

Chapter 5 A Meta-Model for Concurrent Designs

The previous chapter defined a meta-model for describing and analyzing specifications based on data/control flow diagrams. The current chapter defines a meta-model for representing and reasoning about concurrent designs, and for describing the characteristics of target environments in which concurrent designs might execute.

The design process produces and consumes large quantities of information on which design decisions might be based. Whether and how this information can be represented is central to the type of problems that can be considered and solved. The design meta-model provides a basis for describing concurrent designs generated using design-decision knowledge and for reasoning about those designs using automated methods. The design meta-model also provides for traceability between a concurrent design and elements of the data/control flow diagram from which that design is generated. In addition, the design meta-model enables design decisions, made with respect to elements of the design, and design rationale to be captured. Before giving a detailed specification of the design meta-model, an intuitive discussion of the main concepts is presented. The intuitive discussion introduces a diagrammatic notation for the main entities and relationships included in the design meta-model.

5.1 A Diagrammatic View of the Design Meta-Model

Most of the entities and relationships from the design meta-model can be represented in a diagrammatic form using a variation of the graphical notation defined by Gomaa in his book, Software Design Methods for Concurrent and Real-Time Systems.

[Gomaa93] A summary of this notation, and its application to represent concepts in the design meta-model, is provided here for two reasons. First, the notation can be used to represent entities and relationships in the design meta-model in a form that can be grasped easily by readers. Second, a number of examples discussed later, in several appendices to this dissertation, use the notation. Figure 19 illustrates the main elements of the design notation. Any symbol shown in Figure 19 can be given a name that corresponds to the name attribute for the specific design element being represented.

The Task symbol, shown in Figure 19 (a), represents an independent thread of control within a concurrent design. An IHM, or information hiding module, can be drawn using the corresponding symbol in Figure 19 (i). An arbitrary IHM can provide as many operations as required, each represented as a small rectangle protruding from the IHM symbol. Each operation must be labeled. The specific IHM symbol shown in Figure 19 (i) encompasses four operations, although the operations are not labeled in this case. Queues and priority queues, used to buffer messages either in the order of arrival or in a priority order, can be represented using the corresponding symbols provided in Figure 19 (b) and (c), respectively. Each line entering a queue or priority queue represents a distinct type of queued message, that is, a message where the sender does not suspend waiting for

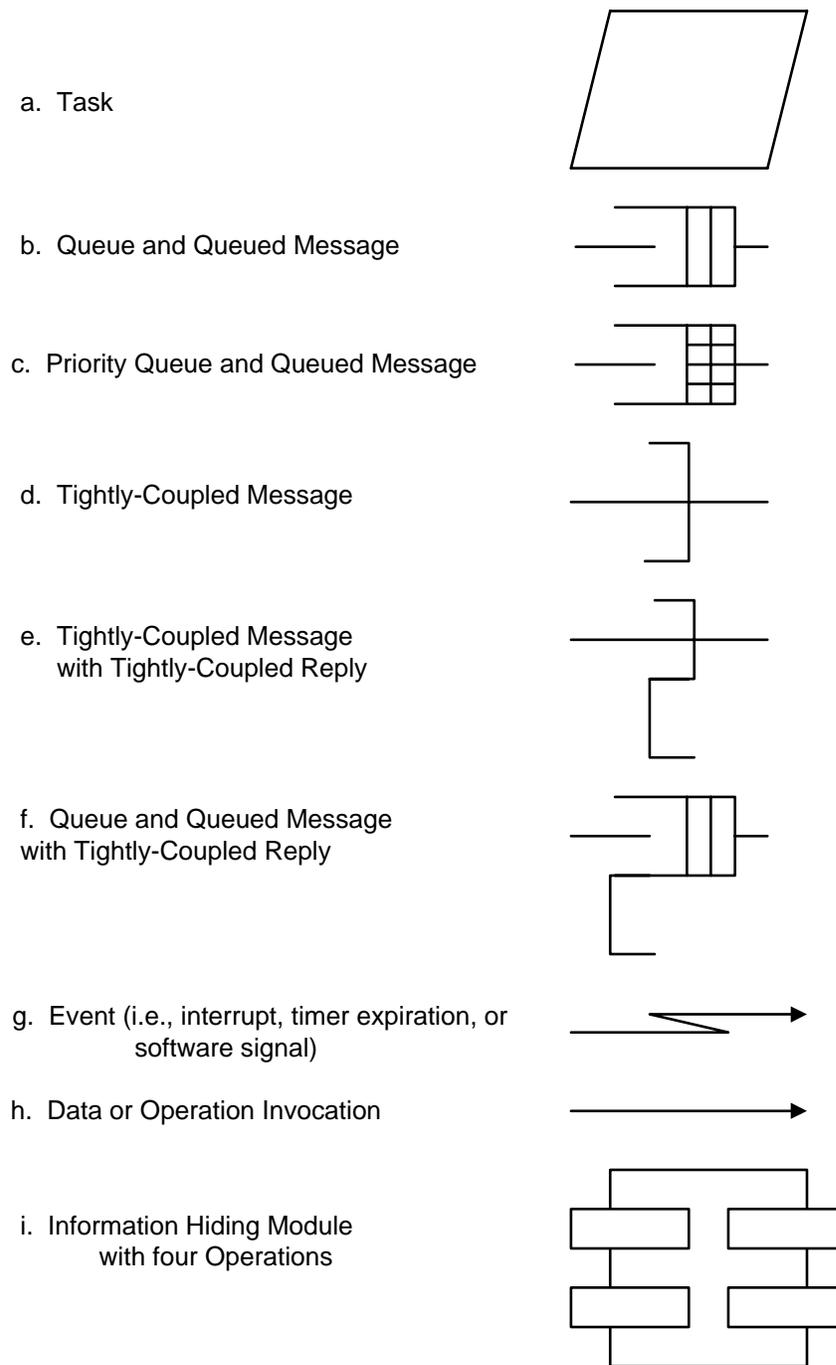


Figure 19. A Diagrammatic Notation for Representing Selected Entities from the Design Meta-Model

