

The Model Integrated Computing Approach to Software Architecture

Ethan K. Jackson

Institute for Software Integrated Systems,
Vanderbilt University, Nashville, TN 37235, USA
ejackson@isis.vanderbilt.edu

Abstract. Model-based design is an approach to software architecture that supports the rapid evaluation of many different design alternatives evolving from a high-level specification. Model integrated computing[1] (MIC) is one manifestation of model-based design, which relies on *semantic units*[2] and *model transformations*[3][4] to rapidly construct and evaluate various refinement paths. Abstract state machines are used at the core of MIC technologies to facilitate this goal. In this paper, we begin from the basic premises exemplified by abstract state machines, and show through examples, why the model-based approaches evolved and how they can be applied to today's architectural challenges.

1 Introduction

Today's software systems pose unique challenges to traditional software engineering methodologies. First, the demands placed on software systems continue to evolve in both the functional and non-functional realms, and along many interacting axes: architectural, temporal, and, physical. Second, the shear scale of software continues to grow, both on a per node basis, and in number of distributed nodes that compose a system. Third, non-engineering disciplines, such as the legal field, are impacting design choices in poorly understood ways. For example, a recently enacted US law, called HIPAA, will impact how medical records can be digitally stored and accessed[5]. On one hand, this trifecta validates exactly what software engineers have always argued: Software must be designed methodically; off-the-cuff implementations will almost surely fail. On the other hand, engineering approaches that focus on sequential systems isolated in a comfortable computational environment are not sufficient for methodically designing today's large-scale and heterogeneous software systems.

The term *model-based design* (also called *platform-based design* [6]) encompasses a spectrum of engineering approaches, all of which address the complexity of modern system design. Intuitively, a model is a light-weight construct that can be rapidly projected onto many possible solution variants. This projection process allows the engineer to explore how complex requirements are affected by the many worlds in which an architecture might be placed. Conversely, it allows the engineer to determine a relevant set of boundaries for further refinement of a

specification. Should distributed system X be implemented with ethernet or via a time-triggered communication protocol? Each choice deeply affects the degree to which requirements can be evaluated [7]. Ethernet allows for collisions, and is inherently probabilistic, while time-triggered protocols impose a rigid temporal structure that avoids these problems. The model-based approaches provide support for answering questions, such as these, early in the design process. In Section 2 we examine the basic design principles for ASM-based software architecture, and we argue why it is reasonable to expand on these principles. Sections 3 and 4 describe the core ideas behind model-based design, with Section 4 focusing on model transformations. We conclude in Section 5.

2 Why More?

Historically, the development of programming languages and software architectures have been tempered by the view that too many features can be dangerous: Dijkstra proclaims “. . . the development of ‘richer’ or ‘more powerful’ languages was a mistake. . .”[8], and Wirth holds that “It is my belief that a tool. . . must be as simple as possible, but no simpler”[9]. Regardless of phrasing, this view requires that we grow our architectural methodologies carefully and with just cause. Thus, before we describe our model-based approach, let us reexamine the founding principles behind abstract state machines, and provide just cause for expanding on these founding principles.

ASM-based software architectures leverage (at least) three basic principles: *precision*, *abstraction*, *refinement* [10][11]. *Precision* means that the semantics of ASM specifications are unambiguous and formally rigorous. This principle resolves the problems associated with natural language specifications, and supports the immediate application of formal methods [12]. *Abstraction* allows the specifications to be written at the required level of detail. This resolves the problem of over-specification. Finally, *refinement* provides a mechanism for incrementally supplying those details left out of the abstract specification. If done correctly, refinement solves the problem of producing an implementation that conforms to the specification. Most would agree that these principles are *necessary* for today’s architectural demands. Nonetheless, the question remains: “Are these principles *sufficient*?”.

In order to explore sufficiency, consider Figure 1 that illustrates a protocol between a software client and server. The user of the software client may open a data session between the client and server by locally producing a **User_Open** message. The client reacts to this local message by broadcasting the **Client_Open** message, which the server observes. Upon observation of **Client_Open**, the server produces an **Open** message local to the server, and changes to the **Serv_Open** state. A similar process occurs when the user wishes to close the session. In this case, the user provides a **User_Close** message. In response, the client broadcasts **Client_Close** and transitions to the **Clnt_Closing** state. The client remains here until it receives a message from the server acknowledging that the session has been terminated.

