

from the array “A” described above, one would write an expression like “A[3]” (depending on the exact programming language used). The programming language knows how to use the name of the array and its format (and the format of the integers) to extract the required information. However, note that the correspondence between 3 a.m. and the “3” in “A[3]” is determined by the abstraction mapping for this particular data structure. If this abstraction mapping were changed, then even if the format (an array of 24 integers) were unchanged, the expression for extracting the reading for 3 a.m. might have to change. For example, suppose that the abstraction mapping were changed such that the last reading of the day, the one taken at 11 p.m., was stored in A[0], the 10 p.m. reading in A[1], and the midnight reading in A[23]. Then to extract the 3 a.m. reading, one would use the expression A[20]. Similarly, if the abstraction function maps an integer t to t degrees Celsius, then to extract the reading in degrees kelvin requires adding 273 to t . But if the abstraction function were changed to storing the temperature readings in degrees kelvin, then extracting the reading in degrees kelvin would not require this addition.

A desire for greater efficiency in the program’s use of time or space is a common reason to change a program’s data structures. For example, if requests for temperature readings in degrees kelvin were more common than requests in degrees Celsius, then it would be more efficient to change the abstraction mapping and to store the readings in degrees kelvin. As another example, the speed of a search through a list of names can be made faster if the data structure is changed from a list to a binary search tree or a hash table. (Searching for a name in a list takes, on the average, time proportional to half of the number of the names in the list. On the other hand, searching for a name in a binary search tree takes, on the average, time proportional to the logarithm of the number of names.)

Another reason for changing a data structure is to allow the program to store or manipulate additional information. For example, to record the humidity during each hour of a day as well, one would need to either change the data structure for the temperature readings or add a new data structure to the program.

Why Changing Data Structures Is Costly. Making changes to a program’s data structures can be costly because information *about* them tends to propagate throughout the program. Of course, the information contained *in* the data structures, the information being manipulated by the program itself, needs to flow throughout the program; that causes no maintenance problems. What causes problems is the use of a data structure’s format and abstraction mapping throughout the program. When every part of the program uses these details, any change to these details could affect all parts of the program; hence any such change requires examining the entire program.

For example, consider a calendar management program. In this program, there will be a data structure used to represent dates. Suppose the format of a date is a record of three fields, each itself an array of two characters, with each character containing a digit. (This might be a reasonable choice if the most common operation performed on dates is to read or write them from or to some external device.) The abstraction mapping is as follows. The two digits in the “month” field’s

array represent the number of the month (with 1 for January), the two digits in the “day” field represent the day of the month, and the two digits in the “year” field represent the year within the current century. Without any further conventions or language support, these details will be used everywhere in the program.

Now suppose that, at some later time, we decide to change the program to deal with dates in more than one century. One way to make this change would be to change the data structure for dates, so that four characters were used to store the year (with the characters read left to right representing a year in the Common Era). However, details of the format and interpretation of dates might have been used anywhere in the program. For example, at one spot in the program, the year in a date, D, might be printed by printing the characters “1” and “9” and then printing the values of the expressions “D.year[0]” and “D.year[1].” However, if the year 2001 is stored with D.year[0] = “2,” D.year[1] = “0,” D.year[2] = “0,” and D.year[3] = “1,” then this way of printing the year is incorrect, because it would print 2001 as “1920”! This kind of dependence on details might occur anywhere in the program, hence changing the representation of dates to use four characters for the year requires an expensive reexamination of the entire program.

When the code that uses a data structure is found in several programs, the problems caused by the propagation of the detailed information about the data structure are essentially unlimited. For example, if a data structure is present in an operating system or a commercial software component, details about it may be written into so many programs that it will be extraordinarily expensive to change.

Information Hiding and Encapsulation

To avoid such difficulties, it is best to hide the detailed information about each data structure in a small section of one’s program (2). Such a small section of a program, which can typically be separately compiled and which typically hides the details of one data structure (or data type, see below), is called a *module*.

A module *hides* or *encapsulates* the details of a data structure if no other part of the program can directly depend on these details. That is, the parts of the program outside the module must manipulate the data structure indirectly. This technique is called *information hiding*, because it hides information about the data structure’s details in a module. It is also called *encapsulation*, because the module is a “capsule” containing the hidden information in the sense that the details are protected from the other parts of the program.

Operations (Methods). Program modules achieve information hiding by restricting code outside the program module itself to the use of a specified set of *operations*. In the jargon of object-oriented programming, these are called *methods*. These operations of a module “know” the details of the data structures that they manipulate, in the sense that they are coded using the details of the format of that data and its abstraction mapping. The other parts of the program manipulate the data structure indirectly, by calling the operations.

These operations can also be looked on as the set of services a module provides to the rest of a program. For example, one might have a module that hides information about the

details of the data structure used for storing the appointments in a calendar manager program. This module might provide operations to add an appointment to the set of scheduled appointments, to cancel an appointment, and to find all the appointments for a given date.

Specifying a Module's Interface. The operations of a module are critical, since they form the interface between the data structures encapsulated by the module and the rest of the program. That is, instead of depending on the details of the data structures, the rest of the program depends on the operations of the module. This means that the operations must be designed to be sufficiently high-level, or abstract, so that the details of the data structures hidden in the module do not leak through the interface. It also means that the exact behavior of the operations must be carefully specified.

A prime example of a set of operations that are too low-level to effectively hide information would be a set that included an operation that returned a pointer to the module's data structure to clients. With such an operation, clients would be able to directly manipulate the data structure. This would provide no information hiding at all. [In the jargon, returning a pointer to a module's internal data structure, or allowing a client to keep such a pointer to an data structure that is later made internal to the module, is called "exposing the representation" (3).]

The *specification* of a module's operations describes how to call each operation and what happens when it is called. Such a specification is best thought of as a contract (3,4). Like any contract, it gives benefits and obligations to each party involved. The two parties are the code that uses the operations (from outside of the module), called *client code*, and the module's own *implementation code*. The obligation of the implementation code is to make the operations behave as specified in the contract. The implementation code benefits by being able to use whatever algorithms or data structures are desired. (Usually the algorithms and data structures will be chosen to minimize costs.) The implementation also benefits by being able to change its algorithms and data structures easily, as described above. The obligation of the client code is to only use the operations provided by the module, and to use them in the ways specified in the contract; that is, the client code must not use any information about the details of the implementation's data structure. The benefit to the client code is that the client code can be written more succinctly, because it is written at a higher level. In addition, since client code is written at a higher level, it is easier to understand than it would be if it were written in terms of the details of the data structures. When, as is often the case, the client code makes up the bulk of the program, the program, as a whole, may become clearer. The program as a whole is also more easily improved. This may include making it more efficient, because efficiency problems with the program's data structures or algorithms may be more easily isolated and fixed, since they are confined to specific modules.

As a contract, the specification fulfills the usual roles familiar from contracts in the legal system. That is, as long as both sides fulfill their obligations, either party may change anything not specified in the contract. In particular, this includes the details of the implementation. In modern object-oriented programming languages, one can use a *class* as the implementation module. If this is done, then the language

will automatically prohibit direct access by clients to data structures in a class. This automatically ensures that the client code fulfills some of its obligations. But even if the contract is only enforced by convention, its use in hiding information about data structures is the key idea that allows them to be easily changed.

We will look at specifications in more detail below.

The Benefits of Information Hiding. If the details of a data structure's format and abstraction mapping are hidden in a module, then when the details need to be changed, one only has to examine and change that module. Because the module is a small part of a larger program, this makes the program easier to change. For example, suppose the calendar manager program has a module that encapsulates the data structure used for storing its user's appointments. Then the details of this data structure can be changed without examining the entire program; only the module in question needs to be examined.

From an economic standpoint, information hiding can be thought of as purchasing an option (5). Recall that, in economics, buying an option on X gives one a guarantee of being able to buy X at a later date for a specified price. Why not just wait until X is needed to buy it? Because either it might not be available at that time or it might cost more than the price specified in the option (plus the cost of the option). In this sense the option purchased by hiding information about the details of a data structure D is the ability to change, at a later date, the details of D's representation without looking at every part of the program. The cost of this option is the cost of writing a module that hides the information about D plus the efficiency loss (if any) that results from manipulating the data structures indirectly through the operations. This cost is justified if the cost of changing the data structure at a later date (i.e., the cost of examining the entire program) is more than the cost of just changing the module (plus the cost of creating and using the module). In a sufficiently large program, the cost of examining the entire program easily outweighs the upfront costs of creating and using a module to hide the data structure's details. Sophisticated compilers can also eliminate much of the cost of manipulating the data structures indirectly.

Other Applications of Information Hiding. This idea of information hiding is not limited to hiding data structure details. For example, it is often used in operating systems to hide information about the detailed management of devices or other resources. In an operating system, a "device driver" provides an interface, with specified operations such as "get" and "put," through which users can manipulate input/output devices. For example, a device driver allows users to manipulate storage media (such as hard disks) from many different manufacturers, without having to rewrite their programs when different media are installed.

The manual that describes the instruction set of a computer can also be seen as a specified interface that allows many different implementations, without requiring new compilers to be written each time a processor's detailed implementation changes. In this case the operations in the interface are the computer's instructions. The assembly language for a computer is a human-readable abstraction of this interface.

Similarly, the reference manual of a programming language also provides a specified interface that insulates the programmer from changes in the computers that the program may run on, and thus it provides a degree of independence from the details of these computers. The operations of this interface are the statements, expressions, and declarations of the programming language, including the ways in which these can be combined.

A programming language may also provide information hiding for its built-in data structures, such as integers and floating point numbers. It does this by specifying the syntax and meaning of the operations that can be used to manipulate them. In this context, information hiding is called *representation independence*, because programs are not dependent on the formats for integers and floating point numbers found on different computers.

Abstract Data Types

Data Types. A family of similar data structures, each with the same format and abstraction mapping, is called a *data type*. The individual data structures in such a family are called *instances* or *objects* of that data type.

Often a program will use many instances of a data type; for example, the calendar manager program might have many instances of the data type Appointment, each one a data structure that represents a separate appointment. It is inconvenient to define a module for each such instance of a data type. Furthermore, if the objects of such a data type are created as the program runs, then it is not possible to define a program module for each object, because the modules are created before run-time.

Because of these problems, it is common to define a module for each data type in a program. In object-oriented programming, this is often taken to an extreme; that is, modules are not created for individual data structures but are only used for data types.

Definition. An *abstract data type* (ADT) is a specified interface for modules that encapsulate a data type. This definition can be seen from two points of view. From the implementation side, such a specification permits many different implementation modules. Thus, an ADT can be viewed as an abstraction of all the modules that correctly implement the specified interface. Each such module consists of code that implements the operations; each of these may use a different format and abstraction mapping for the data structures of the objects of the data type that it manipulates.

However, from the client's side, internal differences in the implementation modules do not matter. Since clients can only depend on the operations in the module's interface and their specified behavior, all modules look the same to the client. Hence, from the client's side, an ADT can be thought of as a set of objects and a set of operations that manipulate them. This agrees with the definition given earlier.

What connects these two points of view is the ADT's specification. The specification describes the operations in terms of the abstract values of objects. For example, the abstract value of an Appointment object might be a tuple of a date, a time, and a string stating the purpose of the appointment. The operation that changes the length of an Appointment is specified in terms of how it affects the time part of this tuple. On

the implementation side, the data structure's abstraction mapping connects the data format to the abstract values; one can check the correctness of the code that implements the operations with respect to the specification by comparing the abstract value at the end of the operation with that specified. On the client side, to produce a desired effect on the abstract value of an object, the client invokes the ADT's operations. An implementation can thus be seen as translating such abstract requests into operations performed at the lower level of its data structure's format.

CREATION AND DESIGN OF ADTs

This section discusses the creation and design of ADTs. It begins with general considerations, and then it describes elaborations found in object-oriented programming.

What ADTs to Specify?

How does one decide what ADTs should be used in a program? A basic strategy is to consider key concepts one would use to describe the workings of an application program (or a set of similar programs) to be candidates for ADTs (16). These may include "tangible things" (such as refrigerators or sensors), "roles" (such as plumber or engineer), "events" (withdrawal, registration), "interactions" (meeting, contract), "places," "concepts" (dates, times), and "organizational units" (the IEEE) (23). Key concepts will be described as important by domain experts. One can evaluate the suitability of an ADT by checking that it can be named with a noun (or sometimes an adjective) and that it has more than one operation (16). The operations will often be named by verbs or verb phrases.

However, both the set of ADTs and the set of operations often need to be expanded with some "internal" types and operations. For example, it may be useful to use a Stack or binary search tree for algorithmic or efficiency purposes, even though these are not nouns used to describe the application. Similarly, it may be useful to have copy or iteration operations, even though these are not verbs used in describing the application.

Finding ADTs in the Calendar Manager Example. For example, consider again the calendar manager program. The key abstractions are easy to identify as concepts (dates and times) and interactions (appointments). Each of these has several operations. Domain experts such as secretaries, would identify these concepts as important. Each can also be named with a noun: Date, Time, Appointment.

We reject creating new ADTs for abstractions that can already be handled by existing ADTs. For example, we do not create an ADT to record the purpose of an appointment, as the existing string abstraction will serve.

Checking a Design by Assigning Responsibilities. One way to refine and check a planned set of ADTs is to look at how the system's "responsibilities" are partitioned among the various ADTs (6).

A responsibility is a task found in the system's requirements. For example, an Appointment object might take responsibility for remembering an appointment's date, time, length, purpose, and other annotations. However, remember-

Class	Responsibilities
Schedule	remember all appointments add or delete an appointment return all appointments for a date return all appointments for a date and time schedule recurring appointments
Appointment	remember date, time, purpose, and place compare date and time to another appointment return an appointment like this, but with a new date
Time	remember hours and minutes past midnight compare to another time return number of hours and minutes to another time return the time a given number of hours and minutes from this time
Date	remember year, month, and day return the day of the week return the next date return the date a given number of days in the future

Figure 1. Classes and responsibilities for the calendar manager example.

ing a collection of Appointment objects and organizing them into a schedule might be the responsibility of a Schedule object.

