World Academy of Science, Engineering and Technology
International Journal of Electrical and Computer Engineering
Vol:3, No:5, 2009

Multi-models approach for describing and verifying constraints based interactive systems

Mamoun Sqali and Mohamed Wassim Trojet

***Abstract*—** The requirements analysis, modeling, and simulation have consistently been one of the main challenges during the development of complex systems. The scenarios and the state machines are two successful models to describe the behavior of an interactive system. The scenarios represent examples of system execution in the form of sequences of messages exchanged between objects and are a partial view of the system. In contrast, state machines can represent the overall system behavior. The automation of processing scenarios in the state machines provide some answers to various problems such as system behavior validation and scenarios consistency checking. In this paper, we propose a method for translating scenarios in state machines represented by Discreet EVent Specification and procedure to detect implied scenarios. Each induced DEVS model represents the behavior of an object of the system. The global system behavior is described by coupling the atomic DEVS models and validated through simulation. We improve the validation process with integrating formal methods to eliminate logical inconsistencies in the global model. For that end, we use the Z notation.

***Keywords*—** Scenarios, DEVS, Synthesis, Validation and Verification, Simulation, Formal Verification, Z Notation.

I. Introduction

A typical development of an interactive system begins with writing scenarios which describe the most important behaviors. They are gradually enriched, specified and composed until describing all the behaviors of the system. A scenario visually describes by means of a sequence diagram, the interaction protocol between objects and the environment. In contrast, a state machine has the vocation to represent the entire behavior of a system and it is hard to be conceived. Moreover, designing the system behavior directly with state-based models is not an intuitive process, since the concept of state is not obvious in the first stages of the development process. The partial character of scenarios makes them easier to be conceptualized. Which why, working in parallel with the requirements of a system expressed in the form of scenarios, and its specification provided by the state machines improves the level and quality of specification. A lot of software engineering approaches synthesize state-based models from scenario-based models with the intent to make the task of describing the dynamic behavior of interactive systems easier [7]. This transformation from scenarios to state machines consists in checking the consistency of the various scenarios and inducing a global behavior for the system from the partial behaviors given in the scenarios. Many problems can arise during synthesis as deadlock or the parallelism which is caused by competition between the events, appearance of the implicit scenarios and other problems of composition which make difficult to apprehend the global behavior of the system.

This article proposes to induce from a set of scenarios expressed in the form of Message Sequence Charts [1], a DEVS [6] model representing the overall behavior of the system. We propose procedures for such transformation. Normally, the obtained simulation models must produce the same sequence of events for the input sequences in the scenarios. Therefore, we use simulation techniques and formal verification (absence of conflicts and incoherencies in system properties) with Z language [15] to ensure the consistency of scenarios. In fact, once the system is modeled with scenarios, our approach automatically generates an equivalent DEVS model. The latter is also automatically transformed to a Z specification.

We present in the following sections, the scenario notation, the Discrete Event Specification (DEVS) formalism, the Z language, the synthesis procedure and an example to illustrate our case study.

II. Recalls

A. Scenarios

The scenarios are effective means to obtain and to validate the requirements. They became the most popular ways to describe systems behaviors. They describe how the components of a system, the environment and the users, work simultaneously and act between them to provide the level of functionality of the system. In particular, they are used at the first phase of the software development that we call requirements engineering, but can appear too in later phases like the validation or maintenance. They can be composed by using flow control operators (alternative, sequence, parallel composition and repetition) in order to form more complex scenarios.

A great number of notations are commonly used for the description of scenarios, like: Message Sequence Charts (MSC) defined within an international standard [1], Live Sequence Charts (LSC) proposed by [2], the UML SD [3], which are a simplified version of basic MSC [4]… All of them are based on a textual and graphical representation. We have chosen Message Sequence Chart to illustrate our approach and represent the requirements of our systems because it is a formal language of which graphical notation is easily understood, and it can be hierarchically composed by using hMSC (hierarchical Message Sequence Chart) in order to form more complex scenarios.

The Message Sequence Charts are composed by hierarchical MSC's (hMSC) and basic MSC's (bMSC). A basic MSC has a structure: (E, A, L, O, ϕ, $\leq$, traj) where:

- E: is a finite set of events divided into a set of sent events SE, and a set of received events RE;
- A: is finite set of actions;
- L: is a finite set of labels;
- O: is a finite set of objects;

- $\leq$: is a partial order relation (antisymmetric, reflexive and transitive) called causal order on events;
  - $\forall(e1) \in E \Rightarrow e1 \leq e1$ (reflexive);
  - $\forall(e1, e2) \in E^2, (e1 \leq e2) \wedge (e2 \leq e1) \Rightarrow e1 = e2$ (antisymmetric);
  - $\forall(e1, e2) \in E^3, (e1 \leq e2) \wedge (e2 \leq e3) \Rightarrow e1 \leq e3$ (transitive);
- ϕ: E $\rightarrow$ O associates an event to an object. Moreover, events belonging to the same object are totally ordered;

$$\forall(e1,e2)\in E^2, \phi(e1)=\phi(e2) \Rightarrow (e1 \leq e2) \vee (e2 \leq e1)$$

- traj: S $\rightarrow$ R is a function which represents the trajectory of the events. This function associates the sending of an event with its reception.

The behavior represented by the bMSC is a set of sequences of events determined by the causal priority. This causal relationship determines a partial order, noted $\leq$, on the events between all objects. The partial order can be derived from the bMSC in respect with two principal rules:

- An event *e* drawn higher than another event *e'* on the same lifeline of an object precedes necessarily *e'*;
- The event associated with a message sending precedes necessarily the event associated with the reception of this message (in the case of an asynchronous communication). For a synchronous communication, the events sending and reception for each message are used to be considered instantaneous.

We will denote by *em(e)* the sending event corresponding to the receiving event *e* and *rec(e)* the reception event corresponding to the sending event *e*. we use label *send(i, j, m)* to denote the event " object *i* sends the message *m* to object *j*" and similarly, *receive(i, j, m)* to denote the event " object *i* receives the message *m* from object *j*". We will often note *!m* the sending event, and *?m* the receiving event for a message *m*.

The hierarchical MSC's were conceived to allow the creation of more complex scenarios [1]. A high-level MSC (hMSC) provides the means for composing bMSCs: it is a digraph where nodes are bMSC's and arcs indicate their possible continuations. It has a special initial and final node that corresponds to the initial and final system states. An execution of an hMSC is obtained by traversing the way starting from initial node to the final one.

