

Effective Modeling of Software Architectural Assemblies Using Constraint Automata

Nikunj R. Mehta

University of Southern California, 941 W. 37th Pl., Los Angeles, CA 90048, USA

Email: mehta@usc.edu, URL:cse.usc.edu/~mehta

Marjan Sirjani¹ and Farhad Arbab

CWI, P. O. Box 94079, 1090 GB Amsterdam, The Netherlands

Email: Farhad.Arbab@cwi.nl, URL:www.cwi.nl/~arbab

ABSTRACT

Alfa is a framework for the construction of software architectures and their elements from architectural primitives. In any system involving events from multiple sources, synchrony and asynchrony between events arise naturally. Support for simultaneous synchrony and asynchrony, and scalability to assemblies of large numbers of architectural primitives are central concerns for effectively modeling software architectural assemblies in Alfa. An increasingly popular formalism for event-based modeling of the behavior of software architectures, labeled transition systems (LTS), was initially chosen to model the behavior of Alfa assemblies. However, this creates an impedance mismatch with the architect's mental model and lacks sufficient scalability. We therefore propose a formal approach to effectively model software architectural assemblies that addresses these limitations, using constraint automata. Constraint automata can be mapped to LTS thus utilizing existing techniques for analysis of behavioral properties. We evaluate the effectiveness of our approach using two application architectures assembled from Alfa's primitives.

2000 ACM Computing Classification System: D.2.4, D.2.11

Keywords & Phrases: software architectures, architectural primitives, timed data streams, constraint automata, labeled transition systems, Alfa.

1 Introduction

Software architectures [19,20] provide high-level abstractions in the form of coarse-grained processing, connecting, and data elements, their interfaces, and their configurations. It is widely believed that compositional approaches to software development (e.g., analogously to how this is done in computer hardware architecture [6]) are key to constructing large, distributed systems [18,19]. Our current work focuses on a constructive, compositional framework for software architectures called Alfa [16]. This technique provides a small number of primitives to assemble increasingly complex architectural elements, and improves the understanding and construction of software architectures in general.

An important objective of software architectures is to enable the reasoning about a system's overall properties. We model the behavior of software architectural assemblies, i.e., the assembly of software architectures and their elements from architectural primitives, as interrelationships among events of architectural significance, similar to some existing techniques [12,14].

Timed data streams [4], which are coordinated pairs of infinite streams of time and data appropriate for event-based modeling of software architectural behavior, form the basis of the behavioral model of Alfa's primitives and their assemblies. In timed data streams, an event is the occurrence of data at a *port* at a certain point in time, and it can be represented as a tuple (α, i) where α represents data contained in the event and i is the time at which the event occurs. For any two distinct events (α, i) and (β, j) , two relations are possible between i and j : 1) $i = j$, and 2) $i \neq j$. When $i = j$, the two events are synchronous, and when $i \neq j$, the two events are asynchronous.

Several analysis techniques have been proposed for determining behavioral properties of software architectures. Various formal methods such as communicating sequential processes (CSP) [1], labeled transition systems (LTS) [14], chemical abstract machines [10], and partially ordered sets [12] have been used to model and analyze the behavior of software architectures. Compositional approaches, such as Alfa, involve large numbers of primitives and significantly increase complexity in such models.

We initially considered LTS, an increasingly popular formalism for event-based modeling of software architecture behavior [15], to be suitable for modeling architectural assemblies because:

¹ Also at Sharif University of Technology, Tehran, Iran

1. LTS can be used to model a system in terms of events [9],
2. a system's behavior in terms of safety and liveness properties can be exhaustively analyzed using its LTS model [8,9],
3. its popularity would ease adoption of techniques that build upon LTS, and
4. extensive tool support is available for automated analysis of LTS models [9,13].

However, our attempts at modeling behavior of architectural assemblies using LTS proved unsatisfactory for two reasons: 1) an *impedance mismatch* between the LTS model and an architect's mental model of a system, and 2) inadequate scalability when modeling assemblies containing large numbers of Alfa primitives.

The first issue can be traced to a fundamental concern in event-based models of systems. In any system involving events from multiple sources, synchrony and asynchrony of their occurrences arise naturally. Any formalism that forces the choice of one as basic and models the other in terms of the basic abstraction falls short of reflecting the system concerns at the proper level of abstraction and causes an impedance mismatch.

In fact, Alfa's primitives allow simultaneous synchrony and asynchrony among events, i.e., if there is synchrony between events a and c , as well as between a and b , it does not imply that occurrences of c are synchronous with occurrences of b . An LTS model of such events represents the synchrony between a and c as a hybrid event ac , and that between a and b as ab , coincidences that correspond to primary events in the system. Further, this impedance mismatch prevents the composition of interacting LTS involving such events, thus disallowing its use for the purpose of modeling software architectural assemblies. In general, the ability to simultaneously model synchrony and asynchrony of events as first-class abstractions is an important need in software architectural modeling that we feel is largely unsatisfied in existing techniques.

The second issue is related to the complexity of LTS models that contain large numbers of Alfa primitives assembled in a software architecture. The attempt to model the correct timing relations among architectural events results in LTS models with a large number of states and transitions, making it difficult to construct and evaluate such models interactively. We describe an example software architecture assembled from Alfa's primitives that illustrates both these issues.

We have attempted to overcome these limitations by defining a novel approach for modeling the behavior of architectural assemblies from Alfa's primitives, using constraint automata. Arbab et al. introduced constraint automata in [5] for model checking Reo connector circuits. Constraint automata can be considered generalizations of probabilistic automata, where data constraints, instead of probabilities, label state transitions and influence their firing. Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. The simplest connectors in Reo are a set of channels with well-defined behavior supplied by users [3].

Although constraint automata can be mapped to LTS, which enables us to tap into existing analysis tools and eases the learning curve for users of our techniques, we also propose that constraint automata are the appropriate conceptual model for assembling architectures from primitives. Further, in our observation the abstraction techniques afforded to us by constraint automata drastically simplify the composition and minimization of state machines that model software architectural assemblies, thus making it possible to interactively create their effective models.

The main contribution of our work presented in this paper is a novel approach for effectively modeling software architectural assemblies to reflect their behavioral properties while maintaining a mapping to the architect's mental model. This approach also integrates with existing LTS-based techniques for determining properties of behavioral models. We evaluate our approach using two different application architectures assembled from Alfa's primitives.

