Submodule Construction for Extended State Machine Models

Bassel Daou and Gregor V. Bochmann

School of Information Technology and Engineering (SITE),
University of Ottawa, Canada
{bdaou, bochmann}@site.uottawa.ca

Abstract. In this paper, we consider the problem of extending existing submodule construction techniques that have been developed for finite state models into more expressive and compact behavioral models that handle data through parameterized interactions, state variables and simple guards. We provide a behavioral model based on extended Input-Output Automata and describe an algorithm that provides the solution to the submodule construction problem in the context of this extended behavioral model. This algorithm is based on abstracting variable configurations using the concept of variable partitions, and splitting of states obtained from the finite state machine model in order to satisfy the constraints imposed by the values of exchanged interaction parameters.

1 Introduction

Submodule construction, also called equation solving or factorization, considers the following situation: An overall system is to be constructed which consists of several components. It is assumed that the specification S of the desired behavior of the system is given, as well as a specification of the behavior of all the components, except one. The process of submodule construction has the objective of finding a specification for the latter component such that all components together provide a behavior consistent with the behavior specification S. If the modeling paradigm for the behavior specifications is sufficiently limited, e.g. finite state models, an algorithm for submodule construction can be defined [MeBo83, Parr89, Shie89, LeQi90, DrBo99]. Submodule construction finds application in the synthesis of controllers for discrete event systems [BrWo94], communication gateway design and protocol conversion [KeHa93, KNM97, TBD97].

In this paper we consider submodule construction techniques for state transition models extended with state variables, interaction parameters and simple guards for transitions. We use a specification paradigm which is an extension of partially specified Input/Output Automata as discussed in [Boch02]. The main difficulties encountered when solving the submodule construction for such extended specification models are the following:

a. One has to keep track of the relationship between the variables of the new module X, the variables of the system specification and the variables of the existing component C.

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 396 – 410, 2005.

b. For each of the input or output transitions of the new component X, one has to decide which local variables should be used to store parameter values received by an input, or which local variable should be used to define the value of an output parameter.

c. There may be many different global system states that may be reached depending on the choices that are taken under point (b) above. We want to find the most general specification for the component X such that without introducing not allowed output to the environment of the system nor unexpected input (for the existing component C) from the component X or from the environment.

Our approach for solving this problem without simply enumerating all possible choices for the new component X is based on the following two ideas:

1. In order to model the equivalence between different variables in a given state, we consider partitions over the set of all variables. A partition defines a set of non-overlapping subsets of variables, and our partitions have the property that all variables that belong to a given subset of the partition are equivalent, that is, known to have the same value.
2. After applying submodule construction for the IOA model (following known methods [QiLe91, BrWo94, KNM97, DriBo99]) we analyze the resulting state machine for X in order to determine which partitions may apply for each of its states. Since for a given state, some partitions may lead to invalid behavior, we introduce a transformation step in which the states of the component X are split according to the possible partitions that can be reached. The purpose of this splitting is to preserve the acceptable behavior (related to a particular partition) and eliminate invalid behaviors (related to other partitions). The splitting of one state often leads to the need for splitting other states from which the former can be reached. We therefore come up with a recursive splitting algorithm which allows us to eliminate all invalid behavior and keep all acceptable behavior, that is, we obtain the most general solution.

To simplify the problem we only deal with safety properties postponing issues related to liveness like blocking and progressiveness that has been solved for finite models [KNM97, DriBo99, BEYB03]. We note, however, that blocking will be partially solved in our context since we assume that the system component can not block any input from its environment.

The paper is structured as follows. In Section 2 we give the definition of our extended IOA specification formalism. Section 3 describes our submodule construction algorithm for extended IOA and gives some examples. Given the limited space in this paper, we concentrate on the definition of variable partitions and the state splitting algorithm.

2 Behavioral Model

We start by adopting a behavioral model that manipulates data and compactly represents large or infinite state systems. The model we suggest is an extension to the IO automata model [LyTu89].

2.1 Behavioral Model Properties

We stress mainly two aspects of the model: data manipulation and value passing, and differentiation between input and output in an assumption guarantee model.

2.1.1 Data Manipulation and Value Passing

The usage of dataflow information in models allows more compact and expressive representation of systems. Data manipulation and value passing can be achieved through extending finite automata models with parameterized interactions, local variables, simple transition guards and variable assignments. Parameterized interactions are used to represent the exchange of data between components and between components and the environment. Processing of data is done through saving values of input parameters in local variables, and using these values later to define the values of output parameters. Our model does not apply any operation on received data, and guards over data are simple equality checks used to represent the assignment of values to parameters, or in other words a restriction to the values that can be assigned to the parameters. Though the formal model that we present later allows for restrictions on input values, we assume that the behavior specification of C and S do not use this feature, since we are only using the guards to represent variable assignment to parameters.

