Hierarchical Modeling and Abstraction of Discrete Event Systems Using Finite State Machines with Parameters†

Yi-Liang Chen†
Rockwell Scientific Co.
Thousand Oaks, CA 91360
ylchen@rwsc.com

Feng Lin
Dept. of ECE, Wayne State Univ.
Detroit, MI 48202
flin@ece.eng.wayne.edu

Abstract
In this paper, a trace-based model abstraction mechanism that aggregates parameters and event sequences to a coarser granularity is presented for discrete event systems modeled as finite state machines with parameters. Using both state transitions and parameter values for representing system behaviors and resources, the finite state machine with parameters (FSMwP) approach has resulted in efficient and compact representations for discrete event systems (in particular, those that cannot be efficiently modeled by the traditional automata-based approach). We propose a hierarchical modeling framework for FSMwPs based on an abstraction mechanism that enables automatic synthesis of models of all the entities in the hierarchy. The characteristics and advantages/disadvantages of the proposed abstraction mechanism and hierarchical framework are also discussed.

Keywords: Discrete event systems, finite state machines, model synthesis, abstraction

1 Introduction
Modeling and control of discrete event systems (DES) have been studied by control engineers for almost 20 years. During this period, many modeling approaches have been proposed, including most notably automata or finite state machines [13, 1], Petri nets (e.g., [12]) and their variations such as vector DES [10] and event graphs [5], queuing systems and generalized semi-Markov processes [1].

Among these models, finite state machines are the most straightforward for control. In fact, the supervisory control theory [13, 11, 1] based on finite state machines has been well developed as it addresses the fundamental issues in the control of DES. As a result, we now have a good understanding of problems such as controllability, observability, coobservability, normality, decentralization, and nondeterminism, etc.

However, for applications, finite state machine models have long suffered the potential problem of state explosion. To alleviate this problem, two effective approaches can be employed. The first approach is to find a more compact representation of DES by grouping similar states into one super-state using parameters. In [3], we propose a modeling approach, termed Finite State Machine with Parameters (FSMwP), that uses both state transitions and parameter values for representing system behaviors and resources. This approach has resulted in efficient and compact representations for discrete event systems (in particular, those that cannot be efficiently modeled by the traditional automata-based approach).

The second approach is to build hierarchical models with different levels of details. For example, a flexible manufacturing system can be modeled in several levels of hierarchy. At the bottom level, each individual machine is modeled in details. At the next level, abstracted and coarser models of machines are used and combined to form the model of a workcell. In this paper, we present a trace-based model abstraction mechanism for discrete event systems modeled as finite state machines with parameters. We propose a hierarchical modeling framework for FSMwPs based on the abstraction mechanism that enables automatic synthesis of models of all the entities in the hierarchy. The characteristics and advantages/disadvantages of our proposed abstraction mechanism and hierarchical framework are also discussed.

Note that, in this paper, our focus is on the hierarchical model synthesis and abstraction for FSMwPs. The issues related to the control and optimization of DES using FSMwPs is not in the scope of this paper. Please refer to [4] for the synthesis of (single-level) safety controllers using FSMwPs.

Proofs, examples (of a logistic system) and other details are documented in the full version of this paper, which is available at www.ece.eng.wayne.edu/~flin.

2 Finite State Machines with Parameters
We summarize the basic results of our modeling primitive, Finite State Machine with Parameters (FSMwP)
proposed in [3]. In short, a finite state machine with parameters can be viewed as a 7-tuple

$$FSMwP = (\Sigma, Q, \delta, P, G, (q_0, p_0), Q_m),$$

where, as in a traditional finite state machine, \( \Sigma \) is the event set, \( Q \) is the state set, and \( Q_m \) is the set of marked (or final) states.

In addition to the states in a finite state machines, a set of parameters is introduced in an FSMwP and \( P \) represents the vector space of these parameters. Note that the parameters in \( P \) may be defined over different domains (e.g., integers, real numbers, enumerated types, etc.). A vector \( p \in P \) represents a particular set of values for these parameters. The initial condition of an FSMwP is then described by the pair, \((q_0, p_0)\), where \( q_0 \) is the initial state and \( p_0 \) contains the initial values for the parameters.