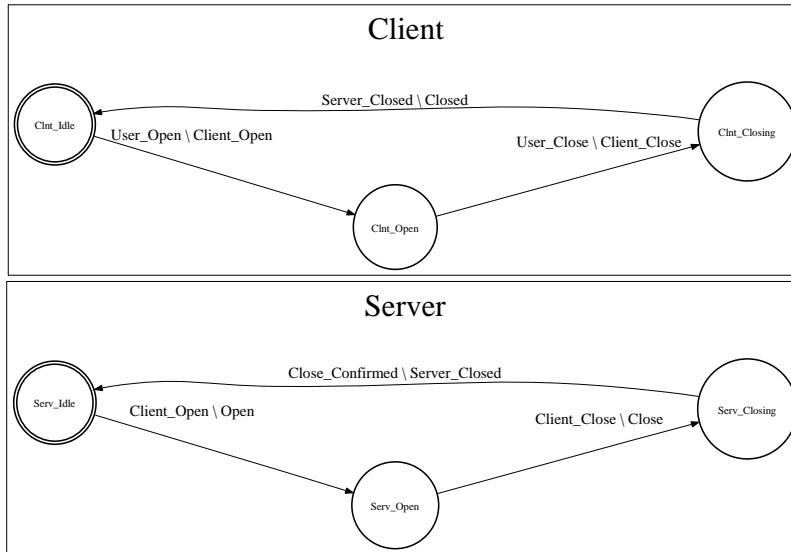


Fig. 1. Example system consisting of client, server, and communication protocol

Figure 1 is fairly precise, in the sense that it uses the notation of *finite state transducers* (FSTs), though it leaves the communication mechanism between FSTs abstract. Collecting these observations, we might specify the client and server by first producing a high-level specification of an FST, as shown in Specification 1. (Specifications are written in ASML-2). This specification is a straightforward translation of FSTs to ASMs. The *Transition* structure has source and destination *States*, and may have a trigger and/or action. If a *Transition* does not have a trigger (action), then the trigger (action) field is *null*. In order to prevent over-specification, the *FST* class is abstract. This supports refinement into one of the many different styles of FSTs. For example, if the FST were to perform actions inside of states (as found in StateCharts [13]), then the *TakeTransition* member would invoke the necessary business logic. The *GetTransitions* method allows an outside observer to determine all of the transitions that would be enabled by a set of events. This method is used by a communication *channel* to calculate the correct global macrostep of a collection of FSTs. Specification 2 shows the high-level specification of the *Channel*. *FSTs* are connected to the *Channel* by providing the constructor with the relevant set of *FST* instances. Calling *Initialize* on a *Channel* puts the global system in the global initial state. Finally, the *Publish* method allows events to be pushed into a *Channel*. This method stores events in the *inputs* variable, then the abstract *Interact* method is called, and the output events are returned. Different channels refine the *Interact* method differently. The output events are those events that an external observer would see after the all FSTs respond to the published

```
1: enum Event
2: enum State
3: structure Transition
4:   src as State
5:   dst as State
6:   trigger as Event or Null
7:   action as Event or Null
8:
9: abstract class FST
10:  public abstract Initialize()
11:  public abstract TakeTransition(t as Transition)
12:  public abstract GetTransitions(inputs as Set of Event) as Set of Transition
```

Spec 1. High-level specification of FST

events. In another words, *Publish* calculates the global macrostep that would occur as a result of the events.

Applying the refinement principle, we shall refine the *FST* class into the simplest class of finite state transducers. (We do not show this specification due to space constraints.) In this refinement, the FST is constructed by providing the set of transitions and an initial state. The *Initialize* method resets the current state to the initial state. Taking a transition *t* causes the current state to change to the destination of the transition. Finally, the *GetTransitions* method returns all the transitions that start on the current state and have no trigger, or have a trigger that is satisfied by the input set of events. Using this refinement, we can easily define the client and server protocols of Figure 1. Specification 3 shows how the client is specified; the server is specified analogously.

Thus far the refinement process has worked as expected. Next, we must refine the communication channel. Specification 4 shows one possible refinement. This first refinement makes a reasonable assumption: The FSTs are asynchronous, so no events occur simultaneously. The *Interact* method collects together all the enabled transitions from the FSTs, takes one transition from each FST, and outputs the actions that occur on the transitions. Though this seems to be a reasonable strategy, a simple test case shows that this refinement is not the correct one. In the test case of Specification 5, *User.Open* and *User.Close* events are put, one-by-one, into the channel, and observed by the client. Since the messages sent by the client are not synchronized with the instances that the server looks for messages, the *Client.Open* and *Client.Close* messages are missed by the server. In the end, the server remains in the idle state throughout the simulation, and the client becomes stuck waiting for the server to indicate that the session has been successfully closed (Figure 2.a).

```

1: abstract class Channel
2:   protected var inputs as Set of Event = {}
3:   protected var outputs as Set of Event = {}
4:   protected fst as Set of FST
5:   protected abstract Interact()
6:   public Channel(s as Set of FST)
7:     fst = s
8:   public Initialize()
9:     step forall fst in fst
10:      fst.Initialize()
11:   public Publish(environ as Set of Event) as Set of Event
12:     step inputs := environ
13:     outputs := {}
14:     step Interact()
15:     step inputs := {}
16:     return outputs