the receiver to accept the message. A tightly-coupled message, that is a message for which the sender suspends until the receiver has accepted the message, can be shown using the corresponding symbol from Figure 19 (d). Figure 19 (e) gives a symbol used to denote a tightly-coupled message sent from left to right with a tightly-coupled reply returning from the right to the left. In cases such as this, the sender of a message suspends not until the receiver accepts the message, but until the receiver returns another message in reply to the message sent by the sender. Figure 19 (f) provides a symbol to denote a queued message sent from left to right for which a reply will be awaited by the sender. Each occurrence of input/output data can be drawn as a directed arc, as in Figure 19 (h), while the existence of an event can be denoted using a directed, zigzag line, as in Figure 19 (g). Only two entities, parameters and message data, from the design meta-model cannot be portrayed directly using the diagrammatic notation given in Figure 19.

Many relationships from the design meta-model can be represented diagrammatically using combinations of the symbols shown in Figure 19. Figures 20, 21, and 22 illustrate some combinations representing relationships included within the design meta-model. A task is said to contain a module whenever that task is the only thread of control that executes code within the module. This relationship, Contains in Figure 20 (a), is depicted by placing an IHM symbol within the symbol for the containing task. If the thread of control for a task executes a specific operation within a module, the task is said to invoke the operation. This relationship, Invokes in Figure 20 (b), is denoted by

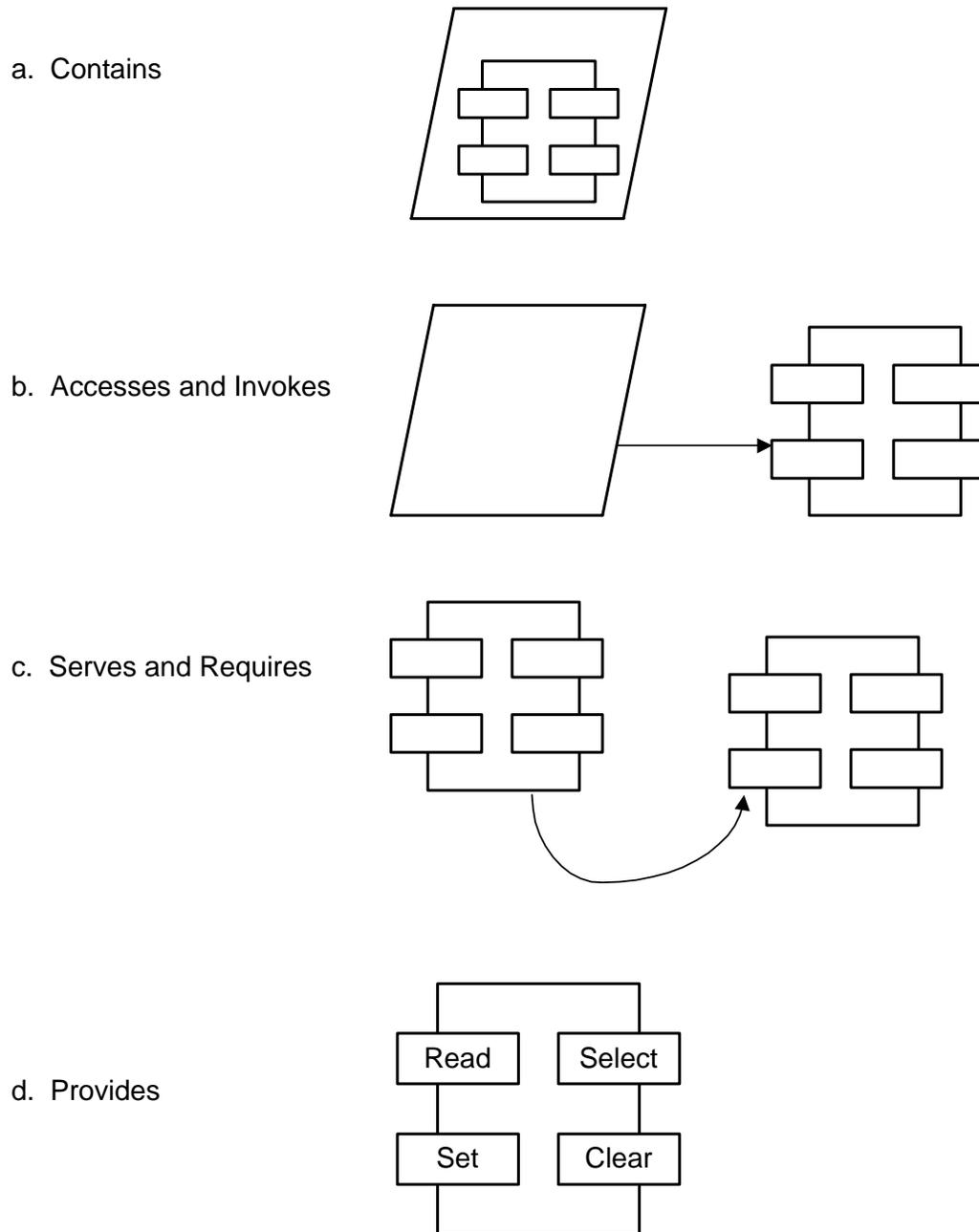


Figure 20. A Diagrammatic Notation for Representing Selected Relationships from the Design Meta-Model (Part One of Three)

drawing a directed arc from a task to a specific operation within an IHM. Whenever a task invokes any operation in a module then the task is said to access the module. This relationship, *Accesses*, is a summary of the *Invokes* relationship between a task and operations within an IHM. Any task that has at least one *Invokes* relationship with an operation within an IHM, also has an *Accesses* relationship with the IHM. Similarly, whenever an operation in one module calls an operation in a second module, the calling operation is said to require the called operation. This relationship, *Requires* in Figure 20 (c), is shown by drawing a directed arc from an IHM to a specific operation within an IHM. Given an IHM that provides at least one operation that is required by another IHM, the first IHM is said to serve the second IHM. An IHM provides one operation for each named entry point that is externally accessible. This relationship, *Provides* in Figure 20 (d), is illustrated simply by labeling the operations protruding from an IHM with the name of each operation.

