This document contains the draft version of the following paper:

C. Xu, S.K. Gupta, Z. Yao, M. Gruninger, and R. Sriram. Towards computer-aided conceptual design of mechatronic devices with multiple interaction-states. *ASME Computers and Information in Engineering Conference*, Long Beach, CA, September 2005.

Readers are encouraged to get the official version from the conference proceedings or by contacting Dr. S.K. Gupta (skgupta@umd.edu).

TOWARDS COMPUTER-AIDED CONCEPTUAL DESIGN OF MECHATRONIC DEVICES WITH MULTIPLE INTERACTION-STATES

Changxin Xu
Mechanical Engineering Department and Institute for Systems Research
University of Maryland
College Park, MD-20742, USA

Satyandra K. Gupta[1]
Mechanical Engineering Department and Institute for Systems Research
University of Maryland
College Park, MD-20742, USA

Zhiyang Yao
Automation and Computer-Aided Engineering Department
Chinese University of Hongkong
Shatin N.T. HKSAR, P.R. China

Michael Gruninger
Institute for Systems Research
University of Maryland
College Park, MD-20742, USA

Ram Sriram
Manufacturing Systems Integration Division
National Institute of Standards and Technology
Gaithersburg, MD

ABSTRACT

In multiple interaction-state mechatronic devices the interactions between elements of use-environment and elements of the device can have different qualitative structures depending upon the modes of the device operation and the states of the use-environment. This paper describes a modeling framework to support conceptual design of such devices using state transition diagrams. We define the primitives and operators needed in the modeling framework, and illustrate the conceptual design process using these primitives and operators. We believe that the framework described in this paper will provide the underlying foundations for constructing the next generation software tools for the conceptual design of mechatronic devices.

Keywords: conceptual design, state transition diagrams, interaction, mechatronic.

1 INTRODUCTION

Increasing autonomy and intelligence in mechatronic devices requires them to be multiple interaction-state devices. In multiple interaction-state devices the interactions between elements of use-environment and elements of the device can have different qualitative structures (i.e., different interaction topologies) depending upon the modes of device operation and the states of the use-environment. For example, consider a hybrid vehicle as shown in Figure 1. When the vehicle is going down a hill, the engine is storing energy into the batteries. While when the vehicle is going up a hill, both the batteries and the engine are providing power to the wheels.

Figure 2 shows an abstraction of the information flow in a typical product development process [Pahl96]. This figure mainly illustrates the information flow and does not show the iterative nature of the design process. The first step is need analysis, which determines the requirements. This step establishes why a device should exist. The second step is to establish behavior specifications, which creates the specifications of the desired observable behavior of the device

[1] Corresponding Author

that satisfy the requirements. This step establishes what a device should do. After that, the conceptual design step analyzes the desired behavior of the device and results in the specifications of the internal structure of the device. Finally the detailed design step completes the design by developing details of every component in the structure. The conceptual and the detailed design establish how the device will provide the desired behavior.

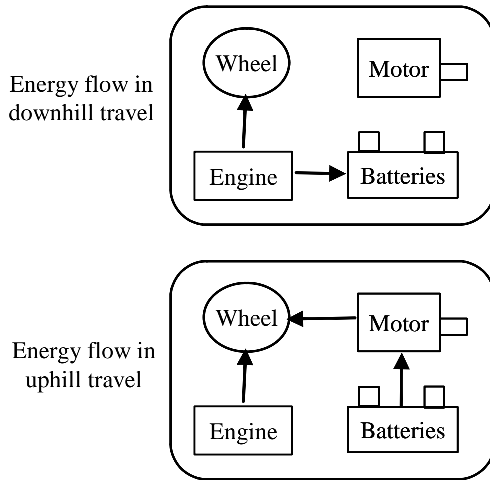


Figure 1: Example of interaction-states in a hybrid car

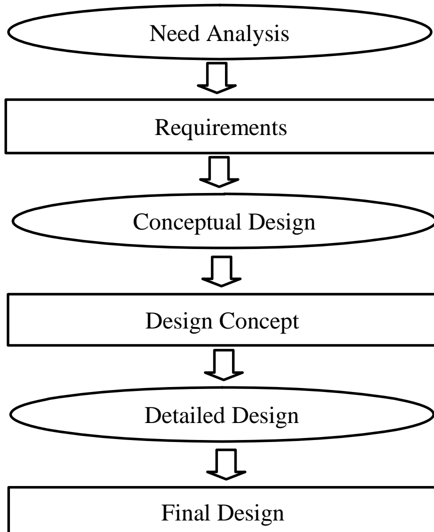


Figure 2: An abstraction of information flow in design (this figure only shows the information flow and does not depict loops generated by the iterative nature of the design process)

Today's intensive competition in the market requires companies to deliver better quality products in shorter lead-times with limited product development budget. Computer aided design (CAD) tools are being used to satisfy such needs. However, most of the commercial CAD systems for mechanical products are aiding designers only in the detailed design step. CAD tools for early stage of mechanical design are either restricted to few specific products or only providing simple sketching functions. CAD models currently only store geometric information and there is no connectivity between the final product geometry and requirements.

For multiple-interaction state mechatronic devices, a satisfactory modeling framework does not exist to support conceptual design. Besides, traditional functional modeling approaches that have been developed for single interaction topology based devices cannot be conveniently applied to multiple interaction-state devices. It is clear that if we were to achieve a high level of automation in design of multiple interaction-state mechatronic devices we will need a new modeling framework to support conceptual design.

State transition diagrams provide a way to represent changes in interactions over time and hence provide a convenient way to model multiple working modes of a device. This paper describes a modeling framework for conceptual design of such devices using state transition diagrams. We define the primitives and operators needed in the modeling framework and give the rationale for their need. We also illustrate the conceptual design process using these primitives and operators on a simple design example. We believe that the framework presented in this paper is a step towards the development of computer-aided tools for aiding conceptual design.

2 RELATED WORK

Pahl and Beitz's widely accepted systematic approach to engineering design defines conceptual design as the feasible combination of working principles for sub-functions. Pahl and Beitz describe functions as the transformation from input to output in three flows: material, energy and signal [Pahl96]. Working principles are sought for low level sub-functions and working structures are formulated by combining working principles.

Grabowski et. al. divides the traditional function model into three layered function models with different levels of abstraction. Logical model, borrowed from electronic domain, is used to present high level topology and connectivity of sub-functions. Status model describes the working state combinations of different components. Relation model defines the mathematical or physical relations between physical variables [Grab99].

Shooter et. al. proposed a model for design information flow [Shoo00]. This model was further refined and resulted in the NIST Core Product Model (CPM) [Fenv01]. This model provides a base-level product model that is open, non-proprietary, generic, extensible, independent of any product development process and capable of capturing the full engineering context commonly shared in product development. The CPM is intended to serve as a generic core representation for design information through the whole product development process. Specialized representations can be developed from it by deriving specialized classes from it.