To double check that the set of ADTs is adequate, one can see if each responsibility in the system's requirements document are partitioned among the various ADTs. Each responsibility should be assigned to a single type of object. Doing so helps keep information about how that responsibility is managed hidden within a single ADT. For example, if changing an appointment's purpose was the responsibility of both a Schedule and an Appointment object, then information about the purpose of appointments could not be hidden entirely within an Appointment (or Schedule) object.

For example, in Fig. 1, several ADTs and their responsibilities for the calendar manager program are described. One notable aspect of this list is that it highlighted the responsibility of remembering the place of an Appointment, which I had previously forgotten. It also becomes clear from this set of responsibilities that no ADT is responsible for communicating with the user. Additional ADTs should be designed to handle the user interface.

Checking a Design by Use-Case Scenarios. Another way to check the suitability of a design is by tracing various "use-case scenarios" (7) in the design. A *use-case scenario* is a particular way in which the system will be used. For example, for the calendar manager program, one use case is that a user will check their appointments for the day. Another use case would be to create a new appointment for a date other than the current date.

The way to use a use-case scenario to check a design is to see how the ADTs in the design are used while playing out the scenario. For example, consider the use case in which the user checks their appointments for the day. In this case the user interface (which is so far missing in our design) gets the command to check appointments, it finds the current date (also missing from our design), and then it asks the Schedule object for a list of appointments for that date. It might also sort the appointments in order by their starting time, which involves asking each appointment object for its starting time.

To display the appointments, it will have to fetch the other information from each: the ending time, the place, and the purpose. This process helps to find missing ADTs and to ensure that all required responsibilities are covered. It also tends to give one a good sense for what operations will be useful for carrying out the required tasks.

Designing Individual ADTs

Once the responsibilities of each ADT are decided, one can think about the detailed design of each ADT. This involves deciding on what information the ADT instances are responsible for holding, and what operations it is able to perform.

Designing Object States. Objects are usually responsible for remembering some information. The client's view of this information is the object's *abstract state*. Another name for this is the object's *abstract value* (8). The abstract value of an object is a mathematical abstraction of its representation in the computer. Recall that the set of abstract values is the target of a data structure's abstraction mapping. In designing an object's state, it is best to focus on this abstract, client-centered point of view and not plunge into details of the format of some particular data structure. Doing so helps ensure that the objects can be described adequately to clients and that the implementation can be changed easily.

As an example, consider designing the state of Time objects. A reasonable set of abstract values might be pairs of integers, each representing the number of hours and minutes past midnight. One can also think of the parts of such a pair as an object's *abstract fields* or *specification variables*. An abstract field does not have to be implemented; such "fields" are only used for specification purposes. Since the set of abstract values is a mathematical concept, the format of an implementation data structure does not need to have two fields, despite the use of two abstract fields in the specification. For example, an implementation might use a single integer field, representing the number of minutes past midnight. The only requirement is that the implementation's data structure have

an abstraction mapping that maps the chosen format to the abstract value set in a way that makes the operation implementations meet their specifications (8).

Why not use a single integer as the abstract value of a Time object? That can certainly be done. However, since abstract values are mathematical concepts, convenience and clarity are their most important attributes. One should not worry about “saving storage space” in the design of abstract values. Space is cost-free in mathematics! (See also the section on “Evaluating ADT Designs” below.)

Compound objects may have other objects as part of their abstract values. For example, consider Appointment objects. A reasonable choice for their abstract values might be a tuple of a Date object, a Time object, and two Strings (for the appointment’s purpose and place). This is clearer than (more high level than) specifying the abstract values as a tuple of 5 integers and two Strings, since the 5 integers have to be separately interpreted as representing the year, month, day, hour, and minutes.

To make such specifications of compound abstract values work, it is necessary to use the concept of object identity. We postulate that each object has a unique *identity*, which can be thought of as its address in a computer’s memory. Two objects with the same abstract value do not necessarily have the same identity. For example, two appointments with the same date, time, and purpose may have different identities; one way this can arise is if a user copies an appointment, perhaps to later change its date. Having two separate objects (i.e., with different identities) is important for making this scenario work. Thus, the abstract values of our Appointment objects would be a tuple containing a Date object’s identity, a Time object’s identity, and a String object’s identity.

It is sometimes useful to distinguish between a collection of object values and a collection of object identities, because these have different kinds of abstract values. For example, it might be reasonable for the abstract value of a Schedule object to be a set of Appointment object identities. This would allow one to store a copy of an existing Appointment in a Schedule object. By contrast, if the abstract value of a Schedule object were a set of values of Appointment objects (i.e., tuples instead of object identities), it would be impossible to store a separate copy of an Appointment object.

Designing Operations. The operations of an ADT correspond to the actions that are required to carry out its responsibilities. We start by describing various kinds of operations and then turn to the design of individual operations.

Kinds of Operations. When beginning to specify the operations of an ADT, it is helpful to think of specifying operations of several standard kinds. The first kind is one that creates or initializes objects. Such an operation is commonly called a *constructor*. An operation that creates an object of type T without using any other objects of type T is called a *primitive constructor*. Some programming languages, such as Smalltalk, allow primitive constructors to create the new objects and return them to the caller. But many languages, such as C++ and Java, take the responsibility upon themselves to create objects and only allow a primitive constructor to initialize them once they are created. In any case, unless one is specifying an abstract type that is not supposed to have objects created for it, one wants some primitive constructor operations. (A data type that is not supposed to have objects

created for it is sometimes useful in object-oriented programming, where it can be used as the supertype of some other type of objects. Classes that implement such types are called *abstract classes*.)

Another kind of operation that is common in ADTs is an *observer*. Observer operations are used to extract information from objects. For example, an operation that would extract the date of an Appointment object would be an observer.

The opposite of an observer operation is an operation to change an ADT’s objects. Such an operation is called a *mutator*. A mutator changes an object’s abstract value. For example, an operation to change the date of an Appointment would be a mutator.

It is also possible to have operations of mixed kinds. Mixed constructors and observers are called *nonprimitive constructors*. For example, an operation that takes a Time object and returns a new one that is for a time one hour later would be a nonprimitive constructor.

Mixes of observers and mutators are sometimes appropriate; but because observers are used in expressions, and side effects in expressions make reasoning about programs more difficult, they should be avoided (16). However, if one is designing an ADT for a programming language that has expressions as its primary syntactic unit, like Smalltalk, then such operations may be needed. For example, in Smalltalk, every operation returns a value; hence mutators are also, in a sense, observers. By convention, in Smalltalk every mutator returns the implicit argument of the operation (named “self”).

Operations that do a mutation and then return some sort of “status code” may often be more appropriately designed as mutators that may throw exceptions.

Errors and Exceptions. Another consideration in designing the operations of an ADT is how to handle errors or exceptional cases. *Errors* arise from misuse of an operation—for example, changing the ending time of an appointment to be before its starting time. *Exceptions* are unusual but not completely unexpected events—for example, reading past the end of a file. There are two general strategies for dealing with errors and exceptions:

1. Have the clients check for them.
2. Have the implementation check for them.

However, for a given ADT, a firm choice between these alternatives should always be made and recorded in the ADT’s specification. If a firm choice is not made, then both the implementation and client code will, for defensive purposes, always check for such conditions. Such duplicate checks can be a source of inefficiency; hence it is always a good idea to decide on one of these two strategies.

If clients must check for errors and exceptional situations, then the specification should use preconditions. A *precondition* is a logical predicate that that says what clients must make true when an operation is called. For example, the operation that changes the ending time of an Appointment object might have a precondition that requires the ending time to be later than its starting time. If this is done, then the operation’s implementation can assume that the ending time given is later than the starting time; hence it need not check for this error. (If the operation is called with inputs that do not satisfy the precondition, then the implementation is not obligated by the specification to do anything useful; it might even

go into an infinite loop or abort the program.) Mathematically, one can view an operation as a relation between its inputs and its outputs. The precondition describes the domain of this relation; that is, the precondition describes what inputs are valid.

If the clients of an ADT cannot be trusted to do the checking themselves, then it is best to specify that the implementation must check for them. Such *defensive* specifications are useful for general-purpose ADTs, those for whom the clients are unknown. In a defensive specification, the operation that changes the ending time of an Appointment checks that the ending time is later than the starting time and throws an exception if it is not. Client code that needs to validate its own input could catch this exception.

If a type is to have both untrusted and trusted clients, then it may be useful to specify both kinds of ADT for the same concept. Untrusted clients can use the ADT with the defensive specification. Trusted clients can use the ADT specified with preconditions. Furthermore, the defensive specification can be implemented by using an implementation of the ADT specified with preconditions. Since this can always be done, when in doubt write the specification using preconditions.

Immutable Objects. For some types, it is reasonable to not have any mutator operations. Objects of such a type are called *immutable*. Since there are no mutators in such a type, an immutable object's abstract value does not change over time. As such, immutable objects often represent pure values. For example, both Time and Date objects are immutable, which matches one's intuition that a specific time or date are unchanging measures. Such objects typically have many non-primitive constructors. Compound objects, however, are typically *mutable*.

Evaluating ADT Designs. The specification that forms the contract between client code and implementations is of the greatest importance. It is most important that it be abstract enough. A specification could fail to be abstract enough by being too close to some particular implementation data structure. One specification, *A*, is more abstract than another, *C*, if *A* has more correct implementations than *C* does. An implementation is correct if its data structures and algorithms meet the specified contract. Implementations will be different if they have different data structures and algorithms. If *A* is more abstract than *C*, then *A* is also said to be a *higher-level* specification than *C*. In this case, *C* is said to be a *refinement* of *A*. Higher-level contracts, since they allow more implementations, allow the data structures and algorithms used in a program to be more easily changed, because any change from one correct implementation of a given contract to another does not affect client code. A contract that only allowed one implementation would thus not allow any changes to the implementation data structures. Such a low-level specification would be an extreme case of *implementation bias* (9).

Intelligence. In designing the operations of an ADT, it is important to try to make them "intelligent" (10). The intelligence of the operations of an ADT can be estimated by how easy it is for clients to make changes to objects that seem common or likely. That is, the operations should not just fetch and set the information stored in the objects, but ideally should perform more complex services for the client code. One can identify some of these by considering various use-case scenarios. By having the operations do more than simply

fetch and set the information, part of the application logic can be handled by the ADT, and the client code will be simplified. However, equally to be avoided is putting all of the application logic into an ADT. Instead, a middle ground is ideal.

As an example, consider the Appointment type. A low-level design might treat an Appointment as a record, with operations to get and set the appointment's date, time, length, place, and purpose. A better design has operations to fetch this information, but would also include operations to (a) change the length of an existing appointment by some specified amount of time, (b) create hour-long or half-hour-long appointments, (c) create a similar appointment for the same time next week, next month, or next year, and (d) compare a given date and time to the appointment's date and time.

Observability and Controllability. One can also think about the design from the perspective of whether it allows access to and control of the state of the objects of that type (or other hidden resources). Good designs are both observable and controllable (11,12). An *observable* ADT allows its clients to extract the intended information from each object. For example, if the date or time of an appointment object could not be obtained by using the operations of the Appointment type, then the Appointment ADT would not be observable. A *controllable* ADT allows its clients to make the type's objects change into any desired state; that is, the object should be able to be put into a state where its observable information has an arbitrary legal value. For example, if the Appointment data type does not provide a way to make an appointment on February 29 of a leap year, then the type would not be controllable.

The notions of observability and controllability can be made more formal by thinking about the abstract values of a type's objects. For example, the Time ADT will be observable if the number of hours and minutes past midnight can be computed from its objects. It will be controllable if objects can be made with any number of hours between 0 and 23 and minutes between 0 and 59. A type with mutable objects is controllable if each object can be mutated to taking on any abstract state.

Observability and controllability have to be kept in balance with the requirements of encapsulation and intelligence. An ideal design will have all of these properties.

Cohesion and Coupling. Each ADT should have a well-defined area of responsibility. An ADT design is *cohesive* if its responsibilities "hang together" because they are closely related. For example, the Appointment ADT would not be cohesive if, besides its responsibilities for manipulating the date, time, place, and purpose of an appointment, it was also responsible for the low-level details of playing a sound file through a computer's speakers. Although such a responsibility might easily evolve from the ability of an appointment to play an alarm, it has little to do with the other responsibilities of the Appointment ADT. If it were included, it would make the ADT less cohesive.

One can check cohesion more carefully when the abstract values of objects are specified using abstract fields. To do this, one checks whether each operation of an ADT reads or writes either just one or every abstract field. There should also be at least one operation that uses all the abstract fields. Operations that use just one abstract field are okay, as are operations that use all the fields. However, if an object has three abstract fields and an operation only deals with two of them, then the abstraction fails the cohesion check. The solution is

to remove the offending operation or to split the abstraction into different ADTs.

Coupling is a measure of how much one module is dependent on another module. Strong coupling between two modules means that when one is changed, then the other should also be changed, or at least checked to see if it needs to be changed. Hence, it is best to avoid strong coupling. Strong coupling may occur when two modules use the same global variables. For example, if the Schedule ADT and the Appointment ADT both used a global variable that holds the current date, then this would provide unnecessarily strong coupling between them. To avoid strong coupling, it is best to pass such information to the relevant operations as an argument or to have both call some other operation, instead of using global variables.

Parameterized (Generic) ADTs

Many ADTs can be generalized to be more reusable by parameterization. For example, consider an ADT for a sequence of 24 hourly temperature readings, `HourlyReadings`. The operations of this type would include: getting a reading for a given hour (“fetch”), setting a given hour’s reading to a given temperature (“store”), and perhaps finding the average, minimum, and maximum temperature in that period. Thus far we have been treating the temperature readings as integers. But suppose that for various applications, we need different amounts of precision. One application might need to keep readings to the nearest integer degree, but others might need floating point numbers. Clearly, the idea of the `HourlyReadings` ADT, which has objects with abstract values that are a sequence of 24 hourly temperature readings, is applicable to both cases. Making two separate ADTs for such closely related concepts would cause duplication in the implementations.