2.1.2 Differentiation Between Input and Output in an Assumption Guarantee Model

As in the IOA model, a system has no control over its input interactions; however, it can assume that certain inputs are not possible. Similarly, a system might be required to give guarantees that it does not send certain interactions at certain states. This concept is generalized in our model to cover parameters sent and received alongside an interaction. So a machine can have assumptions that only specific values can be received using the transition guards and guarantees that it sends only specific values as parameters of output interactions it initiates. We use partial specification to indicate input assumptions and output guarantees, that is, if at a given state there was no transition labeled with a given interaction, then this indicates that the machine assumes that its environment will not generate that interaction at that state. Same applies to the case of output guarantees, where if an output interaction and a given output valuation of the interaction parameters was not specified, then that interaction and parameter valuation is guaranteed not to be generated by the machine.

2.2 Extended Input Output Transition Systems

In the following we present a formal definition of what we call Extended Input-Output Transition System (EIOTS), inspired from the Input Output symbolic Transition System IOSTS in [RBJ00], I/O Automata [LyTu89], and CSP [Hoar85]. An EIOTS is tuple < S, V, s0, Se, $\sum$, T> where

- S is a nonempty finite set of states.
- V is a finite set of variables.
- s0 $\in$ S represents the initial state.

- Se $\subset$ S represents the set of error states resulting from either a not allowed output or an unexpected input. All transitions starting at an error state should lead to an error state.
- $\sum$ is a nonempty, finite alphabet, which is the disjoint union of a set $\sum_{in}$ of input interactions, a set $\sum_{out}$ of output interactions, and a set {ï} which has the special internal interaction ï. For each interaction $\alpha \in \sum_{in} U \sum_{out}$, there is a (possibly empty) ordered set of interaction parameters $Pm_\alpha = <pm_1,\ldots, pm_k>$,
- $T \subseteq S \times 2^{Pmx V} \times \sum \times 2^{VxPm} \times S$. Each tuple $(s, \gamma, \alpha, \theta, s') \in T$ represents a transition where:
  - $s \in S$ is the starting state of the transition.
  - $\gamma \subseteq Pm_\alpha \times V$. A couple $(p,v) \in \gamma$ represents an equality condition of the form (p=v). If α is an input interaction then γ is interpreted as a transition guard formed by the conjunction of all constituting parameter conditions. If α is an output interaction then parameter conditions are interpreted as assignments of variables to parameters or in other words a restriction of the possible values that a parameter can take.
  - $\alpha \in \sum$ is the transition's interaction.
  - $\theta \subseteq V \times Pm_\alpha$. Each couple $(v, p) \in \theta$ represents an assignment of parameter p to variable v. The assignments in θ are executed during the transition after the assignments in γ; they assign new values to some of the variables in V. A variable is allowed to be assigned only once, this makes θ belong to the set of partial mapping relations in $V \times Pm_\alpha$
  - $s' \in S$ is the end state of the transition.

The EIOTS shown in Figure 1 is taken as an example throughout this paper. It represents a desired system behavior S. In our notation, a question mark next to the interaction label represents an input, and an exclamation mark represents an output. Circles represent states and arrows represent transitions.

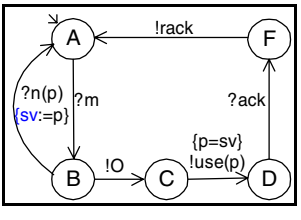

Fig. 1. EIOTS S

3 Submodule Construction Algorithm

The algorithm follows the general steps of submodule construction algorithm for finite state machines namely composition, hiding, determinization and bad or uncontrollable state removal. However, these steps have to be adapted for the new specification paradigm. To allow for determinization to take place we need to remove the effect of hidden guards and hidden variables. This is done through state splitting transformation.

In algorithm 1 we list the general steps of the submodule construction algorithm which basically include computing the unrestricted general behavior formed from the composition of the Chaos machine of X with the specification and the context. The chaos machine represents the most general behavior of X and uses as many variables

as there are in S and C combined. To enable the composition between S and C we need to apply the duality operator to C which gives a machine that has the same structure as C with the exception of interchanging input and output interactions $\sum_{out}$ and $\sum_{in}$. The duality operator is applied as well to the composition so that we can compose it with the Chaos machine. Then, the resulting EIOTS is transformed using state splitting. After splitting we hide all interactions that are not visible to X and all variables coming from S and C. Finally, we handle the nondeterminism introduced by hiding, and we remove all uncontrollable behavior, that is, we mark all states that uncontrollably reach an error state as "bad", and add them to the set of error states.