Stone and Wood have developed a functional basis language that tries to subsume the previous effort in functional modeling and provide a consistent classification scheme for functions. In this approach, functions are characterized using function-flow format and definitions of different classes of functions and flows are provided [Ston00, Hirt01]. Bohm and Stone recently extended this work by including supporting functions [Bohm04].

Zeiny developed a dynamic object-oriented model that stores form, function, behavior, taxonomy, composition and relationships [Zein04].

Chandrasekaran proposed a language called function representation for describing the function of a device, its

structure and the causal processes in the device that culminate in the achievement of the function [Chan94]. The causal process is described using simple state transitions.

Iwasaki et al. proposed the Causal Functional Representation Language (CFRL) [Iwas95]. They argued that this framework allows them to capture the knowledge of how the device is intended to work to achieve its function.

Sasajima et al. proposed Representation Language for Behavior and Function (FBRL) for representing function and behavior with predefined task and domain independent primitives [Sasa96]. Umeda et al. proposed Function-Behavior-State (FBS) modeling and a conceptual design support tool called FBS modeler [Umed96]. In FBS, a state is described by a set of entities and attributes and relationships between them. Behavior is described by a sequence of one or more changes of states. Deng et al. proposed a representation model for desired product in terms of its function, behavior, structure and working environment [Deng99].

Vargas-Hernandez and Shah presents an information model called 2nd-CAD that aims at providing user with catalogs of elements to create interconnected multi layered structures of functions, behaviors, and components. Function, behavior and component are represented in function entity-relationship model, behavior entity-relationship model and component entity-relationship model respectively [Varg04].

Chen and Jayaram extended flow diagram based functional representation schemes into mechatronic system representation by introducing two additional flows (information flow and control flow) and new relationships between functions and flows [Chen02, Jaya03].

Gausemeier et. al. proposed a semi-formal specification language for modeling functions in conceptual design of mechatronic systems. Functions are viewed as transformations of discrete system states described by parameters [Gaus01].

Williams describes design as a process of building a network of qualitative interactions between primitive components. Interactions are described by equations among variables of components [Will92].

Aiming at constructing the logical relationships between sub-functions at the first level of functional decomposition through information flows, Erden et. al. combine Petri nets with hybrid automata to model the logical behavior of mechatronic systems. Hybrid automata are used to model both discrete and continuous state changes and evolution [Erde03].

State transition diagrams (STD), also known as state machines, are a way of describing the time-dependent behavior of a system. STDs are useful for modeling complex system behavior such as multiple entries and exits subject to different conditions. STDs have been formalized in Unified Modeling Language (UML) [Booc98]. Researchers have also used hybrid automata that combine discrete transition diagram with continuous systems in modeling dynamical behavior [Broo04].

Previous research in conceptual design modeling area is mainly focused on single interaction-state systems, where the relationship between different components of the system is fixed during operation of the product. Representing multi-state mechatronic systems not only requires representing the components of the system, but also new features such as changing interaction topologies between system components. Besides, mechatronic product design requires us to consider complex interactions between system elements. In order to describe complex behaviors, we will also need to explicitly model the use-environments. In order to support simulation during the conceptual design, we will also need to formally define the events that trigger different interactions. This requires us to develop a new representation based on the combinations of existing representations to capture the behavior exhibited by multiple interaction-state devices.

3 OVERVIEW OF CONCEPTUAL DESIGN OF MULTIPLE INTERACTION STATE SYSTEMS

While it is well understood what is the outcome of the detailed design, it is not always clear what is the outcome of the conceptual design step. In this paper, we assume the conceptual design step will define the following three main items. First, it will identify the various major components that will be needed to meet the requirements and their roles in meeting the requirements. Second, it will specify basic working principles behind every main component to ensure that the component is realizable. Third, it will specify how various components will interact with each other to achieve the requirements.

In our framework, the conceptual design is carried out using the following two main steps:

Step 1: Define Behavioral Specifications. This step begins by defining the device's main working modes, various events that can happen in the use environment (for example, user pushing the emergency stop), and conditions that lead to unsafe operation conditions. We use state-transition diagrams to show the main working modes of the device and how the working modes change due to events in the use-environment. The modeling primitive event space is used to define all possible events. The modeling primitive unsafe parameter value set is used to define unsafe operation conditions. Primitives are described in Section 4.

We use interaction-states to capture main working modes of the device. Each interaction-state in the behavior specification shows how the device interacts with the use-environment. Use-environment will be defined using a set of components, which are defined by a set of parameters and interactions among them. Figure 3 graphically shows the main primitives needed to define an interaction-state in a transition diagram. Figure 4 shows all the parameters needed in our modeling framework and relationship among them. This figure also provides subsections that give class definitions for these primitives.

Step 2: Elaborate Transition Diagrams: After the initial transition diagram is constructed, the device component may need to be further decomposed such that it can be realized via known working principles. As a result of the component decomposition, interaction-states and transitions may also need to be decomposed to ensure that known working principles can be identified. Section 5 defines three operators for decomposing components, transitions, and transition diagrams. After a detailed transition diagram has been developed as result of the application of these operators, the conceptual design step is completed.