Referring to Figure 21 (a), an example is given of the *Sends* and *Receives* relationships between two tasks and the message around which they synchronize. The task on the left, *Sender*, sends a tightly-coupled message and then waits until that message is received by the task on the right, *Receiver*. The next example, given in Figure 21 (b), illustrates the *Accepts* relationship (incoming zigzag line), where a task receives an event from the external environment, and shows the *Generates* relationship (outgoing zigzag line), where the same task sends an event. Similarly, the next example, shown in Figure 21 (c) shows the *Reads* (incoming directed arc) and *Writes* (outgoing directed arc)

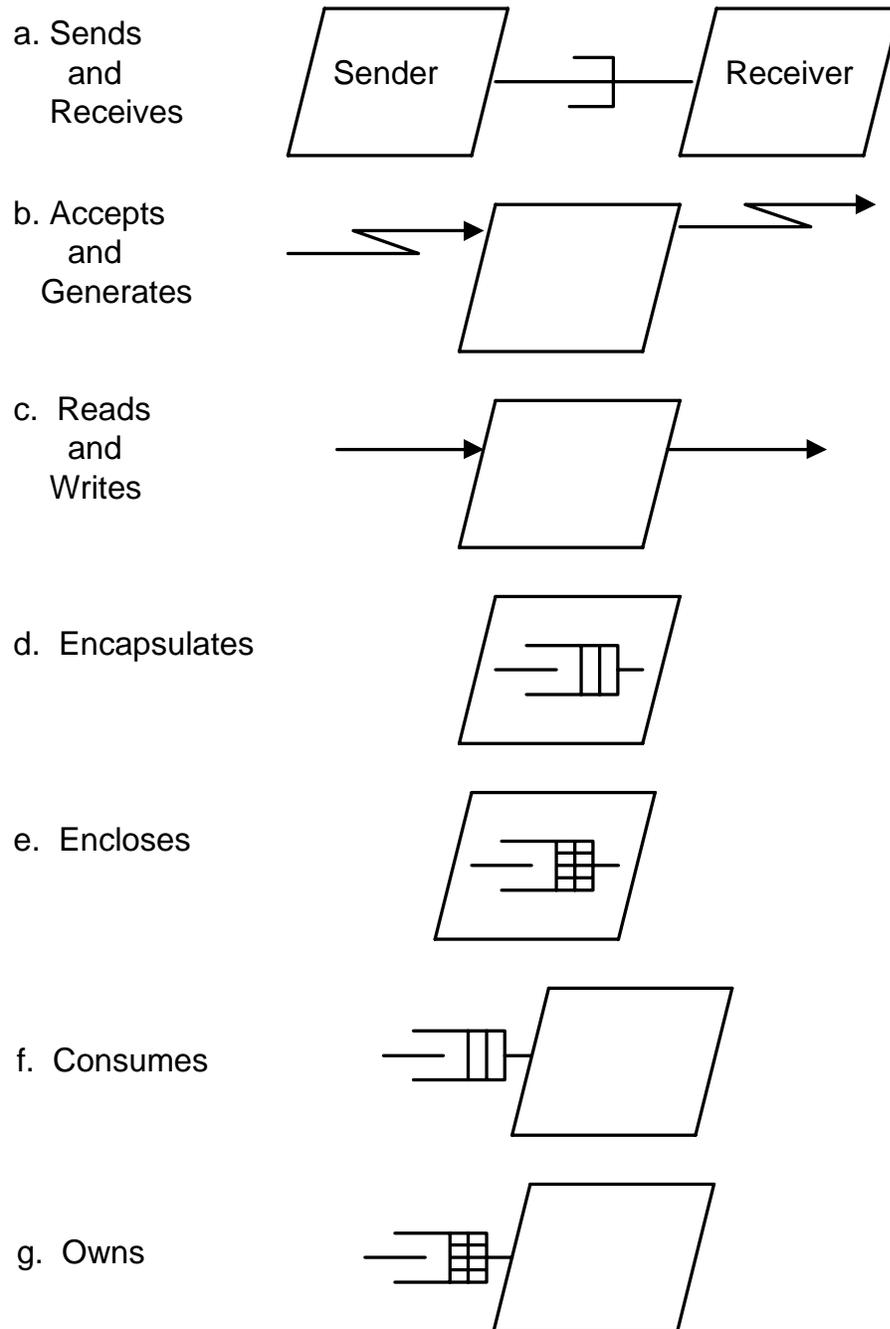


Figure 21. A Diagrammatic Notation for Representing Selected Relationships from the Design Meta-Model (Part Two of Three)

relationships, where a task accepts data from and sends data to the external environment.

The final four examples given in Figure 21 illustrate how to depict various relationships, included within the design meta-model, involving queues and priority queues. The Encapsulates and Encloses relationships, for queues and priority queues, respectively, are used to simulate message queues when the target environment does not provide a message queuing mechanism. Figure 21 (d) shows how to depict the Encapsulates relationship by placing the symbol for a queue within the symbol for the task that controls the queue. Similarly, Figure 21 (e) shows how to depict the Encloses relationship for a priority queue and enclosing task. The Consumes and Owns relationships, as shown in Figure 21 (f) and (g), are used to depict queues and priority queues, respectively, that are processed by a receiving task in cases where the target environment does provide a mechanism for first-in, first-out queues and for priority queues.