The remainder of this paper is organized as follows. In Section 2, we give an overview of Alfa. Next, in Section 3, we motivate the need for an effective formalism for modeling software architectural assemblies. In Section 4, we describe the limitations of LTS that make it unsuitable for modeling simultaneous synchrony and asynchrony in software architectures. Section 5 describes our approach for addressing issues identified in the motivation, using constraint automata and their composition. Section 6 evaluates the effectiveness of our approach using two application architectures assembled from Alfa's primitives. We conclude with pointers to future work in Section 7.

2 Alfa

Compositional approaches toward software architecture have recently emerged as research efforts focus on the process of constructing software architectures [3,11,16,21]. Constructing architectural elements and complete software architectures from primitives improves their systematic understanding. In this section, we describe Alfa [16], an *assembly language for software architectures*, used in the rest of this paper to create architectural assemblies.

2.1 Alfa's primitives

Alfa is a framework for understanding and constructing style-based architectures from a small set of architectural primitives. Alfa's seventeen architectural primitives are classified based on five orthogonal characteristics of architectural styles: structure, behavior, interaction, topology, and data.

Alfa employs point-to-point communication channels, called *ducts*, to tie together computational elements of a style. Each duct provides input/output behaviors to communicate data elements and, as a result, synchronize the communicating elements. In Alfa, similarly to Reo [2], software components are treated as black boxes of functionality that relate inputs and outputs, whereas software connectors have a visible structure made of Alfa's primitives. Alfa's primitives consist of eight nouns, capturing the *form* of architectural style elements, and nine verbs capturing the elements' *function*¹:

1. Data - DATUM
2. Structure - PARTICLE, OUTPUT, INPUT, TWOWAY
3. Interaction - DUCT, RELAY, BIRELAY, HOLDS, LOSES
4. Behavior - CREATE, SEND, RECEIVE, HANDLE, REPLY
5. Topology - CONNECT, DISCONNECT

In a given style, DATUM is the data type of data items exchanged. PARTICLE is the locus of computing in software architectures. It is a container for the behavior of a processing element, and provides INPUT and OUTPUT portals for interacting with its environment. When required a single INPUT and a single OUTPUT portal can be combined into a TWOWAY port for bidirectional communication, which results in a single identity to be assigned to both portals. Both INPUT and OUTPUT portals define DATUMS that can be received from or sent to that portal.

The means of interaction between PARTICLE primitives are DUCT, RELAY, and BIRELAY primitives. A DUCT contains two ends, which are either INPUT or OUTPUT portals. Every DUCT is a FIFO queue that provides two functions—HOLDS and LOSES—to determine its communication characteristics. A DUCT can hold data items up to its HOLDS capacity, which is a whole number. A DUCT with zero (non-zero) capacity is synchronous (asynchronous). A DUCT can lose data items written to it based on its LOSES function, which can take either of four values: *none*, *initial*, *first*, and *last*. The *initial* LOSES characteristic produces a DUCT that is initialized with a single data item. When the DUCT is full, *last* LOSES characteristic allows a DUCT to lose the incoming data item, and *first* LOSES characteristic results in a loss of the oldest data item.

A RELAY contains multiple INPUT and OUTPUT portals. Data items from each INPUT portal are forwarded to every OUTPUT portal of the RELAY. A slightly more powerful primitive is required for bidirectional communication, called a BIRELAY. This primitive performs routing of reciprocal communication back to the TWOWAY port that initiated the communication. A BIRELAY contains multiple initiator TWOWAY (where the bidirectional communication originates) and terminator TWOWAY ports (where the bidirectional communication ends).

Behavioral primitives are used to enact interaction and instantiate primitives. The CREATE function is used to create instances of a PARTICLE, which in turn may result in the instances of the contained forms—any of PARTICLE, INPUT, OUTPUT, TWOWAY, DUCT, RELAY, and BIRELAY—to be created automatically and recursively. The SEND function is used to synchronously write a data item at the OUTPUT end of a DUCT. The RECEIVE function is used to synchronously read a data item at the INPUT end of a DUCT. The HANDLE function allows the PARTICLE to continue its processing while data items are asynchronously removed from the DUCT whenever they become available, which are then handed off to the PARTICLE for processing. Finally, the REPLY function is used to respond to a previously received data item, and results in routing information to be added to the data item being sent in the reply so that it reaches the TWOWAY port where the communication originated. Alfa's topological function primitives are not discussed here since they do not generate any interaction.

Alfa's primitives have been compared with Reo channels in [16] to evaluate their expressiveness. Alfa's primitives INPUT, OUTPUT, DUCT, RELAY, HOLDS, and LOSES can be used to model all primitive Reo channels and its merge and replicate operators described in [3]. Moreover, Arbab shows that Reo's primitive channels together with the merge and replicate operators are expressive enough to model any interactions involving a regular expression of input/output operations on point-to-point channels. By analogy, it can be said that Alfa's set of primitives is just as expressive.

2.2 Modeling Alfa's primitives using FSP

The behavioral semantics of Alfa's primitives are formalized using an LTS-based formal notation, FSP [13], which has been previously used for modeling software architectures. These FSP models serve as the building blocks for

¹ SMALL CAPS are used to identify Alfa primitives. Moreover, FORM primitives are written as nouns with the initial letter capitalized, whereas FUNCTION primitives are written as verbs.

composing models of software architectural assemblies and are used in the rest of this paper. Later we will show that these models produce impedance mismatches that arise when synchrony and asynchrony are not treated as first-class abstractions in labeled transition systems.

The OUTPUT primitive can be modeled as an FSP primitive process as:

```
OUTPUT = (send -> output -> OUTPUT).
```

The use of two actions creates *virtual synchronization* between the ends of a DUCT. In order to approximate the effect of simultaneous synchrony and asynchrony using LTS, we represent synchrony as blocking behavior on the sender(s) until all the synchronized receiving events have occurred. This is necessary to allow events to occur at the other end of the DUCT in virtual synchrony with the events at this OUTPUT. However, only the `send` action is externally visible; the `output` action is artificially inserted to for virtual synchrony. A simpler model is not possible, and such division of a single primitive behavior (SEND) into two FSP actions (`send` and `output`) is the beginning of an impedance mismatch between the architect’s mental model of the system, and its analysis model. Similarly to OUTPUT, an INPUT can be modeled as:

```
INPUT = (input -> receive -> INPUT).
```

Here too, the RECEIVE primitive behavior is split into two FSP actions `input` and `receive`. A DUCT with `HOLDS = 0` and `LOSES = none` is modeled in FSP as a parallel composition of an INPUT and an OUTPUT as:

```
||SYNC_DUCT = (INPUT || OUTPUT) / {send/input, receive/output}.
```

The composition involves relabeling of actions to achieve the desired synchronization. The actions `send` and `receive` are virtually synchronized by the relabeling of `input` as `send` and `output` as `receive`.