Instead of duplicating the code in such cases, it is better to abstract from the family of related ADTs by making a *generic* or *parameterized* ADT that, when instantiated, can generate each member of the family. Such an ADT can be thought of as a function that takes a type and produces an ADT; and so it is an ADT generator in the sense that when the parameters are supplied, it generates an ADT. In the specification, the type parameters are thought of as a fixed but arbitrary types. The usual notation in specifications is to talk about a generic instance of the generator, such as `HourlyReadings[T]`, where “T” is the name of the abstraction’s formal type parameter. Clients instantiate such a type generator, making a type by passing it an actual type parameter. For example, `HourlyReadings[int]` would be an ADT which keeps hourly temperature readings as integers, while `HourlyReadings[float]` would keep them as floating point numbers.

Often some operations are required of the types of objects used as actual type parameters. For example, `HourlyReadings[T]` might require that the type T have the usual arithmetic operations of addition and division (by an integer), which would allow the average of the readings to be computed. The

signatures and behavior of these operations should be stated in the specification of the ADT generator (13).

In programming languages that support parameterized ADTs, such as C++ and Ada, the use of explicit type parameters extends to operations and subroutines. For example, a parameterized function, such as `sort[T]`, can be thought of as a function generator. This style of programming leads to *parametric polymorphism*, in which one piece of code can operate on many different kinds of data. Object-oriented programs exhibit a different kind of polymorphism, which comes from message passing and subtyping.

Refinements for Object-Oriented Programs

Message Passing (Dynamic Dispatch). To explain the message passing mechanism of object-oriented (OO) languages, we first present the problem it solves.

In a non-OO language, one must know the exact type of an object in order to apply operations to its objects. For example, in Ada 83, one would write “Appointment’get_start(myAppt)” to get the starting time from the appointment named “myAppt.” This uses the operation `get_start`, found in the program’s implementation of `Appointment`.

The need to know the exact type of an object makes it more difficult to change programs. For example, suppose that after writing the first version of the calendar manager program, one adds a second type of appointment, `RecurringAppt`. In Ada 83, one would have to write “RecurringAppt’get_start(myRecAppt)” to extract the starting time from an object, `myRecAppt`, of this type. Since the program needs to manipulate both types of appointments at once; and since it cannot know which to expect, it must use a variant record data structure, which can hold either type. A variant record object has an abstract state that consists of two abstract fields: a type tag and an object whose type depends on the type tag. For example, we might have a variant record type, `Appt`, with two possible type tags, “Appt_tag” and “Recurring_tag.” When the type tag is “Appt_tag,” the object stored is an `Appointment` object; when it is “Recurring_tag,” the object is a `RecurringAppt` object.

Using a variant record, “appt,” one might write something like the code in Fig. 2 to extract the start time of `appt`. The code tests the type tag of `appt`, and then it extracts the start time from the object by dispatching to the appropriate type’s `get_start` operation. Not only is this tedious, but it makes the program difficult to change. Imagine what happens if another type of appointment is added to the program: All such dispatching case statements must be found and updated.

The message passing mechanism in OO languages is designed to automate this kind of code. Because of that, it also makes adding new types that are similar to existing types easier. In essence, all objects in an OO language are like the variant record objects in non-OO languages. That is, objects in an OO language allow one to find their exact type at run-time. Each object contains a pointer to some language-specific data structure that represents the class at run-time; typically this includes the name of the class, along with the code for

```
case appt.type_tag of
  Appt_tag => return Appointment'get_start(appt.appointment);
  Recurring_tag => return RecurringAppt'get_start(appt.recurring);
end
```

Figure 2. Ada 83-like pseudocode for dispatching based on the type code in a variant record object.

the instance methods of that class. An *instance method* or *instance operation* is a method that takes an existing object as an argument. The primitive constructors, and other such operations that are still called directly, instead of being dispatched to indirectly, are called *class methods* or *class operations*. (Class methods need not be stored in the class.)

Calls to instance methods are dispatched to the appropriate code based on the run-time type of the object involved. Abstractly, the view presented to the programmer is that objects contain the operations that work on them. In a singly dispatched language, like Smalltalk, C++, and Java, instance methods are dispatched on what would otherwise be the first argument of an operation. This argument is sometimes called the *implicit* or *default argument* to an instance operation. For example, instead of writing “Appointment.get_start(myAppt),” one writes “myAppt.get_start(),” and the object myAppt is the implicit argument. Within the code for an instance method, the default argument is named “self” (in Smalltalk) or “this” (in C++ and Java). The syntax “myAppt.get_start()” embodies the idea that, to invoke a method, one first extracts the method from the object, and then calls it. This mechanism is called *dynamic dispatch* or *message passing*. When the term “message passing” is used, the name of the method invoked (“get_start”) and its arguments are thought of as a *message*, which is sent to the object, to ask it to do something. An invocation such as “myAppt.get_start()” is also thought of as sending the message “get_start()” to the object myAppt. (Note, however, that no concurrency or distribution is necessarily involved.)

Subtype Polymorphism. Message passing allows client code to be written that is independent of the exact types of objects. That is, client code that sends messages to objects is polymorphic, since it can work on objects of different types. This kind of polymorphism is called *subtype polymorphism*. Like parametric polymorphism, it helps make code more general and resistant to change. It is related to parametric polymorphism in that message passing code uses operations of types that are passed to it. However, in subtype polymorphism the types are passed to the code at run-time, in the objects being manipulated, instead of being passed at compile-time, separately from the objects, as in parametric polymorphism.

Message passing and subtype polymorphism focus attention on the instance methods of objects and downplay the role of class methods. Consider the set of all messages that can be sent to an object. This set forms the object’s *instance protocol* (14) and is, in essence, a collection of signature information. It corresponds to a Java interface.

Suppose that objects of type S have an instance protocol that includes all the messages in the instance protocol of type T . Then S objects can be manipulated as T objects without encountering any type errors. This means that the type S is a *subtype* of T . In OO languages, if S is a subtype of T , then objects of type S can be assigned to variables of type T , passed as parameters where T objects are expected, and returned from functions and methods that are declared to return T objects. This is safe because any message that is sent to an object that is supposed to have type T is in the protocol of S objects. For example, RecurringAppt is a subtype of Appointment if any message that can be sent to an Appointment object can be sent to a RecurringAppt object without encountering a type error.

Behavioral Subtyping. Client code does not just depend on the absence of type errors; it also depends on the behavior of objects that it manipulates. An ADT S is a *behavioral subtype* of T if each S object behaves like some T object when manipulated according to the specification of T ’s instance protocol (15). In essence, the objects of a behavioral subtype have to obey the specifications of all the instance methods of their supertypes (4,16–18).

With behavioral subtyping, message passing becomes truly useful, because one can reason about, and test, the correctness of client code in a modular fashion (19). For example, if one can show that an operation that manipulates an Appointment object must accomplish a certain task, based on the specification of the type Appointment, then this conclusion will be valid for all behavioral subtypes of Appointment. Hence this client code will not only be able to manipulate objects of behavioral subtypes—it can do so predictably. It is this property that allows OO programs to be developed in an evolutionary manner.

Inheritance (Subclassing). Strictly speaking, the inheritance mechanism of OO languages has little to do with abstract data types. Recall that a class is a program module that can be used to implement an ADT in an OO language. *Inheritance* is a mechanism that allows one to implement a class by stating how it differs from some other class. A class defined by inheritance is called a *subclass* or *derived class*. A subclass may have (in some languages, like C++) more than one *superclass* or *base class* from which it is derived. A subclass will inherit fields (data structures) and method code from its superclasses. However, code for class methods such as primitive constructors is not inherited by subclasses. A subclass can also add new fields and methods. It is also possible to add some behavior to a superclass’s method without rewriting it completely.

Because of these properties, it is often convenient to implement a behavioral subtype by using inheritance to derive a subclass of a class that implements the supertype. For example, one way to implement RecurringAppt as a behavioral subtype of Appointment is to use a subclass of a class that implements Appointment.

Inheritance is not Behavioral Subtyping. However, it is important to realize that inheritance does not necessarily produce classes that implement behavioral subtypes. For example, if the RecurringAppt class redefines the get_start() method to go into an infinite loop or to always return midnight, then RecurringAppt would not correctly implement a behavioral subtype of Appointment. Indeed, in C++ one can use “private inheritance” to make a subclass that does not produce a subtype. When using C++ or a similar language, one should either make subtypes that are behavioral subtypes or use private inheritance. Doing so ensures that the type system’s checking enforces not only subtyping but also behavioral subtyping. (Of course, the type system will not prove that one type is a behavioral subtype of another, but it will track the declarations given to it.)

RELATED TOPICS

Specifications can be written either formally, using some mathematically well-defined notation, or informally, in En-

glish. Informal specifications suffer from ambiguity, although they are useful for giving overviews and motivation and in situations where the cost of ambiguity and misunderstanding is not high. However, even if you do not use formal specifications, studying them will help you in being more precise in your use of informal techniques. See SPECIFICATION LANGUAGES.

Formal specifications can be used to formally verify the correctness of an implementation of an ADT (8). See PROGRAMMING THEORY.

Some form of specification is needed for testing (validating) code. *Black-box* testing of implementations of an ADT is entirely based on the ADT's specification; the specification is critical for telling what the results of a test should be.

ADTs that are used in concurrent or distributed programs have to deal with issues such as locking, to prevent race conditions such as two clients extracting the same job from a queue. One way to do this is to specify that the ADT makes some client operations wait for some condition to be made true by other clients (20,21). A sampling of formal specification techniques for concurrent and real-time systems is found in the book *Formal Methods for Industrial Applications* (22). See also REAL-TIME SYSTEMS.

BIBLIOGRAPHY

1. B. W. Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
2. D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Commun. ACM*, **15** (12): 1053–1058, 1972.
3. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, Cambridge, MA: MIT Press, 1986.
4. B. Meyer, Applying design by contract, *Computer*, **25** (10): 40–51, 1992.
5. K. J. Sullivan, P. Chalasani, and S. Jha, *Technical Report #97-14*, Computer Science Dept., University of Virginia, Charlottesville, VA: 1997.
6. R. Wirfs-Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Englewood Cliffs, NJ: Prentice-Hall, 1990.
7. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Reading, MA: Addison-Wesley, 1994.
8. C. A. R. Hoare, Proof of correctness of data representations, *Acta Inf.*, **1** (4): 271–281, 1972.
9. C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed., Englewood Cliffs, NJ: Prentice-Hall, 1990.
10. A. J. Riel, *Object-Oriented Design Heuristics*, Reading, MA: Addison-Wesley, 1996.
11. D. L. Parnas and D. P. Siewiorek, Transparency in the design of hierarchically structured systems, *Commun. ACM*, **18** (7): 401–408, 1975.
12. W. F. Ogden et al., Component-based software using RESOLVE I. The RESOLVE framework and discipline—A research synopsis, *ACM SIGSOFT Softw. Eng. Notes*, **19** (4): 23–28, 1994.
13. G. W. Ernst et al., Modular verification of Ada generics, *Comput. Lang.*, **16** (3/4): 259–280, 1991.
14. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.
15. B. Liskov and J. Wing, A behavioral notion of subtyping, *ACM Trans. Program. Lang. Syst.*, **16** (6): 1811–1841, 1994.
16. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Englewood Cliffs, NJ: Prentice-Hall, 1997.
17. A. Wills, Refinement in Fresco, in K. Lano and H. Houghton (eds.), *Object-Oriented Specification Case Studies*, Englewood Cliffs, NJ: Prentice-Hall, 1994, pp. 184–201.
18. K. K. Dhara and G. T. Leavens, Forcing behavioral subtyping through specification inheritance, *Proc. 18th Int. Conf. Softw. Eng., Berlin, Germany*, Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 258–267.
19. G. T. Leavens and W. E. Weihl, Specification and verification of object-oriented programs using supertype abstraction, *Acta Inf.*, **32** (8): 705–778, 1995.
20. B. Liskov and W. Weihl, Specifications of distributed programs, *Distrib. Comput.*, **1**: 102–118, 1986.
21. R. A. Lerner, *Specifying Objects of Concurrent Systems*, Ph.D. thesis, and *Technical Report CMU-CS-91-131*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, 1991.
22. J.-R. Abrial, E. Börger, and H. Langmaack (eds.), *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Controller*, Berlin: Springer-Verlag, 1996.
23. G. Booch, *Object-Oriented Analysis and Design*, 2nd ed., Reading, MA: Addison-Wesley, 1994.

Reading List

Meyer's book, *Object-Oriented Software Construction* (16), is a comprehensive discussion of object-oriented techniques with an extensive bibliography. This book also discusses more formal specification techniques for abstract data types.

A quick introduction to object-oriented design can be found in the book *Designing Object-Oriented Software* (6). Programmers should read the widely acclaimed *Design Patterns: Elements of Reusable Object-Oriented Software* (14).

GARY T. LEAVENS
Iowa State University

ABSTRACT DATA TYPES. See SOFTWARE REUSABILITY.

AC-AC POWER CONVERTERS

Ac-to-ac converters transform electrical power from one ac source to another. Theoretically, each ac source can have a different number of phases and operate at different or variable voltage and frequency. Figure 1 shows the basic block diagram of an ac-to-ac converter where the converter links two different ac sources together.

In fact, a very large number of power electronic converters sold today are ac-to-ac converters because electrical power is distributed mainly via three-phase or single-phase ac networks at a fixed voltage and frequency (50 or 60 Hz) whereas most electrical loads require single- or three-phase power at variable voltage and frequency. Examples are adjustable speed ac motor drives, frequency converters between 50 Hz, 60 Hz and 400 Hz networks, interties between three-phase 50

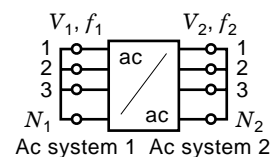


Figure 1. Basic block diagram of an ac-to-ac converter.