Figure 22 illustrates how some more complex relationships can be depicted with diagrams. Figure 22 (a) shows how to depict a request-response transaction between two tasks as the exchange of two tightly-coupled messages, one sent in reply to the other. One task, Requester, sends a tightly-coupled message that another task, Replier, receives. The task Replier answers the received tightly-coupled message; and thus sends a tightly-coupled message (as a reply) that the task Requester receives. A second example, shown in Figure 22 (b), illustrates how a client-server relationship between multiple client tasks and a single server task can be drawn. Each task, Client, sends a queued

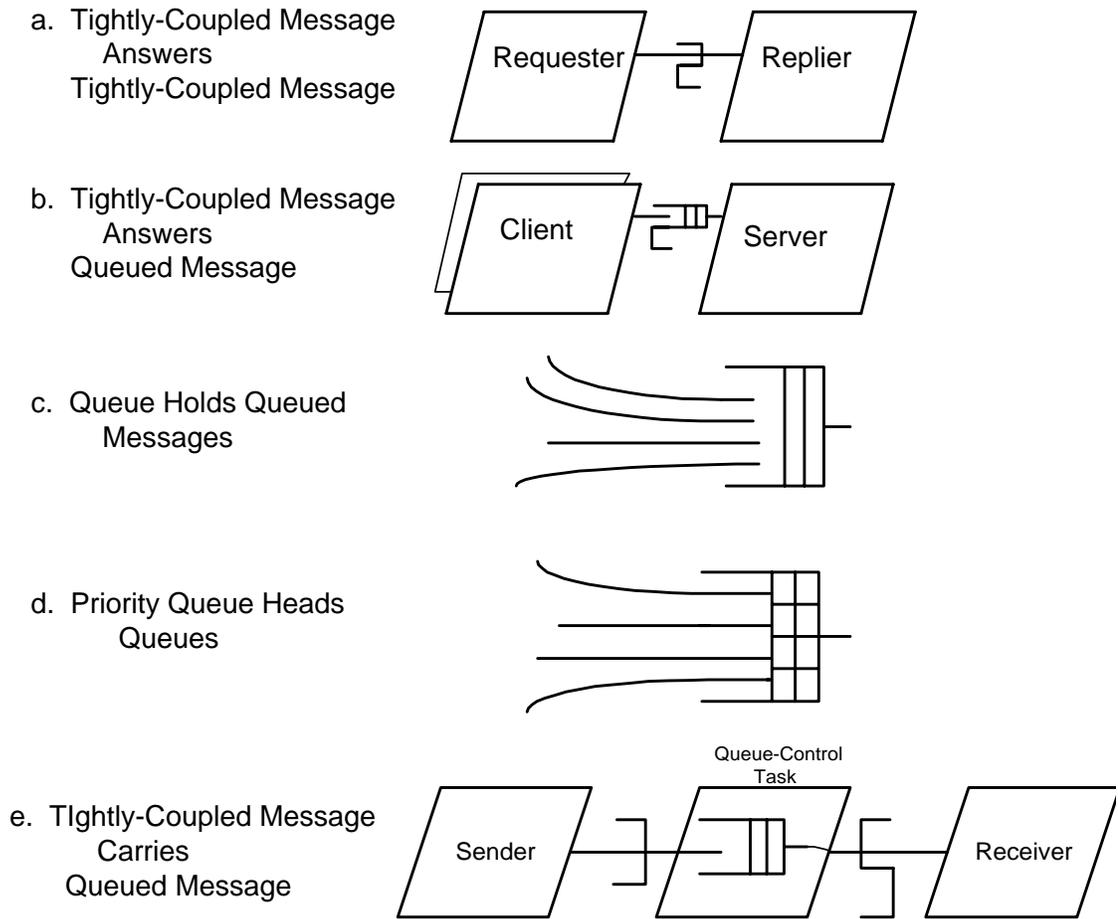


Figure 22. A Diagrammatic Notation for Representing Selected Relationships from the Design Meta-Model (Part Three of Three)

message that the task Server receives. The task Server answers the received, queued message; and thus sends a tightly-coupled message (as a reply) that the task Client receives.

Figures 22 (c) and (d) show how to draw some relationships involving queues, priority queues, and queued messages. Figure 22 (c) shows a single, first-in, first-out (FIFO) queue that holds four types of queued message. Each type of queued message arrives from a separate type of task. Figure 22 (d) shows a single priority queue that accepts queued messages at four distinct priorities. In the example, one type of queued message arrives at each priority; however, multiple types of messages can arrive at any given priority. Also, notice that the symbol for the priority queue can be viewed as if four FIFO queues, placed in parallel, compose the priority queue. This composition represents the Heads relationship between a priority queue and its constituent FIFO queues. Other views of a priority queue are possible, but the parallel FIFO view is adopted for the design meta-model defined in this dissertation.

Figure 22 (e) illustrates how a queue-control task, used to implement a FIFO queue, can be depicted on a diagram. A task, Sender, sends a tightly-coupled message to a queue-control task. The tightly-coupled message is said to carry a queued message within it. The queue-control task removes the queued message and places it within an internal list. When another task, Receiver, wishes to accept the next queued message from the queue-control task, then Receiver sends a tightly-coupled message to the queue-control task and awaits a reply. When a queued message is ready for delivery, the

queue-control task extracts the queued message from its internal list, and then passes that queued message within a tightly-coupled message that replies to the earlier tightly-coupled message sent by the task Receiver.