The FSP model of a DUCT with `HOLDS ≠ 0` and `LOSES = none` involves the use of a buffer to temporarily store events. Using a parameter n to represent the HOLDS property of a DUCT with one INPUT and one OUTPUT, we model this DUCT as:

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N] = (when (i<N) put->COUNT[i+1]
                  |when (i>0) get->COUNT[i-1]).
||FIFO_DUCT(N=5) = (OUTPUT || INPUT || BUFFER(N)) /
                  {send/put, receive/get} \
                  {output, input}.
```

This model explicitly uses the asynchronous form of OUTPUT and INPUT primitives by ignoring the actions `output` and `input`.

A RELAY, which synchronously replicates each received data item to all its outputs, is modeled as a parameterized FSP process in terms of the number of INPUTS (M) and OUTPUTS (N).

```
RELAY(M=5, N=5) = (expect[i:1..M] -> REPLICATE[i][N]),
REPLICATE[i:1..M][j:0..N] =
  (when (j > 0) prop[j]-> ack[j]-> REPLICATE[i][j-1]
   |when (j == 0) join[i] -> RELAY).
```

The RELAY FSP process is, in turn, composed with the necessary number of INPUT and OUTPUT processes to obtain the necessary multi-point interaction behavior.

```
||RELAY_INTERACTION(M=2,N=2) = (input[i:1..M]:INPUT || RELAY(M,N) ||
                                output[j:1..N]:OUTPUT) / {
                                input[i:1..M].input/expect[i],
                                input[i:1..M].receive/join[i],
                                output[j:1..N].send/prop[j],
                                output[j:1..N].output/ack[j]}.
```

Note that the virtual synchrony in this complex model ensures that an INPUT, which receives events, blocks while the event is replicated to all the OUTPUTS of the RELAY, and only once acknowledgements are received from all OUTPUTS reporting the consumption of the event does the original INPUT unblock. This model also introduces an ordering of replication as well as acknowledgements from recipients, which do not exist in the architect’s mind, causing further impedance mismatch as well as a significant addition to the model’s complexity. In the next section, we use these FSP behavior models of some of Alfa’s primitives to compose the model of a software architectural assembly.

3 Motivating Example

In this section, we discuss an example to demonstrate a heretofore unfulfilled need for modeling simultaneous synchrony and asynchrony in software architectural assemblies. We also demonstrate that LTS, a popular formalism for modeling the behavior of software architectures in terms of events [15], causes an impedance mismatch and produces a relatively complex model for a fairly simple software architectural assembly. Effective modeling of software architectural

assemblies requires the selection of an underlying formalism that faithfully and understandably represents primitive and composite behaviors, and produces compact analysis models.

The example application is a *Bank Outdoor Display* system, which alternately displays time and temperature on a screen. The software architecture of this system, as shown in Figure 1a, contains three components namely, *temperature sensor*, *clock*, and *display*, plus an explicit connector called *alternator* to assemble these components. The PARTICLE temperature sensor contains an OUTPUT called *temp*, the PARTICLE clock contains an OUTPUT called *time*, the PARTICLE display contains an INPUT called *text*. Moreover, the PARTICLE alternator contains two INPUTS called *a* and *b*, and one OUTPUT called *c*. In a typical ADL, the behavior of the alternator would be expressed as a formal relationship among its three roles. In a compositional approach, however, the alternator is decomposed further using architectural primitives. Alfa's architectural primitives, described in the previous section, are used to assemble the alternator as shown in Figure 1b.

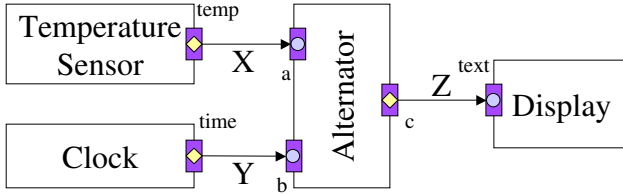


Figure 1a. Bank Outdoor Display

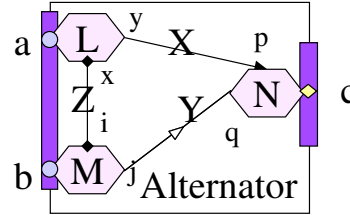


Figure 1b. Alternator assembled from Alfa's primitives

The alternator is assembled from three Alfa RELAYS (L, M, and N) shown as hexagons and three DUCTS (X, Y, and Z) shown as lines with adornments. This assembly alternates the INPUTS *a* and *b* at the OUTPUT *c*, also expressed through the regular expression $c = (ab)^*$ as shown in [3] for a similar Reo connector. DUCT X (holds = 0, loses = none) synchronously transports events from OUTPUT *y* of L to the INPUT *p* of N, DUCT Z (holds = 0, loses = none) requires simultaneous events on OUTPUTS *x* and *i* of RELAYS L and M, respectively, and DUCT Y (holds = 1, loses = none) buffers up to one event between the OUTPUT *j* of RELAY M and the INPUT *q* of RELAY N.

The assembly of the alternator in Figure 1b creates synchrony between *a* and *c*, as well as between *a* and *b*, even though occurrences of *c* are asynchronous with occurrences of *b*. In LTS, this distinction is lost as the synchrony relation is transitive, i.e., if *a* is synchronous with *b*, and *a* is synchronous with *c*, then *b* is synchronous with *c*. This is contrary to the actual relationship among occurrences of *a*, *b*, and *c*, leading to our observation that LTS cannot natively model simultaneous synchrony and asynchrony.

An LTS model of the alternator using virtual synchrony can still be created as a parallel composition of processes representing the primitives used in it. The FSP model of the alternator with proper relabeling of events for virtual synchronization is:

```

||ALTERNATOR = ({a,b}:INPUT || c:OUTPUT || n:RELAY(2,1) || y:FIFO_DUCT(1) ||
  {l,m}:RELAY(1,2))/{
  a.input/l.expect[1], a.receive/l.join[1],
  b.input/m.expect[1], b.receive/m.join[1],
  c.send/n.prop[1], c.output/n.ack[1],
  l.prop[2]/m.prop[2], //Duct Z
  l.prop[1]/n.expect[1], //Duct X
  l.ack[1]/n.join[1], //Duct X
  y.send/m.prop[1], //Duct Y
  y.receive/n.expect[2] //Duct Y
}@{{a,b}.{receive,input},c.{output,send}}.

```

Note, however, that all externally visible ports are modeled with a two-step synchronization as that is necessary if this process is to be composed with other processes. The overall *Bank Outdoor Display* software architectural assembly is modeled using the FSP model of the alternator after relabeling actions corresponding to the three DUCTS (HOLDS = 0 and LOSES = none) as shown below:

```

||BOD = ({temp, time}: OUTPUT || text:INPUT || alternator: ALTERNATOR)/{
  temp.send/a.input, //models duct X
  a.receive/temp.output,
  time.send/b.input, //models duct Y
  b.receive/time.output,
  c.send/text.input, //models duct Z
  text.receive/c.output
}@{{temp, time}.send, text.receive}.

```

Only the externally visible events are defined in the interfaces of the components of the Bank Outdoor Display architecture, which aids in its minimization through abstraction of internal details of the composite process. When composed and minimized incrementally and during composition, this model forms a state space of 2^{10} and results in an unminimized LTS with 80 states and 166 transitions. Upon minimization, the resulting LTS has 25 states and 46 transitions as shown in Figure 2. The synchronization actions at the periphery are hidden and only the externally visible events are present in this LTS.

This approach has two limitations:

1. The FSP model contains an arrangement of events in the resultant state machine that has an *impedance mismatch* with the architect's mental model in terms of synchronization and ordering of events.
2. The relatively large state machine is an indicator of drastically worse problems that occur when dealing with compositions of more primitives. Such rapid increase in complexity to the extent that LTS composition places unrealistic demands on computational resources makes it impractical to interactively design architectural assemblies modeled as LTS.

4 Limitations of Using LTS

We demonstrated the issues that arise when using virtual synchrony in LTS to approximated simultaneous synchrony and asynchrony. Now we formally argue for the insufficiency of LTS for modeling architectural assemblies.

A labeled transition system M can be modeled as a quadruple $\langle \Theta, E, \delta, \theta \rangle$ in terms of events including an externally invisible event τ where:

Θ is a finite set of states,

E is a finite set of externally observable events of M ,

$\delta \subseteq \Theta \times E \cup \{\tau\} \times \Theta$ is a finite set of transitions, and

$\theta \in \Theta$ is the initial state.

In order to represent simultaneous synchrony and asynchrony of externally observable events a, b , we create the powerset of E representing all possible synchronous occurrences of events in E . Then by representing every element in the powerset of E as a single unique event label, we get the set E' , the universe of events in M' .

For example, if $E = \{a, b, c\}$, then

the *powerset* $E = \{\{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$,

and, say, $E' = \{a, b, c, ab, bc, ac, abc\}$.

Now a modified LTS M' which can support simultaneous synchrony and asynchrony is defined as a quadruple $\langle \Theta', E', \delta', \theta_0' \rangle$ where:

Θ' is a finite set of states,

E' is a set of simultaneously synchronous and asynchronous events of M ,

$\delta' \subseteq \Theta' \times E' \cup \{\tau\} \times \Theta'$ is a finite set of transitions, and

$\theta_0' \in \Theta'$ is the initial state.

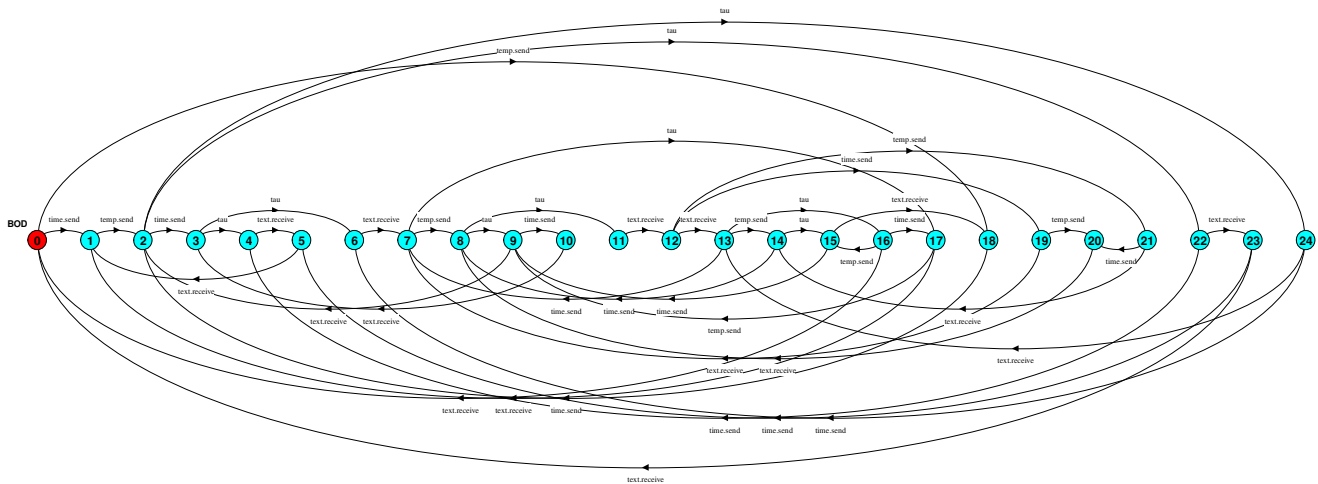


Figure 2. Minimized LTS of Bank Outdoor Display with 25 states and 46 transitions

M' has an impedance mismatch with the architect's mental model due to the fact that it introduces events that are not a part of the set of externally observable events E . Another issue arises with the regular composition of LTS M_1' with another LTS M_2' to produce $M_{12}' = \langle \Theta_{12}', E_{12}', \delta_{12}', \theta_{12}' \rangle$. The parallel composition of interacting LTS is performed by matching events in the processes being composed. The parallel composition of models that contain simultaneous synchrony and asynchrony should take into account transitions that overlap in terms of the events involved. For example, a transition that involves only the event a in E matches the synchrony of events a and b in E' , if $b \notin E$, for the purposes of parallel composition. However, since the events of M_1' and M_2' indirectly represent the externally observed events of M_1 and M_2 , this deduction is not possible in the LTS itself and the parallel composition of M_1' and M_2' is rendered incorrect.

At the core of this problem is the impedance mismatch between the LTS model and the architect's mental model. The impedance mismatch is the relation $\mu: E \rightarrow E'$. In order to correctly compose M_1' and M_2' , the knowledge of μ is required. However, this additional information is absent in the LTS model implying that it is not possible to correctly model the timing behavior of events in composition of regular LTS. This finding lies at the heart of our approach for developing a more powerful scheme, based on constraint automata, for modeling behavior of software architectural assemblies involving simultaneous synchrony and asynchrony.

5 Constraint Automata

As shown in the previous section, labeled transition systems, a common technique for modeling the behavior of software architectures are inadequate for modeling the timing relations between events in software architectural assemblies, leading to a breakdown in their composition. In this section we describe constraint automata as a formalism for modeling such behavior, their systematic composition, and give an algorithm for mapping a constraint automaton to its equivalent LTS model using FSP.

Constraint automata were introduced by Arbab et al. in [5] as a formalism to capture the operational semantics of Reo. Timed data streams, which constitute the foundation of the coalgebraic semantics of Reo, are also the referents in the language of constraint automata.

5.1 Definition of Terms

In this section we introduce the notion of *constraint automata*. Let V be any set. We define the set V^ω of all streams (infinite sequences) over V as $V^\omega = \{\alpha \mid \alpha : \{0, 1, 2, \dots\} \rightarrow V\}$. For convenience, we consider only infinite streams and infinite "runs" of our automata, although finite runs can be modeled as well¹. We denote individual streams as $\alpha = (\alpha_0, \alpha_1, \alpha_2, \dots)$ (or $a = (a_0, a_1, a_2, \dots)$). We call α_0 the *initial value* of α . The (*stream*) *derivative* α' of a stream α is defined as $\alpha' = (\alpha_1, \alpha_2, \alpha_3, \dots)$. Note that $(\alpha')_n = \alpha_{n+1}$, for all $n \geq 0$. We recall the definition of timed data streams from [4]:

$$TDS = \{ \langle \alpha, a \rangle \in Data^\omega \times \mathbb{R}_+^\omega \mid \forall n \geq 0 : a_n < a_{n+1} \text{ and } \lim_{n \rightarrow \infty} a_n = \infty \}$$

A timed data stream $A = \langle \alpha, a \rangle$ represents occurrence of events at a port A and consists of a *data stream* $\alpha \in Data^\omega$ and a *time stream* $a \in \mathbb{R}_+^\omega$ consisting of increasing positive real numbers. The time stream a indicates for each data item α_n the moment a_n at which it occurs at a port A .

Constraint automata can be viewed as acceptors for tuples of timed data streams that are observed at certain ports A_1, \dots, A_n . The rough idea is that such an automaton observes the data occurring at A_1, \dots, A_n and either changes its state according to the observed data or rejects the data if there is no corresponding transition in the automaton. Further, constraint automata are augmented with the names of their ports A_1, \dots, A_n , where A_i stands for the i^{th} TDS. Each transition in a constraint automata is labeled with a pair n, g such that n is a non-empty subset of $N = \{A_1, \dots, A_n\}$, and a guard g that constrains data in the TDS of ports referenced in n .

We recall the definition of a constraint automaton from [5] as a quadruple $C = (Q, N, T, q_0)$ where

Q is a finite set of states,

N is a finite set of names,

$T \subseteq Q \times 2^N \times DC \times Q$, is a finite set of transitions of C ,

$q_0 \in Q$ is the initial state.

We write $q \xrightarrow{n, g} p$ instead of $(q, n, g, p) \in T$ and call n the name set and g the guard set of the transition.

The intuitive operational behavior of a constraint automaton is as follows. It starts in its initial state q_0 . If the current state is q , then C waits until data items occur at some of its ports A_1, \dots, A_n . Suppose data item d_1 occurs at A_1 and data item d_2 at A_2 while (at this moment) no data is observed at the other ports A_3, \dots, A_n . This triggers the automaton to check the

¹ Finite runs of the automata can be transformed to infinite runs using a τ self-transition on terminal states.

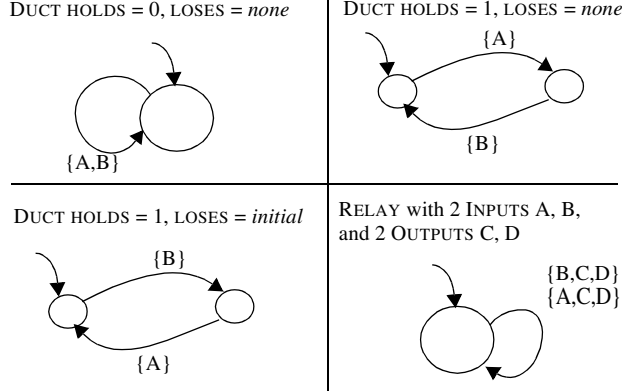


Figure 3. Constraint automata of Alfa's primitives

data constraints of the outgoing transitions of state q with a name set $\{A_1, A_2\}$ to choose a transition t , such that its guard is satisfied by d_1 and d_2 resulting in state p^1 . If there is no $\{A_1, A_2\}$ -transition from q whose data constraint is fulfilled then A rejects.

To see constraint automata in action, consider the behavior models of Alfa's interaction primitives used in this paper as shown in Figure 3. At the top left is the model of an Alfa DUCT used for synchronous communication. Operations on both ends of the DUCT must happen simultaneously for any progress to occur. Thus when an INPUT (A) and an OUTPUT (B) are used on either ends of the DUCT, interaction succeeds only when the OUTPUT is ready to SEND, and input is ready to RECEIVE. This appears to be a rather succinct way of describing the synchronous nature of interaction between A and B. The top right quadrant shows an asynchronous DUCT with an INPUT portal A and an OUTPUT portal B, which can buffer one data item and is initially empty. The bottom left quadrant also shows an asynchronous DUCT that is initially full. Finally, the bottom right quadrant shows a RELAY primitive used with two INPUTS A and B, and two OUTPUTS C and D. SENDING on either INPUT will only succeed only when both OUTPUTS are ready to RECEIVE. Further, the SENDING on INPUT A is asynchronous with SENDING on INPUT B. This is an example of simultaneous synchrony and asynchrony modeled using constraint automata.

5.2 Composition and Hiding

Having defined constraint automata in terms of states and transitions, we now define the composition of two constraint automata from [5] mathematically.

The product automaton of the two constraint automata $C_1 = (Q_1, N_1, T_1, q_{0,1})$ and $C_2 = (Q_2, N_2, T_2, q_{0,2})$ is:

$$C_{12} = (Q_1 \times Q_2, N_1 \cup N_2, T_{12}, q_{0,1} \times q_{0,2})$$

where T_{12} is defined by the following rules:

$$\frac{q_1 \xrightarrow{n_1, g_1} p_1, q_2 \xrightarrow{n_2, g_2} p_2, n_1 \cap N_2 = n_2 \cap N_1}{\langle q_1, q_2 \rangle \xrightarrow{n_1 \cap n_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

and

$$\frac{q_1 \xrightarrow{n_1, g_1} p_1, n_1 \cap N_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{n_1, g_1} \langle p_1, p_2 \rangle}$$

and the latter's symmetric rule.

In addition to composition, we need to hide events that are not externally visible. Hiding leaves the that are externally invisible out of the name sets of transitions. However, reachability of states should not be altered when hiding events. Details of hiding are given in [5].

5.3 Mapping to LTS

Model checking of constraint automata without data constraints does not require the development of new tools since it can be mapped to traditional techniques for non-deterministic finite state machine analysis. This approach leverages existing tools and methods for determining behavioral properties while at the same time providing the right

¹ In the rest of this paper, for simplicity, we use guard conditions that always succeed, i.e. *true*. Arbab et al. [5] give examples of other guard conditions.

1. Label every state in the constraint automaton distinctly.
2. Beginning with the initial state, add all the states of the constraint automaton to a list of untraversed states.
3. Select the first untraversed state.
 - 3.i. Generate a primitive process for the state using its label.
 - 3.ii. For each transition of the untraversed state
 - 3.ii.a. Generate a label for the transition from the onto function μ using the set of synchronous events of the transition.
 - 3.ii.b. Generate an action prefix as an event with the above label leading to a sub-process identified using the label of the resulting state.
 - 3.ii.c. If there is at least one remaining transition, create a choice.
 - 3.iii. Remove the current state from the untraversed list.
 - 3.iv. If there are more untraversed states, mark continuation of the process.
4. Mark an end of the generated process.

Figure 4. Generating FSP from a constraint automaton

level of abstraction for modeling software architectural behavior. We have chosen to transform constraint automata to LTS for the same reasons, as described in the introduction, that make LTS attractive for use in modeling behavior of software architectures. Moreover, we generate the FSP equivalent of an LTS as it allows us to use the LTSA tool to determine safety and liveness properties [14,15].

Generating an FSP process that is equivalent to some constraint automaton (CA) involves naming every state of the CA and mapping every CA transition to an LTS event label. LTS event labels can be generated using an onto mapping function μ from the name sets of CA transitions. The details of an algorithm for generating the FSP from a given CA are given in Figure 4. The algorithm ensures that each state and each transition in the constraint automaton is visited only once. The resulting LTS is already minimized and can be directly analyzed for its safety and liveness properties using existing techniques [14,15]. Of course, the safety and liveness properties must first be mapped from constraint automata to LTS, but a discussion of this issue falls outside the scope of this paper. Results of the analysis can then be inversely mapped to their constraint automata equivalents. For example, a sequence of LTS events leading to a safety property violation can be mapped to their CA counterpart using the inverse of μ described above.

Note, however, that the use of LTS for analyzing software architectural assemblies does not substitute constraint automata, but merely enables its automated analysis. Any further composition of this model with similar models cannot be correctly performed at the level of LTS itself as discussed earlier and visualized in Figure 5.

In summary, constraint automata serve as the appropriate modeling technique for software architectural assemblies involving simultaneous synchrony and asynchrony. We demonstrate this further in the next section using examples on assembling software architectures from primitives.

6 Modeling Software Architectural Assemblies Using Constraint Automata

In this section we illustrate and evaluate constraint automata for modeling software architectural assemblies using two examples: the Bank Outdoor Display system, discussed earlier in Section 3, and a pipe-and-filter style architecture.

$$\begin{array}{ccc}
C_1 = (Q_1, N_1, T_1, q_{0,1}) & \Longrightarrow & M_1 = (\Phi_1, E_1, \delta_1, \theta_{0,1}) \\
\bowtie \text{ compose} & & \bowtie \text{ compose} \\
C_2 = (Q_2, N_2, T_2, q_{0,2}) & \Longrightarrow & M_2 = (\Phi_2, E_2, \delta_2, \theta_{0,2}) \\
= & & = \\
C_{12} = (Q_{12}, N_{12}, T_{12}, q_{0,12}) & \Longrightarrow & M_{12} = (\Phi_{12}, E_{12}, \delta_{12}, \theta_{0,12})
\end{array}$$

Figure 5. Connecting constraint automata to LTS

The effectiveness of constraint automata in modeling software architectural assemblies can be gauged from the understandability of models created using them, and the sizes of resulting models. The greater a model's proximity to concepts being modeled, the higher its understandability. In the case of software architectures, behavioral models are treated as events taking place in the architecture, their ordering and synchronization. The ability to model a system according to its naturally present event order and synchronization leads to greater understandability. The size of a resulting model can be evaluated in terms of the numbers of its states and transitions.

6.1 Bank Outdoor Display

Our first example is the Bank Outdoor Display introduced in Section 3. First consider the alternator in the architecture of this system. The alternator consists of two RELAYS with one INPUT and two OUTPUTS ($(a; x, y)$ and $(b; i, j)$), one RELAY with one INPUT and two OUTPUTS ($(p, q; c)$), a DUCT, which has an INPUT and an OUTPUT, with $HOLDS = 0$ and $LOSES = none$ ($y; p$), a DUCT, which has an two outputs, with $HOLDS = 0$ and $LOSES = none$ ($x; i$), and a DUCT, which has an INPUT and an OUTPUT, with $HOLDS = 1$ and $LOSES = none$ ($j; q$). The constraint automaton for each of these primitives is already given in Figure 3. Here we compose the constraint automata to produce the behavior model of the alternator. Figure 6 shows the process of obtaining this composite constraint automaton.

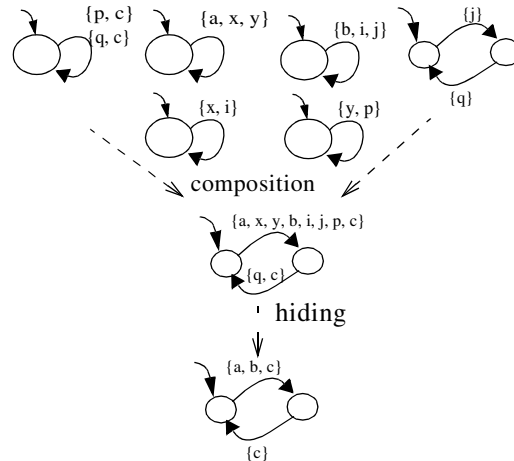


Figure 6. Deriving constraint automaton for alternator

Next the model of the Bank Outdoor Display architecture is obtained using the automaton for an alternator, and three DUCTS. This produces an automaton that looks similar to the one for an alternator, except the event names a and b are replaced by $time$ and $temp$ respectively, and the label c is replaced by $text$. This process is also shown graphically in Figure 7.

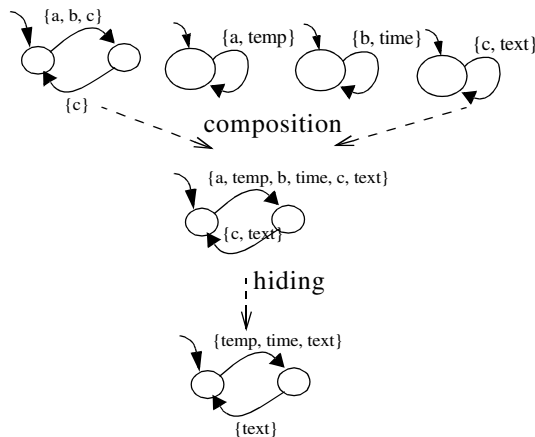


Figure 7. Bank Outdoor Display constraint automaton

The resulting model contains only transition labels that directly correspond to architectural events. Moreover synchronous events are identified as being in the same transition's name set. Events that are always asynchronous never appear together in a transition's name set. Finally, this model is compact as it contains just two states and two transitions.

6.2 Key-word-in-context

To demonstrate the effectiveness of constraint automata for modeling style-based software architecture assemblies, we describe the construction of the key-word-in-context (KWIC) architecture [20] built using the pipe-and-filter style from Alfa's primitives. The KWIC system was originally proposed as a model problem for studying system organizations [18] and has been used for studying style-based architectural design [20]. The KWIC system inputs an ordered set of lines, "circularly shifts" the first word of every line to the end of that line, and outputs all circular shifts of all lines in an alphabetical order. The architecture assembled in this example is due to Shaw [http://www.cs.cmu.edu/~ModProb].

We start with a definition of the composition of elements of the pipe-and-filter style using Alfa's primitives as shown in Figure 8a. This assembly defines the required primitives and their topology as a template which can later be used in architectures built using this style. The style element *pipe*, with a *source* and a *sink* interface, is shown to be decomposed into an *inhibitor* and two RELAYS. Further, the inhibitor is decomposed into interconnected RELAY and DUCT primitives. The style element *filter*, on the other hand, is treated as a black-box component and not decomposed into RELAY and DUCTS. Instead its abstract behavior is modeled through *write* and *read* interfaces. It is possible to add concrete behavior to a filter when used in a particular pipe-and-filter architecture. In the next step, the KWIC architecture is assembled using the pipe-and-filter architectural style, as shown in Figure 8b. The behavior of this architecture can now be modeled in terms of the interfaces of the four filters: *Input*, *Circular Shift*, *Alphabetizer*, and *Output*. This model is derived from the constraint automata of the assembled primitives through composition and hiding discussed in the previous section. The

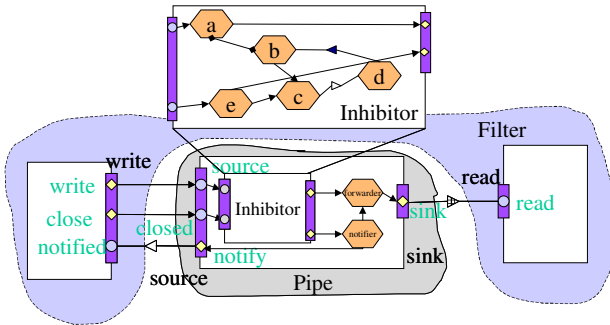


Figure 8a. Composition of pipe-and-filter style

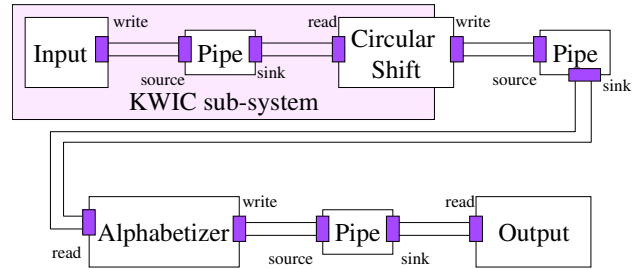


Figure 8b. KWIC architecture in pipe-and-filter style

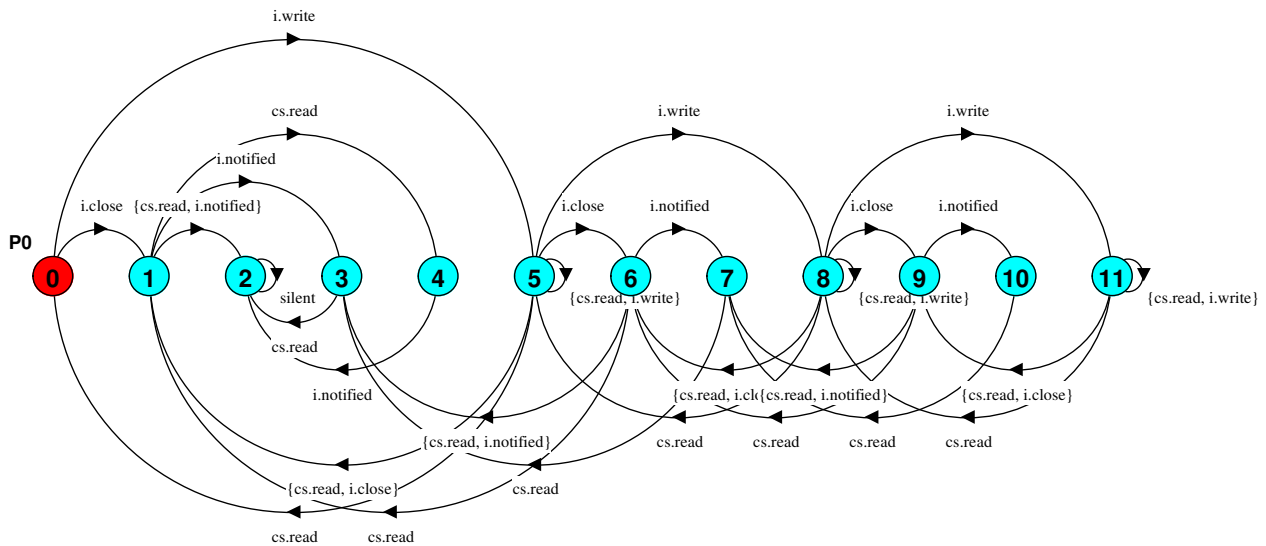


Figure 8c. Constraint automaton of KWIC sub-system

constraint automaton of the KWIC sub-system is shown in Figure 8c. This subsystem comprises the Input and Circular Shift components linked using a pipe instance. The unbounded asynchronous duct connecting the pipe's sink output to Circular Shift's read input is approximated using a finite buffer of 3 elements. The sub-system automaton contains 12 states and 28 transitions, which is eventually composed into an overall KWIC constraint automaton. Such composition requires an automated implementation of composition and hiding, which is a current endeavor of ours. It is remarkable that a minimized LTS model of the same sub-system contains 372 states (the number of transitions in the minimized LTS cannot be obtained in the output of the LTS tool [13]). The understandability of this model can be gauged from Figure 8c where the labels in this composite constraint automaton are the same as the names of portals used in the architectural assembly qualified by a prefix identifying the component in which the portal is located e.g., *cs* for Circular Shift and *i* for Input. Moreover, the synchronization of events is directly visible as transitions with multiple portal names. Some events may be simultaneously synchronous and asynchronous e.g., *cs.read* and *i.notified*. Events that are always asynchronous, on the other hand, never appear together in the constraint automaton. For example, *i.close* and *i.notified* never appear together on a transition, implying that they are always asynchronous.

7 Conclusions and Further Work

In this paper, we emphasized the need for modeling the behavior of software architectural assemblies that contain simultaneous synchrony and asynchrony of events. We showed how a well-understood existing technique for event-based modeling of the behavior of software architectures, LTS, proves inadequate for the above purpose. Our approach, using constraint automata, is based on the notion of timed data streams and explicitly recognizes synchronization in architectural behavior models through first class constructs. Such an approach enables effective modeling of software connector and style-based software architecture assemblies. This is also in concert with recent treatment of software connectors as first class architectural elements (e.g., [17]). Although this paper does not discuss data constraints in constraint automata, their use increases the precision and predictability of behavioral properties, albeit at a higher cost to determine them, compared to regular LTS.

Much work remains to be done in order to determine the effectiveness of the analysis of constraint automata-based behavioral models. We are currently defining a text-based notation to describe constraint automata. We are also exploring means of supporting parameterized constraint automata much like the way parameters are supported in FSP [13]. Parameterized automata are meta-models that can be reused over and over again, an important success factor for primitive-based assembly techniques such as Alfa. Descriptions of constraint automata can be provided to automated implementations of composition/hiding algorithms for constraint automata whose results are then transformed to LTS using the algorithm described in this paper.

As mentioned earlier, specification of safety and liveness properties on constraint automata will form an important step in the effective analysis of constraint automata-based behavior models. Integration with existing techniques for analysis of LTS is an important area of further work. It would be futile to develop these techniques in a vacuum without applying them to solve real problems. It is our intention to model and analyze, using constraint automata, more sophisticated style-based architectures assembled from Alfa's primitives.

8 REFERENCES

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engg. and Methodology*, July 1997.
- [2] F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Proceedings of FMCO 2002*, Leiden, The Netherlands, LNCS vol. 2852, Springer, 2003.
- [3] F. Arbab. Reo: A channel-based coordination model for component composition. To appear in *Mathematical Structures in Computer Science*, Cambridge University Press, 2003.
- [4] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. *Centrum voor Wiskunde en Informatica Report SEN-R0216*. Amsterdam, The Netherlands, Sep. 2002.
- [5] F. Arbab, C. Baier, J. J. M. M. Rutten, and M. Sirjani. Modeling component connectors in Reo by constraint automata. In *Proc. of FOCLASA '03*, Marseilles, France, Electronic Notes in Theoretical Computer Science, Elsevier Science, Sep. 2003.
- [6] C. G. Bell and A. Newell. *Computer structures: Reading and examples*, McGraw-Hill, New York, 1996.
- [8] S. C. Cheung and J. Kramer. An integrated method for effective behavior analysis of distributed systems, In *Proc. ICSE '94*, Sorrento, Italy, May 1994.
- [9] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. of ESEC/FSE '03*, Helsinki, Finland, September 2003.

- [10] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Software Engg.*, 21(4), April 1995.
- [11] A. Lopes, J. L. Fiadeiro, and M. Wermelinger. Architectural primitives for distribution and mobility. In *Proc. FSE '02*, Charleston, SC, USA, Nov. 2002.
- [12] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans. Software Engg.*, 21(4), April 1995.
- [13] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley & Sons. 1999.
- [14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of ESEC'95*, Stiges, Spain, Sep. 1995.
- [15] J. Magee, J. Kramer, and D. Giannakopoulou. Behavior analysis of software architectures, In *Proc. of WICSAI*, San Antonio, TX, USA, Feb. 1999.
- [16] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proc. of ESEC/FSE '03*, Helsinki, Finland, Sep. 2003.
- [17] N. R. Mehta, N. Medvidovic and S. Phadke. Towards a taxonomy of software connectors. In *Proc. of ICSE '00*, Limerick, Ireland, June 2000.
- [18] D. L. Parnas. On the criteria to be used for decomposing systems into modules. *Comm. of the ACM*. 15, 1972.
- [19] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engg. Notes*, 17, 1992.
- [20] M. Shaw and D. Garlan. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.
- [21] B. Spitznagel and D. Garlan. A compositional approach for constructing connectors. In *Proc. WICSA '01*. Oct. 2001.