Hz and 60 Hz power grids, and interties between three-phase 50 Hz systems and single-phase 16 $\frac{2}{3}$ Hz systems.

Most ac-to-ac converters provide only unidirectional power flow because their circuitry and control are less complicated and more economical. Typically, these low-cost converters operate with a two-stage conversion method. First, they transform the ac power into dc power creating an intermediate dc link by using a diode rectifier (ac-to-dc converter). The second conversion stage is accomplished with an inverter (dc-to-ac converter) which generates the variable ac voltage for the load (1–5). Typical examples are motor drives for pumps, blowers, and compressors which do not regenerate power back to the ac grid. To connect ac sources together and to exchange power in both directions, bidirectional topologies are needed. Three distinctively different converter circuits have been developed to this end:

- ac-to-ac bidirectional converters with an intermediate dc link;
- cycloconverters; and
- matrix converters.

Whereas dc-link converters use a two-stage power conversion method (by a rectifier and an inverter), the cycloconverter and the matrix converter transform ac power in a single-stage avoiding dc energy storage devices (capacitors or inductors). All three circuits transform power between ac sources and loads that operate at different or variable voltage and frequency. In addition, several bidirectional power electronic circuits have been developed specifically to transform only the ac voltage amplitude or phase of the ac power system. Direct frequency control of the ac power is not possible with the following dedicated low-cost systems:

- ac voltage regulators;
- ac voltage or tap regulating transformers; and
- ac phase controllers.

AC-TO-AC CONVERTERS WITH AN INTERMEDIATE DC LINK

Three basic types of dc-link systems can be realized depending on the energy storage device used in the dc link. These systems are illustrated in Fig. 2. Large capacitors that store electrostatic energy are used to realize a relatively stable dc-link voltage [Fig. 2(a)]. This type of converter is comprised of two *voltage-source converters* because the dc-link voltage is impressed sequentially on both ac systems. Alternatively, it is possible to use inductors to maintain a stable dc link current. In these current-source systems the energy is stored electromagnetically, and current is impressed on the ac systems by the converters [Fig. 2(b)]. Both systems have dual properties, whereby the properties and characteristics that are true for the voltage quantities in the voltage-source converter are often valid for the currents in the *current-source converter*. Furthermore, it is possible to combine voltage- and current-source converters to construct mixed-dc-link converters as shown in Fig. 2(c).

Presently, costs and technology developments have led users to prefer voltage-source systems in the lower power range from 1 kVA up to 100 MVA, whereas current-source systems

and mixed-dc-link converters are often used at higher power levels ranging from 1 MVA up to 1 GVA.

Voltage-Source ac-to-ac Converters

Typical voltage-source ac-to-ac converter configurations are detailed in Fig. 3. Two converters are linked in parallel to a common dc-link voltage. Usually, these converters are single-phase or three-phase inverters (normal operating mode is from dc to ac). The sign and amplitude of the dc voltage are critical for operating the ac-to-ac converter. The dc voltage must remain positive for both inverters and must be larger than the amplitude of the line-to-line voltage of either ac supply. The dc current of each inverter can be positive or negative. To limit current peaks, the source impedance ought to be inductive because the inverters impress the dc voltage on the ac sources. This property is usually fulfilled in practice because most ac sources have transformers whose leakage inductances are significant. Freewheeling diodes, placed anti-parallel across each power switch, are necessary to provide a current path for the inductive source current to flow whenever a switch turns off. The voltage across each power semiconductor device at turn-off will be clamped to the dc voltage through the freewheeling diodes of the complementary switches.

Modern power devices used in inverter phase legs are power transistors, such as power MOSFETs (up to 1 kV) and insulated gate bipolar transistors (IGBTs). IGBTs are available with blocking voltages ranging from 600 V up to 3.3 kV. At higher power levels gate turn-off (GTO) thyristors up to 9 kV are used. Standard GTO devices require snubbers to limit

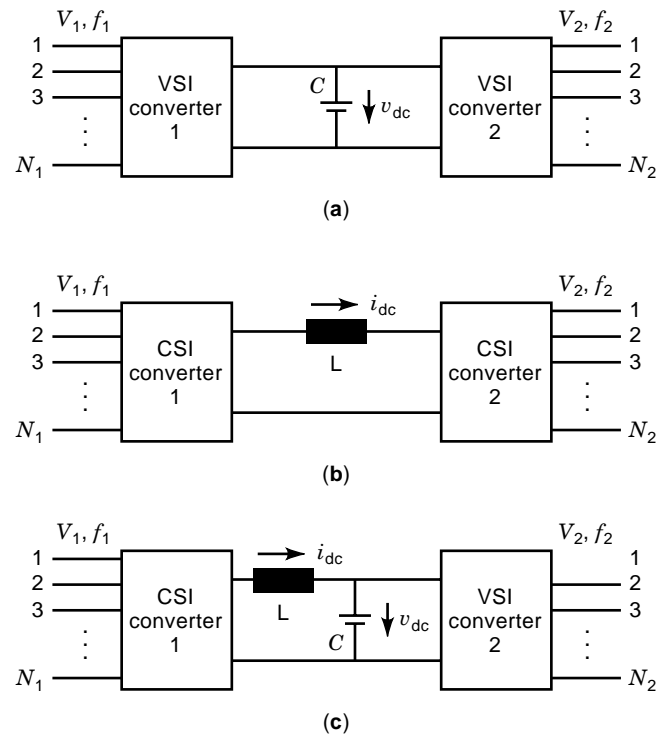


Figure 2. Intermediate dc-link systems. Voltage-source converters (a) use capacitors. Current-source converters (b) use inductors to store dc energy. Mixed-dc-link systems (c) combine voltage- and current-source dc links.

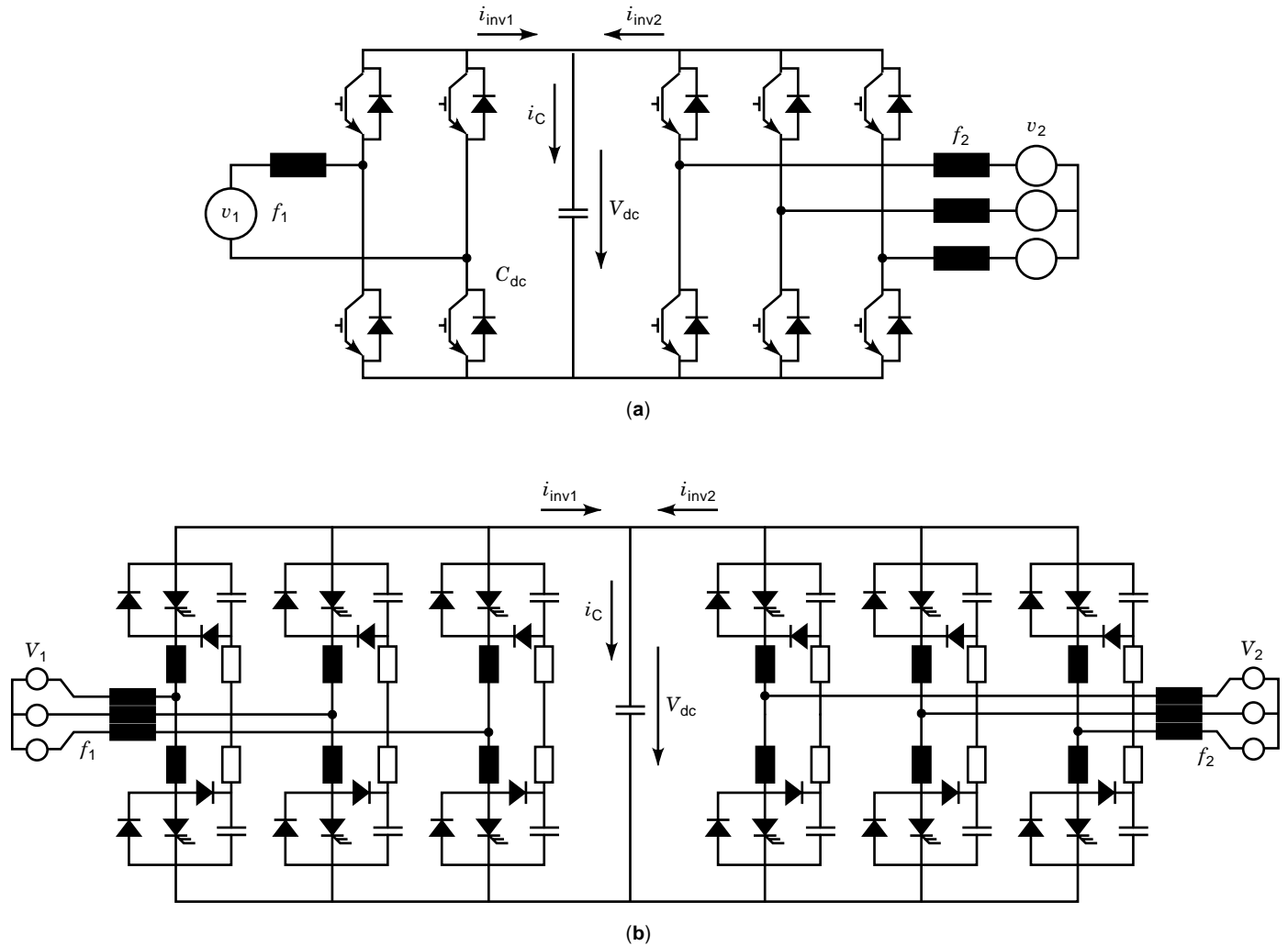


Figure 3. Voltage-source converters. A single-phase to three-phase IGBT converter (a) requires no snubbers. At high power levels, three-phase GTO converters with snubbers (b) are used.

peak power during turn-off. However, recent developments indicate that operation of a GTO without a snubber is possible when special low-inductance gate drivers are used (9).

The converter shown in Fig. 3(a) is a bidirectional converter that uses IGBT devices. Figure 3(a) also illustrates how single-phase ac power is converted to three-phase ac power. The circuit diagram of a high-power, three-phase ac-to-ac converter based on GTO devices is shown in Fig. 3(b). Snubber circuits consisting of capacitors, inductors, and resistors are used to limit the switching stress on the GTO devices.

Each voltage source inverter (VSI) shown in Fig. 3 controls ac voltage amplitude and frequency independently of each other by pulse-width modulation (PWM) control. Several PWM methods have been developed, such as sine-triangle PWM and space-vector modulation (6). With PWM, the peak amplitude of the sinusoidal ac phase voltage of a three-phase system can be controlled linear as a function of the voltage command up to 50% of the dc-link voltage.

The power flow in and out the dc link must be controlled precisely because, in practice, the energy storage capacity of the dc link is limited. The smaller the dc capacitor, the higher the control bandwidth should be for regulating the dc voltage.

For stability reasons, cascaded *current regulation* loops are added to the PWM controller. This current regulator decouples the ac filter inductors from the dc-link capacitor. As a result, the current-regulated, voltage-source converter with the output filter inductors acts as a controlled current source. Hence, this type of control is called current-regulated PWM (CRPWM) control. The complete ac-to-ac control configuration is detailed in Fig. 4.

When the converter is connected to an ac source, the PWM modulator needs to synchronize its pattern with the ac source voltage. Whenever the converter is feeding a passive load, this synchronization is not necessary. A phase-locked loop (PLL) is often used to provide synchronization signals to the zero voltage crossings of the ac source voltages. In Fig. 4, it is assumed that converter 1 (input side) is connected to a three-phase ac system whereas converter 2 (output) is feeding a passive load.

The current commands of converter 2 are generated by an outer control loop that controls external (load side) variables, such as the output ac voltages or, in the case of a machine drive, the torque of an ac machine. The current commands of the input side converter are set by the dc-link voltage regula-

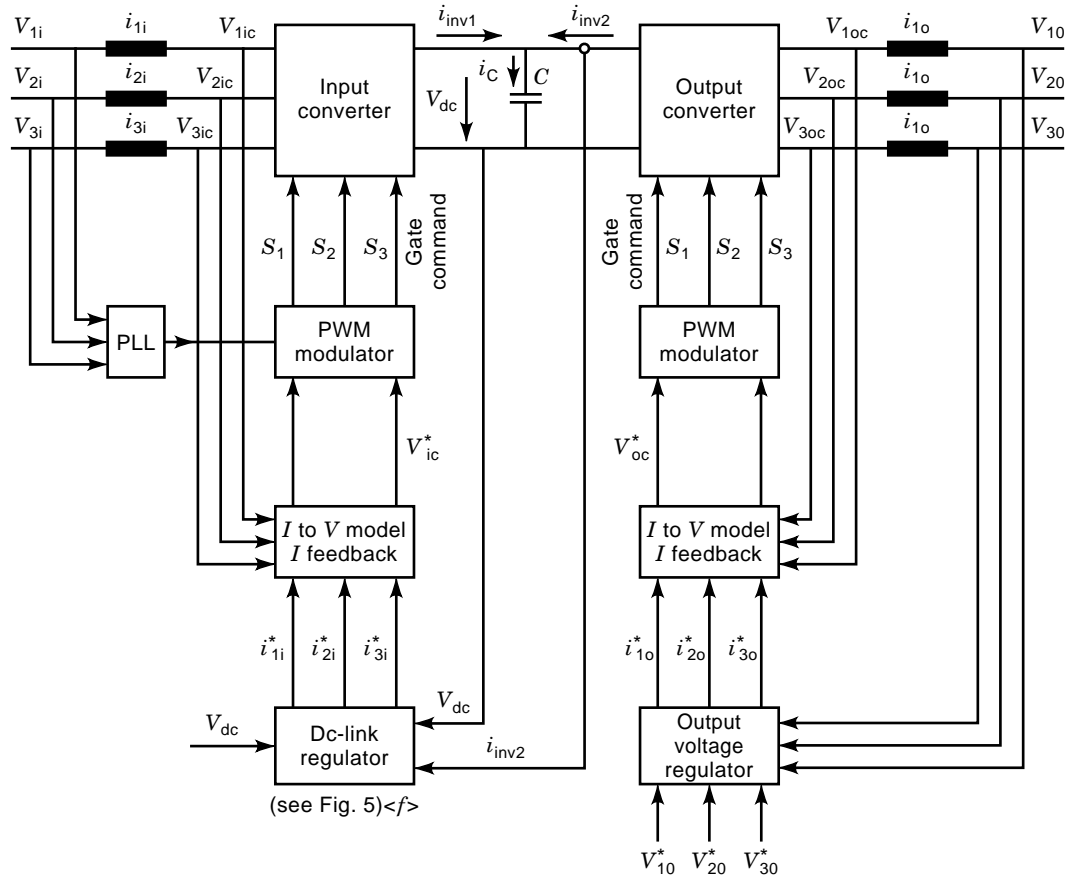


Figure 4. Control diagram of a voltage-source, dc-link, ac-to-ac converter.