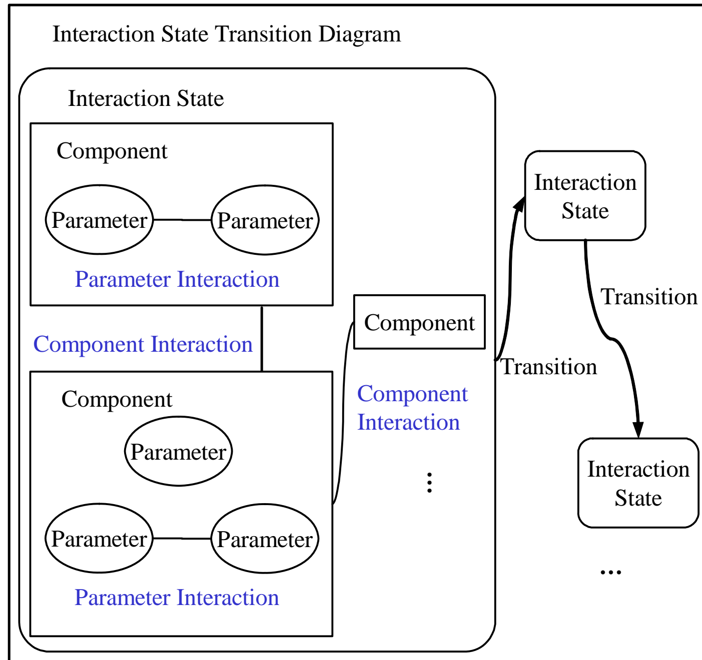


Figure 3: Primitives needed to define an interaction-state

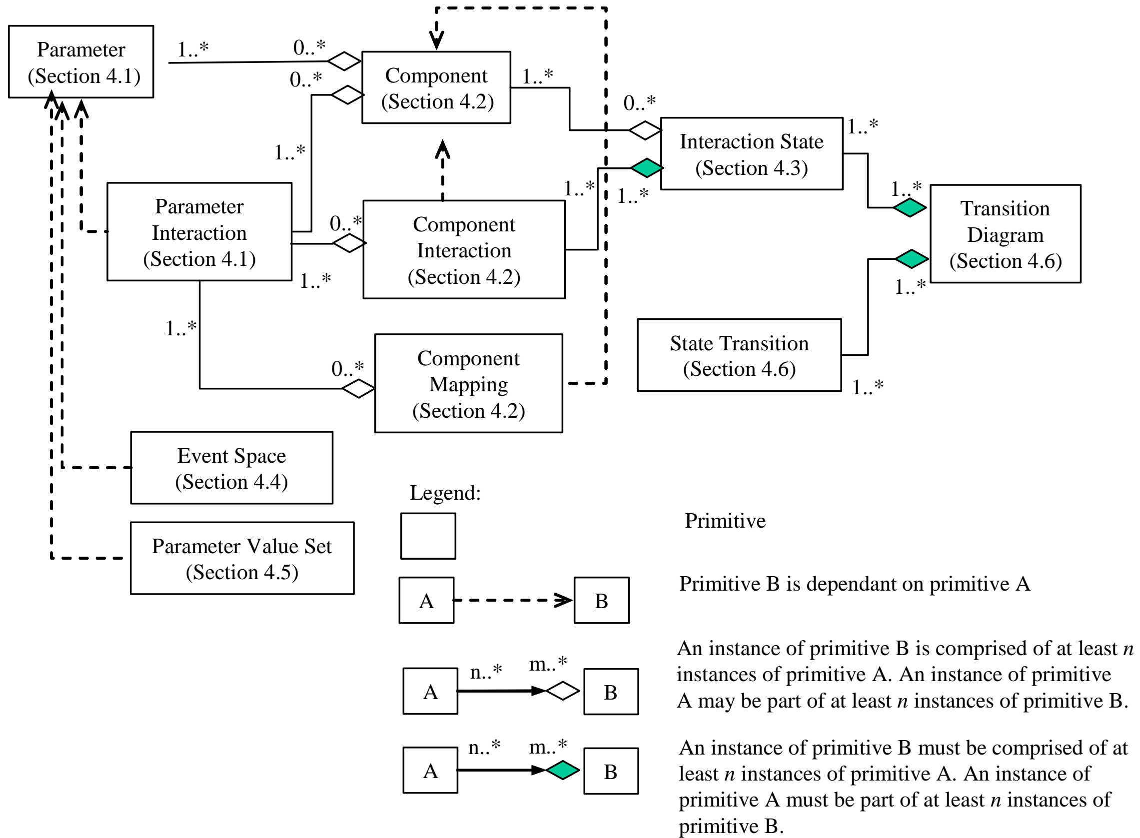


Figure 4: Relationships between main primitives

4 CLASS DEFINITIONS FOR MODELING PRIMITIVES

In this section we define classes for the modeling primitives shown in Figure 4. Every class instance will have a name that will serve as the identification for the class instance. We use notation "name.member" in this paper to refer to a member of a class instance. For example, notation a.p refers to member p of class instance with name a. In the following subsections we introduce the class definitions for various primitives.

We will use the following format and notations in subsequent sections. Class name will be described using Bold Arial font (e.g., **Parameter**). Class member will be described using Italic Times font (e.g., *DataType*). Symbol will be described using Capitalized Italic Times font (e.g., *INTEGER*).

4.1 Classes for Modeling Parameters and Parameter Interactions

A parameter is a type of observation of a component. For example, a motor is a component, and the weight of the motor is a parameter. Class **Parameter** is defined using the following members:

- *DataType* indicates the data type of this parameter. Parameter can be of several different data types. Our framework supports basic data types such as *INTEGER*, *REAL*, *BOOLEAN*, and *STRING*. We also support user-defined data types that are defined by using class **UserDefinedDataType** in terms of basic data types.
- *Unit* is a string that describes the unit of a data element. If the unit is not required, then it is set to *NONE*.

Class **UserDefinedDataType** is defined using member *Fields*, a set of names of Parameter instances.

For example, **Parameter** *position* can be defined in the following manner:

***position* : Parameter** *DataType = positionVector* *Unit = NONE*	***positionVector* : UserDefinedDataType** *Fields* = {*x*, *y*, *z*} *Unit = NONE*	
***x* : Parameter** *DataType = REAL* *Unit* = "mm"	***y* : Parameter** *DataType = REAL* *Unit* = "mm"	***z* : Parameter** *DataType = REAL* *Unit* = "mm"

Parameter may also take *NONE* or *NA* (not available) as a value for convenience. When a parameter does not have a value, we assign its value as *NONE*. When the value of a parameter is not known at the time of modeling, we assign the value as *NA*.

Class **Parameter** can be derived from class **CommonCoreObject** in the NIST CPM [Fenv01].

Relationships among parameters are called parameter interactions. From the perspective of the governing equations behind the relationships, there are two types of interactions:

- *Declarative Interactions*: These can be modeled using algebraic or ordinary differential equations. For example, the interaction of the mass parameter and the volume parameter of a component with uniform density is given by $m=dv$, where m is the mass, d is the density, and v is the volume of the component. However, in conceptual design stage, the exact equation may not be available. A qualitative structure that describes the characteristics of the interaction is then used.
- *Procedural Interactions*: These cannot be modeled explicitly using algebraic or ordinary differential equations during conceptual design. In most of the cases, a procedure is needed to describe these interactions. If simulation is necessary, then simplified numerical simulation can be used as surrogates for these interactions. For example, the interaction among a light source, a person, and, image at the camera lens (i.e., light from the light source reflects from person's face and forms an image at the camera lens) cannot be modeled by algebraic equations or ordinary differential equations.

We define class **ParameterInteraction** using the following members:

- *InteractionReason* is a tag taken from the following options: *ENERGY FLOW*, *SIGNAL FLOW*, *MASS FLOW, SPATIAL CONSTRAINT, LAW,* and *OTHER*.
- *InteractionType* is a tag taken from the following options:
  - *NON-CAUSAL INTERACTION*: For these interactions, there is no need to specify the dependence among parameters. For example, the interaction of the mass and the volume of a component with uniform density is a non-causal interaction.
  - *CAUSAL INTERACTION*: For these interactions, we have to specify the dependent relationships between parameters.
- *ParameterSet* is a set of names of **Parameter** instances that interact with each other.
- *DependantParameter* is the name of a **Parameter** instance whose value is dependent on the other parameters belonging to a *ParameterSet* as a result of the interaction. For non-causal interactions, *DependantParameter* is set to *NONE.*
- *Equation* is an algebraic or ordinary differential equation (in terms of parameters) if the interaction is declarative. In this case, it is defined as an instance of class **Expression**. If the interaction is procedural or the exact form of the equation is not available, then we don't capture the equation. Therefore this field is set to *NA*.

Class **Expression** is defined using a member called *Content*. *Content* is a special type of string that starts and ends with a parenthesis symbol. It includes numbers, standard and user defined function names, logical symbols, and mathematical symbols.

Class **ParameterInteraction** can be derived from class CommonCoreRelationship in the NIST CPM [Fenv01].

4.2 Classes for Modeling Components, Component Interactions, and Component Mappings

A component is a finite collection of parameters and the interactions among these parameters. Class **Component** is defined using the following members:

- *InputParameterSet* is a set of names of **Parameter** instances. These parameters serve as the input ports for flow types of interactions among components.
- *OutputParameterSet* is a set of names of **Parameter** instances. These parameters serve as the output ports for energy and signal flow types of interactions among components.
- *GeneralParameterSet* is a set of names of **Parameter** instances. These parameters do not play input or output role.
- *ParameterInteractionSet* is the set of names of **ParameterInteraction** instances describing interactions among parameters belonging to the component.
- *ComponentType* is a tag assigned to either *USE-ENVIRONMENT* or *DEVICE* to classify two different types of components.

For example, let us consider a DC motor without load. It can be represented by

***motor* : Component**	***c* : ParameterInteraction**
InputParameterSet = {v, k} *OutputParameterSet* = {ω} *GeneralParameterSet* = {*weight*} *ParameterInteractionSet* = {c} *ComponentType* = *DEVICE*	*InteractionReason* = *LAW* *InteractionType* = *CAUSAL INTERACTION* *ParameterSet* = {v, k, ω} *DependantParameter* = ω *Equation* = ($\omega = v/k$)

***v* : Parameter**	***k* : Parameter**	***ω* : Parameter**	***weight* : Parameter**
DataType = *REAL* *Unit* = "m/s"	*DataType* = *REAL* *Unit* = *NONE*	*DataType* = *REAL* *Unit* = "rad/s"	*DataType* = *REAL* *Unit* = "kg"

Where v is the input voltage, k is the motor constant, ω is the no-load speed.

If a is the name of a **Component**, then we use notation $a::p$ to refer to **Parameter** p of **Component** a.

Class **Component** can be derived from class **Artifact** in the NIST CPM [Fenv01].

Components interact with each other to affect their mutual behaviors. Complex components can also be decomposed into simple components. These two relationships about components are modeled using classes **ComponentInteraction** and **ComponentMapping**.

Component interactions usually result due to interactions among their parameters. We define class **ComponentInteraction** using the following members:

- *ComponentSet* is the set of names of the **Component** instances in the interaction.
- *InteractionInfo* is defined as the set of names of **ParameterInteraction** instances that describe the parameter interactions between the components.

All component interactions can finally be modeled as parameter interactions.

A component mapping is defined as the relationship between a component and its children components. The relationship includes component hierarchy and parameter mapping between parent component and children components. We define class **ComponentMapping** using the following members:

- *Component* is the name of the **Component** instance being decomposed.
- *ChildrenComponentSet* is the set of the names of children **Component** instances resulting from the decomposition of *Component*.
- *ParameterMappingSet* is a set of **Expression** instances. Each **Expression** instance defines the relationship between the parent component's parameters and its children components' parameters. For example, suppose parameter p_1 of parent a_1 is mapped to parameter p_2 of children component a_2 and parameter p_3 of children component a_3. Then, a possible expression can be ($a_1::p_1 = a_2::p_2 + a_3::p_3$).

Class **ComponentMapping** can be derived from class **CommonCoreRelationship** in the NIST CPM [Fenv01].

4.3 Classes for Modeling Interaction-States

An interaction-state describes the interactions between a set of components. For example, if a motor is driving a gearbox to transmit mechanical energy, then the interaction-state of this set of components is the description of the motor, the power source, the gearbox, and their interactions. Every component in the component set of this interaction-state must participate in at least one component interaction in this state. A component is ***active*** in the interaction-state if it belongs to the component set of the state. Otherwise, the component is considered *inactive* in the state. Usually when we refer to the components in a state, we refer to the active components in the state.

We use symbol t to denote the time variable associated with the internal clock of the state. We call t the ***local time variable*** because t only exists with respect to a specific state. On the other hand, during simulation we need another variable to indicate the time in the design world, which includes all states of the device. This time variable is denoted as T and is call the ***global time variable***. At a given global time $T=T^*$, the device is in a particular state with its own corresponding local time $t=t^*$. Within a state, t starts from 0. Ending time of a state is denoted by symbol t_e. At a particular time t, the value of a parameter p of component a is denoted by $a::p(t=t')$. $a::p(t)$ is used to represent the value of a parameter parametrically. On the other hand, if the global time variable is used to indicate the value of a parameter, we use $a::p(T=T')$ for a specific time, and $a::p(T)$ to represent it parametrically.

Notation $s::a::p$ will be used to refer to the **Parameter** p of **Component** a in **State** s.

We define class **InteractionState** using the following members:

- *ComponentSet* is a set of names of **Component** instances that are active in the state.
- *ComponentInteractionSet* is a set of names of **ComponentInteraction** instances between the active components in this state.

- *InitialValueSet* is a set of names of class **ValueAssignment** instances that describes how parameter values are initialized.
- *ChangeModeSet* is a set of names of class **ChangeMode** instances that describes how parameter values will change inside of the interaction-state.

Class **ValueAssignment** is defined using the following members:

- *ParameterName* is the name of a **Parameter** instance.
- *InitializationType* is a tag taken from the following options:
  - *INHERIT* indicates that the parameter inherits its value from a previous state. Let the current state be s, and its previous state be s', then the initial value of a parameter $a::p$ belonging to this component can be obtained in the following manner: $s::a::p\ (t = 0) = s'::a::p\ (t = t_e)$, where t_e is the ending time of state s'.
  - *DERIVE* indicates the value of a parameter is derived from other parameter values that belong to some components in the same state.
  - *ASSIGN* indicates the value of a parameter is assigned to a particular value.
- *Value* denotes the value of a parameter. If the *InitializationType* is set to *INHERIT* or *DERIVE*, it is set to *NA*.

Class **ChangeMode** is defined using the following members:

- *ParameterName* is the name of the **Parameter** instance.
- *ChangeType* is a tag taken from the following options:
  - *CONSTANT* indicates the value is a constant within the state.
  - *DERIVE* indicates the value changes according to the values of parameters it interacts with.
  - *EQUATION* indicates the value is changing according to time variable t.
- *Equation* is an equation in terms of a parameter with respect to the local time in a state. In this case, it is defined using class **Expression**. If the *ChangeType* is set to *CONSTANT* or *DERIVE*, it is set to *NA*.

Some limitations may apply on combining initialization types and value-changing modes as shown in Table 1.

States may be inconsistent if the underlying interactions are inconsistent. An *Interaction-state s* is ***inconsistent*** if equations defined in *ComponentInteractionSet* are inconsistent. Equations may turn out to be inconsistent if the system of equations is over-constrained.

Table 1: Limitations on combining initialization types and value-changing modes

Initialization type	Value changing mode
ASSIGN	*CONSTANT*
	EQUATION
INHERIT	*CONSTANT*
	EQUATION
DERIVE	*DERIVE*

4.4 Classes for Modeling Event and Event Spaces

An event occurs when a use-environment component becomes active or inactive, or a parameter or parameters of the use-environment components change their values. Event space refers to the set of all possible events that can happen in the use-environment. Class *Event Space* is defined using the member *ParameterRangeSet,* a set of names of **ParameterValueRange** instances.

Class **ParameterValueRange** is defined using the following members:

- *Parameter* is the name of a **Parameter** instance.
- *RangeType* is a tag taken from the following options:
  - *CONTINOUS* means that values are bound between *ValueLowerLimit* and *ValueUpperLimit.*
  - *DISCRETE* means that values are assigned from a *ValueSet.*
- *ValueSet* is a set of **Expression** instances. If *RangeType* is set to *CONTINOUS*, *ValueSet* is set to *NA*.
- *ValueLowerLimit* is a value for the parameter. If *RangeType* is set to *DISCRETE*, *Value LowerLimit* is set to *NA*.
- *ValueUpperLimit* is a value for the parameter. If *RangeType* is set to *DISCRETE*, *ValueUpperLimit* is set to *NA*.

During the simulation, global time variable T represents time in the design world, which includes the device and the use-environment. Every event happens in the use-environment at a certain specific value of T.

We define class **Event** using the following members:

- *GlobalTime* is the value of the global timer that describes the time when this event happens.
- *EventCondition* is defined as an instance of class **Expression** that describes parameter value changes during the event.

4.5 Classes for Modeling Unsafe Parameter Value Sets

A parameter value set is a snapshot of an interaction-state. In a transition diagram, interaction-states may contain a set (possibly infinite) of parameter value sets. A unique parameter value set can be extracted from an interaction-state by selecting a specific time instant in the interaction-state. For example at $T = 5$, the values of all parameters belonging to both the device and the use-environment components define the world-state at

T = 5. An unsafe parameter value set is a parameter value set that is forbidden by requirements.

Class **UnsafeParameterValueSet** is defined using member *ParameterValueSet*, where *ParameterValueSet* is a set of **Expression** instances that indicates the forbidden parameter values or value ranges by the requirements.

A design concept should never enter an unsafe parameter value set. Therefore, a design concept should be such that in response to all possible events contained in the event space, it should never enter an interaction-state that will contain unsafe parameter value sets.

4.6 Classes for Modeling Interaction-State Transitions and Transition Diagrams

An interaction-state transition is the indication of changes from one interaction-state to another interaction-state. We define class **InteractionStateTransition** using the following members:

- *StartState* is the **InteractionState** instance where the transition starts.
- *EndState* is the **InteractionState** instance where the transition ends.
- *TransitionCondition* is an **Expression** instance that indicates the condition under which the transition occurs. This is a composite expression that may contain (1) sub-expressions indicating the internal time clock of a state reaches a particular value, such as (t=4) or (2) sub-expressions indicating some parameters take particular values, such as (a::$p(t)$=5).
- *ClosureActionSet* is a set of **Expression** instances that describes how the parameters value will be set in the starting state before leaving it. For example, $\{(a_1::p_1(t=t_e) = 1), (a_1::p_2(t=t_e) = 2), (a_2::p_1(t=t_e) = 3)\}$.
- *Initialization Action Set* is a set of **Expression** instances that describe how the parameters value will be set in the ending state before entering it. $\{(a_1::p_1(t=0) = 2), (a_1::p_2(t=0) = 3), (a_2::p_1(t=0) = 3)\}$. Expressions in this set override the initialization expressions defined for a state.

InteractionStateTransition r is ***realizable*** for **InteractionState** s if there exists a sequence of events such that the device reaches s and transition condition for r is satisfied. If a transition is not realizable, then it is called ***unrealizable***. Unrealizable transitions should be eliminated from the design concept, as they do not contribute anything to the behavior.

A transition may be unrealizable because of a variety of reasons. In Figure 5 transitions r_1, r_2, and r_4 are not realizable. Transition r_4 is unrealizable because condition for transition r_3 is always satisfied before condition for transition r_4 is satisfied. Therefore, transition r_4 never takes place.

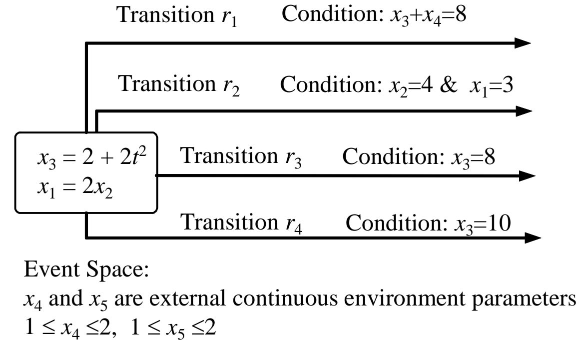


Figure 5: Unrealizable transitions

A transition diagram is a graph whose nodes are interaction-states and edges are interaction-state transitions. We define class **TransitionDiagram** using the following members.

- *InitialState* is the name of a special **InteractionState** instance. Every transition diagram must include an initial state, which is the device interaction-state at T = 0. As a special interaction-state, the initial state has all the components including device components and use-environment components. Parameters of these components are initialized in the initial state. However, all the components remain inactive until events trigger the device to leave the initial state.
- *InteractionStateSet* is the set of names of remaining **InteractionState** instances.
- *InteractionStateTransition Set* is the set of names of **InteractionStateTransition** instances.

A transition diagram is considered ***safe*** with respect to an event space E and a set of unsafe world-states U, if there does not exist a sequence of events E_s that results in one of the unsafe world-states. Figure 6 graphically shows an example of an ***unsafe*** transition diagram that reaches an unsafe world-state.

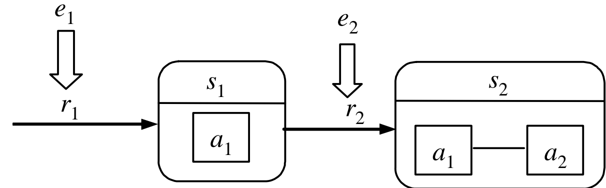


Figure 6: Example of unsafe transition diagram

In this example, p_1 is a parameter of component a_1 and p_2 is a parameter of component a_2. This diagram has four interaction-states including initial state s_0. In each state, the local time variable t is from 0 to some ending time t_e. In state s_1, we have

$s_1::a_1::p_1(t)=s_1::a_1::p_1(t=0) + 1$.

In state s_2, we have

$s_2::a_1::p_1(t)=s_2::a_1::p_1(t=0) + 2t$, $s_2::a_2::p_2(t)=s_2::a_1::p_1(t) + 1$.

As per the definition of the event space, the parameter p_1 can take value between 0 and 10. The unsafe state is and the unsafe parameter is define as $\{(p_1 = 4), (p_2 = 5)\}$. The transition condition from state s_1 to s_2 is defined using expression $(a_1::p_1(t)=3)$. Thus, when an event $(a_1::p_1(t=0):=2)$ happens, it will result in unsafe parameter value set u, which happens in s_2, when t=0.5.

We define a transition diagram as ***valid*** when the following conditions are met:

- Every state in the transition diagram is consistent.
- Every transition in the transition diagram is realizable.
- Given a valid transition diagram and an event space, we can simulate how the transition diagram responds to different events in the event space.

5. ELABORATION OPERATORS

As described in Section 3, in our framework, an initial transition diagram, which represents the specifications of observable behaviors of a device, is constructed first. After that, the conceptual design is performed by elaborating the initial transition diagram and by creating the internal structures of the mechatronic device being designed. The following operators are used for this purpose.

- **Decompose Component**. This operator is called *DECOMPOSE-COMPONENT* and used to decompose a component into a set of components. This operator is defined as the following.
  - Input: component *a*, transition diagram *D* in which *a* exists.
  - Output: a set of components *A*, the component mapping *M* between *a* and *A*, and the new transition diagram *D′* after *a* is decomposed.
  - Action: decompose a into A by establishing a component mapping between a and A. Replace a in D, which leads to D′. The component interactions involving a in D will be converted to component interactions involving A.

Component decomposition should define the parent component's parameters in terms of its children and preserve all interactions present at the parent.

- Decompose State: This operator is called ***DECOMPOSE-STATE*** and used to decompose an interaction-state into several sub-states and state transitions among these sub-states. This operator is defined as the following.
  - Input: state s and a transition diagram D that contains s.
  - Output: new state set S, new state transition set R and a new transition diagram D′.
  - Action: Replace the original state by a new state set and a new state transition set. Redirect transitions that involve the original state to the decomposed state.
- **Decompose Transition**: This operator is called ***DECOMPOSE-TRANSITION*** and used to decompose an interaction-state transition into several states and state transitions among these states. This operator is defined as the following.
  - Input: state transition r.
  - Output: new state set S and new state transition set R.
  - Action: Replace the original state transition by a new state set and a new state transition set. In other words, this operator substitutes a state transition with a new transition diagram.

The transition diagram generated as a result of applying the elaboration operators described above will not violate the behavioral requirements represented in the initial transition diagram and hence it is referred as an ***elaboration*** of the initial behavior specification.

The conceptual design step results in an ordered set (D_i, D_f, *E*, *U*), in which D_i is the initial behavior specification, D_f is the fully elaborated transition diagram, *E* is the event space, *U* is the set of unsafe world-states with respect to *E*. The conceptual design solution is considered ***valid*** if it meets the following conditions:

- D_i and D_f are valid transition diagrams.
- D_i and D_f are safe with respect to *E* and *U*.
- Every component interaction in every state of D_f can be expressed in terms of parameter interactions. For every parameter interaction and state transition, there exists a known working principle.
- D_f is an elaboration of D_i.

6. SIMULATING TRANSITION DIAGRAMS

Transition diagrams either at the beginning of the conceptual design or at the end of the conceptual design can be simulated. This allows designers to understand how the conceptual design solution and/or behavior specifications respond to events in the use environment.

Because of space restrictions, we have omitted the detailed description of the simulation algorithm. But the idea behind simulation algorithm is as follows. First, we generate a sequence of events. This event sequence serves as an input to the simulation algorithm. We initialize the current state with the initial state. Current state is updated as the simulation proceeds.

For the current state, we initialize the state parameters using the appropriate values. If necessary, these values are inherited from the previous state. Then, all the equations that are applicable in this state are collected by considering all the constraints and interactions relevant for this state. Then, we compute the exit time for this state by considering the outgoing transitions for this state. If an event in the event sequence can trigger an outgoing transition, the exit time is determined. However, before exiting the current state, we check to determine if parameters take unsafe values in this state. If unsafe values are encountered, then the algorithm reports those values. Checking of unsafe parameters can be done using either (1) discretising the time and computing state parameters values at these discrete time values, or (2) by using variable substitution techniques and computing the time when the state parameters take unsafe values. Our simulation infrastructure first tries the second method, if it fails then it uses the first methods.

After processing of the current state has been completed, we transit to the next state based on the outgoing transition. The next state is marked as the current state and the steps in the previous paragraph are repeated. After we exhaust the event sequence, the simulation stops.

7. EXAMPLE OF MODELING AUTONOMOUS VACUUM CLEANER

This section describes application of the framework presented in this paper to the design of an autonomous vacuum cleaner (*AVC*). The design task is to develop a device that is able to collect the debris on a surface while avoiding collision from

obstacles on the surface. The requirements are: 1) AVC cleans Surface; and 2) AVC avoids Obstacles. Use-environment components include: Surface, Interface, Obstacles and Power source. The two steps described in Section 3 are carried out in the following manner:

1. **Define Behavior Specifications**: Parameters that are used to define behavior specifications are shown in Table 2. For example, *AVC* stores the debris thus it has a remaining capacity parameter. The possible interactions between *AVC* and its use-environment are summarized into the event space shown in Table 3. Unsafe parameter value sets are described in Table 4. From the requirements, the primary working modes of *AVC* (e.g., interaction-states) are also identified. Figure 7 shows proposed behavior specifications for *AVC*. Detailed descriptions for each interaction-state can be found in [Xu05]. This behavior specification was simulated using a synthetic room with a large obstacle in the middle and a predefined zigzag vacuuming path and was found to be safe.

Table 2: Components and parameters used in AVC behavior specification

Component	Parameter	Type	Convention
AVC	*Speed*	*REAL*	
	RemainingCapacity	*REAL*	0 to 100%
	RemainingEnergy	*REAL*	0 to 100%
	Power	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*
	InputVoltage	*REAL*	
	ObstacleInContact	*BOOLEAN*	*TRUE/FALSE*
Surface	*AreaCovered*	*REAL*	
	LocationVisited	*BOOLEAN*	*TRUE/FALSE*
	MovePossible	*BOOLEAN*	TRUE/FALSE
Power Source	*VoltageOutput*	*REAL*	
Obstacle	*AVCInContact*	*BOOLEAN*	*TRUE/FALSE*
Interface	*Power*	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*

2. **Elaborate Transition Diagram**: Since there is no known component that can fulfill the behavior specification of *AVC* directly, we need to decompose *AVC* into components that can be realized. Starting points of the decomposition are the component interactions between *AVC* and use-environment components. The operator *DECOMPOSE-COMPONENT* is applied to replace the *AVC* in behavior specification with its major components shown in Table 5. *AVC*'s parameters are mapped to the parameters of its children components. In this example the major parameters of *AVC* are directly mapped to one parameter of one child component respectively. Furthermore, the "Waiting" state also needs to be decomposed using operator *DECOMPOSE-STATE*. It is decomposed into "Waiting", "Recharge" and "Empty" states. The corresponding transitions are also redirected and decomposed. A detailed interaction-state transition diagram for conceptual design is shown in Figure 8. Detailed descriptions for each interaction-state can be found in [Xu05].

Table 3: Event space used in AVC behavioral specification

Parameter	Value
Surface::*LocationVisited*	{*TRUE*, *FALSE*}
Surface::*MovePossible*	{*TRUE*, *FALSE*}
Interface::*Power*	{*ON*, *OFF*}
Interface::*PauseStatus*	{*ON*, *OFF*}
Obstacle::*AVCInContact*	{*TRUE*, *FALSE*}

Table 4: Unsafe state used in AVC behavioral specification

AVC::*RemainingEnergy* ≤ 10%
AVC::*RemainingCapacity* ≤ 2%

Table 5: Decomposed components and parameters of AVC

Component	Parameter	Type	Convention
Transporter	*Speed*	*REAL*	
	EnergyInput	*REAL*	
Vacuum	*Remaining-Capacity*	*REAL*	0 to 100%
Controller	*Power*	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*
	ObstacleIn-Contact	*BOOLEAN*	*TRUE/FALSE*
Battery	*InputVoltage*	*REAL*	
	Remaining-Energy	*REAL*	0 to 100%
PathPlanning -Alg. (PPA)	*SpeedOutput*	*REAL*	

8. CONCLUSIONS

This paper describes the class definitions of primitives and elaboration operators needed to support the conceptual design of multiple interaction state devices. We use state transition diagrams as the underlying modeling construct in our framework to capture different working modes of the device. We also provide rationale for each primitive and its role in the conceptual design. The distinction between our approach and traditional functional representation approaches for conceptual design is as follows:

- We use interactions instead of function flows or input/output flows to describe relationships between components. Interactions are more general than flows. In addition to capturing flows, they can also be used to capture non-flow-based relationships such as spatial

constraints. Therefore, our approach is more expressive.

- We use interaction-states to capture the operating modes of a device. Hence we can support devices with multiple interaction-states (i.e., devices whose interactions with the use environment change with time). Therefore, models created using our framework can be simulated more accurately. For example, events can be used to simulate the behavior of a proposed design solution in response to events in the use-environment.
- We believe this new modeling framework will have the following benefits. First, it provides computer interpretable representation schemes for supporting conceptual design of mechatronic systems. Hence it provides an improved support for design information archival and reuse. Second, it provides the foundation for the development of computer aided evaluation and synthesis support during conceptual design.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support provided by NIST's Manufacturing System Integration Division. Any commercial products or company names in this paper are given for informational purposes only. Their use does not imply recommendation or endorsement by the National Institute of Standards and Technology or the University of Maryland.

REFERENCES

[Booc98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Press, 1998.

[Bohm04] M.R. Bohm and R.B. Stone. Representing functionality to support reuse: conceptual and supporting functions. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.

[Broo04] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng (eds.). HyVisual: A Hybrid System Visual Modeler. *Technical Memorandum UCB/ERL M04/18*, June 28, 2004, University of California, Berkeley, CA 94720.

[Chan94] B. Chandrasekaran. Functional representations: a brief historical perspective. *Applied artificial intelligence*, special issue on functional reasoning, 8(2):173-197, 1994.

[Chen02] L. Chen, M. Jayaram and J.F. Xi. A new functional representation scheme for conceptual modeling of mechatronic systems. In Proceedings of the *ASME Design Engineering Technical Conferences*, Montreal, Canada, September 2002.

[Deng99] Y.M. Deng, S.B. Tor and G.A. Britton. A computerized design environment for functional modeling of mechanical products. In Proceedings of the *Fifth ACM Symposium on Solid Modeling,* Ann Arbor, Michigan, USA, 1999.

[Erde03] Z. Erden, A. Erden and A.M. Erkmen. Petri net approach to behavioral simulation of design artifacts with application to mechatronic design. *Research in Engineering Design*, 14(1):34-46 , February 2003.

[Fenv01] S.J. Fenves. A core product model for representing design information. Technical Report, Number NISTIR6736, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2001.

[Gaus01] J. Gausemeier, M. Flath and S. Mohringer. Conceptual design of mechatronic systems supported by semi-formal specification. In Proceedings of the *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, v 2, 2001, p 888-892

[Grab99] H. Grabowski, S. Rude and M. Huang. Supporting early phase of mechatronic product design with layered function models. In Proceedings of the *IEEE International Symposium on Industrial Electronics*, v 2, 1999, p 914-918

[Hirt01] J. Hirtz, R. Stone, D. McAdams, S. Szykman, and K. Wood. A functional basis for engineering design: reconciling and evolving previous efforts. *Research In Engineering Design,* 13(2):65-82, March 2002.

[Iwas95] Y. Iwasaki, M. Vescovi, R. Fikes, and B. Chandrasekaran. Casual functional representation language with behavior-based semantics. *Applied Artificial Intelligence*, 9:5-31.

[Jaya03] M. Jayaram and L. Chen. Functional modeling of complex mechatronic systems. In Proceedings of the *ASME Design Engineering Technical Conferences*, Chicago, Illinois, USA, September 2003.

[Pahl96] G. Pahl and W. Beitz. *Engineering Design: A Systematic Approach*. Springer-Verlag, 1996.

[Sasa96] M. Sasajima, Y. Kitamura, M. Ikeda, and M. Mizoguchi. Representation language for behavior and function: FBRL. *Expert Systems with Applications*, 10(3/4):471-479, 1996.

[Shoo00] S.B. Shooter, W.T. Keirouz, S. Szykman and S. Fenves. A model for information flow in design. In Proceedings of the *ASME Design Engineering Technical Conferences*, Baltimore, Maryland, USA, September 2000.

[Ston00] R.B. Stone and K.L. Wood. Development of a functional basis for design. *Journal of Mechanical Design*, 122(4):359-370, December 2000.

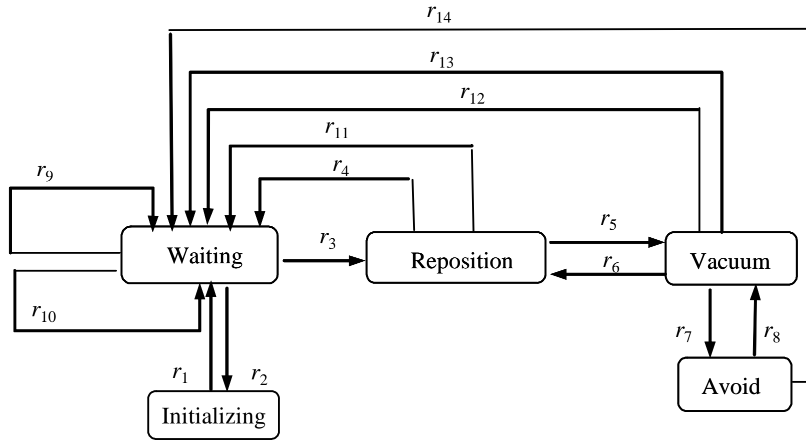
[Umed96] Y. Umeda et al. Supporting conceptual design based on the function-behavior-state modeler. *AIEDAM*, 10(4):275–288, September 1996.

