

## **Chapter 9 Task and Module Integration**

Once the modules and tasks composing a concurrent design are decided, attention must be given to the relationships among the tasks and modules. Modules that are not shared among multiple tasks can be placed, logically, within the task that drives the module, while modules shared by multiple tasks can be placed, logically, outside of any task. This concept of logical placement is simply a means to denote which modules are shared by multiple tasks and which are not. In addition, some modules must remain outside of tasks because they serve other modules that are shared among multiple tasks. Consideration of these issues results in the generation of a software architecture for the evolving concurrent design. The required knowledge is contained in the Task and Module Integration Knowledge base, identified in Chapter 3 of this dissertation.

The Task and Module Integration Knowledge base is assigned the following goals: 1) to determine which information hiding modules in the evolving design should be placed inside tasks and which should be placed outside tasks, 2) to determine, for modules placed outside tasks, the modules accessed and the specific operations invoked by each task, 3) to determine, among the modules placed outside tasks, which modules serve other modules and which specific operations are required by other operations, and 4) to rectify parameter passing for modules, external to tasks, that directly access data stores within modules internal to tasks. To accomplish these goals, the Task and Module

Task And Module Integration Knowledge  
(3 Decision-Making Processes Comprising 17 Rules)

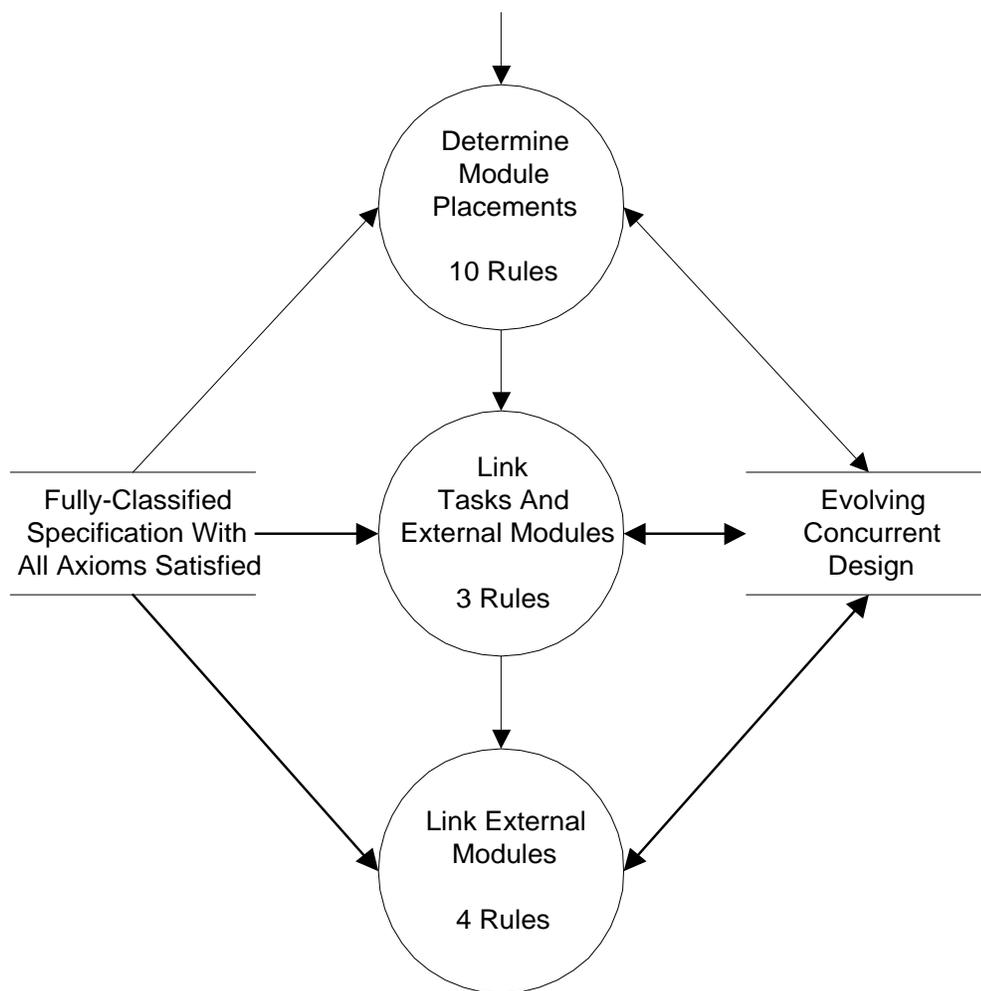


Figure 28. Organization of Task and Module Integration Knowledge

Integration Knowledge base is organized as three decision-making processes, shown in Figure 28. Generation of the software architecture relies upon information contained in a fully-classified data/control flow diagram and on the state of the evolving design. The main information created by the Task and Module Integration Knowledge base can include the following types of architectural relationships, as drawn from the design meta-model defined in Chapter 5 of this dissertation:

- 1) Contains (Tasks contain Modules),
- 2) Accesses (Tasks access Modules),
- 3) Serves (Modules serve other Modules),
- 4) Invokes (Tasks invoke Operations), and
- 5) Requires (Operations require other Operations).

Design-decision rules compose each of the three decision-making processes that lead to a software architecture. The rules composing each process are specified below, beginning with the process that determines module placements.

### **9.1 Determine Module Placements**

The placement of each IHM generated during module structuring must be determined.<sup>1</sup> In general, modules can be analyzed by type. Each device-interface module, for periodic and asynchronous devices, as well as each state-transition module

---

<sup>1</sup>Note that these are decisions about logical placement, not about physical packaging. A module can be placed logically either inside a task or outside any task. Placement within a task denotes that the module is accessed only by that task, while placement outside any task denotes that the module is accessed by multiple tasks. Decisions about physical packaging should be put off until later, when design configuration is addressed.

and each module that provides either user-interface or subsystem-interface services, can be placed directly inside a task. These types of modules are discussed below in Section 9.1.1, Captive Modules. Other types of modules, including device-interface modules for passive devices, as well as data-abstraction, algorithm-hiding, and function-driver modules, discussed in Section 9.1.2 below, Shareable Modules, must be placed outside a task when they are shared, but can be placed within a task when only one task accesses the module. Design-decision rules are required to recognize all of these situations.

### **9.1.1 Captive Modules**

The CODARTS design method provides heuristics for identifying modules that can be placed directly inside a task based on the fact that the task exists primarily to manage the execution of the module. These include modules for asynchronous and periodic devices, modules that contain a state-transition diagram, and modules that provide an interface to external subsystems and to user roles. Design-decision rules are specified below to recognize each of these situations

#### **9.1.1.1 Rules for Placing DIMs for Asynchronous and Periodic Devices**

One CODARTS heuristic places each DIM for an asynchronous device within the task that handles the interrupt for that device. [Gomaa93, pp. 239-240] Another heuristic places each DIM for a polled device within the task that polls the device. [Gomaa93, pp. 240-242] Each of these heuristics leads to a design-decision rule, as specified below.

Rule: Place DIMs For Asynchronous Devices

**if**

IHM<sub>DIM</sub> is a device interface module and  
 IHM<sub>DIM</sub> is derived from an Asynchronous Device Interface Object<sub>ADIO</sub> and  
 Task<sub>AIO</sub> is derived from Asynchronous Device Interface Object<sub>ADIO</sub>

**then**

establish the design relationship Task<sub>AIO</sub> Contains IHM<sub>DIM</sub>  
 record the decision and rationale in the design history for IHM<sub>DIM</sub>

**fi**

An example where this rule applies can be found in the cruise control and monitoring system case study explicated by Gomaa. [Gomaa93, Chapter 22] In the example, a DIM, CC Lever, and a Task, Monitor Cruise Control Lever, are established based on the same asynchronous device interface object, Cruise Control Lever. The rule specified above places the DIM, CC Lever, within the Task, Monitor Cruise Control Lever. A similar rule is specified for periodic devices.

Rule: Place DIMs For Periodic Devices

**if**

IHM<sub>DIM</sub> is a device interface module and  
 IHM<sub>DIM</sub> is derived from a Periodic Device Interface Object<sub>PDIO</sub> and  
 Task<sub>PID</sub> is derived from Periodic Device Interface Object<sub>PDIO</sub>

**then**

establish the design relationship Task<sub>PID</sub> Contains IHM<sub>DIM</sub>  
 record the decision and rationale in the design history for IHM<sub>DIM</sub>

**fi**

An example where this rule applies can also be found in the case study of the cruise control monitoring system. In the example, two DIMs, Brake and Engine, are created based on the existence of two periodic input device objects of the same name. In addition, a Task, Monitor Auto Sensors, is allocated to poll the Brake and Engine. The

rule specified above will place both the Brake DIM and the Engine DIM inside the Task, Monitor Auto Sensors.

### 9.1.1.2 Rule for Placing State-Transition Modules

The CODARTS criteria for integrating IHMs into internal tasks note that each state-transition module is always accessed by a single task and should, thus, be placed inside that accessing task. [Gomaa93, p. 234] The following rule reflects this guidance.

Rule: Place STM

**if**

IHM<sub>STM</sub> is a state transition module and  
 IHM<sub>STM</sub> is derived from a Control Object<sub>CO</sub> and  
 Task<sub>C</sub> is derived from Control Object<sub>CO</sub>

**then**

establish the design relationship Task<sub>C</sub> Contains IHM<sub>STM</sub>  
 record the decision and rationale in the design history for IHM<sub>STM</sub>

**fi**

Two applications of this rule appear in the cruise control and monitoring system discussed by Gomaa. [Gomaa93, Chapter 22] In the example, two state-transition modules (STMs) are defined, one based on the control object Cruise Control and the other based on the control object Calibration Control. Each of these control objects gets mapped to a separate task: Cruise Control and Perform Calibration, respectively. The design-decision rule defined above places each of the state-transition modules inside the task derived from the same control object: the Cruise Control STM is placed inside the task Cruise Control and the Calibration Control STM is placed inside the task Perform Calibration.

### 9.1.1.3 Rules for Placing User and Subsystem Interface Modules

Two additional rules extend the CODARTS heuristic for placing DIMs for asynchronous and polled devices to cover the cases of user-interface modules (UIMs) and subsystem interface modules (SIMs). Since all accesses to modules of these types will be made within the task that drives the interface, each of these IHMs should be placed within the task that interacts with the appropriate user-role and external subsystem. The corresponding design-decision rules are specified below.

Rule: Place UIM

**if**

IHM<sub>UIM</sub> is a user interface module and  
 IHM<sub>UIM</sub> is derived from an User-Role Interface Object<sub>URIO</sub> and  
 Task<sub>UIO</sub> is derived from User-Role Interface Object<sub>URIO</sub>

**then**

establish the design relationship Task<sub>UIO</sub> Contains IHM<sub>UIM</sub>  
 record the decision and rationale in the design history for IHM<sub>UIM</sub>

**fi**

An example where this rule might apply can be found in the distributed factory automation system case study presented by Goma. [Goma93 Chapter 25] In this example, a task is created to drive each instance of a user-role, including Process Engineer, Process Manager, and Factory Operator. A corresponding UIM is defined to manage a dialog with each user role. Since the UIM is only accessed within the task that interacts with the user role, each UIM is placed within the corresponding task.

A similar situation might exist in this example for an external subsystem. Assuming that each Line Workstation Controller subsystem is self-contained and that

such subsystems can be combined sequentially to form an assembly line, then the object, Predecessor Workstation Interface, can be viewed as an external subsystem interface object, this gives rise to a subsystem interface module, SIM, defined during module structuring, and to a subsystem interface task, established during task structuring. Since the SIM is accessed only within the task, the SIM should be placed within the task. The following rule reflects this heuristic.

Rule: Place SIM

**if**

IHM<sub>SIM</sub> is a subsystem interface module and  
 IHM<sub>SIM</sub> is derived from a Subsystem Interface Object<sub>SIO</sub> and  
 Task<sub>SIO</sub> is derived from Subsystem Interface Object<sub>SIO</sub>

**then**

establish the design relationship Task<sub>SIO</sub> Contains IHM<sub>SIM</sub>  
 record the decision and rationale in the design history for IHM<sub>SIM</sub>

**fi**

This rule is not necessary for systems that assume support for distributed message passing among tasks on different nodes in a network. In such systems, the services provided within a SIM are provided directly by the distributed operating system. Since this dissertation does not directly address designs for distributed systems, this rule generates SIMs to encapsulate any special-purpose message passing code.

### 9.1.2 Shareable Modules

The placement of function-driver modules, algorithm-hiding modules, data-abstraction modules, and device-interface modules for passive devices depends upon how such modules are accessed by tasks. The fundamental criterion for placing a module

of these types inside a task is that the module be accessed exclusively by that task. [Gomaa93, pp. 243-244] Whenever a module is accessed by multiple tasks, either directly or indirectly, then the module must be placed outside any task.

When analyzing a data/control flow diagram and the related, evolving design, a module can be recognized as being accessed exclusively by one task when that task is the only task the sends an input to the module and either: 1) the task has a cardinality of one or 2) the task and the module both have an equal cardinality. The first case occurs in situations where a single-instance task accesses either a single-instance or multiple-instance module. Here, the module can be placed inside the task because the task is the sole accessor of the module. The second case occurs when a multiple-instance task accesses a multiple-instance module, and both the task and the module have the same cardinality. In this case, the module can be placed inside the task on the assumption that each instance of the task will contain one instance of the module. This general analysis applies to function-driver modules, algorithm-hiding modules, and data-abstraction modules. The situation with device-interface modules for passive devices becomes more complicated because sets of devices can be aggregated together with other components in a system. For example, each elevator in an elevator control system might be required to have a door, a motor, and a set of lamps. In cases such as this, the modules should be grouped together within the same task regardless of their relative cardinalities. Design-decision rules are specified below for each of these situations.

### 9.1.2.1 Rule for Placing Data-Abstraction Modules

Data-abstraction modules, or DAMs, can be placed based on the same criteria used to place function-driver and algorithm-hiding modules; however, since DAMs include data stores, the analysis applied to the data/control flow diagram and evolving design requires consideration of a different set of details. This requirement leads to a separate design-decision rule for placing DAMs. The rule is specified below.