Algorithm 1. Submodule Construction Algorithm:
Given C, S: EIOTS, $\sum_X$ Interaction Alphabet return EIOTS

- G1 := Chaos($\sum_X$, |S.V| +|C.V|) xDual(SxDual(C))
- G2 := Split(G1,$\sum_X$)
- G3 := Hide(G2, ($\sum_C$ U $\sum_S$) - $\sum_X$, S.V U C.V)
- G4 := Determinize(G3)
- G5 := RemoveUncontrollableBehavior(G4)
- Return EIOTS G5

To illustrate the algorithmic steps we use a submodule construction example. The general system specification S is given in Figure 1. Figure 2a below shows the behavior of the context and Figure 2b describes the general problem architecture. To distinguish variables of various machines, we use the name of the machine as a prefix when naming variables.

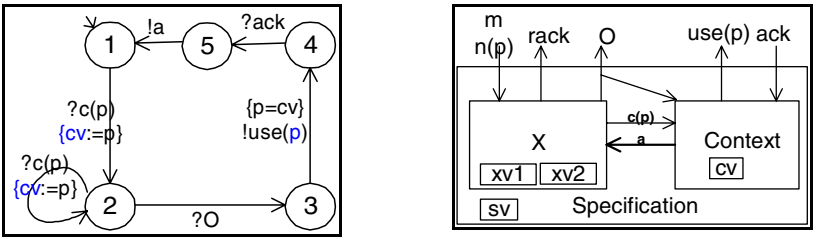

Fig. 2. (a) Context Behavior. (b) Example Architecture.

In the following sub sections we go through the operations on EIOTSs needed for the submodule construction algorithm stressing on composition, and state splitting operations.

3.1 Composition

The composition of EIOTS follows the composition of partially specified IO Automata [KeHa93]. That is, transitions with common interactions are executed synchronously and transition with interactions particular to each machine are executed

independently, assuming that for each interaction there is only one initiator, which is one of the two composed machines or the environment.

Concerning the extended elements of the EIOTS model, the resulting composed machine will have a set of variables which is the disjoint union of the variables of the component machines, assuming that the machines have distinct variable names. When composing two input transitions, the resulting transition will be an input transition. Meanwhile, when composing an input transition with an output transition, the resulting transition will be an output transition. In both cases, the parameter conditions of the resulting transition will be the conjunction of the parameter conditions of the constituting transitions. Similarly, the variable assignments of the resulting transition will be the union of the variable assignments of the constituting transitions.

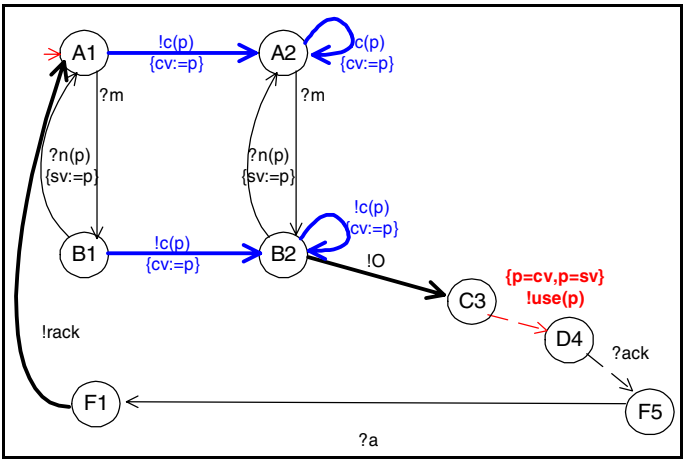

Fig. 3. Composition example. SxDual(C).

The composition of two EIOTSes E1 < S1, V1, s01, Se1, ∑1, T1> and E2 < S2, V2, s02, Se2, ∑2, T2> is an EIOTS E < S, V, s0, Se, ∑, T> that is formally defined as follows:

- S = {(s1,s2)| s1 ∈ S1 and s2 ∈ S2}
- V = V1UV2, the union of variables in E1 and E2.
- s0 = (s01, s02)
- Se = {(s1,s2) | s1∈ Se1 or s2 ∈ Se2}
- $\sum_{in}$. = ($\sum 1_{in}$ - $\sum 2_{out}$) U ($\sum 2_{in}$ - $\sum 1_{out}$) *Note: Input interactions that are neither initiated by E1 nor by E2.*
- $\sum_{out}$. = ($\sum 1_{out}$ U $\sum 2_{out}$). It is assumed that ($\sum 1_{out} \cap \sum 2_{out} = \{\}$)
- T = union of
  - {((s1,s2), γ1, α, θ1, (s1',s2)) | (s1, γ1, α, θ1, s1') ∈ T1, s2 ∈ S2, α ∈ ∑1 - ∑2} *Note: for transition with interactions in ∑1 only.*
  - {((s1,s2), γ2, α, θ2, (s1,s2')) | (s2, γ2, α, θ2, s2') ∈ T2, s1 ∈ S1, α ∈ (∑2 - ∑1)} *Note: for transition with interactions in ∑2 only.*

- {((s1,s2), γ1 U γ2, α, θ1U θ2, (s1',s2')) | (s1, γ1, α, θ1, s1') ∈ T1 and (s2, γ2, α2, θ2, s2') ∈ T2 and α ∈ ∑1 ∩ ∑2} *Note: for transitions with common interactions.*

Figure 3 shows the resulting machine of the composition operation of S and Dual(C) of our example. Notice in particular the output transition (C3, {p=cv; p=sv}, !use(p), {}, D4) which is the result of composing S's output transition (C, {p=sv}, !use(p), {}, D) and Dual(C)'s input transition (1, {p=cv},?use(p), {}, 4). An implicit condition that variables cv and sv should be equivalent to avoid an unspecified reception is created.