The remaining relationships in the design meta-model cannot be illustrated using the symbols given in Figure 19. Relationships involving parameters and message data items cannot be represented because no symbols are defined to represent either entity. While these relationships involving parameters and message data items are not expressible using the diagrammatic notation, they can be represented in the machine-processible form of the design meta-model, and can also be rendered within task behavior specifications and module specifications that are output from the design-generation process.

The previous paragraphs provided an intuitive picture of the entities and relationships that can be used to model concurrent designs. Next, the design meta-model is defined with greater precision.

5.2 Modeling Concurrent Designs

Two entity-relationship (E-R) diagrams and three tables define the design meta-model. The two diagrams and the first table encompass the meta-model for concurrent designs, while the remaining two tables describe a model for target environments. The meta-model for concurrent designs could be described with a single E-R diagram; however, the use of two overlapping diagrams leads to a clearer exposition. The first E-R diagram, shown as Figure 23, identifies the basic elements that make up a

concurrent design, shows the attributes for each of those elements, depicts the inheritance relationships among those elements, and indicates the relationships between any design element and two other entities: 1) decisions made about the design element and 2) specification elements from which the design element derives. The second E-R diagram, shown as Figure 24, depicts the relationships between specific design entities.

The notation used in the E-R diagrams depicted in Figures 23 and 24 is conventional. Entities are shown using rectangles labeled with the entity name; attributes are illustrated with ovals surrounding the attribute name; inheritance relationships are drawn as inverted triangles enclosing the label IS-A, with directed arcs pointing the way up the inheritance hierarchy; relationships other than inheritance are depicted through diamonds that enclose the name of the relationship. A line connects an entity to an attribute when that attribute is a part of the entity. A line connects an entity to a relationship (either an IS-A relationship or an arbitrary relationship) when the entity participates in the relationship. Each line connecting an entity to an arbitrary relationship denotes a cardinality. Specific cardinalities used in Figures 23 and 24 include: 1 (exactly one); 0 or 1 (at most one); N (zero or more); N, $N > 0$ (at least one); and N, $N > 1$ (at least two).

5.2.1 Concurrent Design Entities

Each entity composing the meta-model for concurrent designs is a specialization of the entity named Design Element (see Figure 23). A Design Element possesses two attributes: name provides a unique label for the Design Element among all elements

within a specific design; object identifier provides a unique reference for the Design Element among all objects existing at a given time within the knowledge base that contains the Design Element. Each Design Element participates in at least two relationships. One relationship, shown as Tracks in Figure 23, links a Design Element to zero or more decisions (represented by the entity Decision) made concerning that Design Element. A Decision has three attributes: 1) the rule name attribute identifies the rule that made the decision; 2) the action attribute defines the action taken as a result of the decision; 3) the rationale attribute contains a justification for the decision. A second relationship, shown as Traces To/From in Figure 23, links elements from a data/control flow diagram (specifically, Specification Element, the concept of the same name as defined in the concept hierarchy discussed in Chapter 4) to corresponding elements in the design (Design Element). One instance of Specification Element can lead to several instances of Design Element, and each instance of Design Element can be derived from several instances of Specification Element; therefore, the Traces To/From relationship is many-to-many.

The Traces To/From relationship between instances of Specification Element and instances of Design Element is actually a generalization of several more specific relationships that serve to restrict the traceability between a specification and a concurrent design. The more specific relationships that restrict Traces To/From are shown in Table 3. Each row of the table contains two columns. The first column identifies a type of design element, while the second column identifies a set of specification element types