tor, shown in Fig. 5. The required dc-link current for converter 1 is calculated by two control loops. A fast acting feedforward control loop is implemented using the dc capacitor state equation:

$$C dv_{dc}/dt = i_{inv1} - i_{inv2}$$

The dc-link current i_{inv2} of the load-side converter is measured or calculated from the current signals and the converter state of converter 2. Adding this dc current i_{inv2} to the derivative of the dc voltage command v_{dc}^* leads to a command value for the dc current i_{inv1}^* . Obviously, this feedforward loop is extremely fast and is limited only by the sample delays of the converter

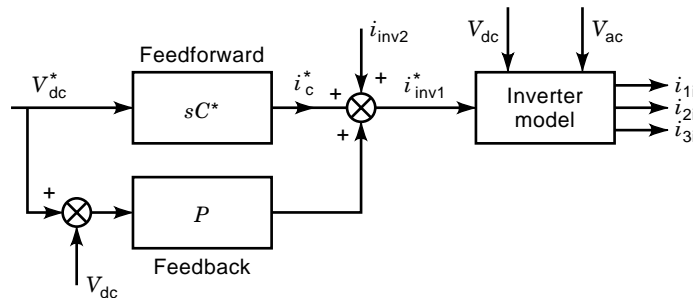


Figure 5. Control block diagram to control the dc-link voltage using feedforward and feedback control.

controller. However, the exact knowledge of the dc capacitor value C is essential to guarantee high accuracy in the feedforward control loop. To compensate for possible detuning errors of C^* in the feedforward path, a feedback loop is added. In this case, to eliminate steady-state error, the feedback can be a simple proportional gain because the capacitor represents a first-order system, that is, an integrator.

Next, the three-phase current commands for the ac converter 1 are computed by using the equality between ac real power and dc power. In addition, reactive components may be superimposed on the input converter current commands if the converter needs to provide reactive power compensation.

Fast acting feedforward control of the dc-link voltage is necessary when the dc-link voltage changes rapidly, for example, when few dc film type capacitors are used. When the dc capacitor is large, for example, when electrolytic capacitors are used, a simple voltage feedback control loop is adequate to regulate v_{dc} .

Typical applications are adjustable speed drives (ASDs) for ac asynchronous and synchronous machines (Fig. 6) that provide bidirectional power flow for centrifuges, elevators, and rolling mills. The output converter controls the torque of the ac machine. The input converter (converter 1 in Fig. 6) is connected to the ac supply and controls the dc bus voltage while maintaining unity power factor at the ac supply.

Modern ac traction systems (Fig. 7) fed by a single-phase, medium-voltage, ac catenary use a single-phase input converter that provides unity power factor at the ac input and

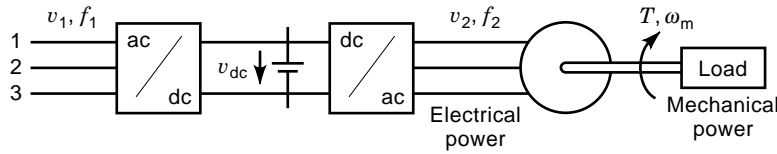


Figure 6. Adjustable speed drives use ac-to-ac converters when power flow can reverse.

regulates the dc voltage in the locomotive. Three-phase output inverters control the torque of the traction machines. While braking, the locomotive feeds energy back into the single-phase ac grid (regenerative braking). In new high-speed locomotives, the trend is to distribute the traction drives over many axles to improve weight distribution and enhance the maximum achievable tractive effort under all weather conditions.

In the United States, large 100 MVA ac-to-ac VSI systems have been commissioned in utility systems to support the voltage of long transmission lines. These static compensators regulate voltage by compensating reactive power demand independent of fast load changes (flicker). Figure 8 illustrates a utility type compensator which compensates for reactive power by injecting current into the ac line with a “shunt” converter. This shunt converter also regulates the dc voltage by drawing current in phase with the ac voltage. An ac voltage in series with the line is induced by using a “boost” converter to compensate for source voltage dips. The boost converter may need to exchange active power with the dc bus when the induced voltage is in phase with the load current. The complete system provides a high degree of controllability against ac voltage sags and swells caused by dynamic reactive load changes and sags in the source voltages. This unified power factor compensator (UPFC) will become a key device in flexible ac transmission systems (FACTS) when voltage and power flow control become more decentralized.

Ac-to-ac Converters with Current-Source dc Link

A simple current-source ac-to-ac converter is obtained by connecting two multiphase thyristor or GTO converter bridges in

series with a dc-link inductor as illustrated in Fig. 9. A typical configuration consists of single-phase or three-phase thyristor converter bridges that can operate in rectifier mode or inverter mode. Figure 9(a) shows a single-phase to three-phase current-source converter based on line-commutated thyristor circuits (snubbers not shown). Figure 9(b) shows a three-phase to three-phase current-source converter. The converter on the left is a forced-commutated thyristor converter whereas the converter on the right uses GTOs to control the output current i_{ac} .

Note that the sign of the circulating dc link current i_{dc} of the circuits shown in Fig. 9 has to be positive. The amplitude of the dc-link current should be equal to the maximum line current of either ac system. When thyristors without commutation circuits are used [Fig. 9(a)], the turn-on instant of the thyristors always lags with respect to the ac supply voltage because thyristors cannot turn off current. As a result, the ac supply needs to provide this reactive power. Hence, as illustrated in the vector diagram of Fig. 9(a), the line-commutated thyristor converter operates only in two quadrants of the voltage-current vector diagram. However, the dc voltage of each converter can be controlled continuously proportional to $\cos \alpha$ by controlling the firing angle α . This firing angle α is measured between the instant where the voltage across the thyristor becomes positive (the device would conduct if a diode were used instead) and the turn-on instant of the thyristor (determined by the gate driver control). Negative dc voltages are obtained whenever the firing angle exceeds 90° . As a result, the converter of Fig. 8(a) is bidirectional for real power as long as i_{dc} remains constant. Assuming no losses in the dc

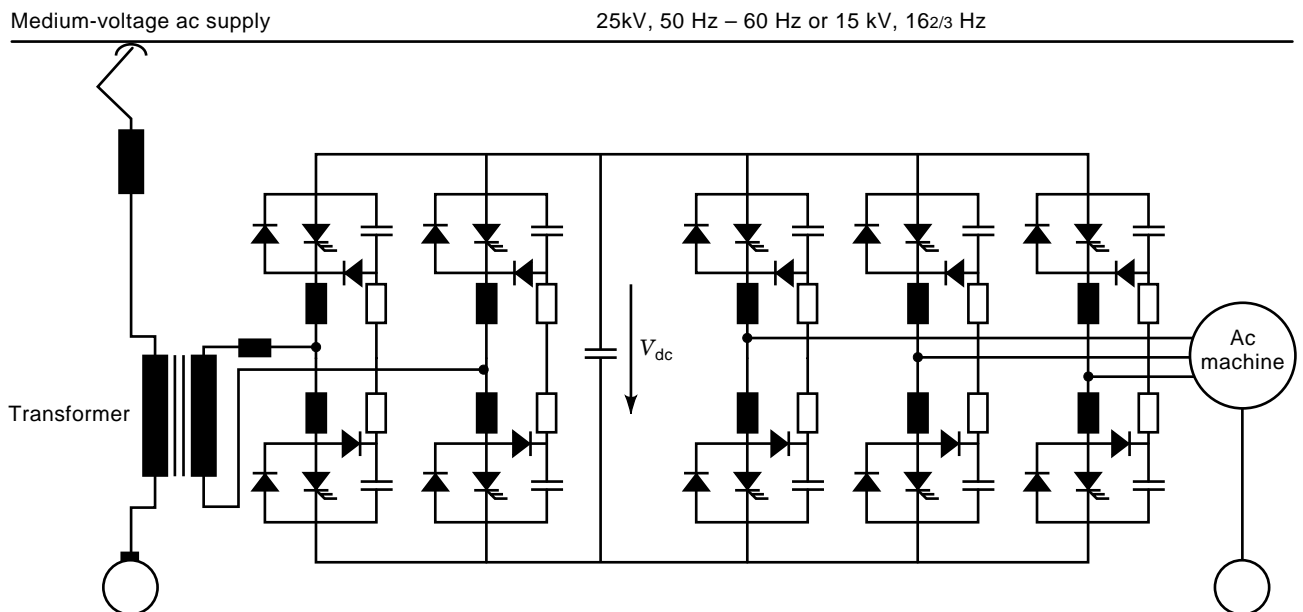


Figure 7. Diagram of a modern traction drive fed by ac catenary.

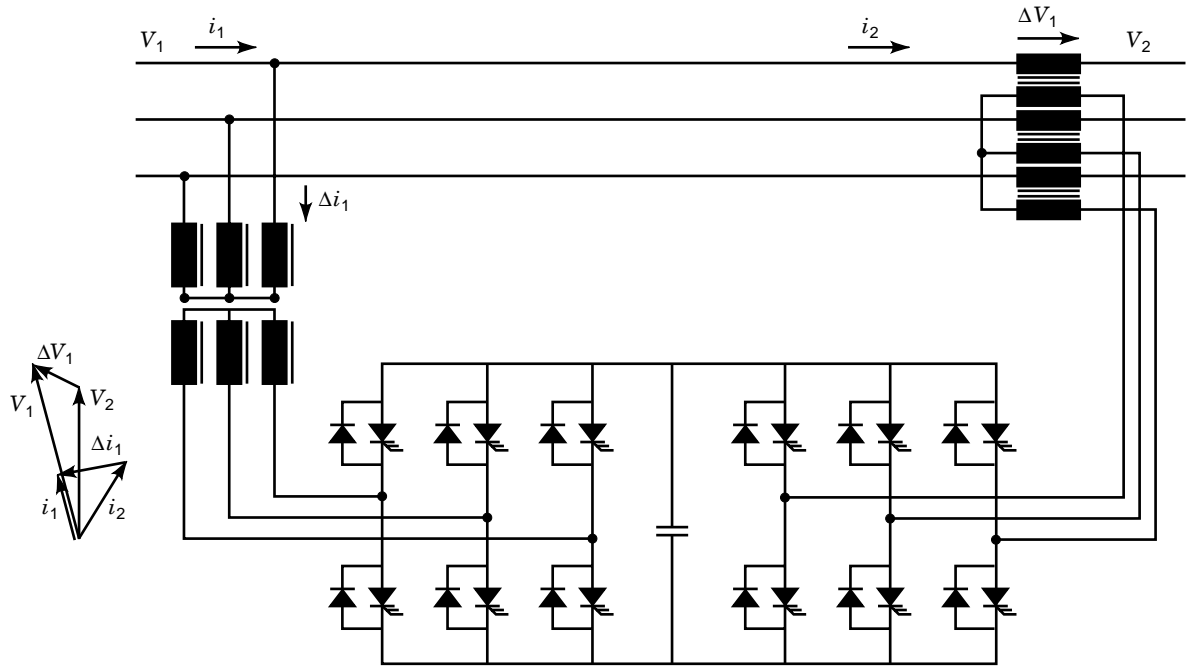


Figure 8. The unified power factor compensator uses a voltage-source ac-to-ac converter to compensate for input line-voltage disturbances and load-side reactive power (including harmonics).

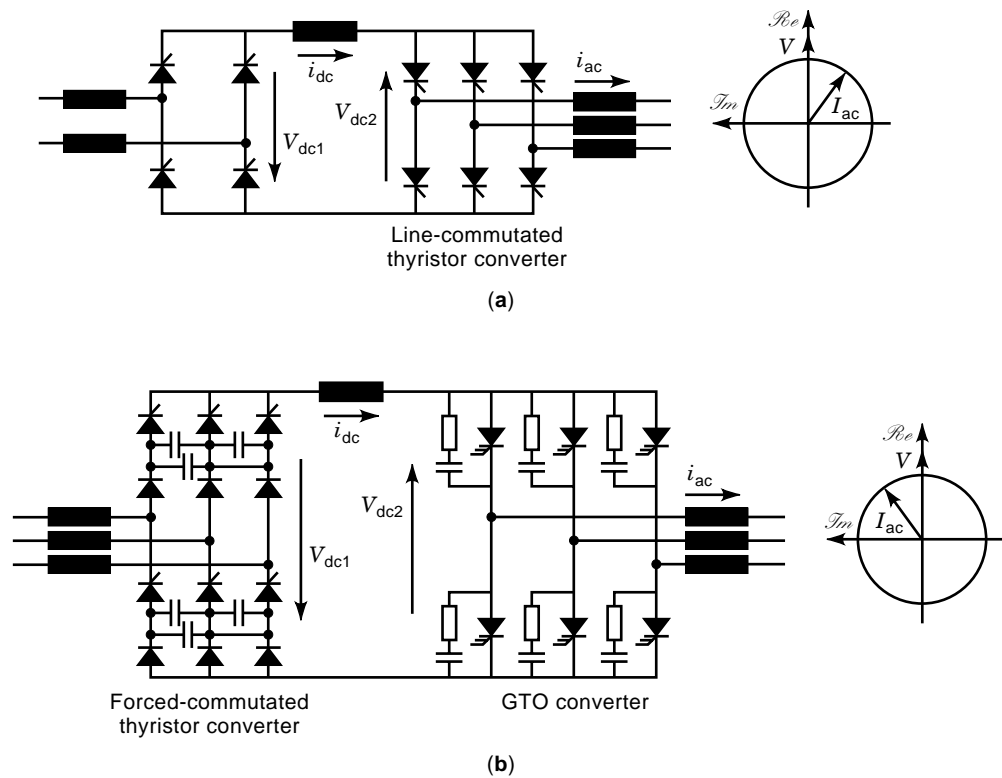


Figure 9. Current-source converters. (a) A single-phase to three-phase converter with line-commutated thyristor bridges operates only in two quadrants (lagging power). (b) The three-phase forced-commutated thyristor converter or the GTO converter operates in all four quadrants.