An hMSC has a structure: (N, A, S0) where:

- N is a finite set of bMSCs with disjoint sets of events;
- A $\subseteq$ (N$\times$ N) is a set of arcs;
- S0 $\in$ N is the initial node.

B. *The DEVS formalism*

The DEVS formalism introduced by [5] provides a means for modeling discrete event system in a hierarchical and modular way. DEVS is a general formalism for discrete event system modeling based on set theory [6]. It allows representing any system by three sets and four functions: Input Set, Output Set, State Set, External Transition Function, Internal Transition Function, Output Function, and Time Advanced Function. DEVS formalism provides the framework for information modeling which gives several advantages to analyze and design complex systems: Completeness, Verifiability, Extensibility, and Maintainability. DEVS has two kinds of models to represent systems. One is an Atomic Model (AM) and the other is a Coupled Model (CM) which can specify complex systems in a hierarchical way [6]. DEVS model processes an input event based on its state and condition, and it generates an output event and changes its state. Finally, it sets the time during which the model can stay in that state.

1) *Atomic model*

An atomic DEVS model describes the behavior of a component, which is indivisible, in a timed state transition level. It is represented by one box comprising inputs and outputs; it allows a system to be described like a set of deterministic transitions between sequential states (Fig. 1). Each transition is labeled by a sending or reception event.

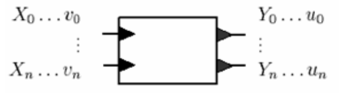


Fig.1 Representation of an atomic DEVS model

Formally, an atomic model is defined by a 7-tuple <X, Y, S, δint, δext, λ, ta> where:

- X is the set of input values;
- Y is the set of output values;
- S is the set of sequential states;
- δint : S $\rightarrow$ S is the internal transition function that defines the state changes caused by internal events;
- δext : Q $\times$ X $\rightarrow$ S is the external transition function, where Q = {(s,e)|s$\in$ S, 0$\leq$e$\leq$ta(s)} is the set of total state; this function specifies the state changes due to external events, with the ability to define a future state according to the elapsed time in the current state;
- λ: S $\rightarrow$ Y is the output function that generates output events;
- ta: S $\rightarrow R^+_{0,\infty}$ gives the lifetime of the states, where $R^+_{0,\infty}$ is the set of positive real numbers between 0 and ∞.

The behaviors of the atomic model are as follows: An atomic model can stay only in one state at any time. The maximum time to stay in one state without external event is determined by *ta(s)* function, it changes its state by *δext* if it gets an external event. If possible remaining time in one state is elapsed, it generates output by λ and changes the state by *δint*. In general, while the internal transition function *δint* expresses the autonomous evolution of the model, the external transition function *δext* defines its evolution when occurring external events.

2) *Coupled model*

The coupled DEVS model is constructed by coupling atomic and/or coupled models. Output events of one model are connected with input events of another. The resulting coupled model can itself be employed as a component in a larger coupled model, by giving rise to the construction of complex models with hierarchical structures (Fig. 2).

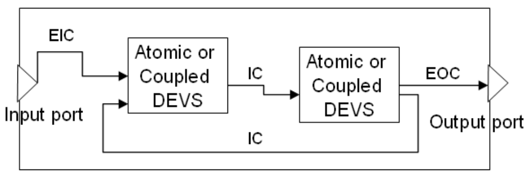


Fig. 2 Representation of a coupled DEVS model

A coupled model is formally defined by a 7-tuple < X, Y, M, EIC, EOC, IC, SELECT > where:

- X is the set of input events;
- Y is the set of output events;
- M is the set of all the DEVS component models;
- EIC $\subseteq$ X $\times$ UiXi is the external input coupling relation;
- EOC $\subseteq$ UiYi $\times$ Y is the external output coupling relation;
- IC $\subseteq$ UiXi $\times$ UiYi is the internal coupling relation;
- SELECT: 2M - ϕ, M is a function which chooses one model when more than 2 models are scheduled simultaneously.

EIC, EOC and IC specify the connections between the input and output ports of the various DEVS models.

C. The Z Specification Language

Z is a formal state-based specification [14] [15]. It is based on predicates logic and set theory. A main ingredient in Z is the way of decomposing a specification into small pieces called schemas. Schemas are used to describe both static and dynamic aspects of a system. The notation of the schema is the following:

Schema Name

declarations (state space)

predicates

1-Declaration of types used into the specification (free type definition).

2- A schema describing the global abstract state of the system:

Abstract_State_Name

declarations of the variables describing the state of the system

predicates (State invariants)

3- A schema describing the initial state of the system:

Initializing_System

Abstract_State_ Name

initialization of states variables

4- List of operations schemas and each one describes the state before and after the operation execution:

Operation Name

Δ*system name(*Δ *: to say that the state of the system is changed)* ***OR*** Ξ *system name (*Ξ *: to say that the state of the system is the same)*

eventual declaration of input variables (? has to be placed after input variable)

eventual declaration of output variables(! has to be placed after output variable)

pre-operation (values of the state variables just before the operation)

post-operation(values of the state variables just after the operation)

eventual values of input variables

eventual values of output variables

5- Treatment of errors that can appear when executing operations.

OperationError

Δ*system name OR* Ξ *system name*

eventual declaration of input variables

eventual declaration of output variables

pre-operation (may be that the operation hasn't to be executed after this pre operation)

post-operation (may be after execution of the operation , the post operation is not available)

eventual value of the input (may be it doesn't satisfy a constraint)

eventual value of the output (can inform that there is an error)

1) Proof Obligation in Z

In traditional Z-based specification methodologies, designers must conduct a set of formal proofs to verify incrementally the consistency of the system being modeled [16] [17]. In state/transition approaches like Z-based model this mostly consists in (1) initialization theorems to ensure that initializations preserve state invariants and (2) pre-condition calculations to enforce the consistency of the operations modifying the state space. Establishing the list of all preconditions ensures that either the state invariant is completely preserved by operation effects or that some other condition must be fulfilled.