The guards \( g \in G \) are predicates on the parameters in \( p \). The transition function \( \delta \) can be defined as \( \Sigma \cup \{ \epsilon \} \times Q \times G \times P \to Q \times P \) (where \( \epsilon \) denotes the empty string) as illustrated in Figure 1. The transition shown is to be interpreted as follows. If at state \( q \) the guard \( g \) is true and the event \( \sigma \) occurs, then the next state is \( q' \) and the parameters will be updated to \( f(p) \). \( \sigma \) is the event (label) of the transition; \( g \) is the guard of the transition; and \( p := f(p) \) is the action of the transition.

\[ \wedge \sigma \]

\[ \longrightarrow \]

\[ \sigma \]

\textbf{Figure 1: A transition in FSMwP.}

If \( g \) is absent in the transition, then the transition takes place when \( \sigma \) occurs. Such a transition is usually referred as an \textit{event transition}. If \( \sigma \) is absent, then the transition takes place when \( g \) becomes true. Such a transition is called \textit{dynamic transition}. If \( p := f(p) \) is absent, then no parameter is updated during the transition. For our abstraction mechanism presented in this paper, we assume that every transition is associated with an event label. Hence, for a dynamic transition, we can either assume its event label to be \( \epsilon \) or assign a fictitious event label to it.

Without much difficulty, we can regard finite state machines with parameters as a special type of the Hybrid Machines (HMs) introduced in [7]. In particular, an FSMwP has no continuous dynamics (i.e., \( \dot{p} = 0 \) at any state). The only way to change parameters is by updating (or re-initialization).

To describe the behavior of an FSMwP, we define a \textit{run} of an FSMwP as a sequence

$$r = (q_0, p_0) \overset{l_1}{\rightarrow} (q_1, p_1) \overset{l_2}{\rightarrow} (q_2, p_2) \overset{l_3}{\rightarrow} (q_3, p_3) \ldots,$$

where \( l_i \) is (the label of) the \( i \)-th transition and \((p_i, q_i)\) is the state and parameter values after the \( i \)-th transition. We denote the set of all possible runs of an FSMwP as

$$R(FSMwP) = \{ r : r \text{ is a run of FSMwP} \}.$$  

A trace of a run is the sequence of event transitions in the run

$$s = \sigma_1 \sigma_2 \sigma_3 \ldots.$$  

That is, \( s \) is obtained from \( r \) by deleting the state information and dynamic transitions (if they are viewed as \( \epsilon \) transitions).

The set of all traces of an FSMwP is a language denoted by

$$L(FSMwP) = \{ s : s \text{ is a trace of FSMwP} \}.$$  

This language is called the language generated by the FSMwP. The language marked by the FSMwP is defined as

$$L_m(FSMwP) = \{ s : s \text{ ends in a marked state } q \in Q_m \}.$$  

Similar to FMSs and HMs, synthesis (or synchronization) among different (sub)system models is achieved by the parallel composition of several FSMwPs to form a composite finite state machine with parameters (CFSMwP)

$$CFSMwP = FSMwP_1 || FSMwP_2 || \ldots || FSMwP_n.$$  

The detailed definition of the parallel composition for FSMwPs can be found in [3].

In practice, it is possible that one event may have different labels in two interacting processes. For example, the output from Workstation A (labeled \( \alpha \)) could be the input to a cart (labeled \( \gamma \)). To model such a system, it is not always convenient to insist that events such as \( \alpha \) and \( \gamma \) be given the same label for proper synchronization. This is because that the cart may move from one station to another. So \( \gamma \) may correspond to the output from Workstation B (labeled \( \beta \)) the next time. In dealing with this situation, we proposed a dynamical masked parallel composition in [3] for FSMwPs that allows dynamically re-labeling and re-mapping of events for synchronization.

### 3 Abstraction of FSMwP

In this section we present our model abstraction mechanism developed for FSMwPs. Our abstraction mechanism takes in a detailed FSMwP model for a (sub)system and, through the use of an abstract mapping which specifies how parameters and event sequences should be aggregated, generates an abstracted (FSMwP) model for this (sub)system. This abstracted FSMwP can then be used in the synthesis of a high level model in a hierarchical modeling framework to be described in Section 4.

To perform the abstraction, we start with a detailed FSMwP model

$$FSMwP_d = (\Sigma_d, Q_d, \delta_d, P_d, G_d, (q_{d_0}, p_{d_0}), Q_{d_m}).$$
Our goal is to obtain an abstracted model

\[ FSMwP_a = (\Sigma_a, Q_a, \delta_a, P_a, G_a, (\eta_{a0}, p_{a0}), Q_{a_m}). \]

which describes the system behavior in a coarser granularity.