3.2 Chaos Machine

The notion of chaos was introduced by Hoare [Hoare85] to denote the most general behavior of a module. It was also used in several papers on submodule construction [PeYe98, DrBo99, Boch02]. For the case of submodule construction we can add variables as much as we want, since we have the full control over the new machine. However, to simulate S and C we only need as many variables as S and C combined.

For our EOITS model the chaos machine has one state which has a looping transition for each input interaction and each combination of assignments of interaction parameters to local variables. Similarly, it has an output transition for each output interaction and each combination of variable assignments to parameters. In Figure 4 we give the Chaos machine for the submodule construction example using two variables which correspond to the two variables of C and S; for the interaction ?n(p), for instance, this machine contains 4 transitions with different variable assignments.

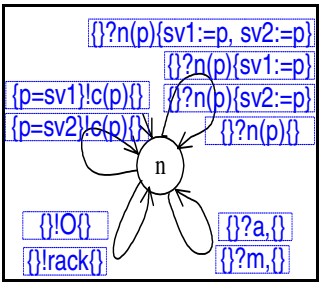

Fig. 4. Example Chaos machine: Chaos($\sum_X$, 2)

3.3 State Splitting

State splitting is done to separate variable configurations that cause guard failure into separate state splits. As outlined in the following algorithm, it is done in three steps, first a variable configuration information collection step in the form of variable partitions followed by two consecutive steps for state splitting.

Algorithm 2. State Splitting Algorithm:
Given G : EIOTS, $\sum_X$ Interaction Alphabet; return an EIOTS

- R := ComputePartitions(G)
- SplitPhaseOne(G, $\sum_X$, R, StateGroups)
- SplitPhaseTwo(G, $\sum_X$, R, StateGroups)

Return EIOTS G

3.3.1 Variable Partition Computation

The concept of variable partition is introduced for the purpose of statically analyzing an EIOTS machine. It mainly helps in providing an abstract representation of all variable configurations that are possible at each state of the machine. We are particularly interested in characterizing variable configurations that cause transition guards to fail.

After the execution of a given transition, some variables will be known to have same values (like those assigned the same input parameter). We say that such variables match, and we are interested in finding all variable matching relations at each and every state of the EIOTS. A variable matching relation is an equivalence relation, since it is reflexive, symmetric and transitive. Therefore, it can be represented by a partition over the set of variables since every equivalence relation over a set defines a unique partition of the elements in that set and vice versa. At any given state, more than one variable configuration or equivalence relation may exist since a state may be reached through different paths, and each execution path can create possibly a new variable configuration. However, since the number of variables is finite, there will be a finite number of possible variable relations and variable partitions at each state. Typically the initial configuration is represented with a single relation where no variable is known to be matching any other variable than itself. So, the initial partition is made of classes that have one variable each.

For a given variable configuration we can tell whether it conforms to a transition guard by checking whether its corresponding variable partition conforms to the guard. In the following we define the conformance predicate.

Definition 1: Partition Conformance Predicate. The predicate Conform is a mapping, $Px2^{PmxV} \rightarrow \{True, False\}$ where P is the set of all partitions of V (the set of all variables), and 2^{PmxV} represents the set of all possible transition guards, such that Conforms(π,γ)=

$$\forall p \in Pm, \forall cl \in \pi, for\ E = \{v \in V \mid (p, v) \in \gamma\}\ (E \cap cl \neq \{\} \Rightarrow E \cap cl = E)$$

Basically it says, if a parameter is restricted the value of several variables then these variables should be equivalent.

When a transition is executed it updates variables thus changing the variable configuration of the machine. In the following we define a transformation function that defines the partition representing the new variable configuration given the partition representing the old configuration and the executed transition.

Definition 2: Partition Transformation function. Each transition in the EIOTS defines a transformation function **Transform**: $Px2^{PmxV}\ x2^{VxPm} \rightarrow P\ U\{\}$, where Transform($\pi$, γ, θ) is defined as follows: Let r be the relation in VxV corresponding to partition π. If Conforms (π, γ) then Transform(π, γ, θ) = π' where π' is the partition corresponding to relation r' in VxV such that for variables v1, v2 $\in$ V, (v1,v2) $\in$ r' if

a) (v1,v2) $\in$ r and not $\exists$ (v1,p1) or (v2, p2) $\in \theta$, or
b) $\exists$ p$\in$ Pm such that (v1,p) and (v2,p)$\in$ θ, or
c) $\exists$ p$\in$ Pm, v3$\in$ V such that (v1,p)$\in$ θ, (p,v3)$\in$ γ, (v3,v2)$\in$ r, and not $\exists$ p' such that (v2,p') $\in$ θ

So there will be a resulting partition if the original partition conforms to the transition guard. And in the resulting partition two variables will be related or in other words, will be in the same class, if (a) they were in the same class of the original

partition and neither is assigned by the transition, or (b) they are assigned the same parameter, or (c) one of them is assigned a parameter that is restricted to the value of a variable that was in the class of the second, while the second is not assigned a new value.

Using the partition transformation function we can define the concept of reachable partitions to a state as follows.

Definition 3: Reachable Partition to a State. We say that a partition π is reachable to a state *s* if and only if there exists a path from *s0* to *s* such that a partition π0={{v}| v∈ V} will be transformed to π after successively applying on π0 all the transformations defined by the transitions in the order defined by the path leading to *s*.

We can compute the set of reachable partitions for each state in an EIOTS machine by a recursive procedure described in the following.

Algorithm 3. Reachable Partitions Computation Algorithm:
Given E<S, V, s0, Se, ∑, T> returns R *//R={(s,π)∈ SxP|π is reachable to state s}.*

- *R* := { }
- π0 := {{v} | v ∈ V}
- *newPartitions* = {(s0, π0)}
- Loop while *newPartitions* ≠ { }
  - Remove a couple (s1, π) from *newPartitions*
  - *R* = *R* U {(s1, π)}
  - For each transition (s, γ, a, θ, s') in {(s, γ, a, θ, s') ∈ T | s = s1}
    - if conforms(π, γ)
      - π' = Transform(π, γ, θ)
      - If (s', π') ∉ *R*
        - *newPartitions* = *newPartitions* U{(s', π')}

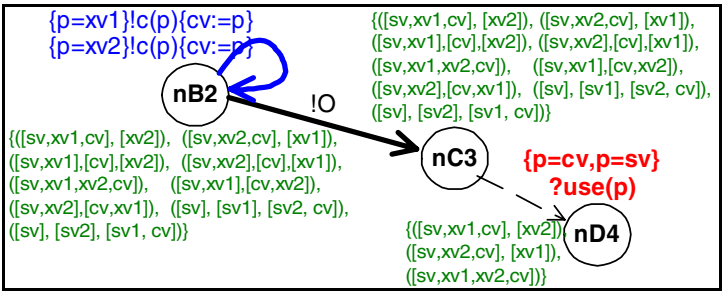

Fig. 5. Variable Partition Computation: part of Chaos(X.∑, 2)xDual(SxDual(C))

The partition computation algorithm is a fixed point algorithm that loops until reaching a point where no progress can be made. Progress is evaluated in terms of finding new partitions possible in some state of the EIOTS. Since this algorithm only adds partitions and since the maximum number of partitions that can be introduced is bounded by the finite set of variable partitions, this algorithm is guaranteed to terminate.

In figure 5 we give part of the results of applying the partition computation algorithm to the composed behavior of X. Note how reachable partitions to state nD4 are only those variable partitions conforming to the guard {p=sv, p=cv} of the incoming transition.

3.3.2 Phase One

Once all variable partitions reachable at the states of the combined behavior are computed, we will be able to proceed with state splitting. In this phase of the algorithm we split into two each state that has an outgoing transition with a guard that fails for certain reachable variable partitions and succeeds for others. One state split will hold the failing partitions and the other will hold the succeeding ones.

Since each original state might have more than one outgoing transition, a state might be split into many state splits according to the different combinations of transition guards' successes and failures, and according to the availability of reachable partitions that satisfy each combination. Each group of states resulting from the splitting of one original state is saved together to be handled collectively in phase two. One element of each group is marked as the first state of the group to which all incoming transitions to the original state are still attached. These incoming transitions are to be handled in the second phase by either redirection or duplication and subsequent recursive state splitting.

Since we know that all uncontrollable behavior leading to an unsafe behavior will be eventually blocked, we treat this behavior collectively by using one state split to represent all partitions causing unsafe behaviors. Therefore, a one element of each state split group is marked as the uncontrollable split state and is used to hold all partitions that cause at least one guarded uncontrollable transition of the original state to fail. Uncontrollability in submodule construction is determined by the ability of the new module to control the execution of a given transition; therefore, we need to provide the algorithm with the set of interactions of the new module. A transition is controllable if (a)its interaction is initiated by the new module, or (b) if the interaction is an input to the new module and it is not the last transition from the same state with the same interaction that is not going to the error state. The second controllability condition is used to represent the ability of the new module to select a particular assignment of parameters to its local variables.

Next we give the algorithm for phase one. We use the function *copyState* as a shorthand for creating a new state as a copy of an existing state without copying incoming or outgoing transitions. We use as well the procedure *ReplaceState*(*old*, *s, G*) to replace state *old* by state *s* in *G* through diverting all incoming transition of state *old* to state *s.* Function *newErrorState* creates a new error state and returns it.

In Figure 6 we show the result of applying phase one of the splitting algorithm to the example used in Figure 5. In particular notice splitting state nC3 to two states nC3.1 and nC3.sE, where nC3.1 holds all partitions that conform to the guarded transition (nC3, {p=sv, p=cv}, ?use(p), {}, D4), and nC3.sE holds all partitions that do not conform, and since the mentioned transition is not controllable, nC3.sE is labeled as the error state of the state split group.

Algorithm 4. SplitPhaseOne Algorithm:
Given G < S, V, s0, Se, ∑ , T>, ∑X Interaction Alphabet, R Set
// Note: R ={(s, π)∈ SxP| π is reachable to state s}.

- StateGroups:= { } *// this variable will hold tuples of the form (sSet: Set of States, f: State, sError: state) to store split state groups that will be later treated in Phase 2. f holds all incoming transitions to the group, and sError is the state with all partitions causing unsafe and uncontrollable behavior*
- For each state curS in G.S
  - sError := CopyState(G, curS)
  - first := curS
  - sSet:={curS} *//will hold states resulting from the splitting of one original state*
  - Tout := { (s, γ, a, θ, s') ∈ T | s= curS }
  - Loop while Tout ≠ { }
    - Remove any element (s1, γ1, a1, θ1, s1') from Tout
    - BadPartitions = { π ∈ P | ∃ (s,π) ∈ R where s1 = s and (not conforms(π, γ1) or s'∈ G.Se)}
    - If BadPartitions ≠ { }
      - R = R – {(s, π) ∈ SxP | s = s1 and π ∈ BadPartitions}
      - If a1 ∈ $\sum X_{out}$ or (a1 ∈ $\sum X_{in}$ and cardinality({ (s, γ, a, θ, s',) ∈ T | s = s1, a = a1 and s' ∉ G.Se}) > *1) // transition is controllable*
      - *//Split the state, duplicate outgoing trans, disable nonconforming trans*
        - η := CopyState(G, s1)
        - sSet := sSet U {η}
        - T:=TU{(η, γ2, a2, θ2,s2') | ∃ (s2, γ2, a2, θ2, s'2) ∈ T where s2 = s1}
        - Tout:=ToutU{(η,γ2,a2,θ2,s2')| ∃ (s2,γ2,a2,θ2,s'2)∈ Tout where s2=s1}
        - T = (T – {(η, γ1, a1, θ1, s1')}) U {(η, γ2, a1, θ2, newErrorState(G))}
      - else *// Transition is uncontrollable*
        - η := sError
      - R = R U {(s,π) ∈ SxP| s = η and π ∈ BadPartitions}
      - If {(s, π)∈ R| s=s1}={ }*// Delete state from group if it has no* π's *left*
        - sSet = sSet – {s1}
        - If s1 = first
          - ReplaceState(first, η, G) *// redirects all incoming trans of first to η*
          - first := η
  - If {(s,p) ∈ R | s = sError} ≠ { }
    - sSet := sSet U {sError}
    - StateGroups:= StateGroups U {(sSet, first, sError)}
  - Else
    - If |sSet| > 1
      - StateGroups:= StateGroups U {(sSet, first, null)}

3.3.3 Phase Two

In this phase of the algorithm each group of split states resulting from phase one is handled separately. The state designated first of its group has all incoming transitions of the original state. Each incoming transition to the group may lead to a recursive creation of a new state split group of the transition origin. For example transition (nB2, { }, !O, { }, nC3) in figure 6 will lead to the splitting of state nB2. At first nB2 is

split into two states since nC3 group has only two states. However, due to incoming transitions to these states, the two nB2 states will be recursively split into states nB2.1, nB2.2, nB2.3, and nB2.4 as shown in figure 7.

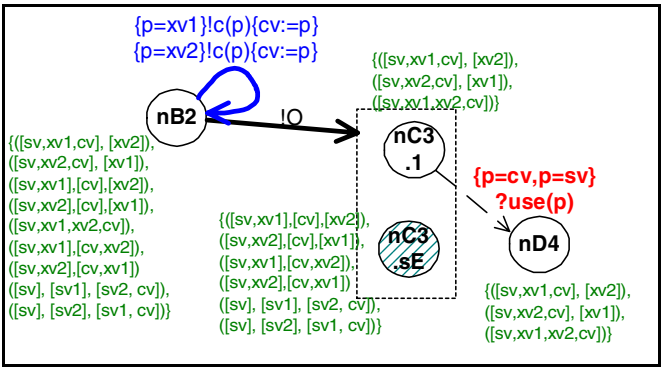

Fig. 6. Phase one of state splitting: part of Chaos(X.∑, 2)xDual(SxDual(C))

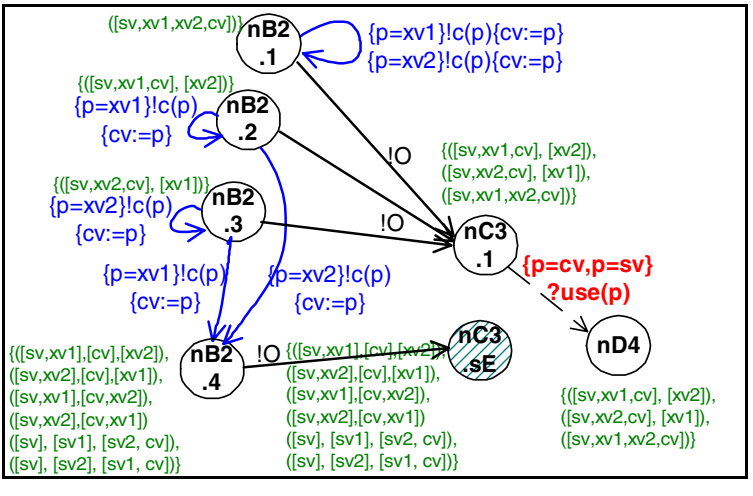

Fig. 7. Phase two of state splitting: part of Chaos(X.∑, 2) x Dual(SxDual(C))

This algorithm is guaranteed to stop since it splits a state only if there are reachable partitions to be split. And it only adds partitions to state split that is designated an error state, but such a state split is not split any further. The maximum number of states that could result from splitting would be |R| where R is the state partition reachability relation. In the extreme, this is the case where each state is split into as many state splits as there are reachable partitions, that is, each split state will be holding a single partition.

Algorithm 5. SplitPhaseTwo Algorithm
Given G < S, V, s0, Se, $\sum$, T>, $\sum_X$ Interaction Alphabet , StateGroups, R Set *// R ={(s, π)∈ SxP| π is reachable to state s}.*

- Loop Until StateGroups= { }
  - Remove some element (StateSet, first, sError) from StateGroups
  - Tin := {(s, γ, a, θ, s')∈ T | s'= first}*// all incoming transition to the current group*
  - Loop Until Tin = { } *//Handle each incoming transition separately*
    - Remove some element (s1, γ1, a1, θ1, s1') from Tin
    - For each state curS in StateSet
      - *// the inverse of Transform function for the partitions of the current state*
      - PartitionSet := { π ∈ P| (s1, π) ∈ R and (curS, Transform(γ1,θ1, π)) ∈ R}
      - If PartitionSet ≠ { }.
        - s1Group = {(sSet,e,f) ∈ StateGroups| s1 ∈ sSet}
        - If s1Group ≠ { }*// s1 the starting state of the current transition belongs to a state group that is waiting to be handled*
          - Remove some (sSet, e, f) from s1Group
          - StateGroups = StateGroups - s1Group
        - Else *// Create new group*
          - sSet := {s1}
          - f := s1
          - e := null
        - If curS = sError and (a1 ∉ $\sum X_{out}$ and (a1 ∉ $\sum X_{in}$ or cardinality({ (s, γ,a, θ, s') ∈ T| s = s1, a = a1 and s' ∉ G.Se }) = 1)*// uncontrollable transition*
          - if e = null
            - e := copyState(G, s1)
          - η:= e
        - Else
          - η:= CopyState(G, s1)
          - T:= TU{(η, γ2, a2, θ2, s2')| ∃ (s2,γ2, a2,θ2, s'2)∈ T where s2 = s1}
          - Tin:=TinU{(η,γ2,a2,θ2,s2')| ∃ (s2,γ2,a2,θ2, s'2)∈ Tin where s2=s1}
          - *// Redirect η's transition that is the duplicate of the current transition to the current state.*
          - T:= (T – {(η, γ1, a1, θ1, s1')}) U {(η, γ2, a1, θ2, curS)}
        - sSet := sSet U {η}
        - R:= R U {(η, π) | π ∈ PartitionSet}
        - R:= R - {(s, π) ∈ R | s = s1 and π ∈ PartitionSet}
        - If {(s, π) ∈ R | s = s1} = { } *//Delete state if it has no partitions left*
          - sSet:= sSet – {s1}
          - If s1 = f
            - ReplaceState(f, η, G) *// redirect all incoming transitions of f to η*
            - f := η
        - If |sSet| > 1 or e ≠ null
          - StateGroups:= StateGroupsU {(sSet, f,e)
  - G.Se = G.Se U {sError}

The state labeled as the error state of the currently handled group (such as state nC3.sE in Figure 8) receives special treatment. When handling an incoming transition to the error state of a group, the new state split of the transition's origin state corresponding to the error state is labeled itself as the error state of its group only if the transition is uncontrollable. In our example, new state split nB2.4 is not marked as the error state of its group since transition (nB2.4, {}, !O, {}, nC3.sE) is controllable.

3.4 Determinization and the Removal of Uncontrollable Behavior

Determinization or internal transition removal uses the usual subset construction algorithm for determinizing finite automata. As mentioned before, this is possible since the splitting algorithm removes the ambiguity created by transition guards.

The determinization results in new unsafe states due to unobservability. These states are removed together with all states from which the unsafe states can be reached through uncontrollable transitions. The same controllability criterion is used as in the case of splitting. This is similar to the case of submodule construction for simple finite state machine models.

4 Conclusion and Future Work

This paper addresses the problem of extending submodule construction techniques for finite state machine models to more expressive behavioral models that use variables, simple guards for transitions and exchange data with the environment through interaction parameters. We have defined a behavioral model with features based on an extended model of Input-Output Automata. The main contribution of this paper is the introduction of dataflow issues to submodule construction problem which has been limited in the past to control flow. However, we have only dealt with the simple usage of data, mainly saving and retransmission. We seek in the future to handle the control flow usage of data through building up on the current approach. We need to ease restrictions on guards such as allowing conjunction, disjunction, explicit negation and state variable equality predicates. This work will be as well the basis for further work on providing more efficient versions of the proposed algorithm through exploring the use of higher abstractions for representing variable partitions and taking into consideration undefined and dead variables.

References

[BEYB03] S. Buffalov, K. El-Fakih, N. Yevtushenko, G. V. Bochmann: Progressive Solutions to a Parallel Automata Equation. FORTE 03, pp. 367-382, 2003.

[Boch02] G. V. Bochmann. Submodule Construction for Specifications with Input Assumptions and Output Guarantees. FORTE 02, pp.17-33, 2002.

[BrWo94] B. A. Brandin, and W.M. Wonham. Supervisory Control of Timed Discrete Event Systems. IEEE Transactions on Automatic Control, Vol. 39, No. 2, pp. 329-342, 1994.

[DrBo99] J. Drissi, and G.V. Bochmann. Submodule Construction for Systems of I/O Automata. Tech. Rep. no. 1133, DIRO, University of Montreal, 1999.

[Hoar85] C. A. R. Hoare. Communicating Sequential Processes, Prentice Hall, Inc., 1985.

[KeHa93] S.G. Kelekar, G. W. Hart. Synthesis of Protocols and Protocol Converters Using the Submodule Construction Approach. PSTV93, pp. 307-322, 1993.

[KNM97] R. Kumar, S. Nelvagal, and S. I. Marcus. A Discrete Event Systems Approach for Protocol Conversion. Discrete Event Dynamical Systems: Theory and Applications, Vol. 7, No. 3, pp. 295-315, 1997.

[LeQi90] P. Lewis and H. Qin. Factorization of finite state machines under observational equivalence. LNCS 458, Springer, 1990.

[LyTu89] N. Lynch and M. Tuttle. An introduction to input/output automata, CWI Quarterly, Vol. 3, No. 2, pp. 219-246, 1989.

[MeBo83] P. Merlin, and G. v. Bochmann. On The Construction of Submodule Specifications and Communication Protocols, ACM Trans. On Programming Languages and Systems. Vol. 5, No. 1, pp. 1-25, 1983

[NeBr95] R. Negulescu, J. A. Brzozowski. Relative liveness: from intuition to automated verification. ASYNC 95, 108-117, 1995.

[Parr89] J. Parrow. Submodule Construction as Equation Solving in CCS. Theoretical Computer Science, Vol. 68, 1989.

[PeYe98] A. Petrenko and N. Yevtushenko. Solving Asynchronous Equations. In Proc. of IFIP FORTE/PSTV'98 Conf., Paris, Chapman-Hall, 1998.

[QiLe91] H. Qin and P. Lewis. Factorisation Of Finite State Machines Under Strong and Observational Equivalences. Journal of Formal Aspects of Computing, Vol. 3, pp. 284- 307, 1991.

[RBJ00] V. Rusu, L. du Bousquet, T. Jéron. An Approach to Symbolic Test Generation. IFM 2000: 338-357, 2000.

[Shie89] M. W. Shields. Implicit system specification and the interface equation. The Computer Journal, Vol. 32, No. 5, pp. 399-412, 1989.

[TBD97] Z. Tao, G. v. Bochmann and R. Dssouli. A Formal Method For Synthesizing Optimized Protocol Converters And Its Application To Mobile Data Networks. Mobile Networks & Applications, Vol.2, No. 3, pp. 259-69, 1997.