[Varg04] N. Vargas-Hernandez and J.J. Shah. 2nd-CAD: a tool for conceptual systems design in electromechanical domain. *Journal of Computing and Information Science in Engineering*, 4(3):28-36, 2004.

[Will92] B.C. Williams. Interaction-based design: constructing novel devices from first principles. In *Intelligent Computer Aided Design*, edited by D.C. Brown, M. Waldron and H. Yoshikawa, pages 255-274, Elsevier Science Publishers, 1992.

[Xu05] C. Xu. *Computational Foundations For Computer Aided Conceptual Design Of Multiple Interaction-State Mechatronic Devices*. Ph.D. Dissertation, University of Maryland, May 2005.

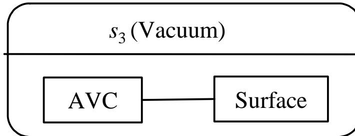
[Zein04] A. Zeiny. Computable dynamic design repository for product data representation. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.



Transition list

Name	Condition	Name	Condition
r_1	*Interface*::*Power* = *ON*	r_8	*Obstacle*:: *AVCInContact* = *FALSE*
r_2	*Interface* ::*Power* = *OFF*	r_9	*AVC*::*RemainingCapacity* ≤ 2%
r_3	*Interface*::*PauseStatus* = *OFF*	r_{10}	*AVC*::*RemainingEnergy* ≤ 10%
r_4	*Surface*::*MovePossible* = *FALSE*	r_{11}	*AVC*::*RemainingEnergy* ≤ 10%
r_5	*Surface*::*LocationVisited* = *FALSE*	r_{12}	*AVC*::*RemainingEnergy* ≤ 10%
r_6	*Surface*::*LocationVisited* = *TRUE*	r_{13}	*AVC*::*RemainingCapacity* ≤ 2%
r_7	*Obstacle*:: *AVCInContact* = *TRUE*	r_{14}	*AVC*::*RemainingEnergy* ≤ 10%

Detailed description of Vacuum state in the above transition diagram



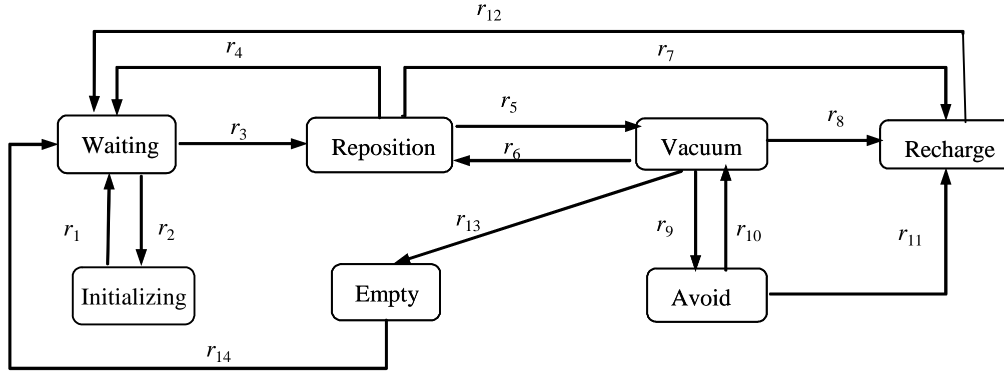
Component Interaction Equations

AVC::*RemainingCapacity*(*t*) =
AVC::*RemainingCapacity*(*t*=0) – *Surface*::*AreaCovered* / 20

Parameters Initialization and Change

Parameter	Initialization Type	Initialization Value	Change Type	Equation
AVC::*Speed*	*ASSIGN*	0.05m/s	*CONSTANT*	*NONE*
AVC::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*NONE*
AVC::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*AVC*::*RemainingEnergy*(*t*) = *AVC*::*RemainingEnergy*(*t*=0) – *AVC*::*Speed* × *t* / 400
AVC::*Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*PauseStatus*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*ObstacleInContact*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*

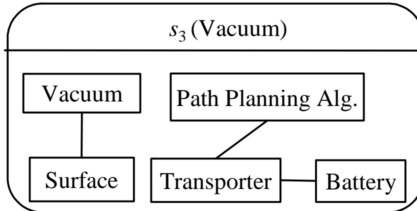
Figure 7: *AVC* behavior specification



Transition list

Name	Condition	Name	Condition
r_1	*Interface*::*Power = ON*	r_8	*Battery*::*RemainingEnergy* ≤ 10%
r_2	*Interface*::*Power = OFF*	r_9	*Obstacle*::*AVCInContact = TRUE*
r_3	*Interface*::*PauseStatus = OFF*	r_{10}	*Obstacle*::*AVCInContact = FALSE*
r_4	*Surface*::*MovePossible = FALSE*	r_{11}	*Battery*::*RemainingEnergy* ≤ 10%
r_5	*Surface*::*LocationVisited = FALSE*	r_{12}	*Battery*::*RemainingEnergy* = 100%
r_6	*Surface*::*LocationVisited = TRUE*	r_{13}	*Vacuum*::*RemainingCapacity* ≤ 2%
r_7	*Battery*::*RemainingEnergy* ≤ 10%	r_{14}	*Vacuum*::*RemainingCapacity* = 100%

Detailed description of Vacuum state in the above transition diagram



Component Interaction Equations

Transporter::*Speed* = *PPA*::*SpeedOutput*
Transporter::*EnergyInput* = *Battery*::*RemainingEnergy*
Vacuum::*RemainingCapacity*(*t*) =
Vacuum::*RemainingCapacity*(*t*=0) − *Surface*::*AreaCovered* / 20
Battery::*RemainingEnergy*(*t*) = *Battery*(*t*)::*RemainingEnergy*(*t*=0)
− *Transporter*::*Speed* × *t* / 400

Parameters Initialization and Change

Parameter	Initialization Type	Initialization Value	Change Type	Equation
Transporter::*Speed*	*DERIVE*	*NA*	*DERIVE*	*NONE*
Transporter::*EnergyInput*	*DERIVE*	*NA*	*DERIVE*	*NONE*
PPA::*SpeedOutput*	*ASSIGN*	0.01m/s	*CONSTANT*	*NONE*
Battery::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*NONE*
Vacuum::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*NONE*

Figure 8: Detailed transition diagram for *AVC*