```

Spec 2. High-level specification of communication channel

```

1: client = new SimpleFST(CInt_Idle, {
2:   Transition(CInt_Idle,CInt_Open, User_Open, Client_Open)
3:   Transition(CInt_Open,CInt_Closing, User_Close, Client_Close)
4:   Transition(CInt_Closing,CInt_Idle, Server_Closed, Closed) })

```

Spec 3. Specification of the client as an instance of *SimpleFST*

```

1: class Asynchronizer extends Channel
2:   protected override Interact ()
3:     step forall fst in fst
4:       choose t in fst.GetTransitions(inputs)
5:         if not (t.action = null) then add t.action to outputs
6:         fst.TakeTransition(t)

```

Spec 4. Refinement of channel into an asynchronous channel

```

1: channel = new Asynchronizer( { server, client })
2: Main()
3:   step channel.Initialize()
4:   step channel.Publish({User_Open})
5:   step channel.Publish({User_Close})
6:   step channel.Publish({Close_Confirmed})

```

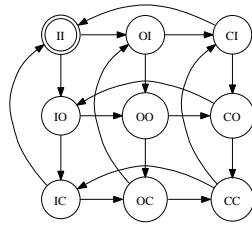
Spec 5. Test case for the asynchronous channel

```

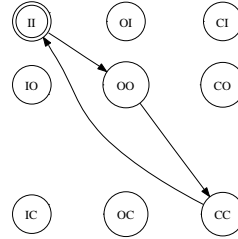
Initialized...
Client in Clnt_Idle, Server in Serv_Idle
Published events {User_Open}, observed events {Client_Open}
Client in Clnt_Open, Server in Serv_Idle
Published events {User_Close}, observed events {Client_Close}
Client in Clnt_Closing, Server in Serv_Idle
Published events {Close_Confirmed}, observed events {}
Client in Clnt_Closing, Server in Serv_Idle

```

(a)



(b)



(c)

Fig. 2. (a) Simulation of the *Asynchronizer* channel (b) Asynchronous product of client/server (c) Synchronous product of client/server

The *Asynchronizer* refinement does not describe the intended synchronization mechanism. Figure 2.c shows the topology of the correct interaction mechanism in terms of the product state-space. The states are label by the abbreviations **I**idle, **O**pen, and **C**losing. The correct refinement synchronizes the client/server so that both start in the idle state (**II**), and then proceed in lockstep to **OO**, and then to **CC**. This product FST precisely corresponds to the well-known synchronous product of finite state transducers. The interaction mechanism implemented by the *Asynchronizer* corresponds to the shuffle product of FSTs, and is shown in Figure 2.b.

Upon realizing this mistake, we must determine if the *Asynchronizer* refinement can be salvaged. Perhaps a further refinement of *Asynchronizer* will lead to the correct *Synchronizer* channel? Exploring this question requires a precise definition of refinement, for which there are many competing notions. For the purposes of illustration, we use a highly generalized notion of refinement. First, recall one definition of an *abstract state machine*.

Definition 1. Let Υ be a signature and \mathcal{U} be a set. Let $\mathcal{C}(\Upsilon, \mathcal{U})$ be the class of all algebras over signature Υ with universe \mathcal{U} . An abstract state machine $A = (L, I, \rightarrow)$ consists of a set of locations $L \subseteq \mathcal{C}(\Upsilon, \mathcal{U})$, a set of initial locations $I \subseteq L$, and a transition relation $\rightarrow \subseteq L \times \mathcal{T}_{\Upsilon}(\mathcal{U}) \cup \{\tau\} \times L$, where $\mathcal{T}_{\Upsilon}(\mathcal{U})$ is the term algebra generated by \mathcal{U} . The special element τ captures the notion of an unobservable transition. We borrow this symbol from automata theory, where $\xrightarrow{\tau}$ is called a silent transition.

Definition 2. Given two abstract state machines A_1 and A_2 , we say that A_2 refines A_1 if there exists a relation $\sim_L \subseteq L_1 \times L_2$ and an equivalence relation $\sim_T \subseteq (\mathcal{T}_T(\mathcal{U}) \cup \{\tau\})^2$ such that:

1. $\forall q \in I_1, \exists q' \in I_2 (q \sim_L q')$
2. $\forall q_1 \xrightarrow{\alpha} q_2 \in \rightarrow_1, \exists q'_1, q'_2 \in L_2 (q_1 \sim_L q'_1, q_2 \sim_L q'_2)$ and
3. $q'_1 \xrightarrow{[\tau]}^* p \xrightarrow{[\alpha]} p' \xrightarrow{[\tau]}^* q'_2 \subseteq \rightarrow_2$

The existence of such equivalence relations does not demonstrate the existence of a meaningful refinement, but rather that there might exist a meaningful refinement of A_1 by A_2 . Commonly employed notions of refinement are specializations of this broad definition. For example, A_2 *strongly simulates* A_1 if $\sim_T = id_{\mathcal{T}}$, no transition is silent, and \sim_L exists. A_2 *weakly simulates* A_1 if \sim_L exists, and $[\tau] = \{\tau\}$, i.e. the equivalence class of τ with respect to \sim_T only contains τ [14]. Property 3 allows the refinement to contain non-trivial events that can be considered equivalent to the silent transition. This captures macrosteps or stable states that are expanded by the refinement. Using this broader definition, we can investigate if some relevant notion of refinement exists between the synchronous and asynchronous channels. The guards (event names) of the synchronous refinement carry the same meaning as those of the asynchronous refinement, thus we shall take $\sim_T = id_{\mathcal{T}}$ to be the identity relation over terms. Given this restriction, there are too many distinct transitions for the asynchronous channel to be refined by the synchronous channel. Therefore, the only solution is to discard the asynchronous refinement, and restart the refinement process. (Strangely enough, it is possible for the asynchronous channel to refine the synchronous one.)

3 Backtracking and The Principle of Alternatives

Our client/server example appears trivial on the surface. However, upon further investigation it clearly illustrates a major obstacle for system architecture: An initial abstract specification characterizes a non-trivial *design space* containing all the possible designs that refine the specification [15]. Furthermore, the structure of this design space can be quite complicated. The refinement mechanism explores paths through the design space, but a single path may lead to a design such that all further refinements are not solutions. When this occurs, we must *backtrack* to a more abstract design, and choose a different refinement path. We call this phenomenon *refinement backtracking*; it caused us to discard the *Asynchronizer* solution. Figure 3.a shows a sketch of a design space in terms of a tree structure. The black dots represent designs that have been constructed, and the dashed regions stand for unexplored subspaces of the design space. The dotted path on the right-hand side of the tree is a correct refinement path to a solution. However, before this path can be found, a number of backtracks occur on the left side of the tree. These backtracking points waste precious design cycles and resources. Unfortunately, today's software systems are so complex that these mistakes are practically unavoidable.

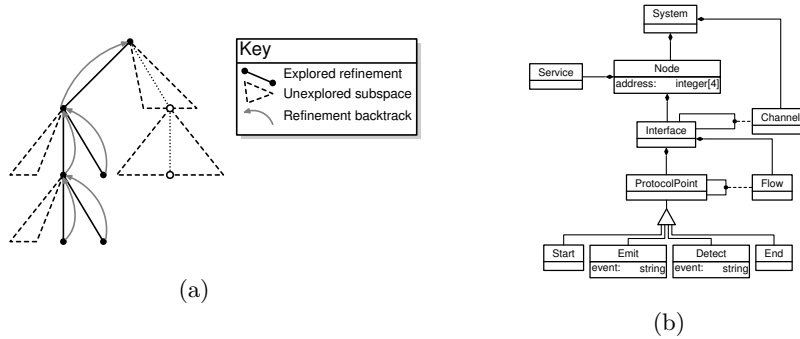


Fig. 3. (a) Example of refinement paths through a design space. (b) Metamodel of the client/server domain

Thus, with just cause, the model-based approaches expand on the principles of precision, abstraction, and refinement with the principle of *alternatives*. This principle accepts that a specification has many possible refinement branches (or alternatives), and provides tool support for simultaneously exploring multiple branches. Of course, accepting this view is the easy part, but providing tool support is more difficult. Ideally, we would like specify a system in such a way that multiple refinements can be automatically generated without significant intervention from the user. Notice that we prevented this in the client/server example by making FST an abstract class. This decision meant that the client/server could not be described (Specification 3) without first refining FST to some degree. In order to introduce the necessary tool support we must represent specifications differently. We now describe the *Model Integrated Computing* (MIC) approach, which represents one direction in model-based design.

MIC supports the principle of alternatives using three techniques. First, the representation of a design (e.g. Figure 1) is decoupled from the behavioral semantics of a design. This is accomplished by characterizing the structure of the problem domain separately from the behaviors of architectural elements. For example, Figure 3.b shows a *metamodel* characterizing the structure of the client/server problem domain. The boxes in the diagram are similar to classes, except that they do not contain any dynamic behavior. Instead, the class-like boxes name the structural primitives that make up *models* in the problem domain. The lines extending between boxes describe the legal mechanisms for instantiating the basic primitives. For example, the line extending from the box labeled *Node* to the box labeled *System* requires every occurrence of a *Node* element to be contained by an occurrence of a *System* element. Structural elements may have *attributes*, i.e. named key-value pairs. The *Node* element has an attribute *address*, which is an array of four integers. Each occurrence of *Node* carries some value for the *address* field. Structural elements may also be *relational*. For example, the structural element called *Channel* exists between two occurrences of *Interface*. Thus, *Channel* can be viewed as a binary relation over *Interfaces*. The *Interface* is the structural element for capturing protocols, which are represented

as flowchart-like structures. Each step of a protocol is a type of *ProtocolPoint* of which there are four specializations: A *Start* point marks the beginning of the protocol, an *Emit* points marks the emission of a message, a *Detect* point marks the detection of a message, and an *End* point marks the successful completion of the protocol. Finally, *ProtocolPoints* are sequenced by the *Flow* relation.

Formally, a metamodel characterizes a set of highly abstract invariants that all designs in the space must satisfy. We call these invariants the *structural semantics*. A *domain* is the collection of all design instances that satisfy the structural invariants defined by a particular metamodel. These invariants can become quite complex, but they can be effectively formalized as a constraint logic over the space of possible designs [16]. Practically, a metamodel can be used to rapidly generate a customized specification tool. For example, the *Generic Modeling Environment* (GME) will generate a customized specification tool from a metamodel [17]. Figure 4 shows the specification tool that was generated from the metamodel in Figure 3.b. Using this tool, we constructed a model $m_{c/s}$ of the client/server system at the purely structural level. The top-left view shows the client and server *Nodes* connected by a *Channel* through *Interfaces* named *Session*. The top-right view shows the inside of the client *Node*. Internally, there is one interface called *Session* and one *Service* called *SystemLogger*. Finally, the bottom view shows the inside of the *Session* interface, which describes the flow of the session protocol for the client. To summarize, by decoupling the structural semantics from the behavioral semantics we can construct abstract design instances without any refinement. One can view these instances (or models) as designs that can be retargeted onto many different refinement paths.

Returning to the principle of alternatives: Metamodeling and model construction is the first pillar for supporting exploration of multiple refinement paths. The second pillar is the *semantic unit* [2]. A semantic unit (SU) is a reusable representation of a particular behavioral semantics, which is defined independently of any problem domain. For example, the *Asynchronizer* implements the shuffle product in the context of a communication channel. However, this is not the only situation where a shuffle product might be useful (or useless), so there is

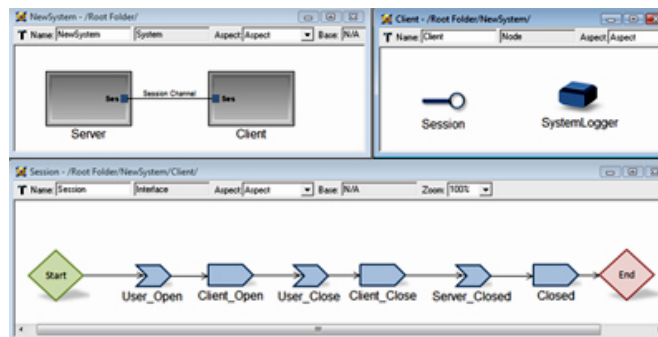
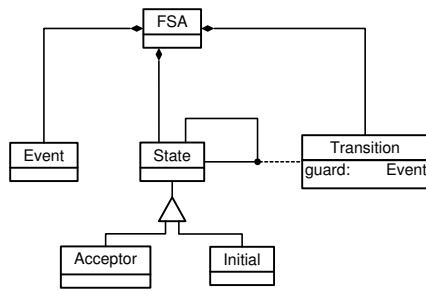
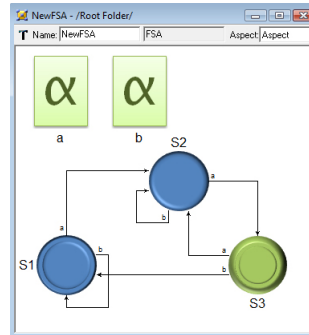


Fig. 4. Abstract model $m_{c/s}$ of the client/server system.

no reason to entwine its specification with the specification of the client/server system. A behavioral semantics is packaged into a reusable semantic unit by defining a triple (D, S, φ) . The set D is a domain, i.e. it is the set of all model structures that satisfy some invariants. The set S is the *semantic domain*, which is a set of ASMs. The mapping $\varphi : D \rightarrow S$ is the *semantic anchor*, which is an onto map from models to ASMs in the semantic domain. For every well-formed model structure $m \in D$, the map $\varphi(m)$ yields an ASM in S that is the behavioral semantics of model m . Figure 5.a shows how we might describe the domain for a



(a)



(b)

Fig. 5. (a) Metamodel of a finite state acceptor domain (b) Example model of a finite state acceptor

finite state acceptor (FSA) semantic unit. Though not shown, we might also add the constraint that every FSA has exactly one initial state, and no state has two transitions guarded by the same event (i.e. the FSA is deterministic). We then take the domain D_{FSA} of the semantic unit to be the set of all model structures that conform to the metamodel in Figure 5.a. Similarly, let the semantic domain S_{FSA} be the set of all finite *control state ASMs* [11] such that the control conditions only compare the current input against a known alphabet of events. The mapping φ_{FSA} is a simple program that generates ASML code from a GME model. It is easy to see that this map is onto. Given a particular FSA model, like that of Figure 5.b, the behavioral semantics can be recovered by applying φ_{FSA} ; Specification 6 shows the control state ASM generated from Figure 5.b. Though this example is fairly simple, ASMs have been used to define a number of non-trivial semantic units including hierarchical and concurrent automata, heterogeneous dataflow, timed automata, and classes of hybrid systems. Many of these semantics units are already available with the MIC toolsuite.

The final step of the MIC strategy is to refine the semantics of an arbitrary problem domain against one of these generic semantic units. Returning to the client/server example, we have defined the problem domain independently of any particular semantics with the metamodel of Figure 3.b, and we have

```

1: var ctl_state as State = S1
2: StepControl (e as Event)
3:   if ctl_state = S1 then
4:     if e = E.A then ctl_state := S2
5:     if e = E.B then ctl_state := S1
6:   if ctl_state = S2 then
7:     if e = E.A then ctl_state := S3
8:     if e = E.B then ctl_state := S2
9:   if ctl_state = S3 then
10:    if e = E.A then ctl_state := S2
11:    if e = E.B then ctl_state := S1

```

Spec 6. Control state ASM generated from model in Figure 5.b

developed a candidate solution (model) $m_{c/s}$ as shown in Figure 4. Imagine that there exist two semantics units $FST_{async} = (D_{async}, S_{async}, \varphi_{async})$ and $FST_{sync} = (D_{sync}, S_{sync}, \varphi_{sync})$. The FST_{async} SU captures the asynchronous product of FSTs for the domain of models D_{async} , while the FST_{sync} SU captures the synchronous product for the domain of models D_{sync} . If we could calculate $\varphi_{async}(m_{c/s})$ and $\varphi_{sync}(m_{c/s})$, then we would simultaneously generate two points along two different refinement paths without wasting the effort of coding these refinements. Unfortunately this calculation cannot be performed because the domains D_{async} and D_{sync} do not contain the same models as the client/server domain $D_{c/s}$. This problem can be fixed by introducing a *model transformation* that translates models from one domain to another domain. A model transformation $\tau : D_1 \rightarrow D_2$ is a mapping between domains. Given the appropriate model transformations $\tau_{async} : D_{c/s} \rightarrow D_{async}$ and $\tau_{sync} : D_{c/s} \rightarrow D_{sync}$, we can project the candidate solution down the two refinement paths via function composition:

$$r_{async} = (\varphi_{async} \circ \tau_{async})m_{c/s}, \quad r_{sync} = (\varphi_{sync} \circ \tau_{sync})m_{c/s}. \quad (1)$$

Thus, the third pillar for supporting alternatives is the ability to construct model transformations. These act as the glue between abstract structural semantics and behaviorally refined semantic units. The effectiveness of this entire process hinges on the cost of constructing model transformations. Thus, it is worthwhile to examine model transformations in more detail.

4 Model Transformations

Model transformations have a long history in model-based design. From the perspective of refinement, a model transformation is a generalization of a *refinement mapping* [18]. A refinement mapping ρ maps abstract specifications to refined specifications using a mechanism that has been fixed *a priori*. For example, a

refinement map ρ_{seq} might replace every parallel *forall* statement by a sequential *foreach* statement, refining the specification for execution on a sequential machine. Such a refinement map can be defined over the structure (or domain) of ASML descriptions, i.e. $\rho_{seq} : D_{ASML} \rightarrow D_{ASML}$. This property is important because ASML descriptions are finite, while the transition system described by an ASML description may be infinite. It would be impractical to define ρ_{seq} as a transformation over transition systems. Model transformations support arbitrary transformations between model structures; refinement mappings form one class of model transformation.

There are number of important issues to consider when using model transformations. How do users construct transformations? What are the semantics of a transformation? How do model transformations relate to the ASM formalism? This last question is important for understanding how model transformations act as reuse and composition mechanisms for semantic units. We begin with the first question, as it is easiest to answer. A model transformation is constructed from a set of rules, where each rule is of the form $m_L \Rightarrow m_R$. When a model transformation is applied to an input model h , all embeddings of m_L in h are found. For each embedding, the right-hand submodel m_r is created in the output model g using m_L as context. For example, the rule in Figure 6.a generates *States* of an FST from an abstract description of a protocol (e.g. Figure 4). A *State* is created for each subsequence of *Detect-Emit* found in a protocol. In order to achieve this, the left-hand pattern m_L , which is shown as boxes with solid borders, contains a *Detect* element connected to an *Emit* element through a *Flow* relation contained in an *Interface*. When a match occurs, the elements in the pattern are *bound* to the matching elements in the input model. The boxes with dashed borders represent the right-hand pattern m_R . These elements are created in the output model. Thus, a *State* element is created, which is contained by the *FST*. Also, relations are created between the new *State* and the matched *Detect*, *Emit* elements. These relations allows the transformation engine to “remember” which *Detect-Emit* pair created the *State*. (This rule is shown using the notation

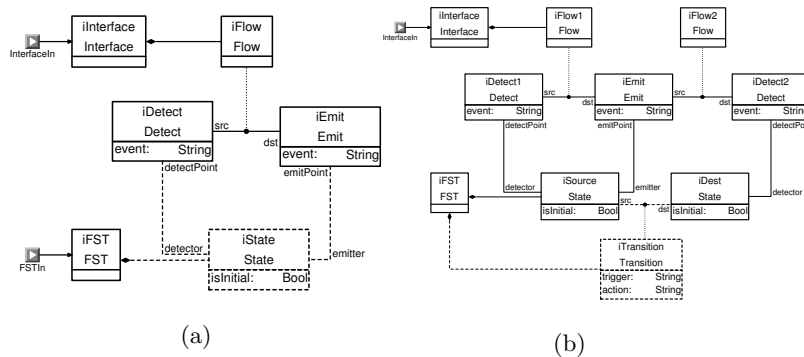


Fig. 6. (a) Transforms a protocol sequence into states (b) Creates transitions between the generated protocol states

of the *Graph Rewriting and Transformation* (GReAT) tool [3]. See also [19]). Transformation rules can be sequenced to execute in a particular order. In this example, the rule in Figure 6.a executes first to create all the *State* elements in the output *FST*. After this rule has completed, the rule in Figure 6.b creates *Transitions* between the *States*. A *Transition* is created for each subsequence of *Detect-Emit-Detect*. The *State* related to the first *Detect* serves as the source of the *Transition*, and the *State* related to the second *Detect* serves as the destination of the *Transition*. The full transformation τ_{FST} from protocols to FSTs requires one additional rule to handle that *Start* and *End* points of the protocol. Using just these three rules, we have anchored the models of domain $D_{c/s}$ to an FST semantic unit. Empirical studies in model transformations have shown them to be a compact and effective way manipulate models [20]. Thus, we argue the model transformation technology is suitable for implementing alternatives.

Model transformations have various mathematical underpinnings. One major class of formalizations grew from the graph-theoretic direction [4]. However, we have been pursuing a logico-algebraic approach that is compatible with abstract state machines. In particular, a metamodel characterizes a signature \mathcal{Y} whose function symbols are used to encode model elements. For example:

$$\begin{aligned} \mathcal{Y}_{Protocol} &= \langle detect(\cdot), emit(\cdot), flow(\cdot, \cdot), start, stop \rangle \\ \mathcal{Y}_{FST} &= \langle state(\cdot, \cdot), transition(\cdot, \cdot, \cdot), initial \rangle \end{aligned} \quad (2)$$

A model is a set of terms from the term algebra[21] of \mathcal{Y} generated by an (infinite) universe \mathcal{U} . A transformation rule is a logical formula for concluding new model terms from existing model terms [16]. The rule in Figure 6.a might be written as the formula (*Horn* clause):

$$\forall x, y \text{ detect}(x), emit(y), flow(x, y) \Rightarrow state(x, y) \quad (3)$$

Both the algebraic signatures and transformation formulas can be generated from the metamodels and graphical descriptions of the transformations. Our algebraic approach allows transformation formulas to be easily converted to an ASM that calculates the result of a model transformation. A *transformation ASM* manages collections of model terms. A *Term* is a structure that contains either a constant value such as a *String* or *Integer*, or it contains some function symbol f along with a list of arguments (which are also *Terms*). (In the interest of space, we do not show the specification of the *Term* structures.) All function symbols from the algebraic signatures are converted into ASM functions that construct the corresponding *Terms*. The specification below shows two such generated functions. The *FuncTerm* constructor takes the name of a function symbol, a list of arguments, and boolean value indicating if the constructed *FuncTerm* is

```

1: detect(t as Term) as Term
2:   return new FuncTerm("detect", [t], true)
3: flow(s as Term, t as Term) as Term
4:   return new FuncTerm("flow", [s, t], true)

```

a member of the input model. This will always be true for *Term* construction functions generated from the signature of the input metamodel, and false for those functions generated from the output metamodel. The transformation ASM must to keep track of two sets of terms. The set M contains all distinct terms that are either in the input model h or the output model g . The set T contains every distinct term in M and all distinct subterms of terms of M . In another words, T contains every term that has been encountered. The *create* function adds a *Term* t to M , if t is not already there, and also adds any new sub*Terms* to T . Using *create*, the *load_model* function constructs the input model. Each transformation

```

1: var M as Set of Term = {}
2: var T as Set of Term = {}
3: load_model()
4:   step create(detect("User_Open"))
5:   step create(emit("Client_Open"))
6:   step create(flow(detect("User_Open"), emit("Client_Open")))

```

rule is converted into a function that quantifies over T to find embeddings of m_L , and then constructs new terms according to m_R . The following statement implements the previous logical formula using the *found* function, which returns true if M contains a particular term t . The transformation rules are executed

```

1: step foreach x in T, y in T where
2:   found(detect(x)) and found(emit(y)) and found(flow(detect(x),emit(y)))
3:   create(state(x,y))

```

until a fix point is reached, at which point M contains all the *Terms* of the output model. This construction converts each transformation formula into exactly one ASML statement, illustrating how easily model transformations fit into the ASM paradigm.

5 Conclusion

The founding principles of abstract state machines - *precision*, *abstraction*, and *refinement* - are essential for architecting today's software systems. These principles allow us to view the architectural process as a large (and unfriendly) design space with potentially mutually exclusive refinement paths. Model-based design, in general, and model integrated computing (MIC), in particular, extend these principles with the principle of *alternatives*, which requires architectural tools to help manage the complex design space. MIC implements alternatives by (1) supporting the construction of models, which are abstract entities that can be easily refined; (2) providing a comprehensive set of semantic units, against which models are rapidly refined; (3) using model transformations as the glue between highly abstract models and highly specialized semantic units. By employing these techniques, many refinement paths can be quickly explored thereby reducing the amount of design cycles spent constructing dead end paths in the design space. ASMs are embedded in the MIC framework in the form of semantic units and transformation ASMs.

References

1. G. Karsai, J. Sztipanovits, A.L.T.B.: Model-integrated development of embedded software. *Proceedings of the IEEE* **91**(1) (January 2003) 145–164
2. K. Chen, J. Sztipanovits, S.N.M.E., Abdelwahed, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*. (September 2005)
3. Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: *ECBS. (2003)* 159–167
4. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)
5. D. Masys, D. Baker, A.B., Cowles, K.: Giving patients access to their medical records via the internet: the pcsso experience. *Journal of the American Medical Informatics Association* **9**(2) (2002) 181–191
6. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. *IEEE Computer* **36**(4) (2003) 45–52
7. Lee, E.A.: Absolutely positively on time: What would it take? *IEEE Computer* **38**(7) (2005) 85–87
8. Dijkstra, E.W.: The humble programmer. *Commun. ACM* **15**(10) (1972) 859–866
9. Wirth, N.: From programming language design to computer construction. *Commun. ACM* **28**(2) (1985) 159–164
10. Gurevich, Y.: *Evolving algebras 1993: Lipari guide*. (1995) 9–36
11. Börger, E.: Abstract state machines: a unifying view of models of computation and of system design frameworks. *Ann. Pure Appl. Logic* **133**(1-3) (2005) 149–171
12. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: *ISSTA. (2002)* 112–122
13. Harel, D., Naamad, A.: The state-mate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.* **5**(4) (1996) 293–333
14. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: *FOCS. (1995)* 453–462
15. Neema, S., Sztipanovits, J., Karsai, G., Butts, K.: Constraint-based design-space exploration and model synthesis. In: *EMSOFT. (2003)* 290–305
16. Jackson, E.K., Sztipanovits, J.: Towards a formal foundation for domain specific modeling languages. *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT'06)* (October 2006) 53–62
17. Institute For Software Integrated Systems: Gme 5 user's guide. Technical report, Vanderbilt University (2005)
18. Burch, J., Passerone, R., Sangiovanni-Vincentelli, A.: *Modeling techniques in design-by-refinement methodologies* (2002)
19. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: Viatra - visual automated transformations for formal verification and validation of uml models. In: *ASE. (2002)* 267–270
20. Neema, S., Kalmar, Z., Shi, F., Vizhanyo, A., Karsai, G.: A visually-specified code generator for simulink/stateflow. In: *VL/HCC. (2005)* 275–277
21. Burris, S.N., Sankappanavar, H.P.: *A course in universal algebra*. Springer-Verlag (1981)