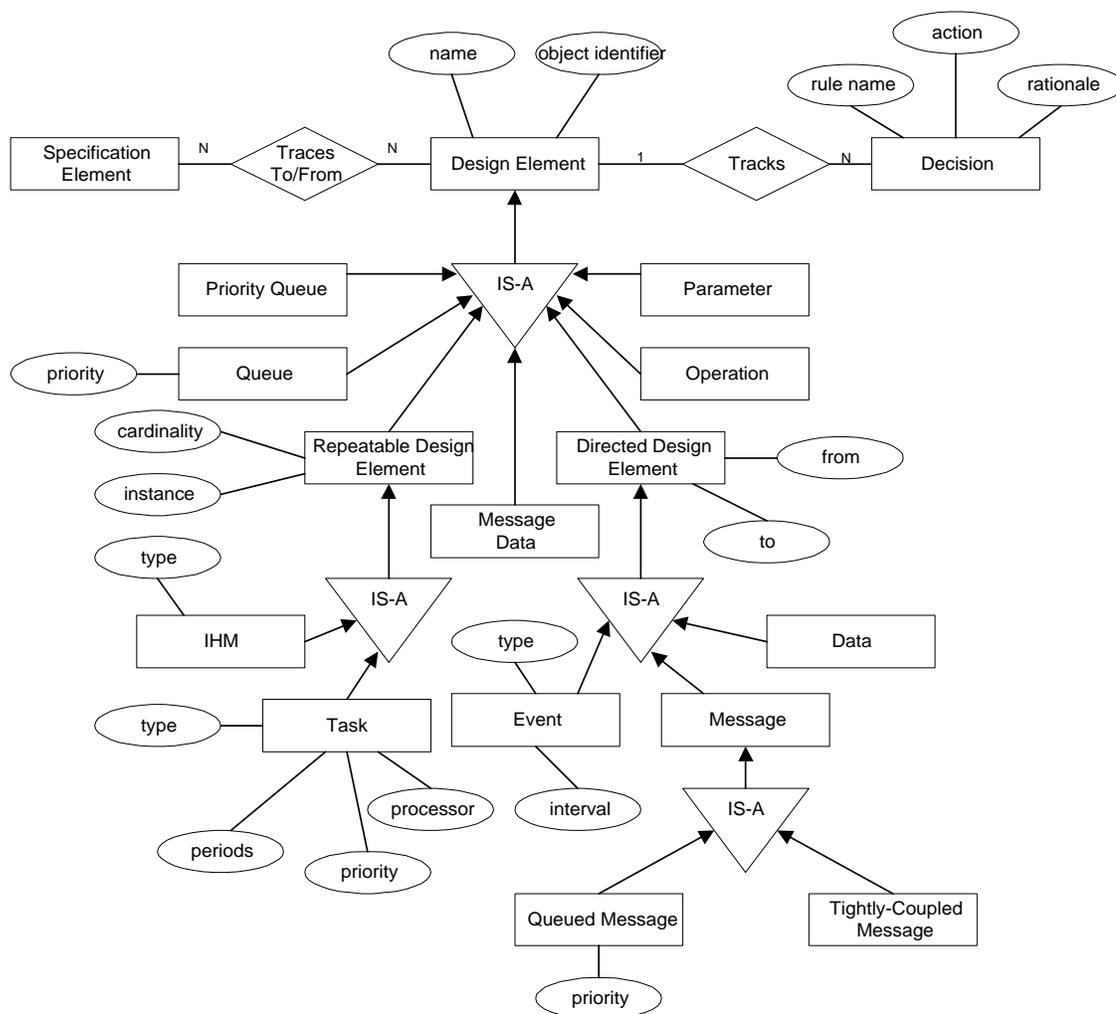


Figure 23. An Entity-Relationship Diagram Depicting the Design Meta-Model (Part One of Two)

from which the design element can be derived. Each row of the table, then, defines a restriction on the traceability between specification elements and design elements. For example, design elements of type Task can only be derived from the specification elements of type Transformation.

Moving down the inheritance tree shown in Figure 23, five leaf-level entities derive directly from Design Element. The entity Priority Queue allows a concurrent design to include a message queue that can hold messages at various priorities. The entity Queue can be used to represent a message queue that can hold messages at one specific priority (denoted by the attribute priority). The entity Message Data represents data that can be included in messages sent between tasks. The entity Operation is used to represent procedures within the software modules composing a concurrent design. The entity Parameter allows parameters for procedures to be depicted within a design.

Two abstract entities are also derived directly from Design Element. The abstract entity Repeatable Design Element can be used to represent elements within a design that can have multiple instances. The attribute cardinality denotes the number of possible instances for a specific Repeatable Design Element. The attribute instance enables specific instances of multi-instance design elements to be identified. The abstract entity Directed Design Element can be used to represent components within a concurrent design that have a source (denoted using the attribute from) and a sink (denoted using the attribute to).

Table 3. Restrictions on Traceability Between Designs and Specifications

Design Element	Specification Elements
Task	Transformation
IHM	Data Store Directed Arc Transformation Two-Way Arc
Message	Control Event Flow Internal Data Flow Signal
Operation	Data-Store Data Flow External Data Flow Interrupt Transformation Update
Parameter	Control Event Flow Data Store External Data Flow Internal Data Flow Signal
Message Data	Control Event Flow Internal Data Flow Signal
Event	Control Event Flow Normally-Named Event Flow
Data	External Data Flow
Priority Queue	Signal Stimulus Transformation
Queue	Signal Stimulus Transformation

Two entities inherit the abstract entity Repeatable Design Element. The entity IHM (for Information Hiding Module) enables software modules to be represented within a concurrent design. Each IHM contains an attribute, type, used to denote the basis for forming the IHM. For example, an IHM can encapsulate a device interface or a user interface, can hide the implementation details of a data structure, can abstract the details of a state-transition diagram, or can enclose the processing associated with an algorithm.

The entity Task enables multiple, independent threads of control to be represented within a concurrent design. Execution models for real-time systems present a rich variety of facilities for inter-task exchange of messages and events. The capabilities and restrictions placed on Tasks regarding the sending and receiving of events and messages are largely outside the scope of the concurrent design meta-model presented in this dissertation; thus, other than the assumptions and restrictions noted below, no specific limitations are assumed regarding the ability of a Task to wait for and respond to arriving events and messages, either queued or tightly-coupled.

The entity Task has several attributes. The attribute type denotes the basis for forming the Task. For example, a Task can perform periodic polling of devices, can process device interrupts, can process asynchronous events, can control access to shared resources or software modules, can decouple communications between tasks, or can perform periodic execution of algorithms. The attribute periods can contain the set of execution cycles for periodic tasks. Multiple periods might be required whenever the logic internal to a task activates itself with various periodicities, depending on specific

requirements of the application. The attribute priority can hold the precedence with which a task will be selected for execution when multiple tasks are eligible to run. The attribute processor enables a task to be assigned to a specific processor when the software will execute on a multiprocessor system.

Three entities inherit the abstract entity Directed Design Element. The entity Data enables information exchanged between the software under design and its environment to be represented. The entity Event allows three forms of asynchronous signal to be depicted: 1) interrupts from hardware, 2) timer expirations, and 3) software signals exchanged between tasks. An attribute, type, is used to distinguish between the three forms of Event. The attribute interval can hold the cycle time for timer expirations. The entity Message enables representation of information exchanged between tasks. Two types of message can be distinguished using separate entities. The entity Queued Message enables the representation of loosely-coupled communications between tasks. The attribute priority allows loosely-coupled messages arriving for a task to be classified by precedence. The entity Tightly-Coupled Message permits the depiction of synchronized message exchange between tasks.