III. THE EXISTING WORKS

The automatic synthesis of conceptions starting from scenarios was a very active field of research during the last years. Many approaches address the scenario synthesis problem and makes possible to induce a total behavior model expressed in a state machine format starting from a set of scenarios [7]. There are two kinds of synthesis: the construction of a global state machine which represents the total behavior of a system directly starting from a set of scenarios, with or without composition mechanism. And the construction of a state machine by object for all the scenarios whose behavior of the system is defined like the parallel composition of all the obtained states machines and which synchronizes on the shared messages. Harel [8] proposed a synthesis approach using the scenario-based language of Live Sequence Charts (LSC) as requirements, and synthesizing a state-based object system composed by a collection of finite state machines. Letier [9] presents a technique to generate Labeled Transition System (LTS) from High Level Message Sequence Chart (hMSC), in this approach; complex system behavior can be modeled by parallel composition of the component LTS models. The

LTS obtained are executed asynchronously but synchronize on shared events; also they present a technique to detect implicit scenarios. Ziadi [10] propose an idea to synthesize statecharts starting from scenarios expressed by UML2.0 Sequence Diagrams, and give an algorithm for synthesizing a composition of statecharts between them. Also, Damas [11] has presented an approach to generate Labeled Transition System from a collection of basics MSC's, and use a technique to merge the identical states. The synthesis approaches differ depending on:

- The choice of the scenarios language;
- Their semantic interpretation;
- The type of target state machines;
- The complexity of the synthesis algorithm implemented;
- If they use or not the techniques of reunification of the identical states.

IV. SYNTHESISING DEVS MODELS FROM SCENARIOS

In this section we discuss a general procedure for deriving DEVS component descriptions from a set of MSC's scenarios. To that end, we give an overview of our translation schema in section A, and present an example of application in section B.

A. *Roadmap for the translation procedure*

The systems we are interested in consist of a set of components and they are described by a set of scenarios expressed in the form of messages sequences charts. We assume given a set of MSC's that describe all the interaction sequences among a set of components named objects. We assume further that we try to obtain an atomic DEVS for exactly one of the objects, say O, occurring in the MSC's diagrams.

The procedure for obtaining that automaton consists of seven successive phases: verification, projection, normalization, transformation into atomic DEVS models, merging all atomic models obtained for each object in one global atomic model, optimization and obtain a global coupled DEVS.

- Verification: This phase consists in checking that the set of the behaviors described by each MSC is a sequence set of events respecting the causal priority. The events associated with one object are totally ordered.
- Projection: During the second phase, we project each of the given MSC's onto the object "O", i.e. we remove all other instance axes, as well as message arrows that neither start, nor end at O. If we use hierarchical MSC, we project each basic MSC onto object "O" by traversing the way starting from initial node to the final node with respecting sequence between basic components.
- Normalization: We identify the events which will make possible to determine the initial and final states of the atomic DEVS models corresponding to "O".
- Transformation into an atomic DEVS model: This phase consists in translating reception events of the object into external transitions in DEVS models and sent events into internal transitions. In the definition of the external transition function, p?v notes the value v of the input event occurring on the input port p of the atomic model (Fig. 3). In the definition of the output function, p!v notes the value v of the output event to be generated on the output port p. If there are actions, we use states variables in DEVS model, and if there are conditions, we add conditions in the equivalents states transitions.

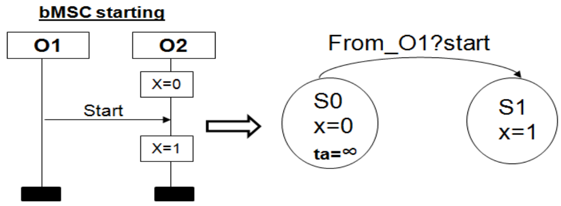


Fig. 3 Projection of starting bMSC onto object O2

In this phase of synthesis, the number of the atomic DEVS models for each object must be equal to the number of the bMSCs.

- Merging all atomic models obtained in one global atomic model: for each object, we merge all resulting atomic models associated to each bMSC, in one global atomic model that represents the global behavior of the object in the system. To that end, we traverse the way of the hMSC starting from the initial bMSC towards the final one with using the following steps. We use scenario semantics restricted to event sequences with the notion of (iteration, alternative and sequence):

  - Seq: Specify a sequence between the behaviors of two operand bMSC (strong sequential composition).

Let Da1 = $<X1, Y1, S1, \delta int1, \delta ext1, \lambda 1, ta1>$ and Da2= $< X1, Y1, S1, \delta int1, \delta ext1, \lambda 1, ta1>$

Da1 seq Da2 = $<X, Y, S, \delta int, \delta ext, \lambda, ta>$ where:

- $S = (S1 \cup S2) - \{s02\}$ if $(Da2 \neq Da\varnothing)$
  - S2 if $(Da1 = Da\varnothing)$
  - S1 otherwise
  - $s0 = s01$ if $(Da1 \neq Da\varnothing)$
    $= s02$ otherwise
- $X = X1 \cup X2$
- $Y = Y1 \cup Y2$
- $\delta int = \delta int1 \cup \delta int1$
- $\delta ext = \delta ext1 \cup \delta ext2$
- $ta = ta1 \cup ta2$

  - Loop : Specify an iteration of an interaction

Let Da1 = $<X1, Y1, S1, \delta int1, \delta ext1, \lambda 1, ta1>$

loop (Da1) = $<X, Y, S, \delta int, \delta ext, \lambda, ta>$ where :

- $S = - (S1-sn1) \cup \{s01\}$
  - $s0=s01$
- $X=X1$
- $Y=Y1$
- $\delta int = \delta int1$
- $\delta ext = \delta ext1$
- $\lambda=\lambda 1$
- $ta=ta1$
- Loop $(Da\varnothing) = Da\varnothing$

  - Alt: Define a choice between a set of interaction operands:

Let Da1 = $<X1, Y1, S1, \delta int1, \delta ext1, \lambda 1, ta1>$ and Da2= $< X1, Y1, S1, \delta int1, \delta ext1, \lambda 1, ta1>$

Da1 alt Da2 = <X, Y, S, δint, δext,λ, ta> where

- $S =$ - S1 if $(Da1 \neq Da\varnothing \wedge Da2 = Da\varnothing)$
  - S2 if $(Da1 = Da\varnothing \wedge Da2 \neq Da\varnothing)$
  - {s0} if $(Da1 = \varnothing \wedge Da2 = \varnothing)$
  - $S1 \cup S2$ - {s02} if $(Da1 \neq Da\varnothing \wedge Da2 \neq Da\varnothing)$
- s0 =A new state if $(Da1 = Da\varnothing \wedge Da2 = Da\varnothing)$
  - s01 if $(Da1 \neq Da\varnothing \wedge Da2 = Da\varnothing)$
  - s02 otherwise
- $X = X1 \cup X2$
- $Y = Y1 \cup Y2$
- $\delta int = \delta int1 \cup \delta int1$
- $\delta ext = \delta ext1 \cup \delta ext2$
- $ta = ta1 \cup ta2$