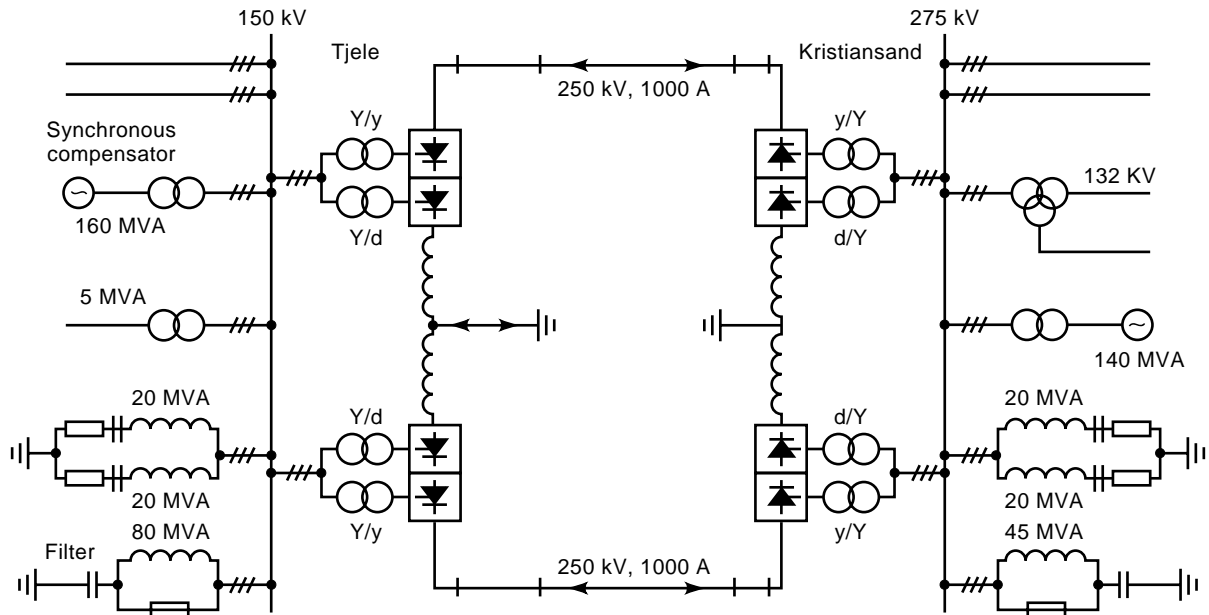


Figure 10. The Skagerrak HVDC system is a classical example of a high-power (500 MW), thyristor-controlled, current-source system (8).

link and neglecting commutation overlap delays, the steady-state voltage v_{dc1} is opposite to the voltage v_{dc2} . In other words, the sum of the firing angles $\alpha_1 + \alpha_2$ equals 180° .

Most dc-link ac-to-ac converters use thyristor devices because thyristors are relatively inexpensive and are available over a wide voltage range (600 V to 9000 V) and current range (10 A to 6000 A). Thyristor converters can be realized from 10 kVA to 10 MVA. In this case, rotating or static VAR compensators are needed because thyristor converters draw lagging currents whenever the firing angle varies between 0° and 180° . Modern current-source converters use GTOs or forced-commutated thyristor circuits [Fig. 9(b)] and turn off current at any instant, enabling full control of the ac line displacement factor $\cos \varphi$.

All current-source converters need capacitive snubbers across the devices because, in practice, ac systems are inductive. This inductive characteristic would induce large voltage spikes across the devices whenever current is turned off. In some cases large ac capacitors are placed at the converter terminals to decouple the ac line inductance and to clamp the voltages across the devices. In addition, these capacitors filter the higher switching harmonics of the converter.

Current source ac-to-ac converters have been used since 1972 in high voltage dc (HVDC) transmission systems and interties. Figure 10 illustrates the HVDC system built between Norway and Denmark in 1976 (8). On each side large current-source converters using line commutated thyristor bridges were installed. Each switch (also called a valve) of the thyristor bridge consists of hundreds of thyristors connected in series. The three-phase voltages of each thyristor converter are phase-shifted by using “wye-delta” and “wye-wye” transformers. This leads to a higher switching pulse rate at the ac side, canceling low frequency harmonics in the ac line current. This harmonic cancellation makes the ac line filters smaller and less expensive. Notice the VAR compensation

systems shown in Fig. 10. They are essential to compensate for the reactive power demand of the two-quadrant thyristor converters which can draw only lagging reactive power.

Today, very large HVDC systems are being built that reach power levels of 2 GW. Transport of electrical power over long distances (more than 500 km) is possible via HVDC lines because the reactance of the line does not cause voltage drops in dc transmission systems. Actually, the dc line reactance adds to the dc-link reactance and helps maintain a constant dc current. Most HVDC systems operate at voltages higher than 300 kV. To block this high voltage, many hundreds of thyristors (up to 500) are connected in series to produce one switch. A complete system may consist of more than 30,000 thyristors. To provide galvanic isolation, these thyristors are optically triggered. To this end, glass fibers are used which carry the light of powerful infrared lasers toward the thyristors.

Current-source converters have been used to recover rotor-slip energy in high-power wound-rotor induction motor drives. Figure 11 illustrates the so-called *Kramer drive*. The converter is handling only the slip energy that is small compared to the total rating of the machine when speed is varied over a narrow range around the synchronous speed.

Despite their simple operating principles, current-source converters have some drawbacks compared with voltage-source converters, such as slower response time for current regulation, higher no-load losses and, in case of thyristors, poor power factor. In the case of back-to-back interties, the dc-link inductor is costly and bulky. Furthermore, current-source converters need symmetrical blocking power devices which have higher conduction losses than asymmetrical devices. Consequently, as the current turn-off capability of GTOs steadily improves, voltage-source ac-to-ac converters are starting to replace current-link converters (except in HVDC systems).

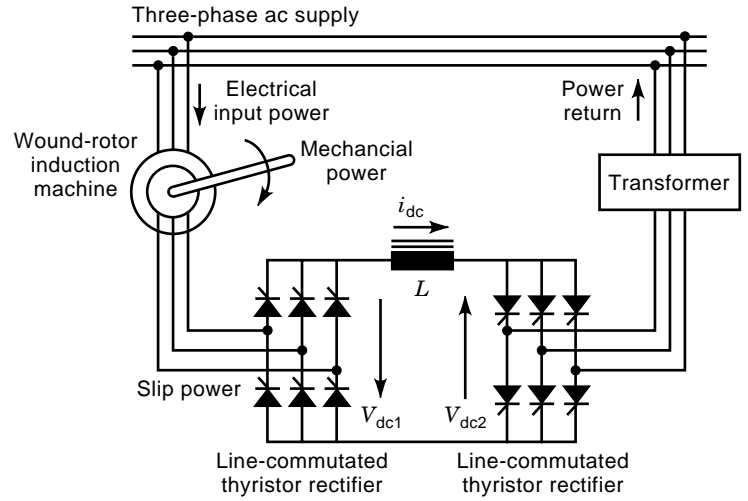


Figure 11. A Kramer drive uses a current-source ac-to-ac converter to recover the slip energy of a wound-rotor induction motor enabling an efficient speed control mechanism.

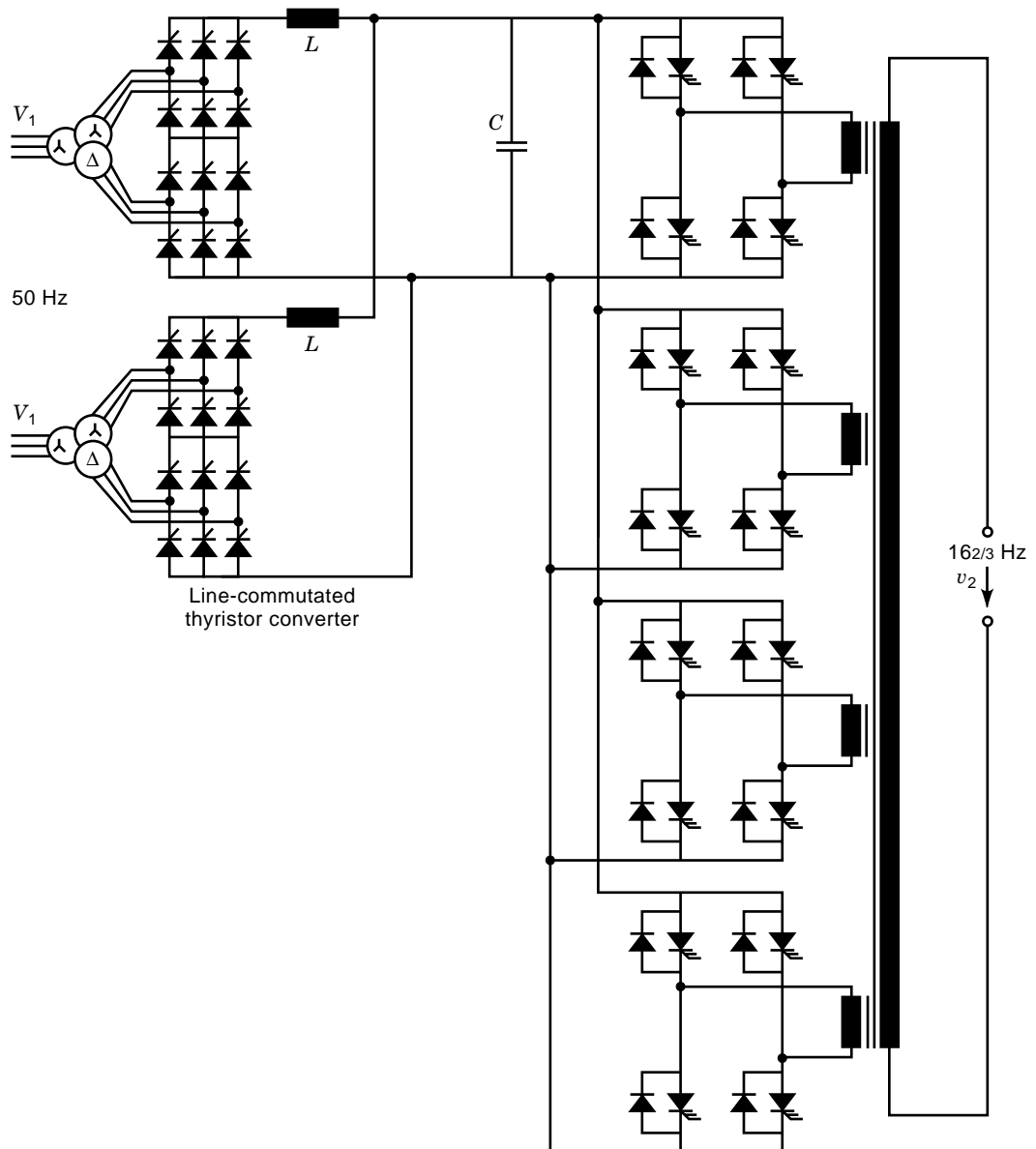


Figure 12. Mixed dc link system used as an intertie between the three-phase, 50 Hz power grid and the single-phase $16\frac{2}{3}$ Hz railway power grid in Germany.

Ac-to-ac Converters with Voltage-Source and Current-Source dc Link

Very high-power 100 MVA mixed-voltage-source and current-source converters have been built to link 50 Hz power systems to the low-frequency 16 $\frac{2}{3}$ Hz train network in Germany (Fig. 12). These converters typically consist of multiple current-source and voltage-source converters linked together (9). To minimize cost, the three-phase, line-side converter is typically based on standard 12-pulse thyristor rectifier bridges, as shown in Fig. 9(a). However, to allow the dc current to reverse sign, additional thyristor converter bridges are placed in anti-parallel. The sign reversal of the dc-link current is needed when power flow reverses because the voltage-source output converter requires a positive dc bus voltage. The voltage-source, railway-side inverter consists of many inverters placed in parallel. Each inverter switch is realized by connecting multiple GTOs in series. Each GTO inverter module feeds

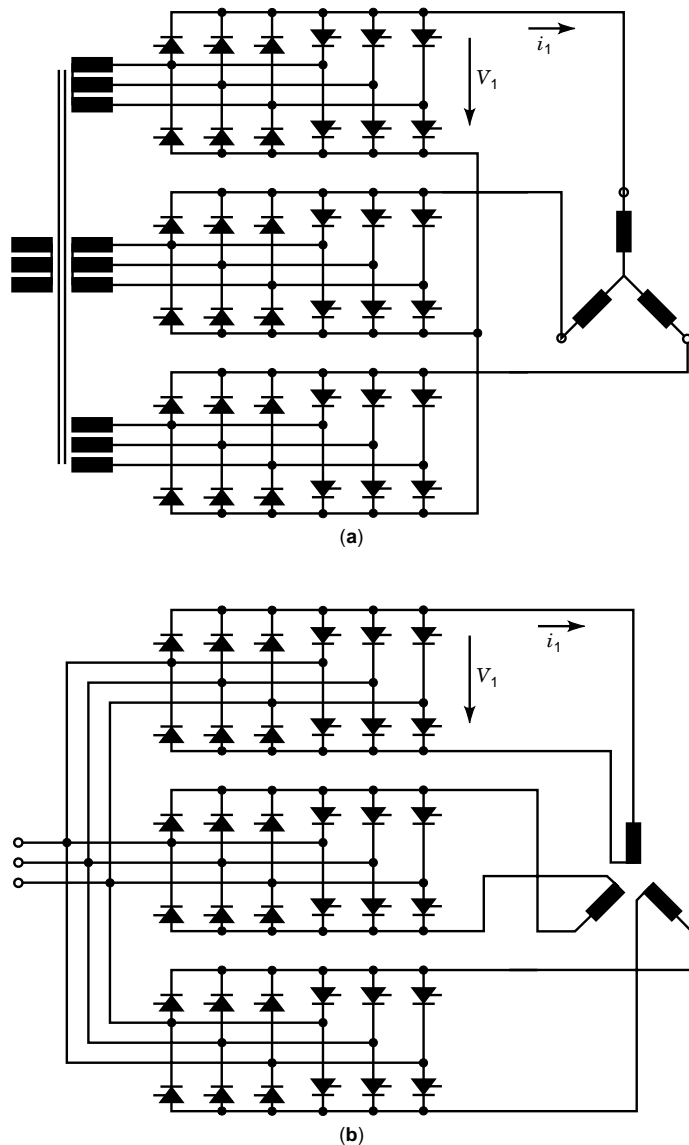


Figure 13. Cycloconverters for three-phase systems. (a) Loads connected in a “wye” configuration require isolation between each set of three phases. (b) The isolation transformer is avoided when using an open, delta-connected load.

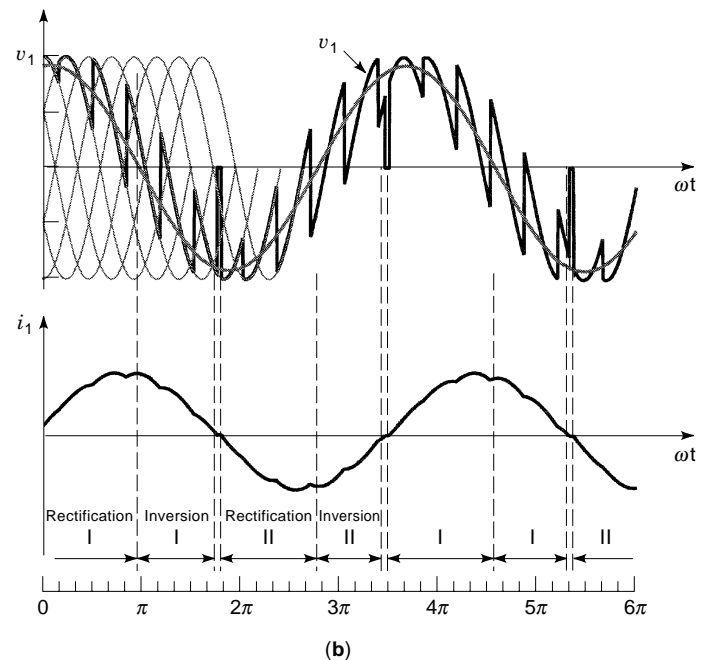
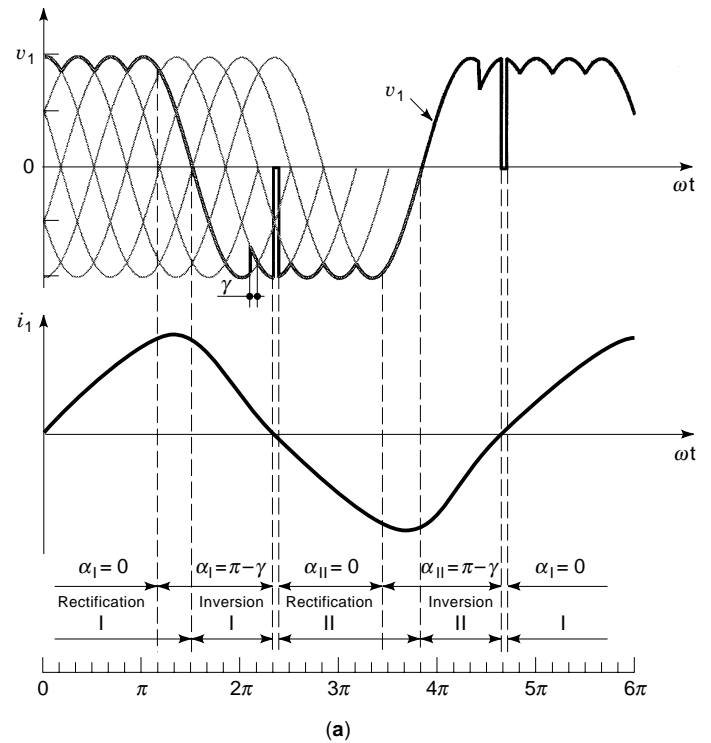


Figure 14. (a) Discrete control of the firing angle leads to trapezoidal output voltage waveforms which can be frequency controlled. (b) Continuous control of the firing angle enables voltage and frequency control of the cycloconverter.

power into a primary winding of the railway-side transformer. The secondary windings of this transformer are placed in series. Switching patterns of each GTO inverter module are shifted to generate a multilevel output pattern that contains few harmonics. This voltage-control method is called harmonic cancellation and features excellent waveform quality, which avoids installing expensive filters on the ac railway

line. These mixed-source inerties are replacing older cycloconverter systems (see later) because external power factor compensators and filters are reduced or eliminated.

CYCLOCONVERTER

The topologies shown in Fig. 9 can be controlled so that little or no circulating dc-link current flows. The firing angle α of the converters can be controlled to vary proportionally with time, leading to a sinusoidal output voltage variation. As a result, a converter is created that has no intermediate dc link because the dc link itself becomes a low-frequency ac output that changes cyclically. This cycloconverter technique was developed in Germany with mercury arc tubes in the early 1930s to produce a single-phase 16 $\frac{2}{3}$ Hz fixed-frequency power system for the railways.

By repeating this configuration for each phase of the second ac grid, a multiphase cycloconverter system is realized. Three-phase versions of cycloconverters with no circulating current are shown in Fig. 13.

Cycloconverters have mostly been built using line-commutated thyristor circuits. Hence, a three-phase ac-to-ac cycloconverter requires 18 thyristors when using half-bridge converters or 36 thyristors when using full-bridge converters. Control of a cycloconverter is illustrated in Fig. 14(a) and 14(b). A simple control method [Fig. 14(a)] is realized when the firing angle α alternates only between the maximum values of 0° and 180° . Each thyristor bridge operates in the rectifier mode when the load current has the same sign as the voltage. When the firing angle α shifts to 180° , a negative voltage is produced by the rectifier. As long as the current remains positive, the converter feeds power back in the input ac source and operates in the inverter mode. As soon as the current crosses zero, the converter is turned off and the second thyristor rectifier is turned on with a firing angle at 0° , repeating the cycle. Figure 14(a) illustrates that a trapezoidal waveform is obtained at a frequency lower than the voltage input ac line frequency. The frequency can be changed only in discrete steps because each period consists of a fixed number of 60° intervals. Variable frequency control with reasonable resolution is realized only when the frequency is very low (below one-third of the fundamental frequency). Voltage control is not possible with this simple discrete control of the firing angle. To overcome this problem, the cycloconverter firing angle α can be programmed to vary proportionally with

time to change the output voltage sinusoidally and control output frequency as shown in Fig. 14(b).

Both methods (discrete and continuous firing-angle control) have found application in low-speed, high-power motor drives used in rolling mills and direct-drive (without gearbox) cement mills. Cycloconverters have been used to recover the rotor slip energy in large doubly fed, wound-rotor induction motor drives. Similar to the CSI in the static Kramer drive, the cycloconverter makes speed control possible. This type of drive is also called the static *Scherbius drive* (7). A Scherbius drive is shown in Fig. 15 which uses half-bridge thyristor converters. Whenever the cycloconverter extracts power from the wound rotor, the machine operates at lower speed (subsynchronous mode). Feeding ac power to the rotor forces the machine to run above the synchronous speed (supersynchronous). The cycloconverter has to be rated at 50% of the rated machine power to enable a $\pm 50\%$ speed variation around the synchronous speed of the drive. In many high-power application, such as pumps, blowers, and wind turbines, this speed variation is adequate because the power demand of these loads varies with the cube of the speed.

Cycloconverters are also used to generate 400 Hz ac power onboard airplanes. A high-speed synchronous generator, driven by the jet engine, delivers three-phase power at high frequency (2 kHz to 3 kHz). Voltage and frequency for the 400 Hz system are controlled by the cycloconverter firing angle.

In general, cycloconverters produce relatively sinusoidal voltage waveforms. However, their disadvantage is the relative high reactive power demand from the input ac supply, especially when the output voltage is low. Furthermore, low-frequency harmonics are generated into the ac source. Bulky filters often need to be installed to compensate for these undesirable effects.

MATRIX CONVERTER

The *matrix converter*, or *Venturini converter*, consists of a matrix of bidirectional current and voltage power devices connecting two multiphase ac sources together. Similar to the cycloconverter, the matrix converter has no internal dc link, thus avoiding bulky energy storage devices. The current and voltage waveforms of both ac systems can be controlled arbitrarily by using PWM control. The matrix converter controls the active power flow between both systems and is capable of individually controlling the reactive power of each ac system.

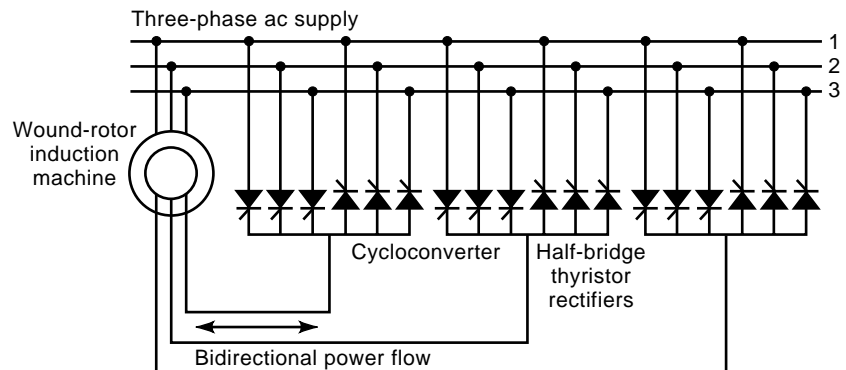


Figure 15. The Scherbius drive uses a cycloconverter to recover the slip energy of a wound-rotor induction machine to provide efficient speed control.

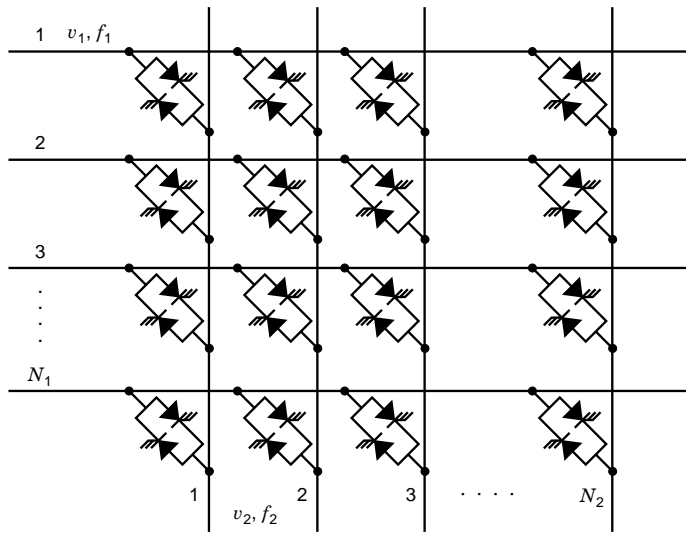


Figure 16. General block diagram of a matrix converter.

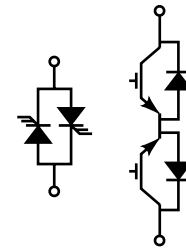


Figure 17. Practical implementation of bidirectional current and voltage switches.

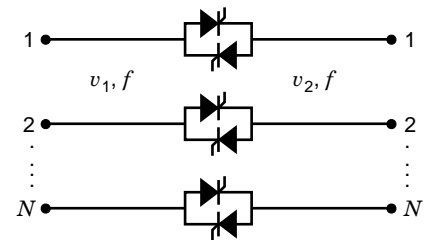


Figure 18. The ac voltage regulator uses antiparallel thyristor switches to regulate output-voltage amplitude.

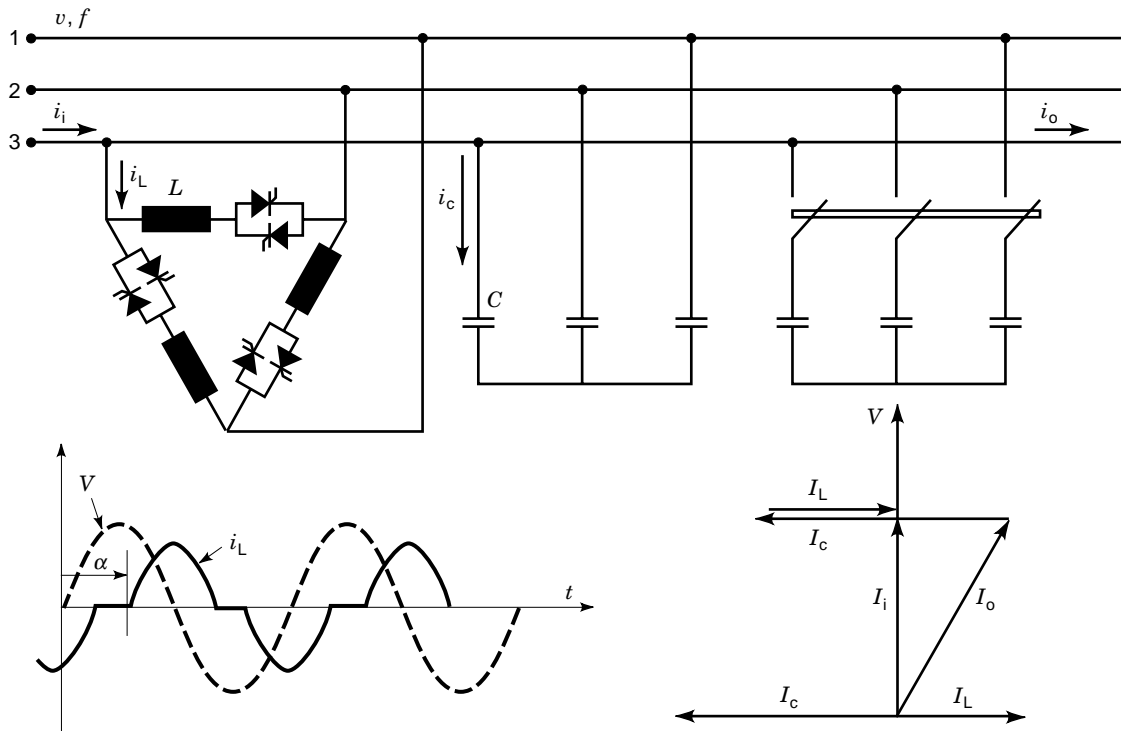


Figure 19. Typical configuration for a static VAR controller. The ac voltage regulator controls the reactive power that flows into the inductors (variable reactance). Fixed or switched capacitors deliver reactive power to offset the reactive power required by the load and the variable reactance.

As illustrated in Fig. 16, each phase of the input ac source can be connected to a phase of the output ac system. It is important that, while turning current on or off, the devices are not connecting solely inductive or capacitive systems together which would lead to uncontrollable current or voltage spikes. As a result, one ac system should have an inductive or current-source behavior while the other behaves as a capacitive or voltage source. As an example, in Fig. 16, the input side is made inductive by adding inductors to the ac supply. The output is made capacitive by adding filter capacitors.

The size of the passive filter components depends on the switching frequency of the converter. The higher the switching frequency, the smaller these components become. At frequencies above 10 kHz, one can consider using the inherent impedance of the ac grid at the input side because the ac supply is typically inductive due to line and transformer leakage inductances.

The matrix converter requires bidirectional power devices that are not available now (as single integrated devices). As a result, *bidirectional power switches* are constructed by connecting symmetrical blocking devices (e.g., GTOs) antiparallel or by connecting asymmetrical devices in series with reverse voltage-blocking diodes (e.g., transistors) as illustrated in Fig. 17. As a result, a three-phase to three-phase matrix converter needs at least 18 power switches whereas a dc-link ac-to-ac converter needs only 12 devices. This increase in the number of components, combined with relatively complex control, makes the matrix converter less attractive in industrial applications. It is anticipated that the matrix converter will gain interest when fully integrated bidirectional ac switches become available, especially for applications that require a high level of integration, for example, integrated into the frame of an electrical machine to reduce volume. Matrix converters may also become attractive in high-temperature applications that require high reliability. Indeed, the maximum allowable ambient temperature of dc-link capacitors used in voltage-source converters is often a limiting factor that prevents voltage-source converters from operating beyond 75°C.

STATIC AC VOLTAGE REGULATORS AND PHASE SHIFTERS

The ac voltage regulator, shown in Fig. 18, is a simplified matrix converter that has only one bidirectional ac switch per phase. Hence, the ac voltage regulator links two ac systems together that have the same frequency and number of phases. In most applications, this low-cost ac-to-ac converter uses line-commutated thyristors. The thyristor firing angle α is controlled to regulate the amplitude of the output ac voltage.

The ac voltage regulator is used in many well-known applications in single-phase circuits, such as light dimmers, speed control of universal dc motors (fed by the ac supply) used in vacuum cleaners, kitchen appliances, blowers, and hand held power tools (saws, drills). Low power factor and relatively high harmonic content of the current waveforms are negative aspects of this simple voltage converter. In high-power applications, three-phase ac voltage regulators are often used as “soft-starters” to limit the in-rush currents during the start-up of induction motors. As soon as the induction motor reaches its rated operating point, the solid-state soft-starter

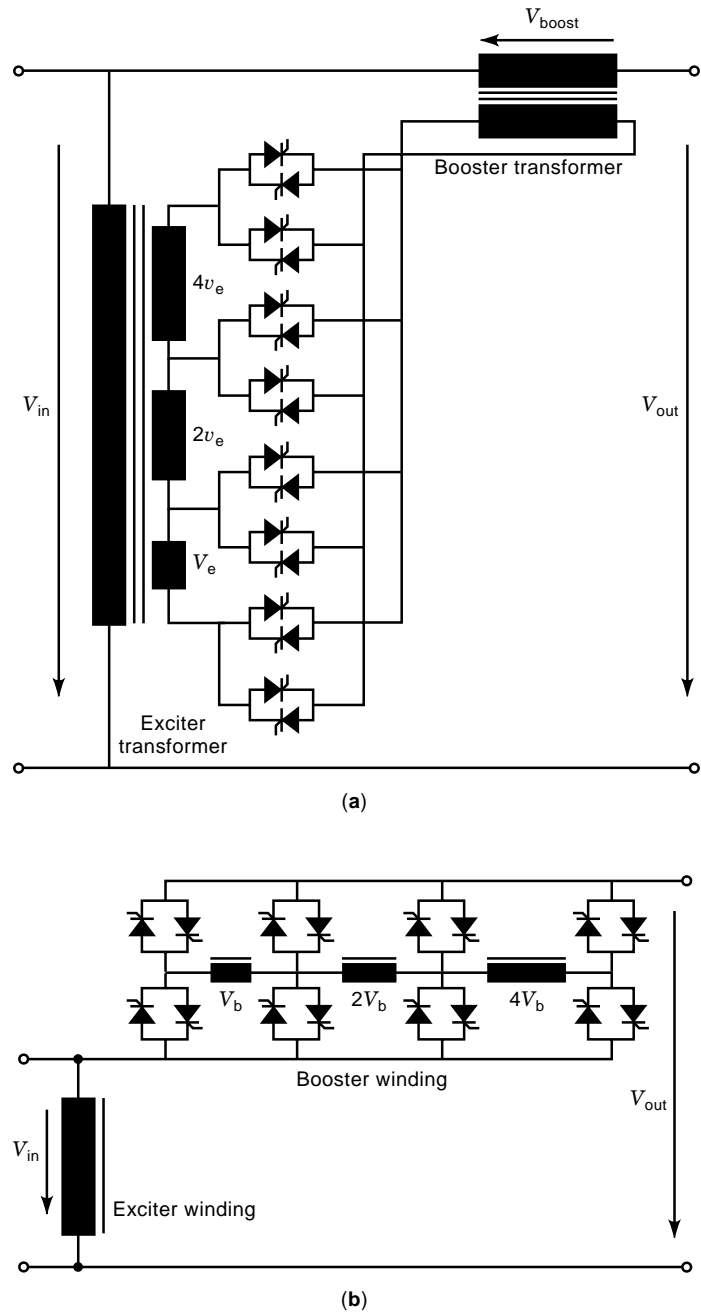


Figure 20. Solid-state, tap changer, single-phase diagram. (a) The booster transformer injects a voltage in series with the ac line voltage and provides isolation for the power electronic circuit. (b) The auto-transformer concept may offer a lower cost alternative. Both circuits increase and reduce the output line voltage.

is bypassed by a mechanical switch to eliminate conduction losses in the thyristors.

Static VAR compensators are another important application for ac voltage regulators (see Fig. 19). Continuous control of reactive power is achieved by controlling the current in the inductors by phase control of the thyristors. The inductor current is lagging and is offset by the capacitive leading current of fixed or switchable capacitor banks.

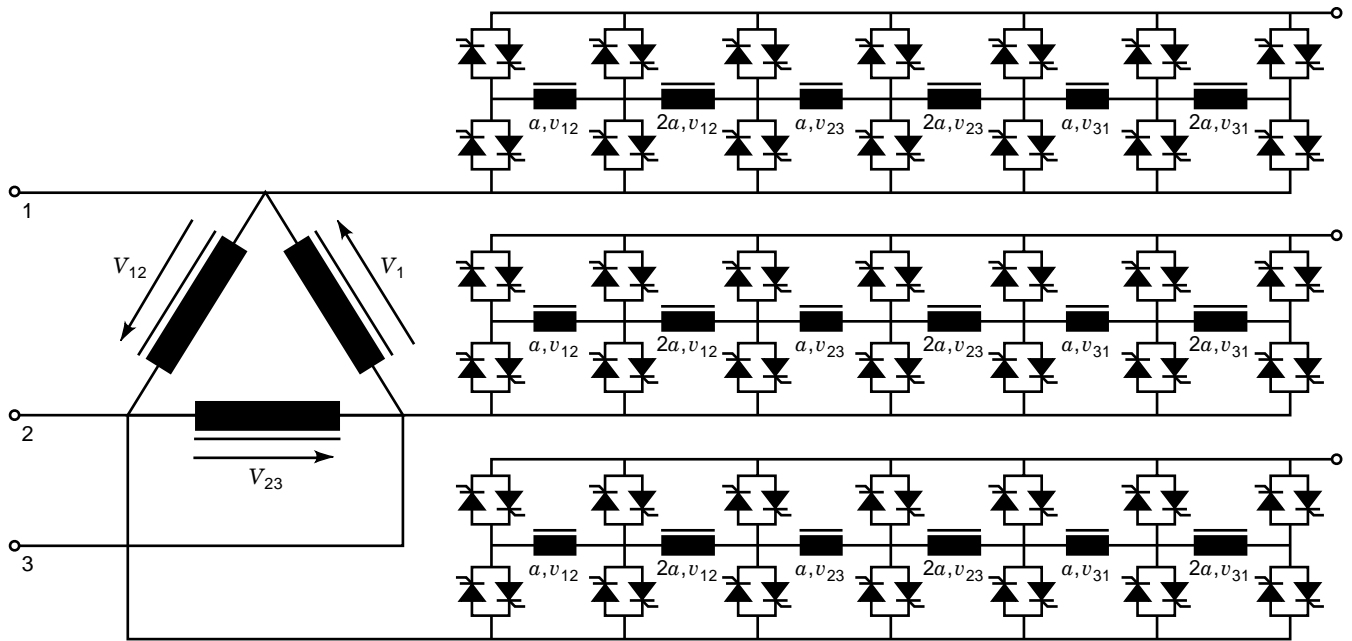


Figure 21. Solid-state voltage regulator and phase shifter.

Accurate control of ac voltages at high power without generating harmonics or creating excessive lagging power is realized by using solid-state transformer tap changers or “load tap changers.” Figure 20 illustrates two tap changer concepts (single-phase diagram). The system shown in Fig. 20(a) uses a separate boost transformer to induce a voltage in series with the ac line. Although the boost transformer can be avoided, as illustrated by the autotransformer circuit shown in Fig. 20(b), it permits isolating the power electronic devices from the medium-voltage ac system, and makes the gate drive circuitry and sensors less expensive. The secondary windings of the exciter transformer are designed to produce voltages that form a binary sequence (1, 2, 4, 8, . . .) of the smallest voltage step that needs to be regulated. Using a matrix of bidirectional thyristor switches, it is possible to generate any (discrete) positive and negative voltage variation in phase with the input voltage. Normally, the thyristors are completely turned on or off, that is, no ac voltage regulation is performed by controlling the firing angle. As a result, the output voltage v_{out} contains no harmonics generated by the regulator. In future applications, the solid-state tap changer will be used as a fast-output voltage regulator whenever the line voltage v_{in} sags. In this case, the circuit can be simplified and fewer thyristors are needed because the secondary voltages are always added and not subtracted from the input voltage.

The solid state tap changer concept can be expanded to create *solid-state phase shifters* that combine phase-shift control and voltage regulation. Indeed, in three-phase systems it is possible to connect secondary windings of other phases of the transformer (which are shifted by 120°) in series with the primary voltage (see Fig. 21). The voltage produced per phase can be any discrete combination of the secondary voltages induced in the windings belonging to that phase. Unbalanced voltages and phase shifts between

ac systems can be completely compensated for with these solid-state phase shifters.

BIBLIOGRAPHY

1. N. Mohan, T. Undeland, and W. Robbins, *Power Electronics*, 2nd ed., New York: Wiley, 1995.
2. J. Murphy and F. Turnbull, *Power Electronic Control of AC Motors*, Pergamon, 1988.
3. J. Van Wyk, Power electronic converters for drives, in B. K. Bose (ed.), *Power Electronics and Variable Frequency Drives*, Piscataway, NJ, IEEE Press, 1997.
4. J. Kassakian, M. Schlecht, and G. Verghese, *Principles of Power Electronics*, Reading, MA: Addison-Wesley, 1991.
5. G. Seguier and F. Labrique, *Power Electronic Converters*, Springer Verlag, 1993.
6. J. Holtz, PWM modulation for electronic power conversion, in B. K. Bose (ed.), *Power Electronics and Variable Frequency Drives*, IEEE Press, 1997.
7. B. K. Bose, *Adjustable Speed AC Drive Systems*, Piscataway, NJ, IEEE Press, 1980.
8. L. Haglöf and B. Hammarlund, HGÜ-Anlage Skagerrak-die längste Starkstrom-Seekabelverbindung der Welt, *Asea Zeitschrift*, **25** (3): 51–59, 1980.
9. P. Steimer, H. Grüning, and T. Werninger, IGCT—a new emerging technology for high power, low cost inverters, *Proc. IEEE-IAS Annual Meeting*, 1997, pp. 1592–1599.

RIK W. DE DONCKER
 Institute for Power Electronics and
 Electrical Drives (of the Aachen
 University of Technology)