5.2.2 Relationships Among Concurrent Design Entities

Aside from the inheritance relationships and the relationships named Tracks and Traces To/From depicted in Figure 23, the design meta-model includes a number of additional relationships, as shown in Figure 24. These additional relationships increase the richness and complexity of the design meta-model. Each relationship in the design

meta-model should be understood to be bi-directional, including both the relationship as shown and its inverse. For example, referring to Figure 23, the relationship Traces To/From can be expressed in two ways: 1) a Specification Element Traces To a Design Element and 2) a Design Element Traces From a Specification Element. Similarly, the relationship Tracks can be expressed in two forms: 1) a Design Element Tracks a Decision and 2) a Decision Is Tracked By a Design Element.¹ While this bi-directionality is an integral part of the design meta-model, each relationship shown in Figures 23 and 24 illustrates only one direction in order to simplify the diagram. The inverse direction for each relationship should be assumed to exist. For the IS-A relationship, the direction shown in the figures corresponds to the view up the inheritance tree, while the assumed inverse relationship corresponds to the view down the inheritance tree. For example, the relationship Queued Message IS-A Message has a corresponding inverse, such as Message Has-A-Descendent Queued Message.

Referring to Figure 24, the entity IHM, representing information-hiding modules, is party to numerous relationships. First, the relationship stating that a Task Contains an IHM denotes that the Task is the sole thread of control that executes the IHM, while its inverse, Contained By, ensures that the related IHM can be executed through the control of only one task. The meta-model does not currently provide a relationship for tasks that can be placed within IHMs because such arrangements are considered only to be one

¹The E-R Notation, intended to show the relationships among entities in a natural, readable fashion, leads to ambiguities within the diagrams depicted here. For example, in Figure 24, does a Task Contain an IHM or does an IHM Contain a Task? These ambiguities are resolved in the accompanying textual description.

means of implementing synchronization internal to an IHM and, thus, becomes a detailed design issue. Second, the relationship *Accesses* represents the use by a task of the services of an IHM that is shared among multiple tasks. The inverse relationship, *Accessed By*, can be used to show that an IHM provides services to a task. A third relationship of interest, *Serves*, can be used to show when an IHM, shared by multiple tasks, provides operations used by another IHM, which is then also shared, indirectly by multiple tasks. The inverse relationship is known as *Served By*. On a more detailed level, an IHM can use zero or more specific operations provided by another IHM. The relationship *Requires* and its inverse *Required By* allow depiction of these more detailed relationships between IHMs. Conversely, an IHM can provide specific operations for use by client tasks and by other IHMs. The *Provides* relationship, and its inverse, *Provided By*, allows operations to be allocated to specific IHMs. Each operation can be provided only by a single IHM; each IHM must provide at least one operation. Operations provided by IHMs can be accessed by multiple tasks; the relationship *Invokes*, and its inverse, *Invoked By*, enables these accesses to be depicted. Zero or more tasks can invoke a specific operation. Each operation can be invoked by zero or more tasks.

A set of three relationships enable parameters to be associated with operations. An input parameter to an operation is denoted using the relationship *Takes*, and its inverse, *Taken By*. An output parameter from an operation is depicted with the relationship *Yields*, and its inverse, *Yielded By*. An input/output parameter for an operation is shown through the relationship *Alters*, and its inverse, *Altered By*. Any

operation can be involved in zero or more relationships of each of these types. A specific parameter can be either an input parameter, an output parameter, or an input/output parameter to one, and only one, operation. For messages, the relationship Includes, and its inverse, Included By, allows a message to be given zero or more units of data². A specific unit of data can be included in at most one message.

The preceding discussion identified three relationships involving the entity Task. Task is also party to a number of other relationships. For one, a task can send and receive zero or more messages, while a message must be sent and received by one task only (this design meta-model requires that broadcast and multicast communications be replaced by individual message communications). These facts can be represented with the relationships Sends (inverse is Sent By) and Receives (inverse is Received By). A task may also accept events of any allowed type (that is, timer expiration, interrupt, or software signal) and may generate software signal events. These capabilities can be shown using the Accepts (inverse is Accepted By) and Generates (inverse is Generated By) relationships. Accepting a timer expiration event places is assumed to place a task in a state where it is ready to execute. Accepting an interrupt event is assumed to vector the execution of a task to a specific location within the task logic. Accepting a software signal event is assumed to require a task to be waiting for the event, as one among possibly other events and messages, to arrive.

²A message without data consists solely of a message type (or name). A message without data can be used to simulate an internal software event.

A task may accept data from or send data to the external environment. These situations can be represented via the Reads (inverse is Read By) and Writes (inverse is Written By) relationships, respectively.

The remaining relationships involving the entity Task deal with message queues. A task that receives queued messages can own zero or one priority queue (shown through the relationship Owns), and a priority queue can be owned by at most one task (represented with the inverse of Owns, Owned By). A task that receives queued messages can also consume the contents of zero or more FIFO message queues (depicted using the relationship Consumes), while any message queue must be consumed by a single task (indicated using Consumed By, the inverse relationship of Consumes).

When a priority queue exists it is modeled as a parallel set of FIFO queues for which the priority queue is said to provide a head (the relationship Heads). Each FIFO queue can be a component of at most one priority queue (denoted with the inverse of Heads, Headed By). A message queue can also hold queued messages from more than one source, where each message held has the same value for its priority attribute as the value for the priority attribute of the message queue that holds the message. These connections can be represented using the Holds relationship, and its inverse, Held By.

When the target environment for the system being designed does not provide message queues, an intermediary task can be constructed to decouple message communications that would normally be sent via message queues. In such cases, two relationships provide alternatives to Owns and Consumes. The relationships Encloses

and Encapsulates, and their inverses Enclosed By and Encapsulated By, allow a priority queue or FIFO queue, respectively, to be hidden inside an intermediary task. Communication between two decoupled tasks occurs through the intermediary task. Communications with the intermediary task occur through synchronized message exchange using tightly-coupled messages. In such circumstances, the messages sent between the decoupled tasks are carried within tightly-coupled messages exchanged with the intermediary task. The relationship Carries (and its inverse, Carried By) can be used to show that a Queued Message is transferred within a Tightly-Coupled Message.

Both tightly-coupled and queued messages can be answered with a tightly-coupled message. Should the sender of a tightly-coupled message require a reply from the receiver before additional processing can occur, the message sender will wait for a message to be returned from the receiver before continuing. Similarly, should the sender of a queued message need a reply from the receiver before continuing, then the sender waits until the receiving task provides the necessary reply. The relationship Answers (and its inverse, Answered By) allows a request message to be linked to a tightly-coupled message that serves as a corresponding reply.

5.2.3 Assumptions Underlying the Design Meta-Model

Specific real-time operating systems, run-time environments, and programming languages define the semantics for exchanging messages and, where applicable, signals between tasks. In order to apply to a wide range of systems, the design meta-model defined in this dissertation makes only a weak set of assumptions about the specific

semantics associated with message and signal exchanges. Most real-time operating systems, run-time environments, and programming languages define a more detailed set of semantics. Designs described using the design meta-model can be elaborated using the semantics of any specific operating system, run-time environment, or programming language that does not violate the weaker assumptions of the design meta-model. The relevant assumptions are given below.

- ◆ The sender of a tightly-coupled message cannot proceed until the receiver of the message accepts the message.
- ◆ The sender of a queued message will block when sending to a full queue. The receiver of a queued message will block when receiving from an empty queue.
- ◆ Messages are read from a queue in first-in, first-out order. If the queue is a priority queue, then messages are read highest priority first, but in first-in, first-out order within each priority.
- ◆ Tasks request their own suspension when awaiting a timer expiration event. When a timer expiration event occurs, the suspended task commences execution from the statement following the suspension request.
- ◆ Tasks provide labeled entry points for execution upon the receipt of interrupt or software-signal events.
- ◆ A task can selectively await the arrival of one or more messages, either queued messages, tightly-coupled messages, or both. Arrival of any one of the awaited messages allows the waiting task to continue.

5.3 Describing Target Environments

The research described in this dissertation models target environment descriptions, or TEDs, as: 1) a set of constraints imposed by the intended hardware architecture and operating system that will host a concurrent design and 2) a set of guidelines used to make some design decisions. At appropriate times in the design process these constraints and guidelines are consulted by design-decision rules in order to make sensible decisions. Table 4 presents the hardware and operating system imposed constraints that are included with the TED model. Table 5 gives the design guidelines that can be represented with the model.

Two constraints relate to the ability of target hardware and operating systems to provide access to memory shared between tasks. The constraint called intra-processor shared memory denotes the availability or unavailability of shared memory for tasks that execute on the same processor. If intra-processor shared-memory mechanisms are not provided in a target system, then, since message communications among tasks is assumed to exist, the designer can place IHMs accessed by multiple tasks on the same processor into server tasks. The constraint called inter-processor shared memory denotes the availability or unavailability of shared memory for tasks that execute on separate processors. If inter-processor shared-memory mechanisms are not available in the target environment, then the designer must place IHMs accessed by tasks executing on separate processors inside a server task in the local memory of one of the processors. If inter-processor shared-memory mechanisms are available, then the designer can place

Table 4. Target Environment Constraints

Constraint	Representation	Possible Values
intra-processor shared memory ¹	SYMBOL	Available (default) or Unavailable
inter-processor shared memory ¹	SYMBOL	Available or Unavailable (default)
maximum number of inter-task signals	NATURAL NUMBER	Default is 2
message queues	SYMBOL	Available (default) or Unavailable
priority queues	SYMBOL	Available or Unavailable (default)
number of processors ¹	NATURAL NUMBER	Default is 1
number of task priorities ¹	NATURAL NUMBER	Default is 64

Table 5. Design Guidelines

Guideline	Representation	Possible Values
task-inversion threshold	NATURAL NUMBER	Default is 4
priority-assignment algorithm ¹	SYMBOL	Manual or Algorithm Name
task-allocation algorithm ¹	SYMBOL	Manual or Algorithm Name

¹ These constraints and guidelines are not used by any design-decision rules specified in this dissertation. They are included to support future research regarding the automated configuration of designs for a variety of hardware architectures.

IHMs accessed by tasks executing on separate processors into a shared-memory area accessible by multiple processors.

Another constraint defines the maximum number of inter-task signals supported by the target operating system. This constraint can be used by a designer to decide whether events between two tasks can be allocated to software interrupts, or whether inter-task messages must be used to transport the events.

Two additional constraints denote the availability or unavailability of various queuing facilities in the target operating system. The constraint called message queues indicates whether or not the target operating system allows tasks to consume first-in, first-out (FIFO) queues of incoming messages. The constraint called priority queues indicates whether or not the target operating system allows tasks to consume queues of incoming messages where the queue contents are segregated based on priority. Using these two constraints, a designer can decide what type of interface should be provided to tasks that require queues of messages. A designer might choose one of the following task interfaces: 1) a single FIFO queue, 2) a priority queue, 3) multiple FIFO queues simulating a priority queue, 4) a queue-control task simulating a FIFO queue, or 5) a queue-control task simulating a priority queue.

The two remaining constraints define the number of processors available in the target environment and the number of task priorities that can be supported for each processor. This information can be used in making design decisions regarding the assignment of tasks to processors and the assignment of priorities to tasks on the same

processor. In fact, the values held by these constraints might be used in conjunction with two of the three guidelines shown in Table 5.

The task-allocation algorithm guideline indicates whether tasks are to be allocated to processors manually or by using an automated algorithm. When an automated algorithm is available, the guideline names the algorithm. Similarly, the priority-allocation algorithm guideline indicates whether task priorities are to be assigned manually or by using an automated algorithm. The final guideline, task-inversion threshold, indicates when multiple instances of the same task should be merged into a single task that switches internally among the various instances of the task.