- Optimization: To optimize the resulting global atomic DEVS we use standard algorithms of optimization from automata theory in order to make our models deterministic and have the minimum number of states and transitions. To that end, we merge states that receive, send the same events and have the same variables number and values:
  - Case external transition + internal transition: if δint(Si)=Sj / λ(Si)=pi!vi and δint(S'i)=S'j / λ(S'i)=pi!vi and δext(Sk,e,pk?vk)=Si and δext(S'k,e,pk?vk)=S'i then Si=S'i;
  - Case external transition + external transition: if δext(Si, e, pi?vi)=Sj and δext(S'i, e, pi?vi) =S'j and δext (Sk, e, pk?vk)=Si and δext(S'k, e, pk?vk)=S'i then Si=S'i;
  - Case internal transition + external transition: if δext(Si, e, pi?vi)=Sj and δext(S'i,e,pi?vi)=S'j and δint(Sk)= Si / λ (Sk)=pk!vk and δint(S'k)=S'i / λ(S'k)=pk!vk then Si=S'i;
  - Case internal transition + internal transition: if δint(Si)=Sj / λ(Si)=pi!vi and δint(S'i)=S'j / λ(S'i)=pi!vi and δint(Sk)=Si / λ(Sk)=pk!vk and δint(S'k)=S'i / λ(S'k)=pk!vk then Si=S'i.
- Generating the global coupled DEVS: in this final phase, the final coupled DEVS model can be obtained by coupling the various global atomic models for each object. In that end, if an object O1 sends an event to another object O2, we connect the output port of O1 with the input port of O2, and vice versa. The final coupled DEVS obtained describes the overall behavior of the system.

To illustrate our approach we have used scenario semantics restricted to event sequences with the notion of (repetition, alternative and sequence). The advantage of the use of coupled DEVS and not of the total state machines is on the first hand, to make possible to simulate and to validate the behavior of each object of the system. And in addition this type of transformation gives flexibility to the process of the synthesis. Indeed, any modification, addition or removal of an object in the system do not influence on the process of the synthesis. We have just to modify, add or remove the corresponding atomic DEVS model. The next section provides an example application of the procedure we have outlined here. An algorithm which translates a basic SD (Sequence Diagram) into an atomic DEVS model by object was presented in [12]. Also, the principle of the construction of an atomic DEVS model by object starting from several composed SD, and the construction of coupled DEVS model is described in [13].

B. *Case Study*

In the previous section, we proposed a method for translating a set of scenarios into state machines represented in the formal DEVS specification. To illustrate our approach, we use a hybrid car interactive based system.

A hybrid car has an engine that runs with fuel and a rechargeable battery. Hybrids are preferred because all-electric cars rarely get above speeds of 50-60 miles per hour (mph). They also need to be recharged between 50 and 100 miles. The battery system in hybrid cars is recharged from the car itself. Electrical hybrid engine can take the kinetic energy that comes from applying the brakes and charge the battery. The originality of this car is the presence of two engines, one run with fuel (thermal engine) and the other is electric. The assumptions linked to the hybrid car are the following:

- When the driver starts the car or the speed is lower than 50mph: Only the electrical engine is moving with rechargeable battery (Fig. 4). As long as the car runs, the battery loses energy. This system is necessary for low speeds. In this case, the thermal engine is completely inactive, no dioxide carbon is emitted. A great advantage for planet, and the pocket of the driver.

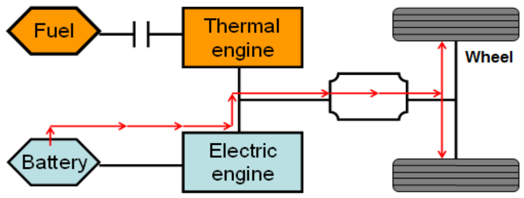


Fig. 4 When the driver starts the car or the speed is lower than 50mph

- When the speed is higher than 50mph: The electrical engine is in stand by. Only the thermal engine works. When speed exceeds the 100 mph, part of the driving energy provided by the fuel is used to reload the battery, via a generator (the electrical engine) (fig. 5). All is recycled, contrary to a traditional car.

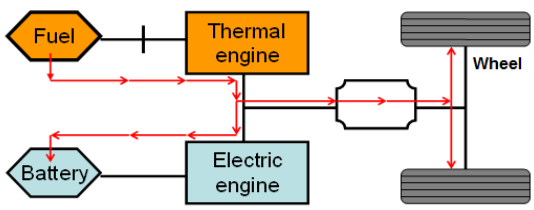


Fig. 5 When the speed is higher than 50mph

- The deceleration phases: When the driver brakes, thekinetic energy resulting from the movement of the vehicle, is directly sent towards the battery of the electrical engine (Fig. 6).

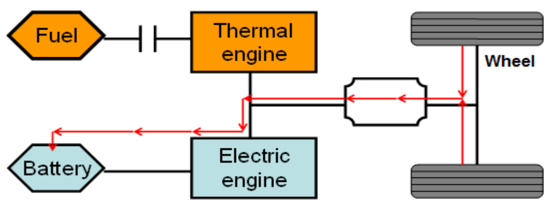


Fig. 6 Sending kinetic energy to the batteries

The behavior of this hybrid system is represented in the hMSC of Fig.7. This hMSC is composed by three objects: the Driver which is considered as a control device, the Electric and the Thermal engines that represent the operative system.