Rule: Place DAMs

```

if
  IHMDAM is a data-abstraction module
then
  if IHMDAM is accessed by multiple tasks
  then
    denote that IHMDAM is not contained within a task
  else
    if the cardinality of the accessing TaskAT exceeds one but
      does not equal the cardinality of IHMDAM
    then
      denote that IHMDAM is not contained within a task
    else
      establish the design relationship TaskAT Contains IHMDAM
      record the decision and rationale in the design history for IHMDAM
    fi
  fi
fi

```

This rule applies to each DAM in an evolving design. Access to the DAM is first analyzed to determine if separate tasks, apart from considerations of cardinality, access the DAM. If so, the DAM is placed outside any task. If only one task accesses the

DAM, then the issue of cardinality is examined. If the sole accessing task has a cardinality of one or if the cardinality of the sole accessing task is identical to the cardinality of the DAM, then the DAM is placed within the task. In the case of incompatible cardinalities, then the DAM is placed outside any task.

Determining if a DAM is accessed by multiple tasks becomes quite complex. Any task in the evolving design accesses a DAM whenever one of the following conditions exists:

- ◆ The specification element(s) from which the task and the DAM derive intersect.
- ◆ The task is derived from a solid transformation in the data/control flow diagram and either: 1) that solid transformation writes to a data store within the DAM, 2) that solid transformation reads from a data store within the DAM, or 3) that solid transformation updates a data store within the DAM.

Each task in the design that meets one of these conditions with respect to a given DAM is an accessor of the DAM. If more than one task meets one of these conditions, then the DAM is accessed by multiple tasks.

An example where this rule detects a shared DAM can be found in the cruise control system offered by Goma. [Gomaa93, Chapter 22] In the example, a DAM, Desired Speed, is formed from a data store, Desired Speed, and two transformations, Select Desired Speed and Clear Desired Speed. A task, Cruise Control, is also derived from these two transformations; thus, the task is an accessor of the DAM, Desired Speed. Another task, Auto Speed Control, is derived from three transformations: Resume

Cruising, Increase Speed, and Maintain Speed. Although none of these three transformations form any part of the Desired Speed DAM, two of the transformations, Resume Cruising and Maintain Speed, do read information from the data store contained within the DAM. This means that the task Auto Speed Control is also an accessor of the Desired Speed DAM. Since Desired Speed is accessed by two tasks the DAM must be placed outside any task.

An example where this rule detects a captive DAM can be seen in a robot controller case study discussed by Gomaa. [Gomaa93, Chapter 23] In this example, a DAM, Robot Program, is formed solely from a data store of the same name. The DAM is accessed via a read from the transformation Interpret Program Statement. A task, Interpreter, is derived in part from Interpret Program Statement. The task is the sole accessor of Robot Program, so the DAM can be placed within the task.

#### **9.1.2.2 Rules for Placing Remaining Information-Hiding Modules**

Placement decisions regarding function-driver and algorithm-hiding modules and device-interface modules for passive devices require four design-decision rules. Each rule is assigned a preference value in order to simplify the specification of the rules. By considering each situation in turn, lower preference rules do not have to exclude explicitly cases already recognized by higher preference rules. The first rule, specified below, recognizes when a relevant module is accessed by multiple tasks of different types.

Rule: Place Selected Shared IHMs (Second Preference)

**if**

IHM<sub>1</sub> is a function-driver, device-interface, or  
algorithm-hiding module and  
IHM<sub>1</sub> is derived from a Specification Element<sub>SE1</sub> and  
IHM<sub>1</sub> is derived from a Specification Element<sub>SE2</sub> and  
Task<sub>T1</sub> is derived from Specification Element<sub>SE1</sub> and  
Task<sub>T2</sub> is derived from Specification Element<sub>SE2</sub>  
(where Task<sub>T1</sub> is not Task<sub>T2</sub>)

**then**

denote that IHM<sub>1</sub> is not contained within a task  
record the decision and rationale in the history for IHM<sub>1</sub>

**fi**

This rule considers the elements of a single module and of two different tasks. Whenever any element or elements of the module are also elements of two separate tasks, then the module is shared and must be placed outside of any task. Assigning this rule second preference achieves two objectives. First, other rules dealing with device-interface modules for periodic and asynchronous devices take precedence over this rule, so only device-interface modules for passive devices remain to be allocated when this rule becomes effective. Second, rules with lower preference values than this rule can be specified more simply because all cases of shared access by two different tasks will have already been considered before the less preferred rules.

An example where this rule applies appears in the cruise control and monitoring system as described by Gomaa. [Gomaa93, Chapter 22] In the example, a passive device input object, Gas Tank, is accessed from two transformations: Initialize MPG and Compute

Average MPG. Each of these transformations compose part of a different task: Monitor Reset Buttons and Compute Average Mileage, respectively. Using the rule specified above, the DIM derived from the Gas Tank is placed outside any task.

A separate rule recognizes when a device-interface module for a passive device belongs to an aggregation group, as specified in the input specification. In other cases, device-interface modules for passive devices are treated the same as algorithm-hiding and function-driver modules; however, when a passive device belongs to an aggregation group, issues of cardinality are ignored and the device-interface module is placed inside the same tasks as other members in its aggregation group. The rule to recognize these cases is specified below.

Rule: Place DIMs For Aggregated, Passive Devices

(Third Preference)

**if**

IHM<sub>DIM</sub> is a device interface module and  
 IHM<sub>DIM</sub> is derived from a Passive Device Interface Object<sub>PDIO</sub> and  
 (Passive Device Interface Object<sub>PDIO</sub> receives an Input from Device<sub>D</sub>  
 or sends an Output to Device<sub>D</sub>) and

Task<sub>C</sub> is derived from a Control Object<sub>CO</sub> and  
 IHM<sub>STM</sub> is derived from Control Object<sub>CO</sub> and

Task<sub>C</sub> contains IHM<sub>STM</sub> and

Control Object<sub>CO</sub> and Device<sub>D</sub> are members of the same **Aggregation  
 Group**

**then**

establish the design relationship Task<sub>C</sub> Contains IHM<sub>DIM</sub>  
 record the decision and rationale in the design history for IHM<sub>DIM</sub>

**fi**

An example where this rule applies appears in the elevator control system case study considered by Gomaa. [Gomaa93, Chapter 24] In the example, each physical elevator in the application is composed of: 1) elevator control logic (represented by the control object, Elevator Control), 2) a motor (represented by the device, Elevator Motor), 3) a door (represented by the device, Elevator Door), 3) one elevator request button per floor (represented by the device, Elevator Buttons), and 4) one elevator lamp per floor (represented by the device, Elevator Lamps). These elements from the data/control flow diagram lead to the creation of several modules. Of particular interest for the current rule are the DIMs for the passive device input/output objects, Motor and Door, and for the passive output device, Elevator Lamps. Since these DIMs interface to devices that are replicated with each elevator, the DIMs should be placed, whenever possible, within any task that is created to drive an elevator. The rule specified above ensures that this placement is made.

Another rule for placing shareable modules recognizes when a relevant module is accessed by a single, multiple-instance task that has a cardinality unequal to the cardinality of the module. The rule is specified below.

Rule: Place Selected IHMs With Incompatible Cardinality (Fourth Preference)

**if** IHM<sub>I</sub> is a function-driver, device-interface,  
or algorithm-hiding module and  
IHM<sub>I</sub> is derived from a Specification Element<sub>SE</sub> and  
Task<sub>T</sub> is derived from Specification Element<sub>SE</sub> and  
the cardinality of Task<sub>T</sub> exceeds one and is not equal to the  
cardinality of IHM<sub>I</sub>  
**then** denote that IHM<sub>I</sub> is not contained within a task  
record the decision and rationale in the history for IHM<sub>I</sub>  
**fi**

This rule finds any multiple-instance task that accesses a relevant module and that has a cardinality that differs from that of the module. Situations where this rule would apply should be extremely rare, however, the rule is included to increase the scope of situations that are covered. For example, this rule prevents a task with three instances from containing an algorithm-hiding module with five instances, forcing the algorithm-hiding module to be placed outside all accessing tasks and then shared among them. This rule is given fourth preference in order that the next rule can be specified simply to assume that any remaining tasks that access a relevant module have a compatible cardinality.

The final rule that deals with the placement of shareable modules can simply assume, by process of elimination, that any unplaced module of a relevant type is contained by an accessing task because the cardinalities of the task and module must be compatible, that is, either the cardinality of the accessing task is one or the cardinality of the accessing task and the accessed module are identical. The rule is specified below.

Rule: Place Selective Captive IHMs (Fifth Preference)

**if**

IHM<sub>i</sub> is a function-driver, device-interface,  
or algorithm-hiding module and  
IHM<sub>i</sub> is derived from a Specification Element<sub>SE</sub> and  
Task<sub>T</sub> is derived from Specification Element<sub>SE</sub>

**then**

establish the design relationship Task<sub>T</sub> Contains IHM<sub>i</sub>  
record the decision and rationale in the design history for IHM<sub>i</sub>

**fi**

An example where this rule applies can be found in the cruise control and monitoring system presented by Gomaa. [Gomaa93, Chapter 22] In the example, a function-driver module, Speed Control, is formed from three transformations: Resume Cruising, Increase Speed, and Maintain Speed. This module is accessed solely by the single-instance task, Auto Speed Control; thus, the Speed Control module is placed within the Auto Speed Control task.

## **9.2 Link Tasks and External Modules**

After module placement decisions are completed, some modules, called external modules, remain outside any task. In order to determine the specific operations, provided by external modules, that are invoked from each task, the tasks and external modules composing the evolving design must be examined. Any task that invokes at least one operation in an external module can also be said to access that module. Three rules are defined to draw the needed inferences. One rule links a transformation allocated to a module contained in a task to a transformation or to directed arcs already allocated to an operation provided by an external module. The rule, although somewhat complicated, identifies the two main cases where a task invokes directly an operation in an external module, while excluding cases, dealt with in another rule, where an invocation by a task flows from one external module to another. One case where a task can be inferred to invoke an operation in an external module occurs when a transformation is allocated both to a task and to an operation in an external module. An example of such a case appears in the cruise control and monitoring system explained by Gomaa. [Gomaa93, Chapter 22] In the

example, an external data-abstraction module, Distance, provides an operation, Determine Distance, derived from a transformation of the same name, and a task, Determine Distance and Speed, is derived from the same transformation. Since the transformation Determine Distance does not receive a signal, stimulus, or control flow from any transformation that is allocated to both the task Determine Distance and Speed and to an external module other than Determine Distance, the task Determine Distance and Speed can be inferred to invoke directly the operation Determine Distance. A complex rule, specified below, recognizes such situations.

Rule: Invocation Via Transformation

**if**

IHM<sub>EM</sub> is not contained in any task and  
 IHM<sub>EM</sub> provides an Operation<sub>EF</sub> and  
 Transformation<sub>OP</sub> is allocated to Task<sub>T</sub> and  
 (Transformation<sub>OP</sub> is allocated to Operation<sub>EF</sub> or  
 (Transformation<sub>OP</sub> is the sink of Directed-Arc<sub>CALL</sub> and  
 Directed-Arc<sub>CALL</sub> is a Signal or Disable or Stimulus and  
 Directed-Arc<sub>CALL</sub> is allocated to Operation<sub>EF</sub>)) and  
 (Transformation<sub>OP</sub> does not receive a Directed-Arc<sub>OA</sub>,  
 where Directed-Arc<sub>OA</sub> is a Signal or Control-Event-Flow or Stimulus,  
 from a Transformation<sub>CALLER</sub> that is allocated both to Task<sub>T</sub> and  
 to IHM<sub>OM</sub>, where IHM<sub>OM</sub> is not IHM<sub>EM</sub> and where IHM<sub>OM</sub> is not  
 contained in any task) and  
 Task<sub>T</sub> does not already invoke Operation<sub>EF</sub>

**then**

establish the design relationship Task<sub>T</sub> Invokes Operation<sub>EF</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>

**if** Task<sub>T</sub> does not already access IHM<sub>EM</sub>

**then** establish the design relationship Task<sub>T</sub> Accesses IHM<sub>EM</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>

**fi**

**fi**

A second case where the same inference can be drawn occurs when a transformation is allocated to a task and where that transformation receives a signal, disable, or stimulus that is allocated to an operation in an external module. No example of this second case is readily available in Gomaa's case studies; however, the case can be explained by making an assumption about Gomaa's cruise control and monitoring system case study. Assume that the module Speed Control was placed outside any task. The module contains an operation, Deactivate, that is derived from three disables, where each disable arrives at a separate transformation, Resume Speed, Increase Speed, and Maintain Speed. Under these assumptions, the Deactivate operation is invoked by each task. Cases such as this are also covered by the rule.

Another rule, specified below, recognizes situations where a task invokes an operation derived (in Chapter 8) from direct access to a data store.

Rule: Invocation Via Data Store

**if**

IHM<sub>ANY</sub> is contained in Task<sub>T</sub> and  
 Transformation<sub>DSA</sub> is allocated to IHM<sub>ANY</sub> and  
 Transformation<sub>DSA</sub> connects via Arc<sub>A</sub> to a data store and  
 Arc<sub>A</sub> is allocated to Operation<sub>DSOP</sub> and  
 IHM<sub>DAM</sub> provides Operation<sub>DSOP</sub> and  
 IHM<sub>DAM</sub> is not contained in any task and  
 Task<sub>T</sub> does not already invoke Operation<sub>DSOP</sub>

**then**

establish the design relationship Task<sub>T</sub> Invokes Operation<sub>DSOP</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>

**if** Task<sub>T</sub> does not already access IHM<sub>DAM</sub>

**then** establish the design relationship Task<sub>T</sub> Accesses IHM<sub>DAM</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>

**fi**

**fi**

An example where this rule applies can be found in Gomaa's cruise control and monitoring system case study. [Gomaa93, Chapter 22] In the example, a module, Speed Control, is contained in the Auto Speed Control task. Three transformations, Resume Speed, Increase Speed, and Maintain Speed, are allocated to Speed Control. Two of these transformations, Resume Speed and Maintain Speed, read the desired speed by directly accessing a data store contained within an external data-abstraction module (that is, a data abstraction module placed outside any task). Each of these read accesses is allocated to a Get operation on the data-abstraction module Desired Speed; thus, the task, Auto Speed Control, accesses the module Desired Speed and invokes the Get operation provide by the module Desired Speed.

Another situation can occur where a task invokes an operation in an external module that in turn passes some data to an operation in another external module. For example, in Gomaa's cruise control and monitoring system case study an external data-abstraction module, Distance, provides an operation, Compute, derived from a transformation, Determine Distance. Determine Distance passes a data flow, Incremental Distance, to a transformation Determine Speed, which is the basis for an operation Update in another external data-abstraction module, Current Speed. This situation could be resolved in several ways in a design. One possibility, the one shown in Gomaa's case study, assigns an operation, Read Incremental Distance, to the Determine Distance module, which the Update operation in the Current Speed module can use as required. In this case, both the Compute operation in the Distance module and the Update operation in

the Current Speed module are called directly from the Determine Distance and Speed task. In a variant of this approach, not addressed in Gomaa's case study, the operation Compute in the Distance module could return Incremental Distance as a parameter when invoked by the Determine Distance and Speed task. The Determine Distance and Speed task could then pass Incremental Distance as an input parameter when invoking the Update operation in the Current Speed module. A third approach, also not addressed in Gomaa's case study, has the Compute operation in the Distance module invoke the Update operation directly in the Current Speed module, passing Incremental Distance as an input parameter. Rather than present the designer with multiple options, the following rule adopts the second approach as a standard solution.

Rule: Invocation Via Parameter Matching

**if**

Task<sub>ANY</sub> invokes an Operation<sub>A</sub> and  
 Transformation<sub>S</sub> is allocated to Operation<sub>A</sub> and to IHM<sub>I</sub> and  
 Transformation<sub>R</sub> is allocated to Operation<sub>B</sub> (where Transformation<sub>R</sub>  
 is not Transformation<sub>S</sub> and Operation<sub>A</sub> is not Operation<sub>B</sub>) and to  
 IHM<sub>J</sub> (where IHM<sub>J</sub> is not IHM<sub>I</sub>) and  
 IHM<sub>J</sub> is not contained in any task and  
 Transformation<sub>S</sub> sends an Arc<sub>DATA</sub> to Transformation<sub>R</sub> and  
 Arc<sub>DATA</sub> is a Stimulus or a Signal and  
 Arc<sub>DATA</sub> is allocated to a Parameter<sub>O</sub> yielded by Operation<sub>A</sub> and  
 Arc<sub>DATA</sub> is allocated to a Parameter<sub>I</sub> taken by Operation<sub>B</sub> and  
 Task<sub>T</sub> does not already invoke Operation<sub>B</sub>

**then**

establish the design relationship Task<sub>T</sub> Invokes Operation<sub>B</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>  
**if** Task<sub>T</sub> does not already access IHM<sub>J</sub>  
**then** establish the design relationship Task<sub>T</sub> Accesses IHM<sub>J</sub>  
 record the decision and rationale in the history for Task<sub>T</sub>

**fi**

**fi**

### 9.3 Link External Modules

Further analysis is required to determine when an operation provided by one external module requires an operation provided by another external module. In such cases, the second external module can be said to serve the first external module in a client-server relationship. Two rules are defined to make the needed inferences. One rule, specified below, establishes when one operation requires another by recognizing arcs that connect transformations in different external modules, where the connected transformations are each allocated to a module operation.

Rule: Transformation Requires Transformation

**if**

Transformation<sub>S</sub> is allocated to IHM<sub>CLIENT</sub> and to Operation<sub>CALLING</sub> and  
 IHM<sub>CLIENT</sub> is not contained in any task and  
 Transformation<sub>R</sub> is allocated to IHM<sub>SERVER</sub> and to Operation<sub>CALLED</sub> and  
 IHM<sub>SERVER</sub> is not contained in any task and  
 IHM<sub>CLIENT</sub> is not IHM<sub>SERVER</sub> and  
 Transformation<sub>S</sub> is not Transformation<sub>R</sub> and  
 Operation<sub>CALLING</sub> is not Operation<sub>CALLED</sub> and  
 (Transformation<sub>S</sub> sends a Arc<sub>A</sub> to Transformation<sub>R</sub>, where Arc<sub>A</sub> is a  
 Signal, or (Transformation<sub>S</sub> sends an Arc<sub>A</sub> to Transformation<sub>R</sub>, where  
 Arc<sub>A</sub> is a Stimulus, and Transformation<sub>R</sub> sends a Response to  
 Transformation<sub>S</sub>)) and  
 Arc<sub>A</sub> is allocated to Operation<sub>CALLED</sub> and  
 Arc<sub>A</sub> is not allocated to Operation<sub>CALLING</sub> and  
 Operation<sub>CALLING</sub> does not already require Operation<sub>CALLED</sub>

**then**

establish the design relationship Operation<sub>CALLING</sub> Requires Operation<sub>CALLED</sub>  
 record the decision and rationale in the history for Operation<sub>CALLING</sub>

**if** IHM<sub>SERVER</sub> does not already serve IHM<sub>CLIENT</sub>

**then**

establish the design relationship IHM<sub>SERVER</sub> Serves IHM<sub>CLIENT</sub>  
 record the decision and rationale in the history for IHM<sub>SERVER</sub>

**fi**

**fi**

An example where this rule applies appears in the cruise control and monitoring case study provided by Goma. [Goma93, Chapter 22] In the example, two external modules, a device-interface module for a Gas Tank and a data-abstraction module, Average MPG, exhibit a suitable relationship. Two transformations, Initialize MPG and Compute MPG, each allocated to separate operations provided by Average MPG, send the stimulus Fuel Request to, and receive the response Fuel Level from, the transformation Gas Tank, allocated to the device-interface module. The rule specified above will recognize that both the operation derived from the transformation named Initialize MPG and the operation derived from the transformation named Compute MPG require the operation, Get\_Fuel\_Level, derived from the stimulus-response pair, Fuel Request and Fuel Level. The rule will also establish that the device-interface module, Gas Tank, serves the data-abstraction module, Average MPG.

A separate rule is needed to recognize that an operation in one external module requires an operation in another external module when the required operation results from direct access to a data store. Cases where this type of relationship exists can be found in the cruise control and monitoring case study described by Goma. [Goma93, Chapter 22] For example, an external data-abstraction module, Calibration, provides two operations, Start and Stop, that each result from transformations that read directly from a data store contained inside another external data-abstraction module, Shaft Rotation Count. The direct reads of the data store indicate that both the Start operation and the Stop operation provided by the Calibration module require the Read operation provided by the Shaft

Rotation Count module, thus; the Shaft Rotation Count module serves the Calibration module. A rule to recognize and act upon such situations is specified below.

Rule: Transformation Requires Data Store

**if**

Transformation<sub>s</sub> is allocated to IHM<sub>CLIENT</sub> and to Operation<sub>CALLER</sub> and  
 IHM<sub>CLIENT</sub> is not contained in any task and  
 Transformation<sub>s</sub> connects to a Data Store via Arc<sub>DATA</sub> and  
 Arc<sub>DATA</sub> is allocated to Operation<sub>CALLED</sub> and  
 IHM<sub>SERVER</sub> provides Operation<sub>CALLED</sub> and  
 IHM<sub>SERVER</sub> is not contained in any task and  
 Operation<sub>CALLER</sub> does not already require Operation<sub>CALLED</sub>

**then**

establish the design relationship Operation<sub>CALLER</sub> Requires Operation<sub>CALLED</sub>  
 record the decision and rationale in the history for Operation<sub>CALLER</sub>

**if** IHM<sub>SERVER</sub> does not already serve IHM<sub>CLIENT</sub>

**then**

establish the design relationship IHM<sub>SERVER</sub> Serves IHM<sub>CLIENT</sub>  
 record the decision and rationale in the history for IHM<sub>SERVER</sub>

**fi**

**fi**

Another interesting situation can arise involving modules external to any task. Suppose, the Calibration DAM, referred to in several earlier examples involving a cruise control and monitoring system, had not been created, but that, instead, two separate modules, Calibration Start Count and Calibration Constant are created. This situation arises when an experienced designer decides not to combine the two modules, or if no experienced designer is available. In this case, one DAM, Calibration Start Count, consists of a transformation, Record Calibration Start, and a data store, Calibration Start Count, while the second DAM consists of a transformation, Compute Calibration Constant, and a data store, Calibration Constant. During module placement, one module,

Calibration Start Count, is placed within a task, Perform Calibration, while the other module, Calibration Constant, is placed outside any task, because multiple tasks, including Perform Calibration and Determine Distance and Speed, access the module. Note that the transformation, Compute Calibration Constant, requires access to the data from a data store, Calibration Start Count, within a module contained within a task, Perform Calibration. Rather than have Compute Calibration Constant access Calibration Start Count using a direct read, a rule can be defined to rectify the parameters used by any operation that requires data from a data store within a task so that the needed data can be passed to the operation as an input parameter. The necessary rule is specified below.

Rule: Operation Takes Data

```

if
    IHMEXT is not contained within a task and
    OperationOP, provided by IHMEXT, is derived from TransformationT and
    IHMINT, derived from Data StoreDS, is contained within a task and
    TransformationT reads data from Data StoreDS and
    OperationOP Takes no appropriate input parameter
then
    create and assign a name to a ParameterIN
    establish the design relationship OperationOP Takes ParameterIN
    record the decision and rationale in the design history for OperationOP
fi

```

A similar situation can occur where a write to a data store within a task can be mapped instead to an output parameter from an operation provided by a module external to a task. The following rule addresses such situations.

Rule: Operation Yields Data

**if**

IHM<sub>EXT</sub> is not contained within a task and  
Operation<sub>OP</sub>, provided by IHM<sub>EXT</sub>, is derived from Transformation<sub>T</sub> and  
IHM<sub>INT</sub>, derived from Data Store<sub>DS</sub>, is contained within a task and  
Transformation<sub>T</sub> writes data to Data Store<sub>DS</sub> and  
Operation<sub>OP</sub> Yields no appropriate output parameter

**then**

create and assign a name to a Parameter<sub>OUT</sub>  
establish the design relationship Operation<sub>OP</sub> Yields Parameter<sub>OUT</sub>  
record the decision and rationale in the design history for Operation<sub>OP</sub>

**fi**