## Chapter 8 Module Structuring

While task structuring requires a designer to search data/control flow diagrams for threads of control, module structuring requires the designer to take an orthogonal view, searching the data/control flow diagrams instead for reasons to combine transformations and data stores based on information-hiding criteria. This goal is accomplished by applying Module Structuring Knowledge, which is organized as seven, distinct, decision-making processes, shown in Figure 27. The main information used to structure modules consists of a fully-classified data/control flow diagram, as described in Chapter 4, and the state of the evolving, software design. The main information output from the module-structuring phase consists of a set of information hiding modules, or IHMs, that become components in the evolving concurrent design.

The decision-making strategy used to structure modules begins by identifying elements in the data/control flow diagram that form the basis for IHMs. These include any interface and control objects and any data stores that are accessed by multiple transformations. In addition, some special configurations of state-dependent functions form modules. The concept classification, performed earlier during the analysis of the input specification, facilitates identification of candidate modules. Once candidate modules are identified, an attempt is made to allocate functions to existing data-abstraction modules. Any remaining functions are then examined. Functions that

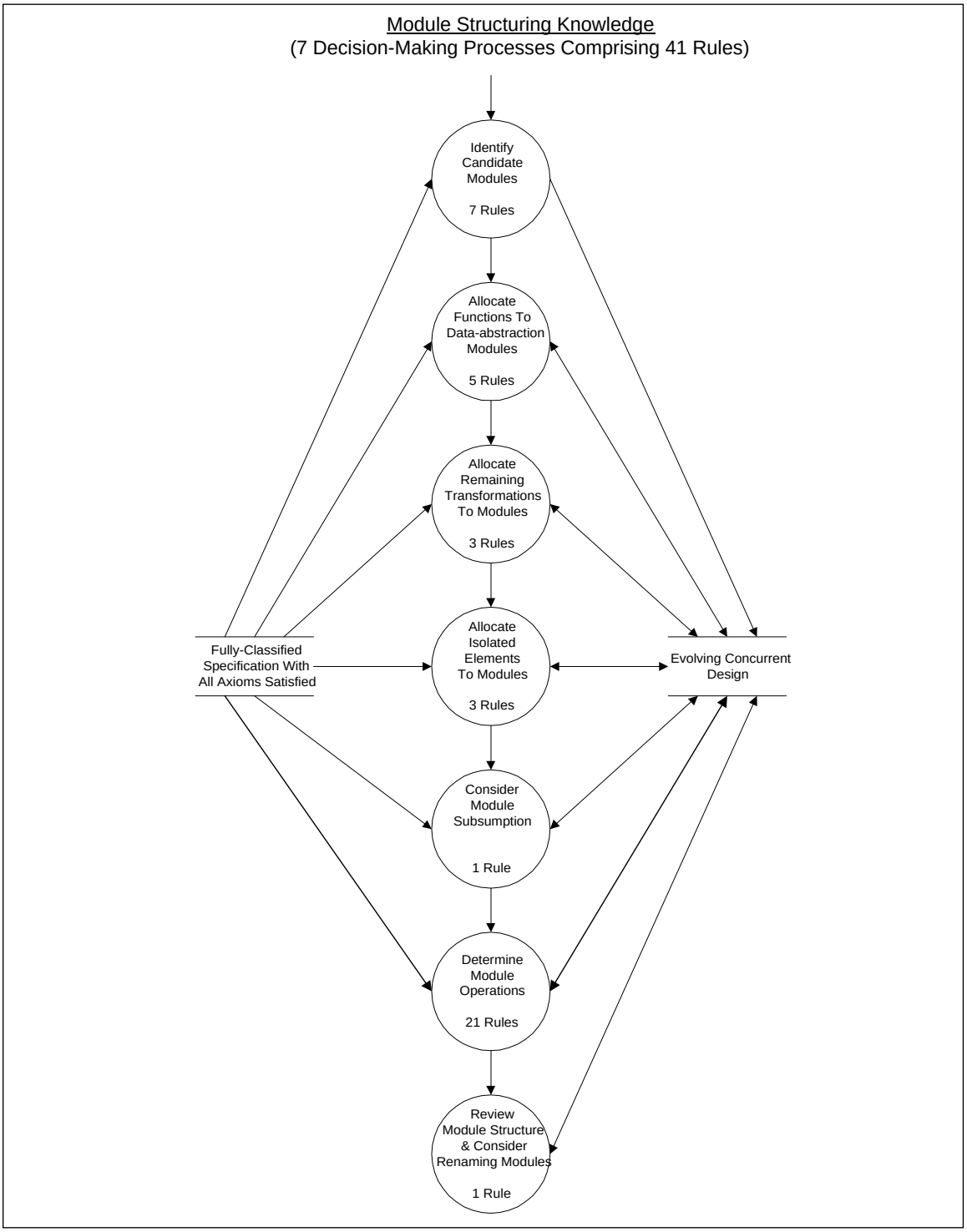


Figure 27. Organization of Module Structuring Knowledge

hide complex algorithms are allocated to algorithm-hiding modules, while functions that cannot be definitively allocated are referred to an experienced designer, if available, for assistance. Any remaining nodes, called isolated elements, must connect to only one other node in the specification. These isolated elements are examined last. An isolated data store is incorporated into an existing module as local storage, unless an experienced designer indicates that the data store should be allocated to its own data-abstraction module. Each remaining, unallocated function is mapped to a corresponding algorithm-hiding module.

After all elements of the input specification are allocated to modules in the design, then selected modules are examined to identify those that might be combined together. Each case identified is referred to an experienced designer for a decision; thus, this portion of the strategy is invoked only when an experienced designer is available. To complete module structuring, module operations are determined and then the designer is offered an opportunity to review the module structure, and to consider whether any of the modules should be renamed.

### 8.1 Identify Candidate Modules

Module structuring begins with a decision-making process that: 1) identifies those transformations and data stores in the input specification that can be allocated to separate modules, 2) makes the necessary allocations, 3) captures the decisions and rationale, and then 4) denotes the traceability between existing specification elements and the newly created modules. This initial decision-making process consists of a set of rules that

identify device-interface modules, other interface modules, state-transition modules, data-abstraction modules, and state-driven, function-driver and algorithm-hiding modules.

### 8.1.1 Rule for Identifying Device-Interface Modules

The CODARTS design method identifies a criterion for allocating a module to provide a virtual interface for each device, thus, insulating other software modules from changes in the actual devices, where such changes do not alter the virtual interface. [Gomaa93, pp. 226-227] A single design-decision rule reflects this criterion.

Rule: Device Interface Module

**if**

Transformation$_{DIO}$ is a Device Interface Object

**then**

create a device interface module, IHM$_{DIM}$
record the design decision and rationale in the design history for IHM$_{DIM}$
denote the traceability between Transformation$_{DIO}$ and IHM$_{DIM}$

**fi**

This rule recognizes each transformation in an input specification that inherits the concept Device Interface Object. Each such transformation forms the basis for a Device Interface Module, or DIM.

### 8.1.2 Rules for Identifying Other Interface Modules

While not explicitly addressed in the CODARTS module structuring criteria, other interface objects could form the basis for modules that provide a virtual interface to the external elements with which they interact. Two design-decision rules address this extension of the CODARTS device interface module criterion.

Rule: User Interface Module

**if**

Transformation$_{URIO}$ is a User-Role Interface Object

**then**

create a user interface module, IHM$_{UIM}$
record the design decision and rationale in the design history for IHM$_{UIM}$
denote the traceability between Transformation$_{URIO}$ and IHM$_{UIM}$

**fi**

This rule could apply, for example, in a factory automation application as described by Gomaa. [Gomaa93, Chapter 25] In the example, three user roles are specified: process engineer, production manager, and factory operator. An interface object is defined for each user role: Process Planning User Services, Production Management, and Operator Services, respectively. A User Interface Module would be allocated for each of these user-role interface objects.

Another rule recognizes each object that provides an interface to an external subsystem. For each such object recognized a Subsystem Interface Module is allocated.

Rule: Subsystem Interface Module

**if**

Transformation$_{SIO}$ is a Subsystem Interface Object

**then**

create a subsystem interface module, IHM$_{SIM}$
record the design decision and rationale in the design history for IHM$_{SIM}$
denote the traceability between Transformation$_{SIO}$ and IHM$_{SIM}$

**fi**

As an example where this rule might apply, consider the factory automation system described by Gomaa. [Gomaa93, Chapter 25] Suppose that each Line Workstation

Controller is defined to be a distributed subsystem, rather than an internal subsystem. In the decomposition of each Line Workstation Controller subsystem, the predecessor and successor workstations could each be represented as a Terminator that interacts with an interface object, the Predecessor Workstation Interface and the Successor Workstation Interface, respectively. Two Subsystem Interface Modules would then be defined to provide a virtual interface for sending messages to and receiving messages from the appropriate Link Workstation Controller subsystems.

### 8.1.3 Rule for Identifying State-Transition Modules

The CODARTS design method includes a criterion for allocating a behavior hiding module for each control object in an input specification. [Gomaa93, p. 230] This criterion is echoed in the following rule.

Rule: State-Transition Module

**if**

Transformation$_{CO}$ is a Control Object

**then**

create a state transition module, IHM$_{STM}$
record the design decision and rationale in the design history for IHM$_{STM}$
denote the traceability between Transformation$_{CO}$ and IHM$_{STM}$

**fi**

A cruise control and monitoring system described by Gomaa includes two control objects: Cruise Control and Calibration Control. [Gomaa93, Chapter 22] The rule that allocates state-transition modules maps each of these control objects to a state-transition module.

### 8.1.4 Rule for Identifying Data-Abstraction Modules

The CODARTS design method recognizes that each data store in a data/control flow diagram potentially forms the basis for creating a data-abstraction module, or DAM. [Gomaa93, pp. 228-230] A DAM is used to hide the internal structure of a data store, and to provide access to the information in the data store through operations. The design-decision rules defined for module structuring consider two types of data stores: those that are accessed by multiple transformations and those that are accessed by only a single transformation. The former data stores serve as the basis for establishing a DAM. Consideration of the latter data stores is deferred until a later decision-making process that examines isolated elements. The rule to establish a DAM is given below.

Rule: Data Abstraction Module

**if**

$Node_{DS}$ is a Data Store and
$Node_{DS}$ is accessed by multiple transformations

**then**

create a data abstraction module, $IHM_{DAM}$
record the design decision and rationale in the design history for $IHM_{DAM}$
denote the traceability between $Node_{DS}$ and $IHM_{DAM}$

**fi**

Each access to a data store consists of either a Store, a Retrieve, or an Update, each a concept specified in the semantic meta-model for specifications. Any transformation that connects to a data store via one of these concepts constitutes an accessing transformation. An example where this rule applies can be found in a robot controller system described by Gomaa. [Gomaa93, Chapter 23] In the example, a data store,

Sensor/Actuator Data Store connects via an Update with the transformation Process Sensor/Actuator Command, via a Store with the transformation Input From Sensors, and via a Retrieve with the transformation Output To Actuators. Since three transformations access the data store, the rule allocates a DAM based on Sensor/Actuator Data Store.

### 8.1.5 Rules for Identifying State-Driven Modules

Although not strictly called for in the CODARTS module structuring criteria, the design-decision rules defined in this dissertation include two rules to form modules from special configurations of state-driven transformations. State-driven transformations include those that are triggered or enabled by a Control Object. One rule recognizes state-driven transformations that send outputs only to Device Interface Objects. This rule applies the CODARTS function-driver module criterion [Gomaa93, p. 231] to state-driven transformations. The second rule groups state-driven transformations that do not connect to data stores, but that send outputs to transformations other than Device Interface Objects into a single algorithm-hiding module. This rule adapts the CODARTS criterion for algorithm-hiding modules [Gomaa93, pp. 231-232] to situations where state-driven transformations can be grouped together based on their functional relationship as support algorithms for a state-transition module. Each rule is specified in turn below.

Rule: State-Dependent Function Driver Module

**if**
Transformation$_{SDF}$ is a State-Dependent Function with cardinality, F, and
Transformation$_{CO}$ is a Control Object and
Transformation$_{SDF}$ receives a Control Event Flow or a Signal from Transformation$_{CO}$ and
Transformation$_{SDF}$ sends a Signal or Stimulus to a Transformation$_{DIO}$ and
Transformation$_{DIO}$ is a Device Interface Object and
Transformation$_{SDF}$ does not write to any Data Store
**then**
create a function driver module, IHM$_{FDM}$
record the design decision and rationale in the design history for IHM$_{FDM}$
**for every** Transformation$_{E}$ that is a State-Dependent Function and
that has a cardinality, F, and
that receives a Control Event Flow or Signal from Transformation$_{CO}$ and
that sends a Signal or Stimulus to Transformation$_{DIO}$
and that does not write to any Data Store
allocate Transformation$_{E}$ to IHM$_{FDM}$
record the design decision and rationale in the design history for IHM$_{FDM}$
denote the traceability between Transformation$_{E}$ and IHM$_{FDM}$
**rof**
**fi**

A situation where this rule applies can be found in a cruise control and monitoring system presented by Gomaa. [Gomaa93, Chapter 22] In the example, a Control Object, Cruise Control, manages three State-Dependent Functions: Maintain Speed, Resume Cruising, and Increase Speed. Each of these State-Dependent Functions generates outputs for the same Device Interface Object, Throttle, and none of them write to any Data Store. Here, the three State-Dependent Functions would be mapped into a single function-driver module.

The second rule to group state-driven transformations into a module is specified below.

Rule: Triggered Algorithm-Hiding Module

**if**
Transformation$_{TSF}$ is a Triggered Synchronous Function
with cardinality, F, and
Transformation$_{CO}$ is a Control Object and
Transformation$_{TSF}$ receives a Control Event Flow or a Signal from
Transformation$_{CO}$ and
Transformation$_{TSF}$ sends a Signal or Stimulus to a Transformation$_{F}$ and
Transformation$_{F}$ is a Function and
Transformation$_{TSF}$ does not write to any Data Store
**then**
create an algorithm hiding module, IHM$_{AHM}$
record the design decision and rationale in the design history for IHM$_{FDM}$
**for every** Transformation$_{E}$ that is a Triggered Synchronous Function and
that has a cardinality, F, and
that receives a Control Event Flow or Signal from
Transformation$_{CO}$ and
that sends a Signal or Stimulus to any Transformation$_{A}$
where Transformation$_{A}$ is a Function and
that does not write to any Data Store
allocate Transformation$_{E}$ to IHM$_{AHM}$
record the design decision and rationale in the design history for
IHM$_{AHM}$
denote the traceability between Transformation$_{E}$ and IHM$_{AHM}$
**rof**
**fi**

To understand the distinction between this rule and the previous rule, a slightly modified version of the same example is used. In the aforementioned cruise control and monitoring system assume that, rather than sending outputs to the same Device Interface Object, the three State-Dependent Functions, Increase Speed, Resume Cruising, and

Maintain Speed, each send Stimuli to one or more Functions within the input specification. These three transformations would then form the basis for an algorithm hiding module in support of the state-transition module derived from the Control Object Cruise Control.

### 8.2 Allocate Functions to Data-Abstraction Modules

Once a candidate set of modules exists, consideration should be given to allocating functions that interact with data stores to the same IHM as the data store. These functions can be allocated as operations on DAMs, using criteria included in the CODARTS design method. [Gomaa93, pp. 228-230]. The CODARTS criteria identify a number of situations that should be considered. For example, if an entire transformation operates only on a data store, then the transformation should be allocated as an operation on the DAM that contains the data store. As another example, if a transformation reads from one data store and updates another data store, then the transformation should be allocated to the DAM that contains the data store updated by the transformation. The CODARTS criteria also state that when a transformation updates multiple data stores that the transformation should not be allocated to any DAM.

The design-decision rules specified below attempt to reflect the heuristic guidance given in the CODARTS criteria. In general, the rules use information-hiding as the criteria for creating modules, and so are biased to allocate a function to some data store whenever possible. The first rule allocates a function to a data store when that function connects to no other data store.

Rule: Function Connects With Only One Data Store

**if**

$IHM_{DAM}$ hides a $Data\ Store_{DS}$ and
$Function_F$ writes to or reads from $Data\ Store_{DS}$ and
$Function_F$ writes to or reads from no other Data Store

**then**

allocate $Function_F$ to $IHM_{DAM}$
record the design decision and rationale in the design history for $IHM_{DAM}$
denote the traceability between $Function_F$ and $IHM_{DAM}$

**fi**

The function in question is allocated to the same DAM as the data store it accesses even when that function sends outputs to other transformations. An example where this rule applies can be found in the elevator control system recounted by Gomaa. [Gomaa93, Chapter 24] In the example, a transformation, Update Status, writes to one and only one data store, Elevator Status and Plan. The rule defined above maps Update Status to an operation of the DAM that contains Elevator Status and Plan.

A second rule reflects the CODARTS heuristic that advises mapping a transformation to an operation of a DAM whenever the transformation writes only to the data store contained within the DAM.

Rule: Function Writes To Only One Data Store

**if**

$IHM_{DAM}$ hides a $Data\ Store_{DS}$ and
$Function_F$ writes to $Data\ Store_{DS}$ and
$Function_F$ writes to no other Data Store that is hidden in a DAM

**then**

allocate $Function_F$ to $IHM_{DAM}$
record the design decision and rationale in the design history for $IHM_{DAM}$
denote the traceability between $Function_F$ and $IHM_{DAM}$

**fi**

This rule causes a function to be mapped to a DAM operation whenever the function writes to a data store hidden within the DAM, provided the function writes to no other data store. Even though the function may read from other data stores, this rule ensures that the function will be mapped to an operation of the DAM that it updates. An example where this rule applies can be found in the cruise control and monitoring system detailed by Gomaa. [Gomaa93, Chapter 22] In the example, a transformation, Record Calibration Start, reads from a data store, Shaft Rotation Count, and writes to a data store Calibration Start Count. The rule defined above maps the transformation, Record Calibration Start, to an operation of the DAM hiding the data store, Calibration Start Count.

A third rule recognizes situations where a function might read from multiple data stores and write to no data store, but where that function is the sole reader from one data store, but not the sole reader from any other data store.

Rule: Function Is The Sole Reader From A Data Store

**if**

$IHM_{DAM}$ hides a $Data\ Store_{DS}$ and
$Function_F$ reads from $Data\ Store_{DS}$ and
$Function_F$ writes to no data store and
no other transformation reads from $Data\ Store_{DS}$ and
$Function_F$ is the sole reader from no data store other than $Data\ Store_{DS}$

**then**

allocate $Function_F$ to $IHM_{DAM}$
record the design decision and rationale in the design history for $IHM_{DAM}$
denote the traceability between $Function_F$ and $IHM_{DAM}$

**fi**

In such cases, the function is allocated to the DAM that hides the data store for which the function is the sole reader. This rule is based on the idea that a function might cohere more closely with a data store for which it is the only reader than to another data store for which many readers exist. An example where this rule would apply appears in a cruise control and monitoring system elaborated by Gomaa. [Gomaa93, Chapter 22] In the example, the function, Check Oil Filter Maintenance, reads from two data stores: Cumulative Distance and Miles at Last Oil Filter Maintenance. Check Oil Filter Maintenance writes to no data stores. The data store Cumulative Distance is accessed by many readers. The data store, Miles at Last Oil Filter Maintenance, however, is accessed by only one reader, Check Oil Filter Maintenance. The rule defined above allocates Check Oil Filter Maintenance to the DAM that contains the data store Miles at Last Oil Filter Maintenance.

The remaining situations where functions access multiple data stores might be decided by an experienced designer based on knowledge about the application. Where an experienced designer is not available, unallocated functions are mapped, in subsequent decision-making processes, by default to algorithm-hiding modules. Where an experienced designer is available, however, two rules are defined to enable the designer to make an allocation of functions to DAMs, where appropriate. The first rule recognizes the presence of the ambiguity and also the availability of an experienced designer. In such a situation, the ambiguity is referred to the designer for a decision.

Rule: Connects With Multiple Data Stores (Last Preference)

**if**
an experienced designer is present and
$Function_F$ accesses multiple Data Stores where those Data Stores are already allocated to a DAM
**then**
show the designer the $Set_{DS}$ of Data Stores that are already allocated to a DAM and that are also accessed by $Function_F$
ask the designer to indicate to which Data Store from $Set_{DS}$, if any, the $Function_F$ should be allocated as an operation
**if** the designer selects a $Data\ Store_A$ from $Set_{DS}$
**then** denote for use in a related rule, Function Allocated To Data Store, that $Function_F$ is an operation of $Data\ Store_A$
**fi**
**fi**

This rule is given last preference to ensure that it does not conflict with the other, more specific, rules that are able to allocate a function to a data store even when that function connects with multiple data stores. Should the designer indicate a decision, then a second rule recognizes that a decision was taken and implements that decision. This second rule is specified below.

Rule: Function Allocated To Data Store

**if**
$IHM_{DAM}$ hides a $Data\ Store_{DS}$ and
$Function_F$ is an operation on $Data\ Store_{DS}$, as indicated by the designer
**then**
allocate $Function_F$ to $IHM_{DAM}$
record the design decision and rationale in the design history for $IHM_{DAM}$
denote the traceability between $Function_F$ and $IHM_{DAM}$
**fi**

An example where these two rules, taken together, would apply can be found in the cruise control and monitoring example, assuming that the previously defined rule, Function Is The Sole Reader From A Data Store, does not exist. With that assumption, the function, Check Oil Filter Maintenance, referred to previously, is recognized by the rule, Connects With Multiple Data Stores, because the function reads from two data stores: Cumulative Distance and Miles at Last Oil Filter Maintenance. Assuming that an experienced designer is available, this function, and the two data stores, are referred to the designer for resolution. The designer could map the function to an operation on one of the two data stores, in which case, the rule, Function Allocated To Data Store, would be invoked. The designer might also choose to leave the function for later decision-making processes.

### 8.3 Allocate Remaining Transformations to Modules

After the object-based IHMs and the DAMs are created, a number of transformations might remain unallocated to any module. These unallocated transformations must now be examined and allocated. The CODARTS algorithm hiding module criterion provides the basis for allocating certain transformations to modules. [Gomaa93, pp. 231-232] A rule is specified to mirror this criterion.

Rule: Active Function

**if**

Transformation$_{AF}$ is a Periodic Function or an Asynchronous Function
or a Triggered Asynchronous Function
or a Triggered Synchronous Function

**then**

create an algorithm hiding module, IHM$_{AHM}$
allocate Transformation$_{AF}$ to IHM$_{AHM}$
record the design decision and rationale in the design history for IHM$_{AHM}$
denote the traceability between Transformation$_{AF}$ and IHM$_{AHM}$

**fi**

This rule creates an algorithm-hiding module for each unallocated function of the indicated types. An example where this rule applies can be found in the robot controller system described by Gomaa in a case study for the CODARTS design method. [Gomaa93, Chapter 23] In the example, an Asynchronous Function, Interpret Program Statement, forms the basis for an algorithm-hiding module, according to the rule given above.

After algorithm-hiding modules are allocated, the remaining, unallocated transformations in the input specification are Synchronous Functions. In general, the remaining Synchronous Functions might be allocated to existing modules where application-specific knowledge can be used to make such allocations. When an experienced designer is available, each unallocated Synchronous Function is referred to the designer for a decision. The following rule reflects these cases.

Rule: User Specifies Allocation (Last Preference)

**if**

an experienced designer is present and
$Transformation_{SF}$ is a Synchronous Function and
$Transformation_{F}$ is a Function and
$Transformation_{F}$ sends a Signal or Stimulus to $Transformation_{SF}$ and
$Transformation_{F}$ is allocated to an IHM

**then**

show the designer the $Set_{F}$ of Functions that connect to $Transformation_{SF}$ and that are already allocated to an IHM
ask the designer to select the Function from $Set_{F}$ , if any, that should be in the same IHM as $Transformation_{SF}$
**if** the designer indicates that $Transformation_{SF}$ should not be allocated to the same IHM as a Function in $Set_{F}$
**then**
create an algorithm hiding module, $IHM_{AHM}$
allocate $Transformation_{SF}$ to $IHM_{AHM}$
record the design decision and rationale in the design history for $IHM_{AHM}$
denote the traceability between $Transformation_{SF}$ and $IHM_{AHM}$
**else**
denote for use in the rule Synchronous Function Allocated To An IHM that $Transformation_{SF}$ should be placed in the same IHM as the Function selected from $Set_{F}$

**fi**

**fi**

An example where this rule applies can be found in the robot controller case study detailed by Gomaa. [Gomaa93, Chapter 23] In this example, a Synchronous Function, Receive Acknowledgment, might be allocated to the same algorithm-hiding module as either Generate Axis Command or Interpret Program Statement, or might be allocated to an algorithm-hiding module of its own. According to the rule defined above, the allocation of Receive Acknowledgment would be referred to an experienced designer, if available, for a decision.

Whenever an experienced designer does allocate a Synchronous Function to an existing IHM, the following rule implements that decision.

Rule: Synchronous Function Allocated To An IHM

**if**

Transformation$_{SF}$ is a Synchronous Function and
Transformation$_{F}$ is a Function and
Transformation$_{F}$ is allocated to IHM$_{AHM}$ and
Transformation$_{SF}$ should be allocated to the same IHM as
Transformation$_{F,}$ as indicated by the designer

**then**

allocate Transformation$_{SF}$ to IHM$_{AHM}$
record the design decision and rationale in the design history for IHM$_{AHM}$
denote the traceability between Transformation$_{SF}$ and IHM$_{AHM}$

**fi**

Assuming, from the preceding example, that the designer indicates that the Synchronous Function Receive Acknowledgment should be allocated to the same IHM as the Function Generate Axis Command, then this rule makes the necessary allocation.

### 8.4 Allocate Isolated Elements to Modules

Some elements from the specification might remain unallocated following completion of the first three decision-making processes contained within the Module Structuring Knowledge base. In general, these elements will be either: 1) data stores that are accessed by only one transformation and that, thus, were not used to establish data-abstraction modules or 2) unallocated Synchronous Functions that were not considered because an experienced designer was unavailable. Failure to use a data store as the basis for a data-abstraction module during the process of identifying candidate IHMs isolates these data stores from further decision-making. The main purpose of the

next decision-making process is to make allocation decisions regarding these isolated data stores. In addition, any Function that remains unallocated to an IHM is recognized and allocated to an algorithm-hiding module. First, the rules for unallocated data stores are explained.

### 8.4.1 Rules to Allocate Isolated Data Stores

Data Stores that connect via an Update arc to a single transformation, and that are accessed by no other transformation, serve most likely as local memory for the updating transformation, but might instead provide a placeholder for a data interface to an external subsystem, being designed separately, that is not shown on the data/control flow diagram. Since a different allocation of the data store is indicated in each of these cases, a rule is defined to consult with an experienced designer,[1] where available, about which use is intended. Where no experienced designer is available, the data store is mapped by default to local storage within an existing module. The rule is specified below, following a brief example of where the rule might apply.

In a cruise control and monitoring system case study, described by Gomaa, a function, Determine Distance, updates a data store, Last Distance. Last Distance connects to no other data store. In this situation, Last Distance might be allocated to a DAM or might be incorporated into an IHM that includes Determine Distance. The proper decision depends upon whether Last Distance serves only as a local memory for Determine Distance. An experienced designer can be asked about the correct

[1] In some situtations, the use of RTSA notation leads to ambiguity. The experienced designer referred to in such situations is the author of the data/control flow diagram who, one hopes, can resolve the ambiguity.

interpretation of Last Distance. Lacking an experienced designer, the most likely interpretation can be made by default; thus, the following rule is defined.

Rule: Isolated Update Of A Data Store

**if**
Transformation$_{ST}$ is a Solid Transformation and
Transformation$_{ST}$ is allocated to IHM$_{I}$
Transformation$_{ST}$ updates a Data Store$_{DS}$ and
no other transformation accesses Data Store$_{DS}$
**then**
**if** an experienced designer is available
**then**
ask the designer whether Data Store$_{DS}$ is used as local memory for Transformation$_{ST}$ or as an interface to another subsystem
**else**
assume Data Store$_{DS}$ is used for local memory
**fi**
**if** Data Store$_{DS}$ is used for local memory
**then**
allocate Data Store$_{DS}$ to IHM$_{I}$
record the design decision and rationale in the design history for IHM$_{I}$
denote the traceability between Data Store$_{DS}$ and IHM$_{I}$
**else**
create a data abstraction module, IHM$_{DAM}$
allocate Data Store$_{DS}$ to IHM$_{DAM}$
record the design decision and rationale in the design history for IHM$_{DAM}$
denote the traceability between Data Store$_{DS}$ and IHM$_{DAM}$
**fi**
**fi**

Any data stores that remain unallocated after this rule is applied must be read-only or write-only data stores that connect to a single transformation. Data stores in this configuration might represent, for example, parameter files or audit trail files, or might

provide a data interface to another subsystem. In any of these cases, the isolated data store forms the basis for a data-abstraction module, according to the following rule.

Rule: Isolated Data Store (Last Preference)

**if**
Data Store$_{DS}$ is not allocated to an IHM
**then**
create a data abstraction IHM$_{DAM}$
allocate Data Store$_{DS}$ to IHM$_{DAM}$
record the design decision and rationale in the design history for IHM$_{DAM}$
denote the traceability between Data Store$_{DS}$ and IHM$_{DAM}$
**fi**

An example where this rule applies can be found in the robot controller case study provided by Gomaa. [Gomaa93, Chapter 23] In the example, Robot Program, a read-only data store that is accessed only by the transformation Interpret Program Statement, represents an input file that is created off-line by another subsystem in the application. The rule specified above allocates the data store Robot Program to a data-abstraction module.

### 8.4.2 Rule to Allocate Isolated Functions

Any remaining elements that require allocation must be Synchronous Functions. Synchronous Functions only remain unallocated if no experienced designer is available for consultation. Each unallocated Synchronous Function forms the basis for an algorithm-hiding module. The rule defined to recognize this situation is specified below.

Rule: Isolated Function

**if**
Transformation$_F$ is a Function and
Transformation$_F$ is not allocated to an IHM
**then**
create an algorithm hiding module, IHM$_{AHM}$
allocate Transformation$_F$ to IHM$_{AHM}$
record the design decision and rationale in the design history for IHM$_{AHM}$
denote the traceability between Transformation$_F$ and IHM$_{AHM}$
**fi**

An example where this rule might apply can be illustrated using the robot controller case study referred to earlier. [Gomaa93, Chapter 23] In the example, assume that the transformation Process Motion Command is classified as a Synchronous Function; further assume that no experienced designer is available to decide that Process Motion Command should be allocated to the same IHM as the transformation Interpret Program Statement. With these assumptions, the rule defined above will form an algorithm-hiding module based on the existence of the unallocated Synchronous Function, Process Motion Command.

### 8.5 Consider Module Subsumption

The fifth decision-making process within the Module Structuring Knowledge base considers whether any pairs of information-hiding modules are candidates to be combined. This phase only applies if the designer is experienced. This constraint exists because any decision to combine two information-hiding modules requires a judgment on the part of the designer, a judgment that novice designers should not be called upon to

make. Currently, only data-abstraction modules, or DAMs, are considered. If the designer is not experienced, then DAMs will not be combined. If the designer is experienced, then when one DAM exists that is read solely by another DAM, consideration should be given to combining the DAMs, depending upon application-specific knowledge and upon other design considerations not contained within design-decision rules. The rule defined below identifies situations where a designer might want to consider combining DAMs and then allows the designer to review the structure of the modules and to make a decision to combine the modules or to keep them separate.

Rule: Exclusive Read Between Data-Abstraction Modules

**if**
designer is experienced and
$IHM_{DAM1}$ is a data-abstraction module and
$IHM_{DAM2}$ is a data-abstraction module and
$IHM_{DAM2}$ reads from $IHM_{DAM1}$ and
no other IHM reads from $IHM_{DAM1}$
**then**
report to the designer that $IHM_{DAM2}$ is a candidate to subsume $IHM_{DAM1}$
depending on application-specific knowledge
offer the designer a chance to review the components of $IHM_{DAM1}$
and $IHM_{DAM2}$
ask the designer whether $IHM_{DAM2}$ should subsume $IHM_{DAM1}$
**if** the designer says to subsume $IHM_{DAM1}$
**then**
merge $IHM_{DAM1}$ into $IHM_{DAM2}$
record the design decision and rationale in the design history for
$IHM_{DAM2}$
**fi**
**fi**

A situation where this rule might apply appears in the cruise control and monitoring system, as specified by Gomaa. [Gomaa93, Chapter 22] Assuming that a design is under construction, previously explained rules would have created a number of DAMs. Three of these DAMs are relevant to the current discussion: 1) a DAM, consisting of Calibration Start Count (a data store) and Record Calibration Start (a transformation that writes to Calibration Start Count), 2) a DAM consisting of Calibration Constant (a data store) and Compute Calibration Constant (a transformation that writes to Calibration Constant), and 3) a DAM consisting of Cumulative Distance (a data store), Determine Distance (a transformation that writes to Cumulative Distance), and Last Distance (a data store providing local memory for Determine Distance). The DAM containing Cumulative Distance is the only DAM containing an operation that reads the DAM containing Calibration Constant; thus, the Calibration Constant DAM might be subsumed by the Cumulative Distance DAM. An experienced designer might decide, however, that the functional nature of these DAMs is too different to warrant combining them. Consider, though, another situation involving the DAM containing Calibration Constant, which is the only DAM containing an operation that reads the DAM containing Calibration Start Count. In this case, an experienced designer might decide that the functional similarity between the two DAMs warrants combining them.

### 8.6 Determine Module Operations

The sixth decision-making process that composes the Module Structuring Knowledge determines the operations provided by each module and maps relevant

specification elements to operation parameters. Before operations and parameters are determined, arcs that flow internally within a module are allocated to that module and dropped from further consideration. Depending upon the type of module under consideration, different strategies are used to determine module operations. Interface modules are treated as objects, with incoming event flows and data flows used to identify operations. Outgoing event flows and data flows from interface modules are mapped to operation parameters. In certain cases, where no incoming event flow or data flow exists for an interface module, operations are created based upon the data flows exchanged between the module and its associated terminator. State-transition modules are treated as objects with two standard operations, Process Event and Get Current State. All event flows entering the control object associated with a state-transition module are mapped onto an incoming parameter for the Process Event operation. For any data-abstraction module, each direct access from outside the module to a data store encapsulated within the module is mapped to an appropriate operation on the data store, that is, a Get, Put, or Update operation. Any function within a module that is accessed from outside the module is mapped to an operation for the module. For operations derived from functions, incoming and outgoing event flows and data flows are mapped, in most cases, to operation parameters. The design-decision rules that implement these strategies are specified and described below.

### 8.6.1 Rule to Allocate Arcs Internal to Modules

Event flows and data flows between specification elements within the same information hiding module can be allocated to the module and then dropped from further consideration because such flows are not visible outside the module. This rule ensures that an appropriate allocation is made for each arc within the input specification, even when that arc is not significant outside an information hiding module. The rule to identify such flows and to make the necessary allocation is specified below.

Rule: Allocate Arc Internal To IHM (First Preference)

**if**
a $Node_A$ is in an $IHM_I$ and
a $Node_B$ is in an $IHM_I$ and
an $Arc_{AB}$ is a Directed-Arc or Two-Way-Arc and
$Arc_{AB}$ connects $Node_A$ and $Node_B$
**then**
allocate $Arc_{AB}$ to $IHM_I$
record the decision and rationale in the history for $IHM_I$
**fi**

An example where this rule applies can be found in an elevator control system case study described by Gomaa. [Gomaa93, Chapter 24] In the example, a data-abstraction module, Elevator Status and Plan, comprises a data store and four transformations: Check This Floor, Update Status, Check Next Destination, and Accept New Request. All accesses to the module come through one of the four transformations; thus, each data flow between the transformations and the data store that compose the module is not visible outside the module. The rule specified above allocates each of these data flows to

the data-abstraction module, Elevator Status and Plan. The rule is given first preference in order that arcs internal to IHMs can be eliminated from further consideration. Such arcs do not lead to module operations, nor to parameters.

### 8.6.2 Rules to Determine Operations for Interface Modules

Each interface module, that is, each module derived from an object that exchanges data or events with a terminator, is viewed as an object. For each such module, an initialization operation is created and the various incoming and outgoing event flows and data flows are mapped to appropriate operations and parameters. A number of situations must be anticipated. One rule, specified below, allocates an initialization operation for each interface module in an evolving design.

Rule: Interface Module Skeleton

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$
**then**
create an $Initialize\ Operation_{IOP}$
establish the design relationship $IHM_{IM}$ Provides $Operation_{IOP}$
allocate $Object_{IO}$ to $Operation_{IOP}$
record the decision and rationale in the history for $IHM_{IM}$
create an $Initialization\ Data\ Parameter_{P}$
establish the design relationship $Operation_{IOP}$ Takes $Parameter_{P}$
record the decision and rationale in the history for $Operation_{IOP}$
**fi**

Many examples where this rule applies can be seen in a cruise control and monitoring system presented by Gomaa. [Gomaa93, Chapter 22] In the example, a number of device interface objects can be found: Brake, Engine, Shaft, Cruise-Control Lever,

Calibration Buttons, Gas Tank, Mileage Display, Maintenance Display, Maintenance Reset Buttons, and Mileage Reset Buttons. Each such device interface object results in the creation of a device-interface module. The rule specified above creates an operation, Initialize, for each device-interface module in the example.

A second rule creates an operation for each uniquely-named event flow to an interface object from another transformation. Multiple occurrences of an incoming event flow with the same name are mapped to the same operation. The rule is specified below.

Rule: Signal To Interface Module

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the sink for a $Signal_S$
**then**
**if** an $Operation_{SOP}$ with the same name as $Signal_S$ is not already provided by $IHM_{IM}$
**then**
create $Operation_{SOP}$ with the same name as $Signal_S$
establish the design relationship $IHM_{IM}$ Provides $Operation_{SOP}$
record the decision and rationale in the history for $IHM_{IM}$
**else**
use existing $Operation_{SOP}$
**fi**
allocate $Signal_S$ to $Operation_{SOP}$
record the decision and rationale in the history for $Operation_{SOP}$
**fi**

An example where this rule applies can be found in Gomaa's elevator control system case study. [Gomaa93, Chapter 24] In the case study, two device-interface objects, Elevator Door and Motor, receive event flows. Elevator Door receives Open Door and

Close Door, while Motor receives Up, Down, and Stop. The rule specified above maps each of these event flows to an operation with the same name as the event flow.

A third rule creates a Put operation for each uniquely-named data flow received by an interface object from another transformation, where the incoming data flow is not associated with an outgoing response. Each such operation created is named with a concatenation of two symbols: Put_ and data-name, where data-name is the name of the incoming data flow. Multiple occurrences of an incoming data flow with the same name are mapped to the same operation. The rule is specified below.

Rule: Stimulus Without Response To Interface Module

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the sink for a $Stimulus_S$ and
a $Transformation_T$ is the source for $Stimulus_S$ and
no $Response_R$ flows from $Interface\ Object_{IO}$ to $Transformation_T$
**then**
**if** an $Operation_{POP}$ named Put_Data that Takes a $Parameter_P$ named Data, where Data is the name of $Stimulus_S$, is not already provided by $IHM_{IM}$
**then**
create $Operation_{POP}$ named Put_Data
establish the design relationship $IHM_{IM}$ Provides $Operation_{POP}$
record the decision and rationale in the history for $IHM_{IM}$
create $Parameter_P$ with the same name as $Stimulus_S$
establish the design relationship $Operation_{POP}$ Takes $Parameter_P$
record the decision and rationale in the history for $Operation_{POP}$
**else** use existing $Operation_{POP}$ and $Parameter_P$
**fi**
allocate $Stimulus_S$ to $Operation_{POP}$
allocate $Stimulus_S$ to $Parameter_P$
record the decision and rationale in the history for $Parameter_P$
**fi**

An example where this rule applies appears in the cruise control and monitoring system case study presented by Gomaa. [Gomaa93, Chapter 22] In the example, two device interface objects, Mileage Display and Maintenance Display, receive five incoming data flows: Average MPG, Average MPH, Oil Filter Status, Air Filter Status, and Major Service Maintenance Status. The rule specified above creates an operation for each of these incoming data flows, for example, Put_Average_MPG, with each created operation taking an incoming parameter, for example, Average_MPG.

Another example found in the same case study illustrates the compressing effect obtained from the rule. The device interface object Throttle receives three incoming data flows, each named Throttle Value. The rule specified above creates an operation, Put_Throttle_Value with a parameter, Throttle_Value, and then, with repeated application of the rule, maps each instance of the Throttle Value data flow to the same operation and parameter.

A fourth rule creates an operation for interface objects that receive a data flow and emit an associated response, where the name of the incoming data flow is not the same as the name of the associated response.[2] In such cases, the incoming data flow and its associated response are viewed as a request for data, possibly based upon information provided with the incoming data flow. For this reason, such stimulus-response pairs are mapped to a Get_Data operation, where Data is the name of the response data flow, that

[2] Note that the RTSA notation, and the specification meta-model as defined in Chapter 4, do not insist that the name of a Stimulus and Response be different.

has an incoming parameter, named for the incoming data flow, and an outgoing parameter, named for the response. The rule is specified below.

Rule: Stimulus With Response To Interface Module

**if**

an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the sink for a $Stimulus_S$ and
a $Transformation_T$ is the source for $Stimulus_S$ and
a $Response_R$ flows from $Interface\ Object_{IO}$ to $Transformation_T$ and
$Response_R$ and $Stimulus_S$ do not have the same name

**then**

**if** an $Operation_{GOP}$ named Get_Resp, where Resp is the name of $Response_R$ that Takes a $Parameter_S$ named Stim, where Stim is the name of $Stimulus_S$, and Yields a $Parameter_R$ named Resp, is not already provided by $IHM_{IM}$

**then**

create $Operation_{GOP}$ named Get_Resp
establish the design relationship $IHM_{IM}$ Provides $Operation_{GOP}$
record the decision and rationale in the history for $IHM_{IM}$
create $Parameter_R$ with the same name as $Response_R$
establish the design relationship $Operation_{GOP}$ Yields $Parameter_R$
create $Parameter_S$ with the same name as $Stimulus_S$
establish the design relationship $Operation_{GOP}$ Takes $Parameter_S$
record the decision and rationale in the history for $Operation_{GOP}$

**else**

use existing $Operation_{GOP}$ and $Parameter_S$ and $Parameter_R$

**fi**

allocate $Response_R$ to $Operation_{GOP}$
allocate $Stimulus_S$ to $Operation_{GOP}$
allocate $Response_R$ to $Parameter_R$
record the decision and rationale in the history for $Parameter_R$
allocate $Stimulus_S$ to $Parameter_S$
record the decision and rationale in the history for $Parameter_S$

**fi**

An example where this rule applies can be seen by examining the Gas Tank device interface object in Gomaa's cruise control and monitoring system case study. The Gas Tank receives the Fuel Request data flow and responds with the Fuel Level data flow. In this situation, the rule specified above creates an operation, Get_Fuel_Level, with an input parameter, Fuel Request, and an output parameter, Fuel_Level. In addition, since the Gas Tank exchanges two instances of the Fuel Request and Fuel Level data flows, one with the transformation Initialize MPG and another with the transformation Compute Average MPG, the rule maps each of these instances to the same operation.

A fifth rule attends to situations where an interface object receives a data flow and emits an associated response, but where the incoming and responding data flow have the same name. Such situations are viewed as operations where a data value is passed in, possibly modified, and then returned to the caller of the operation. With this view, such situations are mapped to an Update_Data operation, where Data is the name of the incoming and associated responding data flows. Each such operation created will also be assigned an input/output parameter, Data, named from the incoming and responding data flows. As with previous rules, multiple instances of a stimulus-response pair with the same name, are mapped to a single Update_Data operation.

To fabricate an example where this rule could apply assume that an Interface Object, Password_Generator, accepts a data input, Password, and returns a data output, Password. Perhaps the intent of such an operation is to validate an old password before generating a new password. If this fabricated example existed, then an operation,

Update_Password, would be created with an input/output parameter named Password. The rule to make these mappings is specified below.

Rule: Same Stimulus And Response With Interface Module

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the sink for a $Stimulus_S$ and
a $Transformation_T$ is the source for $Stimulus_S$ and
a $Response_R$ flows from $Interface\ Object_{IO}$ to $Transformation_T$ and
$Response_R$ and $Stimulus_S$ have the same name
**then**
**if** an $Operation_{UOP}$ named Update_Data, where Data is the name of $Response_R$, that Alters a $Parameter_A$ named Data, is not already provided by $IHM_{IM}$
**then**
create $Operation_{UOP}$ named Update_Data
establish the design relationship $IHM_{IM}$ Provides $Operation_{UOP}$
record the decision and rationale in the history for $IHM_{IM}$
create $Parameter_A$ with the same name as $Response_R$
establish the design relationship $Operation_{UOP}$ Alters $Parameter_A$
record the decision and rationale in the history for $Operation_{UOP}$
**else**
use existing $Operation_{UOP}$ and $Parameter_A$
**fi**
allocate $Response_R$ to $Operation_{UOP}$
allocate $Stimulus_S$ to $Operation_{UOP}$
allocate $Response_R$ to $Parameter_A$
allocate $Stimulus_S$ to $Parameter_A$
record the decision and rationale in the history for $Parameter_A$
**fi**

Event flows and data flows, other than responses, from an interface object to another transformation are allocated to Get_Name operations, where Name is Status, in

the case of event flows, or the name of the flow, in the case of data flows. A rule is specified below for each of these situations, beginning with outgoing event flows.

Rule: Signal From Interface Module

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the source for a $Signal_S$
**then**
**if** an $Operation_{GOP}$ with the name Get_Status that yields a $Parameter_O$ named Status is not already provided by $IHM_{IM}$
**then**
create $Operation_{GOP}$ named Get_Status
establish the design relationship $IHM_{IM}$ Provides $Operation_{GOP}$
record the decision and rationale in the history for $IHM_{IM}$
create $Parameter_P$ name Status
establish the design relationship $Operation_{GOP}$ Yields $Parameter_P$
record the decision and rationale in the history for $Operation_{GOP}$
**else**
use existing $Operation_{GOP}$ and $Parameter_P$
**fi**
allocate $Signal_S$ to $Operation_{GOP}$
allocate $Signal_S$ to $Parameter_P$
record the decision and rationale in the history for $Parameter_P$

**fi**

An example where this rule applies can be found in Gomaa's elevator control system case study [Gomaa93, Chapter 24] where the device interface object Motor emits event flows Elevator Started and Elevator Stopped and where the device interface object Elevator Door emits event flows Door Opened and Door Closed. The rule specified above will create a Get_Status operation for the device-interface module, or DIM, derived

from Motor and for the DIM derived from Elevator Door. Each of these operations will yield a parameter, Status, and the corresponding event flows will be mapped to both the operation and the parameter.

A similar rule, specified below, creates operations for data flows emitted by an interface object, where such data flows have no associated response. When an outgoing data flow does have an associated response, then the outgoing data flow is viewed as an invocation of an operation provided by the responding transformation.

Rule: Stimulus Without Response From Interface Module

**if**
an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an $Interface\ Object_{IO}$ and
$Interface\ Object_{IO}$ is the source for a $Stimulus_S$ and
a $Transformation_T$ is the sink for $Stimulus_S$ and
no $Response_R$ flows to $Interface\ Object_{IO}$ from $Transformation_T$
**then**
**if** an $Operation_{GOP}$ named Get_Data that Yields a $Parameter_P$ named Data, where Data is the name of $Stimulus_S$, is not already provided by $IHM_{IM}$
**then**
create $Operation_{GOP}$ named Get_Data
establish the design relationship $IHM_{IM}$ Provides $Operation_{GOP}$
record the decision and rationale in the history for $IHM_{IM}$
create $Parameter_P$ with the same name as $Stimulus_S$
establish the design relationship $Operation_{GOP}$ Yields $Parameter_P$
record the decision and rationale in the history for $Operation_{GOP}$
**else** use existing $Operation_{GOP}$ and $Parameter_P$
**fi**
allocate $Stimulus_S$ to $Operation_{GOP}$
allocate $Stimulus_S$ to $Parameter_P$
record the decision and rationale in the history for $Parameter_P$
**fi**

An example where this rule applies appears in Gomaa's elevator control system case study. [Gomaa93, Chapter 24] The example contains a device interface object, Floor Arrival Sensor, that emits a data flow, Floor Number, which has no associated response. The rule specified above maps this data flow to the operation Get_Floor_Number that yields a parameter, Floor_Number.

A final rule is specified to determine operations for interface objects where no event flows or data flows are exchanged between the interface object and any other transformation. Such situations arise whenever an interface object interacts solely with a data store. In these cases, any input data flow from a terminator to the interface object is mapped to a Read operation with the input data flow as an output parameter. Where no input data flow exists but where an interrupt event flow exists, then a Read_Count operation is created with an output parameter named Count. This operation allows the count of interrupts to be accessed. Any output data flow to a terminator from the interface object is mapped to a Write operation with the output data flow as an input parameter.

Several cases where these situations arise can be found in the case studies described by Gomaa. For example, Gomaa's robot controller system [Gomaa93, Chapter 23] includes two device interface objects, Output To Actuators and Input From Sensors, that interact solely with a data store, Sensor/Actuator Data Store. For the Output To Actuators object, an operation is needed to write the data flow Actuator Output. For the Input From Sensors object, an operation is needed to read the data flow Sensor Input. A

different example appears in the cruise control and monitoring case study where the Shaft object interacts solely with the data store Shaft Rotation Count but receives no input data flows. Here, an operation can be provided to read the count of interrups. A rule that addresses these cases is specified below.

Rule: Interface Module Without Drivers

**if**

an $IHM_{IM}$ is a device-interface module or a user-interface module or a subsystem-interface module and
$IHM_{IM}$ is derived from an Interface $Object_{IO}$ and
Interface $Object_{IO}$ is not the source for any Signal or Stimulus and
Interface $Object_{IO}$ is not the sink for any Signal or Stimulus

**then**

**if** Interface $Object_{IO}$ receives an $Input_I$

**then**

create an $Operation_{ROP}$ named Read
establish the design relationship $IHM_{IM}$ Provides $Operation_{ROP}$
record the decision and rationale in the history for $IHM_{IM}$
allocate $Input_I$ to $Operation_{ROP}$
record the decision and rationale in the history for $Operation_{ROP}$
create a $Parameter_O$ with the same name as $Input_I$
establish the design relationship $Operation_{ROP}$ Yields $Parameter_O$
record the decision and rationale in the history for $Operation_{ROP}$
allocate $Input_I$ to $Parameter_O$
record the decision and rationale in the history for $Parameter_O$

**fi**

**if** Interface $Object_{IO}$ receives an $Interrupt_I$ and receives no Input

**then**

create an $Operation_{ROP}$ named Read_Count
establish the design relationship $IHM_{IM}$ Provides $Operation_{ROP}$
record the decision and rationale in the history for $IHM_{IM}$
allocate $Interrupt_I$ to $Operation_{ROP}$
record the decision and rationale in the history for $Operation_{ROP}$
create a $Parameter_O$ named Count
establish the design relationship $Operation_{ROP}$ Yields $Parameter_O$
record the decision and rationale in the history for $Operation_{ROP}$
allocate $Interrupt_I$ to $Parameter_O$
record the decision and rationale in the history for $Parameter_O$

**fi**
**if** Interface Object$_{IO}$ emits an Output$_{O}$
**then**
create an Operation$_{WOP}$ named Write
establish the design relationship IHM$_{IM}$ Provides Operation$_{WOP}$
record the decision and rationale in the history for IHM$_{IM}$
allocate Output$_{O}$ to Operation$_{WOP}$
record the decision and rationale in the history for Operation$_{WOP}$
create a Parameter$_{I}$ with the same name as Output$_{O}$
establish the design relationship Operation$_{WOP}$ Takes Parameter$_{I}$
record the decision and rationale in the history for Operation$_{WOP}$
allocate Output$_{O}$ to Parameter$_{I}$
record the decision and rationale in the history for Parameter$_{I}$
**fi**
**fi**

### 8.6.3 Rule to Determine Operations for State-Transition Modules

Each state-transition module, formed from a control object, is treated as a special-purpose module encapsulating a state-transition diagram. State-transition modules are always allocated two operations. One operation, Process Event, accepts an input parameter, Event, and then looks up the appropriate set of actions for the value associated with Event, depending on the current state, as maintained within the state-transition module. The second operation, Get Current State, simply returns the current state of the encapsulated state-transition diagram.[3] This manner of addressing state-transition modules is taken from the CODARTS module structuring criteria. [Gomaa93, pp. 230-231] Only one rule is needed to determine the operations for state-transition modules. The required rule is specified below.

---

[3] When not needed, this operation can be eliminated later by the designer.

Rule: STM

**if**

an $IHM_{STM}$ is a state-transition module and
$IHM_{STM}$ is derived from an $Control\ Object_{CO}$ and

**then**

create an $Operation_{PE}$ named Process_Event
establish the design relationship $IHM_{STM}$ Provides $Operation_{PE}$
record the decision and rationale in the history for $IHM_{STM}$
create a $Parameter_{E}$ named Event
establish the design relationship $Operation_{PE}$ Takes $Parameter_{E}$
record the decision and rationale in the history for $Operation_{PE}$
**for** each $Signal_{S}$ such that $Control\ Object_{CO}$ is the sink for $Signal_{S}$
allocate $Signal_{S}$ to $Parameter_{E}$
record the decision and rationale in the history for $Parameter_{E}$
**rof**
create an $Operation_{GCS}$ named Get_Current_State
establish the design relationship $IHM_{STM}$ Provides $Operation_{GCS}$
record the decision and rationale in the history for $IHM_{STM}$
create a $Parameter_{S}$ named STD_State
establish the design relationship $Operation_{CGS}$ Yields $Parameter_{S}$
record the decision and rationale in the history for $Operation_{GCS}$

**fi**

This rule applies for each control object that exists in an input specification because each control object is mapped to a state-transition module and each state-transition module provides two operations. Several of the case studies presented by Gomaa [Gomaa93, Chapters 22, 23, and 24] contain instances of a control object; for example, Cruise Control and Perform Calibration in the cruise control and monitoring system, Control Robot in the robot controller case study, and Elevator Control in the elevator control system. The rule specified above applies to each of these instances.

### 8.6.4 Rules to Determine Operations for Direct Access to Data Stores

A set of three rules addresses situations where a data store within a data-abstraction module connects directly via a directed arc with a transformation that is outside the data-abstraction module. Each direct access from outside a data-abstraction module requires that an appropriate operation be provided by the data-abstraction module. One rule, specified below, creates for an outgoing data flow a Get_Data operation, where Data is replaced by the name of the data flow or, if the data flow has no name, by the name of the data store. The operation yields a parameter with the same name as that used in place of Data in the operation name.

Rule: DAM Get

**if**
an $IHM_{DAM}$ is a data-abstraction module and
$IHM_{DAM}$ is derived from a $Data\ Store_{DS}$ and
$Data\ Store_{DS}$ is the source of $Retrieve_{R}$ and
$Retrieve_{R}$ is not internal to $IHM_{DAM}$
**then**
determine a name for $Operation_{GOP}$ and $Parameter_{O}$
**if** an $Operation_{GOP}$ yielding $Parameter_{O}$ is not already provided by $IHM_{DAM}$
**then**
create $Operation_{GOP}$
establish the design relationship $IHM_{DAM}$ Provides $Operation_{GOP}$
record the decision and rationale in the history for $IHM_{DAM}$
create $Parameter_{O}$
establish the design relationship $Operation_{GOP}$ Yields $Parameter_{O}$
record the decision and rationale in the history for $Operation_{GOP}$
**else** use existing $Operation_{GOP}$ and $Parameter_{O}$
**fi**
allocate $Retrieve_{R}$ to $Operation_{GOP}$
**fi**

An example where this rule applies can be found in the cruise control and monitoring system explained by Gomaa. [Gomaa93, Chapter 22] In the example, a data store, Cumulative Distance, forms the basis for a data-abstraction module, Distance. Cumulative Distance is accessed directly via a Retrieve from ten transformations that are not a part of the Distance DAM. The rule specified above recognizes each of these accesses. The first execution of the rule creates a Get_Cumulative_Distance operation that yields a parameter, Cumulative_Distance. Each subsequent execution simply maps a Retrieve onto the existing operation. A similar rule, specified below, addresses direct writes to a data store.

Rule: DAM Put

**if**
an $IHM_{DAM}$ is a data-abstraction module and
$IHM_{DAM}$ is derived from a $Data\ Store_{DS}$ and
$Data\ Store_{DS}$ is the sink of $Store_S$ and
$Store_S$ is not internal to $IHM_{DAM}$
**then**
determine an appropriate name for $Operation_{POP}$ and $Parameter_I$
**if** an $Operation_{POP}$ taking $Parameter_I$ is not already provided by $IHM_{DAM}$
**then**
create $Operation_{POP}$
establish the design relationship $IHM_{DAM}$ Provides $Operation_{POP}$
record the decision and rationale in the history for $IHM_{DAM}$
create $Parameter_I$
establish the design relationship $Operation_{POP}$ Takes $Parameter_I$
record the decision and rationale in the history for $Operation_{POP}$
**else** use existing $Operation_{POP}$ and $Parameter_I$
**fi**
allocate $Store_S$ to $Operation_{POP}$
**fi**

This rule creates a Put_Data operation, where Data is replaced by the name of data flow or, if the data flow has no name, by the name of the data store. The operation takes a parameter with the same name as that used in place of Data in the operation name. An example where this rule applies appears in the robot controller system explained by Gomaa. [Gomaa93, Chapter 23] In this example, the Sensor/Actuator Data Store forms the basis for a data-abstraction module. A device interface module, based upon the transformation Input From Sensors, executes a direct store to the Sensor/Actuator Data Store. The rule specified above creates an operation, Put_Sensor/Acutator_Data_Store, with an input parameter so that the data-abstraction module can provide an operation to write to the data store. To complete direct data store accesses, a final rule, specified below, handles update operations.

Rule: DAM Update

**if**
an $IHM_{DAM}$ is a data-abstraction module and
$IHM_{DAM}$ is derived from a $Data\ Store_{DS}$ and
$Data\ Store_{DS}$ connects to an $Update_{U}$ and
$Update_{U}$ is not internal to $IHM_{DAM}$
**then**
determine an appropriate name for $Operation_{UOP}$ and $Parameter_{IO}$
**if** an $Operation_{UOP}$ taking $Parameter_{IO}$ is not already provided by $IHM_{DAM}$
**then** create $Operation_{UOP}$
establish the design relationship $IHM_{DAM}$ Provides $Operation_{UOP}$
record the decision and rationale in the history for $IHM_{DAM}$
create $Parameter_{IO}$
establish the design relationship $Operation_{UOP}$ Alters $Parameter_{IO}$
record the decision and rationale in the history for $Operation_{UOP}$
**else** use existing $Operation_{UOP}$ and $Parameter_{IO}$
**fi**
allocate $Update_{U}$ to $Operation_{UOP}$
**fi**

A situation where this rule applies can be fabricated by altering some assumptions about Gomaa's cruise control and monitoring system. [Gomaa93, Chapter 22] Assume that, rather than its intended use as local memory for the transformation Determine Distance, the data store Last Distance in the example provides an interface to an external subsystem that is being designed separately. Under such an assumption, Last Distance forms the basis for a separate data-abstraction module. In such a case, the rule specified above creates an Update_Last_Distance operation with an input/output parameter.

### 8.6.5 Rules to Determine Operations from Functions

Module operations can be determined from one other source -- Functions (see Chapter 4 for a definition) allocated to a module that are then accessed from outside the module. Two situations must be recognized. One situation involves a function that receives an event flow or data flow from a transformation within another module or that receives no event flow or data flow at all. A rule for this case is specified below.

Rule: External Function

**if**

$IHM_I$ is an algorithm-hiding module or a function-driver module or a data-abstraction module and
$Function_F$ is allocated to $IHM_I$ and
(($Function_F$ receives a Trigger or Enable or Stimulus or Signal from $Transformation_T$ and $Transformation_T$ is not allocated to $IHM_I$) or
($Function_F$ does not receive any Trigger or Enable or Stimulus or Signal))

**then**

create an $Operation_{EF}$ with the same name as $Function_F$
allocate $Function_F$ to $Operation_{EF}$
establish the design relationship $IHM_I$ Provides $Operation_{EF}$
record the decision and rationale in the history for $IHM_I$

**fi**

Several examples where this rule applies, in various forms, can be found in Gomaa's cruise control and monitoring system case study. [Gomaa93, Chapter 22] Consider for example the information hiding module, Average MPG, composed of two functions, Initialize MPG and Compute Average MPG and a data store, Initial Distance and Fuel Level. One function, Initialize MPG, receives a signal from the transformation Mileage Reset Buttons, which is allocated to a device-interface module. The rule specified above creates an operation based on the function Initialize MPG. The other function, Compute Average MPG, composing Average MPG receives no signal or stimulus or enable or trigger. The rule specified above also creates an operation based on this function.

A second situation to consider involves disables that enter a module. For any module that receives a disable from outside the module, a deactivate operation is created. Multiple disables entering a single module are mapped to a single deactivate operation. For example, in the cruise control and monitoring system discussed above, a function-driver module, Speed Control, is formed from three functions: Maintain Speed, Resume Cruising, and Increase Speed, each of which receives a Disable from the control object, Cruise Control. Since Cruise Control forms the basis for a state-transition module, the disables received by Speed Control originate from outside the Speed Control module. The CODARTS module structuring criteria indicate that a single operation to deactivate a module should be provided for each module that receives a disable from outside the module. [Gomaa93, pp. 231-232] The rule to achieve this mapping is specified below.

Rule: Deactivate Module

**if**
$IHM_I$ is an algorithm-hiding module or a function-driver module or a data-abstraction module and
$Function_F$ is allocated to $IHM_I$ and
$Function_F$ receives a $Disable_D$ from $Transformation_T$ and
$Transformation_T$ is not allocated to $IHM_I$
**then**
**if** $IHM_I$ does not already provide an $Operation_{DOP}$ named Deactivate
**then**
create an $Operation_{DOP}$ named Deactivate
establish the design relationship $IHM_I$ Provides $Operation_{DOP}$
record the decision and rationale in the history for $IHM_I$
**else**
use the existing $Operation_{DOP}$
**fi**
allocate $Disable_D$ to $Operation_{DOP}$
**fi**

Once an operation is created based on a function, the event flows and data flows entering and leaving the function must be analyzed in order to allocate operation parameters, where appropriate. A number of mapping strategies might be considered by a human designer. Because a range of strategies could be adopted, allocation of event flows and data flows to parameters might be left for a human designer. Instead, the approach adopted defines a single set of mappings that can be reviewed later by a designer to consider whether different mappings are preferred. The mappings are achieved through a set of six rules.

One rule identifies cases where an operation is invoked by a single control flow. In such cases, the control flow is not treated as a parameter but, rather, is allocated to the

operation itself. This allocation ensures that these control flows are mapped to an appropriate design element. The rule is specified below.

Rule: External Function Invocation

**if**

$IHM_I$ is an algorithm-hiding module or a function-driver module or a data-abstraction module and
$Function_F$ is allocated to $IHM_I$ and
$Function_F$ is allocated to $Operation_{EF}$ and
$Function_F$ receives an $Arc_A$ from $Transformation_T$ and
$Arc_A$ is a Trigger or an Enable or a Signal and
$Transformation_T$ is not allocated to $IHM_I$ and
$Function_F$ does not receive a $Signal_S$ with a different name from $Arc_A$ from $Transformation_{ANY}$ where $Transformation_{ANY}$ is not allocated to $IHM_I$

**then**

allocate $Arc_A$ to $Operation_{EF}$
record the decision and rationale in the history for $Operation_{EF}$

**fi**

An example where this rule applies appears in the cruise control and monitoring system discussed previously. In the example, a data-abstraction module, Desired Speed, consists of two functions, Select Desired Speed and Clear Desired Speed, and a data store. Each of the functions receives a trigger from a control object, Cruise Control, in a state-transition module. The rule specified above maps each of these triggers to the operation associated with the function stimulated by each trigger. Another example in the same case study can be seen where a single event flow, MPG Reset, is sent from Mileage Reset Buttons to Initialize MPG. The sending and receiving transformations are allocated to different modules, so the rule specified above maps the MPG Reset event flow to the operation associated with Initialize MPG.

Multiple event flows that enter a function associated with an operation are viewed as incoming parameters meant to influence the processing within the operation. A rule to achieve this mapping is specified below.

Rule: Multiple Signals To Function

**if**
an $Operation_{EF}$ is derived from a $Function_F$ and
$IHM_I$ provides $Operation_{EF}$ and
$Function_F$ receives $Signal_{S1}$ and
$Function_F$ receives $Signal_{S2}$, where $Signal_{S2}$ is not SignalS1, and
$Signal_{S1}$ and $Signal_{S2}$ do not have the same name and
$Signal_{S1}$ is not allocated to $IHM_I$ and
$Signal_{S2}$ is not allocated to $IHM_I$ and
$Operation_{EF}$ does not already take a parameter named Condition
**then**
create a $Parameter_C$ named Condition
establish the design relationship $Operation_{EF}$ Takes $Parameter_C$
record the decision and rationale in the history for $Operation_{EF}$
**for** each $Signal_S$ received by $Function_F$ where $Signal_S$ is not allocated to $IHM_I$
allocate $Signal_S$ to $Parameter_C$
record the decision and rationale in the history for $Parameter_C$
**rof**
**fi**

An example where this rule applies can be found in the elevator control system explicated by Gomaa. [Gomaa93, Chapter 24] In the example, a data-abstraction module, Elevator Status and Plan, provides an operation derived from the function Update Status. Update Status receives two distinct event flows, Arrived and Departed. The rule specified above maps these two event flows onto an input parameter named Condition for the operation derived from Update Status.

Another rule simply allocates each data flow arriving at a function composing a module operation to an input parameter. The rule to make these mappings is specified below.

Rule: Stimulus To Function

**if**
an $Operation_{EF}$ is derived from a $Function_{F}$ and
$IHM_{I}$ provides $Operation_{EF}$ and
$Function_{F}$ receives $Stimulus_{S}$ and
$Stimulus_{S}$ is not allocated to $IHM_{I}$ and
**then**
**if** $Operation_{EF}$ does not already take $Parameter_{I}$ with the same name as $Stimulus_{S}$
**then**
create $Parameter_{I}$ with the same name as $Stimulus_{S}$
establish the design relationship $Operation_{EF}$ Takes $Parameter_{I}$
record the decision and rationale in the history for $Operation_{EF}$
**else**
use existing $Parameter_{I}$
**fi**
allocate $Stimulus_{S}$ to $Parameter_{I}$
record the decision and rationale in the history for ParameterI
**fi**

An application of this rule can be illustrated with Gomaa's elevator control system case study. [Gomaa93, Chapter 24] In the example, the transformation Scheduler forms the basis for an algorithm-hiding module containing a single operation. The rule defined above maps each incoming data flow, Elevator Status, Elevator Commitment, and Service Request, to a separate input parameter for the sole operation in the Scheduler module.

Another rule identifies cases where a stimulus to a function is answered by an associated response, and where the stimulus and response have the same name. In such cases, the stimulus and response are mapped to a single input/output parameter for the operation derived from the function. The rule, specified below, is given first preference so that the incoming stimulus and the outgoing response will not be mapped to an input parameter and an output parameter, respectively, by other rules.

Rule: Same Stimulus Response With Function (First Preference)

**if**
an $Operation_{EF}$ is derived from a $Function_{F}$ and
$IHM_{I}$ provides $Operation_{EF}$ and
$Function_{F}$ receives $Stimulus_{S}$ from a $Transformation_{T}$ and
$Function_{F}$ sends a $Response_{R}$ to $Transformation_{T}$ and
$Stimulus_{S}$ has the same name as ResponseR and
$Stimulus_{S}$ is not allocated to $IHM_{I}$ and
$Response_{R}$ is not allocated to $IHM_{I}$
**then**
**if** $Operation_{EF}$ does not already alter $Parameter_{IO}$ with the same name as $Stimulus_{S}$
**then**
create $Parameter_{IO}$ with the same name as $Stimulus_{S}$
establish the design relationship $Operation_{EF}$ Alters $Parameter_{IO}$
record the decision and rationale in the history for $Operation_{EF}$
**else**
use existing $Parameter_{IO}$
**fi**
allocate $Stimulus_{S}$ to $Parameter_{IO}$
allocate $Response_{R}$ to $Parameter_{IO}$
record the decision and rationale in the history for $Parameter_{IO}$
**fi**

No example where the rule applies can be found among the case studies provided by Gomaa but an example can be manufactured by making a slight alteration to the robot

controller system specification. [Gomaa93, Chapter 23] Assume that the function Process Sensor/Actuator Command is replaced with a function Update Sensor Value. Assume further that the function receives a stimulus, Sensor Value, from the function Interpret Program Statement and returns a response, Sensor Value. The rule specified above maps the stimulus and response that are named Sensor Value to an input/output parameter for the operation derived from the function Update Sensor Value.

Two remaining rules provide mappings for event flows and data flows emanating from any function that forms a module operation. The first rule maps such event flows to an output parameter.

Rule: Signals From Function

**if**
an $Operation_{EF}$ is derived from a $Function_F$ and
$IHM_I$ provides $Operation_{EF}$ and
$Function_F$ receives $Signal_S$ and
$Signal_S$ is not allocated to $IHM_I$ and
$Operation_{EF}$ does not yield a $Parameter_O$ named Status
**then**
create a $Parameter_O$ named Status
establish the design relationship $Operation_{EF}$ Yields $Parameter_O$
record the decision and rationale in the history for $Operation_{EF}$
**for** each $Signal_S$ sent by $Function_F$ where $Signal_S$ is not allocated to $IHM_I$
allocate $Signal_S$ to $Parameter_O$
record the decision and rationale in the history for $Parameter_O$
**rof**
**fi**

An example where this rule applies appears in Gomaa's elevator control system case study. [Gomaa93, Chapter 24] In the example, the function Check Next Destination emits

three event flows, Down Request, Up Request, and No Request. The function Check Next Destination becomes an operation provided by a data-abstraction module. The rule specified above creates a output parameter named Status for the operation and allocates the three event flows to that parameter.

A final rule maps each data flow, whether stimulus or response, emitted from a function associated with an operation to an output parameter for the operation. The rule is specified below.

Rule: Stimulus Or Response From Function

**if**
an $Operation_{EF}$ is derived from a $Function_F$ and
$IHM_I$ provides $Operation_{EF}$ and
$Function_F$ sends an $Arc_A$ to a $Transformation_T$ and
$Arc_A$ is a Stimulus or Response and
$Arc_A$ is not allocated to $IHM_I$ and
no Response goes from $Transformation_T$ to $Function_F$
**then**
**if** $Operation_{EF}$ does not already yield a $Parameter_O$ with the same name as $Arc_A$
**then**
create a $Parameter_O$ with the same name as $Arc_A$
establish the design relationship $Operation_{EF}$ Yields $Parameter_O$
record the decision and rationale in the history for $Operation_{EF}$
**else**
use existing $Parameter_O$
**fi**
allocate $Arc_A$ to $Parameter_O$
record the decision and rationale in the history for $Parameter_O$
**fi**

A case where this rule applies can be found in Gomaa's cruise control and monitoring system case study. [Gomaa93, Chapter 22] In the example, the function Determine

Distance leads to an operation provided by the data-abstraction module named Distance. Determine Distance emits a data flow named Incremental Distance. The rule specified above maps Incremental Distance to an output parameter for the operation derived from Determine Distance.

### 8.7 Review Module Structure and Consider Renaming Modules

After all the transformations and data stores that can be allocated to modules are allocated, the module structuring for the evolving design is essentially complete. The designer is then given an opportunity to review the module structure. If the designer is dissatisfied with the results, then they can be discarded. In addition, the designer is given an opportunity to rename any module. A single rule, not shown here, drives the module review and renaming that completes the structuring of modules.