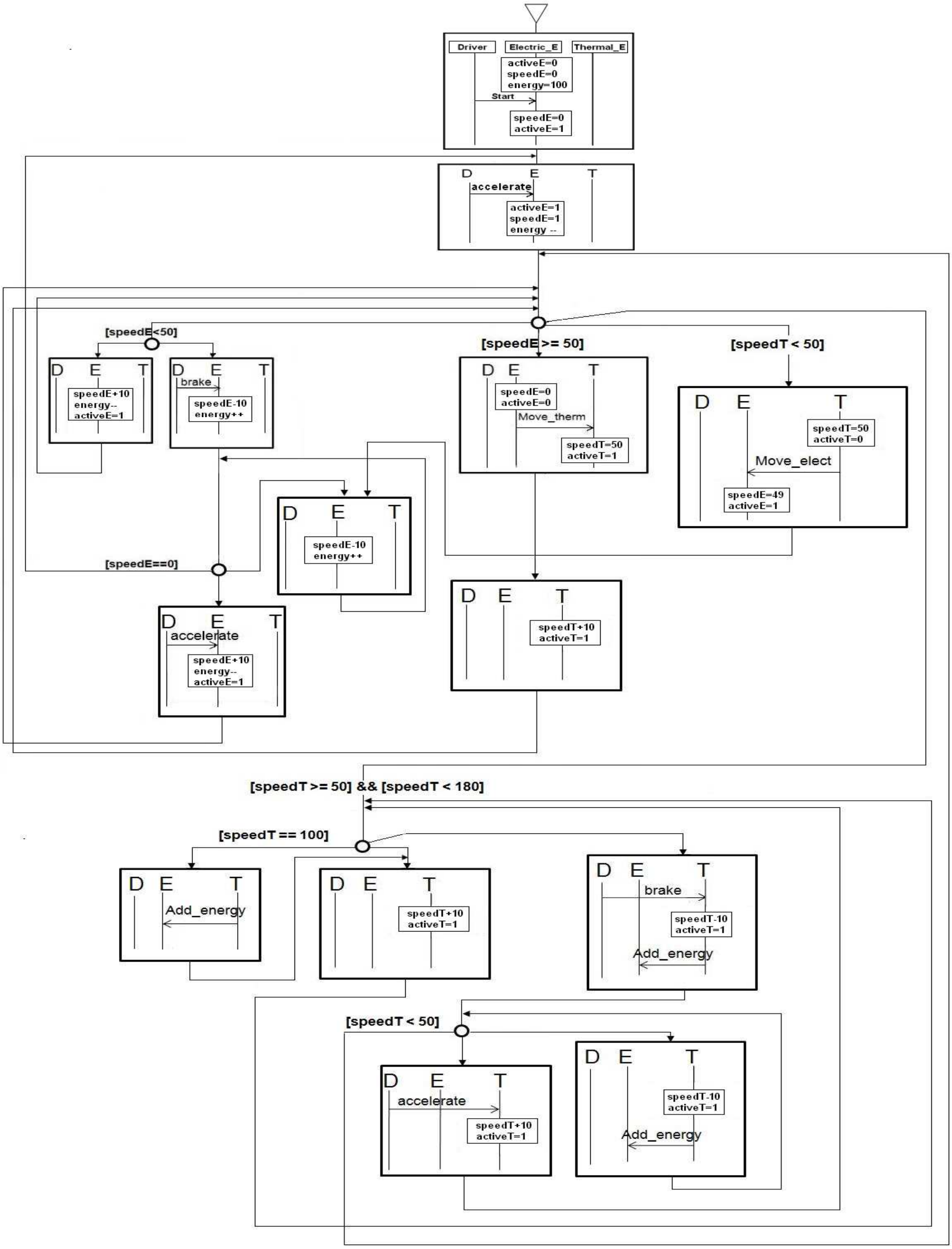


Fig. 7.A hybrid car MSC

<u>Details and constraints:</u>

- In starting, it is the electrical motor which provides the traction power. activE=1, energy = 100 and speedE = 0;
- The driver either he accelerates or he brakes. If the driver accelerates, he is considered that it is always in the acceleration phase (speedE++ or speedT++) until he brakes. And vice versa. It is supposed that an acceleration phase lasts 1 u.t. idem for a braking phase.
- If speed $\in$[0 mph – 50 mph[=> activeE=1, activeT = 0 and energie -- .
- If speed $\in$[50 mph – 180 mph[=> activeE=0 and activeT =1.

- If the driver brakes: speed-- and energy++.
- The passage from speed <100 to speed>=100: energy++.
- We suppose that speed increases by 10 mph while accelerating, and decreases by 10 mph while braking.

By using the previous steps of transformation into DEVS models, we obtain the DEVS atomic models represented in the Fig. 8 for the electrical engine, and Fig. 9 for the thermal engine. After the global atomic models for all objects of the system have been built, we construct the coupled model given in the Fig. 10 who describes the overall behavior of the hybrid car system, by connecting the outputs of the electrical engine model with the input of the model representing the thermal engine and vice versa, because the objects communicate between them and also to allow the system to work automatically.

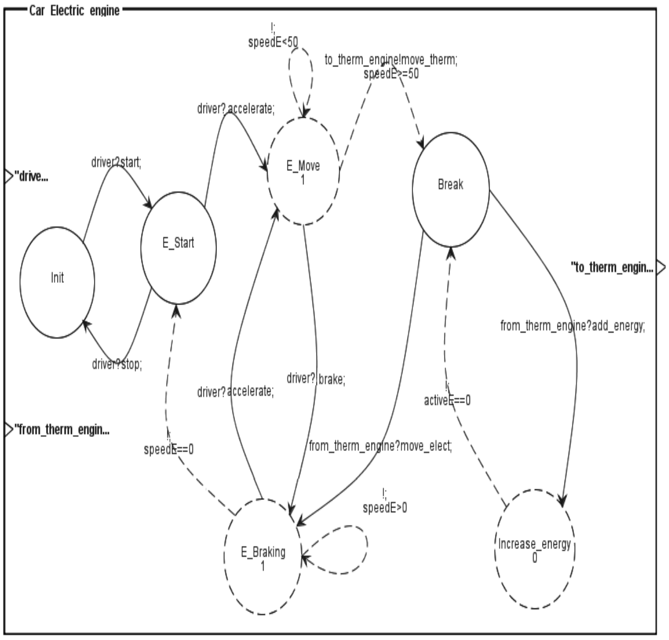


Fig. 8 The global atomic DEVS model for the electrical engine

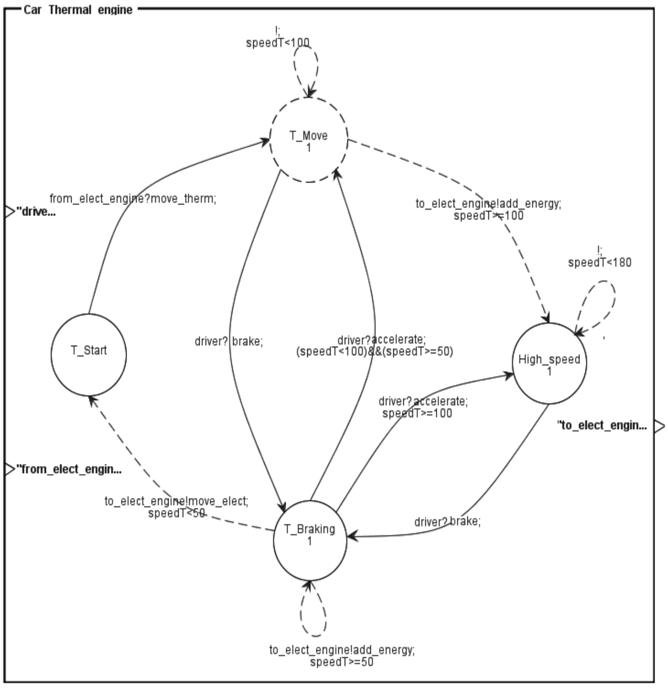


Fig. 9 The global atomic DEVS model for the thermal engine

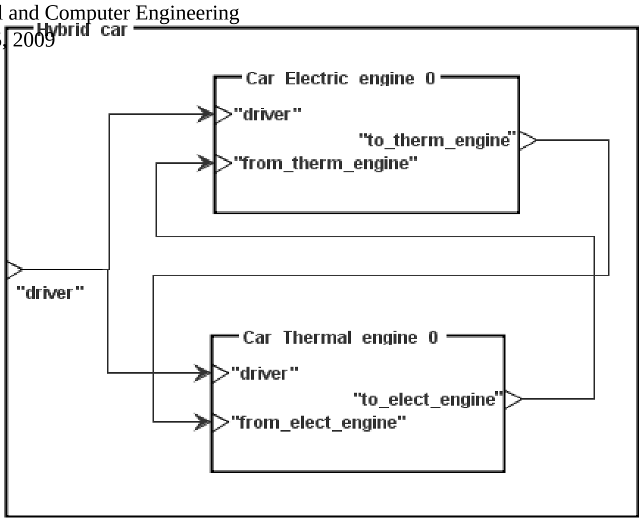
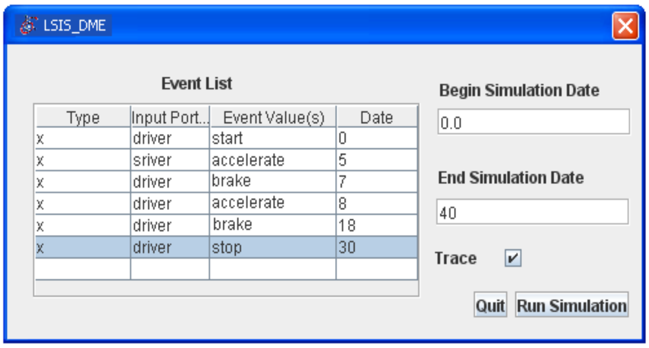


Fig. 10 The final coupled DEVS model for a hybrid car

To validate the specification of the behavior of the system obtained with the final coupled DEVS model, we simulate the model results. For that we use the LSIS-DME tool [21]. This simulator was developed by team members of our laboratory; it's composed of two parts: a model editor, and simulator. Then, from the final model and a set of data, the simulator provides the simulation results (Fig. 12). The dataset is defined by the driver who enters all external events supposed to occur during the simulation (Fig. 11). During the simulation, we have to check that all events that should be treated were treated, and all events that should not be treated were not treated.

We suppose that the driver do the following scenario:



Type	Input Port...	Event Value(s)	Date
x	driver	start	0
x	sriver	accelerate	5
x	driver	brake	7
x	driver	accelerate	8
x	driver	brake	18
x	driver	stop	30

Fig. 11 Example of filling input schedules

In this scenario, the driver start the car at 0 u.t, accelerate at 5u.t, brake at 7 u.t, accelerate at 8 u.t, barke at 18, u.t and stop the car at 30 u.t. Normally in simulation , when driver stat, only electric engine must run (activeE=1) and the termal engine must be inactive (activeT=0); when the driver accelerate in 5 u.t, the speedE must increase by 10 unity(speedE+10) and the energy must decrease (energy--) until ta =7u.t. When the driver brake in 7 u.t the speedE must decrease by 10 unity (speedE-10) and the energy must increase (energy++). At 8 u.t, when the driver accelerate again, speedE increase until 50 mph, after this, the thermal engine must be inactive (activeE=o), and the thermal engine begin active (activeT=1). SpeedT must increase until ta=18 u.t. when the speedT exceed 100mph, the thermal engine sent enrgy to electric engine. When the driver brake at 18 u.t, the energy must increase and the speed must decrease until speedE=0. And finally the driver stops the car.

The following figure show the simulation results of the hybrid car.

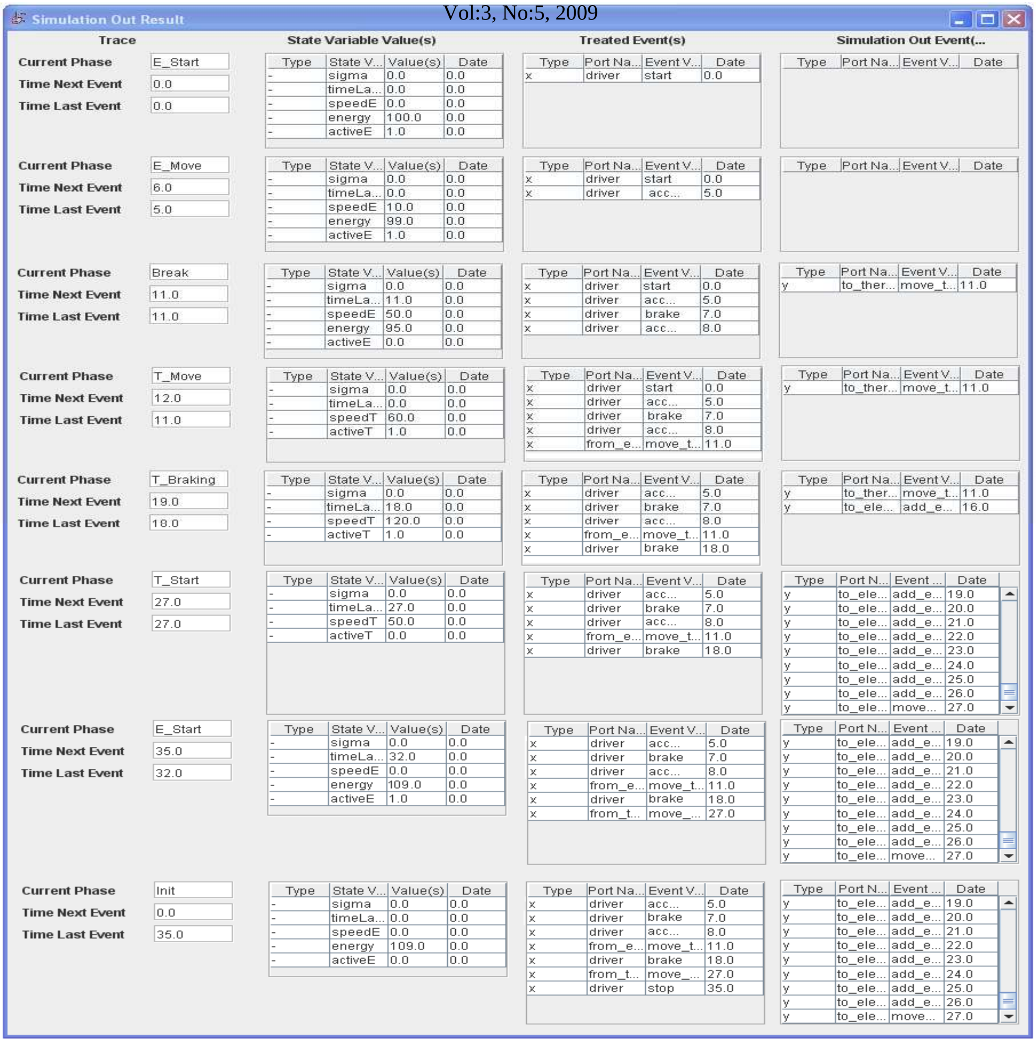


Fig. 12 The simulation results

The description of the system behavior can be established on a set of scenarios. Since each scenario is usually written in isolation, bringing many scenarios together will result in inconsistencies which have to be detected and resolved. A final coupled DEVS model inferred from a set of scenarios and representing the overall behavior of the system should be established by atomic models representing the behavior of each component appearing in the scenarios. In addition, each atomic model should exhibit as sequences of events at least all scenarios projected to the time line of its component. This consistency constraint between a MSC and an inferred atomic DEVS model is defined as follows [9]:

Definition1 (Scenarios-DEVS consistency):

Let SC a scenario represented in the form of a MSC model with components 1, …,n. An atomic DEVS model D = (D1 ‖… ‖ Dn) is consistent with SC if, and only if, for each component i, $Bh(SC)_{/events(i)} \subseteq Bh(Di)$ where events(i) is the set of events involving components i in the scenarios. And for the coupled final DEVS model CD = (D1 *coupling* … *coupling* Dn), $Bh(SC) = Bh(CD)$.

Since, scenarios-based models describe only examples of system behaviors; it is possible that the atomic DEVS models consistent with those scenarios produce more behaviors than those explicitly captured in the scenarios. However, some of these additional behaviors may be present in every atomic DEVS model that is consistent with the specified scenarios. Such scenarios are called implied scenarios.

- What is an implied scenario? An implied scenario is a behavioral path that can be extracted from the DEVS model but does not exist in the MSC specification.

Some state transition paths which are not explicit in MSC can occur by merging similar state in optimization phase of synthesis process. Such state transitions are called unexpected state transition.

The implied scenarios can be constructed from the unexpected state transitions and can help to complete the requirements specification with unforeseen situations or indicate that the specification must be refined to prevent unwanted executions.

Henry Muccini [19] and Felipe [20] had presented an approach to detect implied scenarios in state machines extracted from hierarchical Message Sequence charts, and has proved that there is a strict correlation between implied scenarios and non-local branching choices in hMSC "An implied scenario may be found in the MSC specification when a nonlocal choice occurs that lets processes keep extra information that is lately used for a communication".

V. Formal Verification with Z Language

In this section we show how to prove formally static system properties with Z language. We will use the approach given in [18], which consists in translating DEVS model to Z specification. The equivalent Z specification of the DEVS model of the hybrid car system is divided into three parts: Electric Engine Z specification (equivalent to Electric engine DEVS model), Thermal Engine Z specification (equivalent to Thermal Engine DEVS model) and Z Hybrid Car specification (equivalent to the coupled DEVS model).

A. Z specification of the Electric Engine

The following free type definitions represent finite sets of state values, input and output variables:

$*E\_PHASE ::= Init \mid E\_Start \mid E\_Move \mid Break \mid Increase\_Energy \mid E\_Braking$

$*CAR\_INPUTS ::= start \mid stop \mid accelerate \mid brake$

$*SET\_IN\_OUT ::= add\_energy \mid move\_elect \mid move\_therm$

The global state schema containing the state variables describing the electric engine is:

Elect_Engine

$phaseE: E\_PHASE;\ speedE, energy, activeE: \mathbb{N}$

$0 \leqslant speedE < 50 \wedge 0 < energy < 100$

The initial state schema containing initial values of the state variables is:

Init_Elect_Engine

$Elect\_Engine$

$phaseE = Init \wedge speedE = 0 \wedge energy = 100 \wedge activeE = 0$

There are two major operations schemas deducted from the DEVS model of the Electric Engine: (i) Internal_Transit_Elect schema which contains all the internal transitions of the DEVS model and eventual outputs generated with some of theses transitions, and (ii) External_Transit_Elect schema which contains all the external transitions of the DEVS model. Each transition is presented by (values of state variables before transition and eventual inputs $\Rightarrow$ values of states variables after transition and eventual outputs)

Internal_Transit_Elect

$\Delta Elect\_Engine;\ to\_therm\_engine!: SET\_IN\_OUT$

$phaseE = E\_Move \wedge activeE = 1 \wedge speedE < 50$
$\Rightarrow phaseE' = E\_Move \wedge energy' = energy - 1$
$\wedge speedE' = speedE + 10 \wedge activeE' = activeE$
$phaseE = E\_Move \wedge activeE = 1 \wedge speedE \geqslant 50$
$\Rightarrow phaseE' = Break \wedge speedE' = 50 \wedge activeE' = 0$
$\wedge to\_therm\_engine! = move\_therm$
$phaseE = Increase\_Energy \wedge activeE = 0$
$\Rightarrow phaseE' = Break \wedge speedE' = 50 \wedge activeE' = activeE$
$phaseE = E\_Braking \wedge speedE = 0 \wedge activeE = 1$
$\Rightarrow phaseE' = E\_Start \wedge speedE' = 0 \wedge activeE' = activeE$
$phaseE = E\_Braking \wedge activeE = 1 \wedge speedE > 0$
$\Rightarrow phaseE' = E\_Braking \wedge speedE' = speedE - 10$
$\wedge energy' = energy + 1 \wedge activeE' = activeE$

External_Transit_Elect

$\Delta Elect\_Engine;\ driver?: CAR\_INPUTS$
$from\_therm\_engine?: SET\_IN\_OUT$

$phaseE = Init \wedge speedE = 0 \wedge energy = 100$
$\wedge activeE = 0 \wedge driver? = start$
$\Rightarrow phaseE' = E\_Start \wedge speedE' = 0 \wedge activeE' = 1$
$phaseE = E\_Start \wedge speedE = 0 \wedge activeE = 1$
$\wedge driver? = stop$
$\Rightarrow phaseE' = Init \wedge speedE' = 0 \wedge activeE' = 0$
$\wedge energy' = 100$
$phaseE = E\_Start \wedge speedE = 0 \wedge activeE = 1$
$\wedge driver? = accelerate$
$\Rightarrow phaseE' = E\_Move \wedge speedE' = speedE + 10$
$\wedge energy' = energy - 1 \wedge activeE' = activeE$
$phaseE = E\_Move \wedge activeE = 1 \wedge driver? = brake$
$\Rightarrow phaseE' = E\_Braking \wedge speedE' = speedE - 10$
$\wedge activeE' = activeE$
$phaseE = E\_Braking \wedge activeE = 1$
$\wedge driver? = accelerate$
$\Rightarrow phaseE' = E\_Move \wedge speedE' = speedE + 10$
$\wedge energy' = energy - 1 \wedge activeE' = activeE$
$phaseE = Break \wedge speedE = 50 \wedge activeE = 0$
$\wedge from\_therm\_engine? = add\_energy$
$\Rightarrow phaseE' = Increase\_Energy \wedge energy' = energy + 1$
$\wedge activeE' = activeE$
$phaseE = Break \wedge speedE = 50 \wedge activeE = 0$
$\wedge from\_therm\_engine? = move\_elect$
$\Rightarrow phaseE' = E\_Braking \wedge speedE' = speedE - 10$

B. Z specification of the Thermal Engine

The same rules are applied to the thermal engine, thus:

$*T\_PHASE ::= T\_Start \mid T\_Move \mid T\_Braking \mid High\_Speed$

$*CAR\_INPUTS ::= accelerate \mid brake$

$*SET\_IN\_OUT ::= move\_elect \mid add\_energy \mid move\_therm$

Therm_Engine

$phaseT: T\_PHASE;\ speedT, activeT: \mathbb{N}$

$50 \leqslant speedT \leqslant 220$

Init_Therm_Engine

$Therm\_Engine$

$phaseT = T\_Start \wedge speedT = 0 \wedge activeT = 0$

Internal_Transit_Therm

Δ*Therm_Engine ; to_elect_engine!: SET_IN_OUT*

phaseT = T_Move $\wedge$ *activeT* = 1 $\wedge$ *speedT* $\geqslant$ 100
$\Rightarrow$ *phaseT' = High_Speed* $\wedge$ *speedT' = speedT* + 10
$\wedge$ *activeT' = activeT* $\wedge$ *to_elect_engine! = add_energy*
phaseT = T_Braking $\wedge$ *activeT* = 1 $\wedge$ *speedT* $\leqslant$ 50
$\Rightarrow$ *phaseT' = T_Start* $\wedge$ *speedT'* = 50 $\wedge$ *activeT* = 0
$\wedge$ *to_elect_engine! = move_elect*
phaseT = T_Move $\wedge$ *activeT* = 1
$\Rightarrow$ *phaseT' = T_Move* $\wedge$ *activeT' = activeT*
$\wedge$ *speedT' = speedT* + 10
phaseT = High_Speed $\wedge$ *activeT* = 1
$\Rightarrow$ *phaseT' = phaseT* $\wedge$ *activeT' = activeT*
$\wedge$ *speedT' = speedT* + 1
phaseT = T_Braking $\wedge$ *activeT* = 1 $\wedge$ *speedT* > 50
$\Rightarrow$ *phaseT' = phaseT* $\wedge$ *activeT' = activeT*
$\wedge$ *speedT' = speedT* – 1 $\wedge$ *to_elect_engine! = add_energy*

External_Transit_Therm

Δ*Therm_Engine; driver?: CAR_INPUTS*
from_elect_engine?: SET_IN_OUT

phaseT = T_Start $\wedge$ *speedT* = 50 $\wedge$ *activeT* = 0
$\wedge$ *from_elect_engine? = move_therm*
$\Rightarrow$ *phaseT' = T_Move* $\wedge$ *speedT' = speedT* + 10
$\wedge$ *activeT'* = 1
phaseT = T_Move $\wedge$ *activeT* = 1 $\wedge$ *driver? = brake*
$\Rightarrow$ *phaseT' = T_Braking* $\wedge$ *speedT' = speedT* - 10
$\wedge$ *activeT'* = 0
phaseT = T_Braking $\wedge$ *activeT* = 1$\wedge$ *driver? = accelerate*
$\wedge$ *speedT* > 50 $\wedge$ *speedT* < 100
$\Rightarrow$ *phaseT' = T_Move* $\wedge$ *speedT' = speedT* + 10
$\wedge$ *activeT'* = 1
phaseT = High_Speed $\wedge$ *activeT* = 1 $\wedge$ *driver? = brake*
$\Rightarrow$ *phaseT' = T_Braking* $\wedge$ *speedT' = speedT* - 10
$\wedge$ *activeT'* = 1
phaseT = T_Braking $\wedge$ *activeT* = 1
$\wedge$ *driver? = accelerate* $\wedge$ *speedT* $\geqslant$ 100
$\Rightarrow$ *phaseT' = High_Speed* $\wedge$ *speedT' = speedT* + 10
$\wedge$ *activeT'* = 1

C. Z specification of the Hybrid_Car

First the free type definition of the inputs of the coupled model is presented.

**CAR_INPUTS::= start / stop / accelerate / brake*

Z schemas of the Electric Engine and the Thermal Engine (stated below) are used.

The global state schema of the hybrid car is presented by the following conjunction of schemas:

Hybrid_Car $\widehat{=}$ *Elect_Engine* $\wedge$ *Therm_Engine*

The initial state of the Hybrid_Car state is given by the conjunction of both electric and thermal engines initial states schemas:

Init_Hybrid_Car $\widehat{=}$*Init_Elect_Engine* $\wedge$ *Init_Therm_Engine*

The couplings between DEVS model components are represented as following:

Coupling1 $\widehat{=}$*Internal_Transit_Elect*
$\gg$ *External_Transit_Therm*

Coupling2 $\widehat{=}$*Internal_Transit_Therm*
$\gg$ *External_Transit_Elect*

In fact, the outputs of the Internal_Transit_Elect schema are the inputs of the External_Transit_Therm and inversely.

D. Proof Obligation of the Hybrid_Car

We have used Z/EVES – a Z editor used for writing Z specification and making proofs, to prove that the initial state and operations of the Hybrid_Car preserve state invariants:

- *Proving Initial state:*

  theorem *Can_Init_Hybrid*
  $\exists$*Hybrid_Car'* • *Init_Hybrid_Car*
  Prove by reduce - