowed state transitions to other states.

For each state of the statechart notation a row with the desired control actions (outgoing signal pattern) and the event pattern of incoming signals and data are specified (Figure 9). Thereby, a unique set of incoming and outgoing signals of bit-values defines a disjoint set of controller states. The number of theoretical possible states is an exponential function of the number of I/O bits (2^n), and is much more larger than the number of actual designed controller states. Therefore, I/O-patterns not explicitly defined in the state transition table are treated as error states. In addition, "don't care" states, denoted as $X = 0 \vee 1$, are introduced to represent the state hierarchy.



	FRs	DPs	PVs					x1	x2	x21	x22	
			I.n0	I.n1	...	O.n0	O.n1					
x1	FR.x1	S.x1 Available	0	X		X	X	1	1			
x2	FR.x2	S.x2 Requested	1	X		X	X	1	1			
x21	FR.x21	S.x21 Idle	1	0		1	0			1	1	
x22	FR.x22	S.x22 Active	1	1		0	1			1	1	
...	...	...										

Figure 9. State transition design table.

The table is segregated into four parts, on the left-hand side the FRs, followed by the DPs, the PVs, and the state transition matrix on the right-hand side. An '1' in the state transition matrix indicates what are the possible subsequent states of each state. Shaded areas illustrated prohibited transitions.

4 CASE STUDY

We will illustrate the developed method by the means of a program segment, which is a part of the PLC program in our FAS line in the HKUST manufacturing systems laboratory. The flexible assembly system consists of several workstations: a loading/unloading robot, several manual workstations for commissioning, assembling, and packaging. These stations are interconnected with a computer-controlled transfer system (CTS).

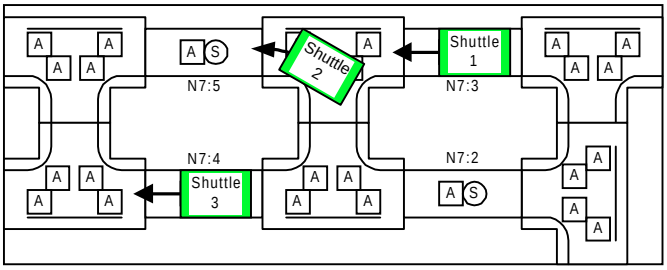


Figure 10. Transport system.

An essential part of the CTS is to control the transfer of a transport shuttle from one workstation to the next. A shuttle staying in one workstation has to wait until the next workstation is free (Figure 10).

Stopping and releasing of the shuttle is performed with a stopper unit at each workstation, consisting of a proximity sensor and a pneumatic actuator. In order to design this seemingly simple task we have to make sure that the workstation, and therefore the stopper unit, is requested only by one shuttle at a time. In addition, the stopper unit has to control the correct

transport to the next station because no further sensor is installed. A watchdog timer monitors the regular time of the transportation process. If the time exceeds the normal time, an alarm signal is triggered.

The high level functional requirement (FR.x) is to temporary stop the transport shuttle at the workstation location. The corresponding design parameter is a PLC controlled stopping state-module (S.x). The decomposition procedure has been carried out in many iterative design cycles and through zigzagging between FRs and DPs. The table in Figure 11 shows the result of the decomposition process of the FRs and DPs including the related statechart. An analysis of all state patterns (PVs) confirms that the state set is non-overlapping.

This case study also supports the reuse in an analogous design problem. For instance, a reusable structure is the state-pair *available* and *requested*. These two states wraps up the acting substates of shared resources. Likewise, the states *idle* and *active* encapsulate the control actions. The linear sequence of control actions within the substate *active* indicates a typical decoupled design with a starting state and a terminal state.

5 CONCLUSION

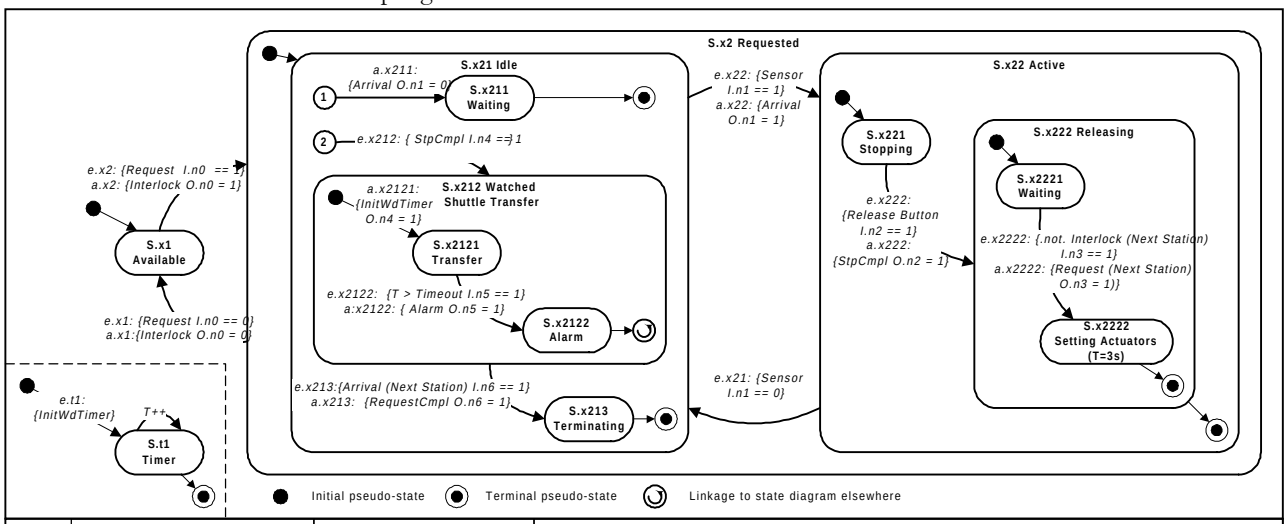
In this paper, we analyzed the applicability of axiomatic design to PLC software design. It has been shown that statechart notation supports the decomposition and zigzagging procedure of the FRs and DPs in an explicit way. Using the statechart notation leads inherently to an uncoupled or at least decoupled design solution. This is because statecharts disallow overlapping of states which is an indication of state-coupling.

An ideal design is the total uncoupling of states, meaning that the states are independent of the sequence of occurrence. Interestingly, the independence axiom might provide a scientific explanation why relay ladder logic is still the preferred language in programming automation controllers. The rung-structure of relay ladder enforces decoupling of the program into independent elements. However, this paper has also shown that without applying the hierarchical state decomposition procedure, programming in relay ladder logic is error-prone. Augmenting the power of relay ladder logic with the conceptual and decomposable nature of axiomatic design could be an important step to ease the programming, debugging, and testing, particularly in a complex and large system. Furthermore, applying a systematic approach is conducive to the development of a computer-aided design tool that could generate the program code automatically.

The outlined example also illustrates that the principles of axiomatic design contribute a clear repeatable state decomposition procedure to the computer science community. The general principle of designing "good" statecharts has not been reported until now.

6 ACKNOWLEDGMENTS

The authors would like to acknowledge Rockwell Foundation for financial support (HKUST grant RAHK 97/98) and their information and feedback. The authors would like to express their gratitude to Professor N. P. Suh, K. D. Lee, and the anonymous reviewers of this paper for their valuable comments.



	FRs	DPs	PVs																
			I.n0	I.n1	I.n2	I.n3	I.n4	I.n5	I.n6	O.n0	O.n1	O.n2	O.n3	O.n4	O.n5	O.n6	...		
x	Temporary stop shuttle at workstation	PLC controlled stopping module	X	X	X	X	X	X	X	X	X	X	X	X	X	X			
x1	Make resource available	S.x1 Available	0	X	X	X	X	X	X	0	X	X	X	X	X	X			
x2	Provide secure (encapsulated) service	S.x2 Requested	1	X	X	X	X	X	X	1	X	X	X	X	X	X			
x21	Provide idle period	S.x21 Idle	1	0	X	X	X	X	X	1	X	X	X	X	X	X			
x211	Provide arrival waiting time	S.x211 Waiting	1	0	X	X	X	X	X	1	0	X	X	X	X	X			
x212	Provide monitored shuttle transfer	S.x212 Watched Transfer Shuttle	1	0	X	X	1	X	0	1	X	X	X	X	X	X			
x2121	Transfer Shuttle	S.x2121 Transfer Shuttle	1	0	X	X	1	0	0	1	X	X	X	1	0	X			
x2122	Alert if timeout / transfer failure	S.x2122 Alarm	1	0	X	X	1	1	0	1	X	X	X	0	1	X			
x213	Terminate safely release action	S.x213 Terminating	1	0	X	X	X	X	1	1	X	X	X	X	X	1			
...																			

Figure 11. Statechart for a stopper system.

7 REFERENCES

[1] Berry G., and Cosserat I., "The ESTERAL Synchronous Programming Language and its Mathematical Semantics," *Seminar on Concurrency*, Lecture Notes in Computer Science 197, Springer, Berlin, 1985.

[2] Douglass B.P., *Doing Hard Time*, Massachusetts: Addison-Wesley, 1999.

[3] Dijkstra E.W., "Cooperating Sequential Processes", *Programming Languages* ed. F. Genuys, Academic Press, London, 1968.

[4] Harel D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[5] Hintersteiner J.D., and Nain A., "Integrating Software into Systems: An Axiomatic Design Approach", *Proceedings of the 3rd International Conference on Engineering Design and Automation*, Vancouver, B.C. Canada, August 1-4, 1999.

[6] Hintersteiner J.D., Tate D., "Command and Control in Axiomatic Design Theory: Its Role and Placement in System Architecture", *Proceedings of the 2nd International Conference on Engineering Design and Automation*, Maui, HI, August 9-12, 1998.

[7] IEC 61131-3, *Programmable Controllers Part 3: Programming Languages*, International Electrotechnical Commission, Geneva, 1993.

[8] Kopetz H., *Real-Time Systems – Design Principles for Distributed Embedded Applications*, Boston: Kluwer Academic Publishers, 1997. ISBN 0-7923-9894-7

[9] Suh N.P., *Axiomatic Design: Advances and Applications*, New York: to be published by Oxford University Press, 1999.