We adopt a “trace-based” approach for our abstraction mechanism. That is, each high level transition should correspond to a set of traces at the low level. Occurrence of any trace in the set should signal the occurrence of a high level event. As explained in Section 2, we assume that, without loss of generality, for our abstraction mechanism, each transition in the detailed FSMwP, \( FSMwP_d \), has a corresponding event label.

The aggregation of parameters and event sequences for a detailed FSMwP model is specified in its corresponding abstract mapping. Since an FSMwP consists of a set of parameters and a finite state structure that describes traces of the system, the abstract mapping consists of two parts: a parameter mapping and a trace mapping. The parameter mapping is rather straightforward: it simply maps old (i.e., detailed) parameters into new (i.e., abstracted) parameters. Formally, for each new parameter \( p_{a_i} \) in \( P_a \), \( p_{a_i} \) is expressed by a function of old parameters:

\[ p_{a_i} = c_i(p_{d1}, p_{d2}, ..., p_{dn}). \]

Hence, if we define \( E = [e_1, e_2, ..., e_m] \), then \( E \) is a mapping from \( P_d \) to \( P_a \). Note that the type of functions used for the parameter mapping depends on the problem domain and may vary from simple algebraic functions to conditional statements. We provide the following examples for illustration.

1. If we want to total the outputs of two machines, denoted by \( p_1 \) and \( p_2 \), respectively, then we can map \( p_1 \) and \( p_2 \) to a new parameter \( p = p_1 + p_2 \).
2. If we want to take the average speed of two cars, denoted by \( v_1 \) and \( v_2 \), respectively, then we can map \( v_1 \) and \( v_2 \) to a new parameter \( v = (v_1 + v_2) / 2 \).
3. Other possible mappings include \( p = \min\{p_1, p_2, ..., p_n\} \), and \( p = \max\{p_1, p_2, ..., p_n\} \).

The basic idea of the trace mapping is as follows. Suppose after the occurrence of a trace \( \sigma_1 \sigma_2 ... \sigma_k \) at the low level, a high level event \( \sigma_a \) should be generated with guard \( g_a \) and parameter (re-)assignment (i.e., action) \( p_a := f_a(p_a) \), then the FSMwP describing the trace mapping should have the transition structure as shown in Figure 2. In the figure, \( \mu_a \) is associated with \( g_a \wedge \sigma_a \Rightarrow p_a := f_a(p_a) \).

To represent mapping of all possible traces, we use a special FSMwP

\[ FSMwP_t = (\Sigma_t, Q_t, \delta_t, P_t, G_t, (\eta_{t0}, p_{t0}), Q_{t_m}). \]

where \( FSMwP_t \) has the following special properties:
(1) \( P_t = P_a \) (i.e., its parameters are new parameters); (2) \( G_t = \emptyset \) (i.e., transitions in \( FSMwP_t \) have no guards); (3) \( Q_{t_m} = Q_t \) (i.e., all its states are marked); and (4) transitions in \( FSMwP_t \) have no (re-)assignment of parameters.

The event set \( \Sigma_t \) includes all the low level events in \( \Sigma_d \) together with a special set of \( \mu \) events that will be translated into high level events in \( \Sigma_a \). The \( \mu \) transitions have the following special properties:
(1) We associate each \( \mu \) transition with an event label in \( \Sigma_a \), a guard defined on \( P_a \) and a (re-)assignment of parameters in \( P_a \); (2) The destination state of a \( \mu \) transition is always the initial state \( q_{t0} \); (3) Each state in \( FSMwP_t \) (except for \( q_{t0} \)) has one \( \mu \) transition. The last property is necessary in order to prevent the artificial “deadlock” in the procedure of abstraction.

Note that the trace mapping \( FSMwP_t \) may be viewed as a “generalized” Mealy machine (with parameters) where \( \mu \) transitions are the only vocal transitions which output their corresponding new event labels, guards, and actions.

Figure 3 shows a simple \( FSMwP_d \) model that we use in this section to illustrate our abstraction procedure. We omit the guards and the parameter (re-)assignments (i.e., actions) of each transition in the figure.

Figure 4 shows a trace mapping \( FSMwP_t \) for our detailed model depicted in Figure 3. We denote a \( \mu \) transition that associates with a high level event label, say, \( \alpha \) in \( \Sigma_a \), as \( \mu_\alpha \) in the figure.

To obtain \( FSMwP_a \) from \( FSMwP_d \) and \( FSMwP_t \), we first ignore all the guards in \( FSMwP_d \) (i.e., assume that they are all true). We then perform a special type of parallel composition on \( FSMwP_d \) and \( FSMwP_t \):

\[ FSMwP_d \parallel_a FSMwP_t. \]

In this special parallel composition \( \parallel_a \), rules of parallel
composition \parallel apply to all transitions except for the μ transitions (in FSM\textsubscript{wP}\textsubscript{1}) and \( \epsilon \) (the empty event) transitions (in FSM\textsubscript{wP}\textsubscript{2}).

A μ transition is defined at \((q_\alpha, q_t)\) if μ is defined at \(q_t\) and (1) there is no transitions out of \(q_t\) or (2) there is a non-μ transition out of \(q_t\) but there is no corresponding transition of the same event label out of \(q_t\). The next state after the μ transition is \((q_\alpha, \delta_\mu(q_t))\).

Transitions of the empty event \( \epsilon \) also require special treatment in this composition. First, we consider in FSM\textsubscript{wP}\textsubscript{2} only the \( \epsilon \) transitions that are explicitly defined (e.g., for a dynamic transition). That is, we ignore all the default self-loop transitions of \( \epsilon \) from some state \(q_t\) to itself unless they are defined as \(\delta_\epsilon(q_t, \epsilon) = q_t\) explicitly. Moreover, in FSM\textsubscript{wP}\textsubscript{2}, we consider only the \( \epsilon \) transitions that not only are defined explicitly but also bring the machine to a different state. With these two assumptions, the explicitly defined \( \epsilon \) transitions are treated in the same way as the rest of the non-μ transitions.

Figure 5 shows the resulting FSM\textsubscript{wP} of the special parallel composition \( \parallel a \) between FSM\textsubscript{wP}\textsubscript{1} in Figure 3 and FSM\textsubscript{wP}\textsubscript{1} in Figure 4.

After obtaining FSM\textsubscript{wP}\textsubscript{2} \( \parallel a \) FSM\textsubscript{wP}\textsubscript{1}, we construct FSM\textsubscript{wP} as follows: First, we "project out" all the non-μ transitions (for example, all the dashed transitions in Figure 5). That is, we replace all the non-μ transitions in FSM\textsubscript{wP}\textsubscript{2} \( \parallel a \) FSM\textsubscript{wP}\textsubscript{1} by empty strings \( \epsilon \) and leave all the μ transitions unchanged. This results in a nondeterministic machine with \( \epsilon \) transitions.

Next, we convert the nondeterministic machine into a deterministic machine using, for example, the procedure outlined in [1]. Finally, we replace each μ transition in the resulting machine by its corresponding high level transitions \( q_a \wedge \sigma_a/p_a := f_a(p_a) \). The final abstracted model FSM\textsubscript{wP}\textsubscript{a1} of our illustrated detailed model FSM\textsubscript{wP}\textsubscript{1} using the trace mapping of FSM\textsubscript{wP}\textsubscript{1} is shown in Figure 6.

The design of FSM\textsubscript{wP} is problem-dependent. We believe that this abstraction mechanism is suitable for a large class of practical problems. To demonstrate this, let us discuss the theoretical connections between the detailed model FSM\textsubscript{wP}\textsubscript{2} and the abstracted model FSM\textsubscript{wP}\textsubscript{a}. For simplicity of our discussions, we assume that no parameters are involved in any FSM\textsubscript{wP}.

Suppose that the goal of abstraction is to map a set of strings \( S = \{s_1, s_2, \ldots, s_m\} \subseteq \Sigma^*_d \) into abstracted events \( \{\sigma_1, \sigma_2, \ldots, \sigma_n\} \subseteq \Sigma^*_a \), that is

\[
\{s_1, s_2, \ldots, s_m\} \overset{h}{\longrightarrow} \{\sigma_1, \sigma_2, \ldots, \sigma_n\}.
\]

The mapping \( h \) could be many to one, in other words, two strings may be mapped into the same abstracted event. Without loss of generality, we assume that none of the strings in \( S \) is a prefix of another string in \( S \). (If \( a \) is mapped into \( \alpha \) and \( ab \) is mapped into \( \beta \), then there is ambiguity in the specification.) Define FSM\textsubscript{wP}\textsubscript{1} = \((\Sigma_t, Q_t, \delta_t, P_t, G_t, (q_{t0}, p_{t0}), Q_{tm})\) as follows \(^1\).

- \( \Sigma_t = \Sigma_d \cup \{\mu_1, \mu_2, \ldots, \mu_n, \mu_e\} \), where \( \mu_i \) corresponds to \( \sigma_i \in \Sigma_a \), \( i = 1, 2, \ldots, n \) and \( \mu_e \) corresponds to \( \epsilon \), an abstracted event signaling that a string generated by FSM\textsubscript{wP}\textsubscript{2} is not in \( S \) (we consider this as an error in the specification).
- \( Q_t = \{q_{tu}: u < s, \text{for some } s \in S\} \). That is, for each prefix \( u \in S \) (i.e., the set of prefixes of \( S \)), we assign in \( Q_t \) a unique state \( q_{tu} \).
- \( P_t = \emptyset, G_t = \emptyset, p_{t0} = \emptyset, Q_{tm} = Q_t \).

\(^1\) Alternatively, the FSM\textsubscript{wP}\textsubscript{1} can be built by a procedure similar to the construction of the realization of the input map described in [2].
• For \( \sigma \in \Sigma_d \),
  \[
  \delta_d(\sigma, q_{i0}) = \begin{cases} 
    q_{i0}, & \text{if } u \in \overline{S} \\ 
    \text{undefined}, & \text{otherwise}
  \end{cases}
  \]

For \( \mu_i \in \{ \mu_1, \mu_2, \ldots, \mu_m \} \),
  \[
  \delta_d(\mu_i, q_{i0}) = \begin{cases} 
    q_{i0}, & \text{if } h(u) = \sigma_i \\ 
    \text{undefined}, & \text{otherwise}
  \end{cases}
  \]

Finally, \( \delta_d(\mu_i, q_{i0}) = q_{i0} \) is defined if and only if for all \( i \in \{1, 2, \ldots, n\} \), \( \delta_d(\mu_i, q_{i0}) \) is undefined.

Clearly for any state (except for the initial state), one and only one \( \mu \) transition is defined in \( FSM\_P_1 \). The following theorem shows that \( FSM\_P_1 \) indeed achieves the mapping \( h \).

**Theorem 1** For any \( FSM\_P_2 \) with \( P_2 = \emptyset \) (and hence \( G_d = \emptyset \), \( P_{2a} = \emptyset \)), if \( FSM\_P_1 \) is defined as above, then the resulting \( FSM\_P_2 \) has the following property.

\[
\sigma_{i1} \sigma_{i2} \ldots \sigma_{i_k} \in L(FSM\_P_2) \iff (3i_1, s_{i1} \ldots s_{i_k} \in L(FSM\_P_2)) \land (h(s_{i1}) = \sigma_{i1}) \land (h(s_{i2}) = \sigma_{i2}) \land \ldots \land (h(s_{i_k}) = \sigma_{i_k}).
\]

Let us compare our trace-based abstraction mechanism to other state-based abstraction mechanisms (e.g., [8]), where a subset of states in the detailed machine (in this case, an FSM) are combined into one state in the abstract model. The main advantages of our trace-based abstraction are as follows:

1. **Our trace-based abstraction is machine independent.** In other words, when we define the abstract mapping, we do not need to consider which particular detailed \( FSM\_P_2 \) is being abstracted. Rather, we only need to consider which traces are being abstracted (to what).

This property allows us to pre-construct an abstract mapping “template” for a class of different \( FSM\_P \) detailed models with similar characteristics and/or scope (even prior to the synthesis/construction of these detailed models). To achieve that, we can “over-specify” the trace mapping template that may contains traces not in a specific detailed model (but may appear in other detailed models). In Section 4, we describe how we can utilize this property to automatically synthesize models for aggregated (higher-level) entities in our \( FSM\_P \) hierarchical modeling mechanism.

2. **Our abstract models are always deterministic.** It is well known that state-based abstraction often leads to nondeterminism, unless some consistency conditions are satisfied. By virtue of our construction of \( FSM\_P_2 \), our abstract models are guaranteed to be deterministic. Control of nondeterministic systems is usually more difficult and complex [6].

The main disadvantage of our trace-based abstraction is that we cannot guarantee that the abstracted model will always have less states than the detailed model, since the focus of our abstraction is on the aggregation of parameters and event sequences to a coarser granularity. In other worlds, if we randomly pick some \( FSM\_P_2 \) and some \( FSM\_P_1 \), then in the worst case the resulting \( FSM\_P_2 \) will likely have more states. However, in practice, we do not construct the \( FSM\_P_2 \) and \( FSM\_P_1 \) randomly without taking advantage of the intrinsic structure of the systems and application domains. In [9] we describe a series of experiments we conducted, over mainly the military air operation domain, which showed that the more structure the systems have, the less states their abstracted models may usually result.

### 4 Hierarchical Modeling of Finite State Machines with Parameters

We now propose a hierarchical modeling framework for complex systems modeled as \( FSM\_P \)s. We assume that a hierarchy of the system is given, which describes the relationship and groupings of various subsystems/components (we term them as “entities”) in the system. We classify the entities in the system to be either component entities (which represent the finest granularity of objects of interest in the given application domain) or aggregated entities (whose behaviors are characterized through aggregating the behaviors of other entities belonging to them). For example, in military air operation systems, the component entities may include individual fighters or bombers while the aggregated entities may be fighter squadrons, flight of several bombers, or strike packages; in manufacturing systems, the component entities may include individual machines while the aggregated entities may be workstations, assembly lines, etc. Clearly, the structure of a hierarchy depends highly on the application domain.

We adopt a bottom-up approach in our hierarchical modeling mechanism as outlined in Figure 7. The \( FSM\_P \) model for a component entity can usually be obtained by instantiating and customizing an \( FSM\_P \) model (of its corresponding type) from a predefined model library in the application domain. Due to the use of parameters and the explicit definitions of constant values as parameters, customizing an \( FSM\_P \) model for a specific component entity can easily be achieved. For example, an \( FSM\_P \) model of a specific F-16 fighter for a specific mission (with specific payloads) can be obtained from a generic F-16 \( FSM\_P \) model with its corresponding parameters properly set according to the given mission and payloads.

We use the parallel composition (described in [3]) and the abstraction mechanism (described in Section 3) as the main tools in synthesizing various aggregated entities as follows. First, given the model of a lower-level entity (either component or aggregated) and its associated abstract mapping, we apply the abstraction mechanism to generate its abstracted \( FSM\_P \) model. The detailed model of an \( n \) level aggregated entity is then synthesized from the abstracted models of all its \( n - 1 \) level constituents using the parallel composition. This
process is illustrated in Figure 7.

Note that, since the component entity (CE) model is usually customized (mainly on the parameter values) from the model of its corresponding type in a domain model library, we need to specify only an abstract mapping (AM) for each model type (e.g., AM in Figure 7) which can then be applied to all the component entities of the same type (e.g., CE in Figure 7) during the abstraction. Meanwhile, because of the trace-based nature of our abstraction mechanism as discussed at the end in Section 3, if the two types of models share the same characteristics and/or scope (e.g., F-16 fighters or F-15 fighters), they may employ the same abstract mapping (possibly over-specified) for the abstraction (e.g., components of both CE and CE types use the same AM). Furthermore, we may pre-specify an abstract mapping template that can be used for the abstraction of future aggregated entities (AE) synthesized that may share the similar characteristics and/or scope (e.g., an abstract mapping for various compositions of fighter squadrons). This is also illustrated in Figure 7 where both aggregated entities AE in Figure 7 and AE are associated with the same pre-specified abstract mapping AM. With these characteristics, the hierarchical modeling framework can synthesize models for all the entities automatically, given the model hierarchy, component and abstract mapping libraries. Please refer to [9] for our implementation of this modeling framework.

Since the parallel composition and the abstraction procedure both result in FSMWP models, our hierarchy modeling mechanism can accommodate as many levels of entities as the application domain may require. This mechanism can not only handle homogeneous hierarchy but also heterogeneous hierarchy, where an aggregated entity may have both other aggregated entities and component entities as its constituents. This property is illustrated in Figure 7 where both CE in Figure 7 (a component entity) and AE [r] (an aggregated entity) belong to the aggregated entity AE[v].

5 Conclusion

In this paper, we have presented a trace-based model abstraction mechanism for discrete event systems modeled as finite state machines with parameters. We have proposed a hierarchical modeling framework for FSMWPs based on an abstraction mechanism that enables automatic synthesis of models of all the entities in the hierarchy. We have also discussed the characteristics and advantages/disadvantages of the proposed abstraction mechanism and hierarchical framework.

References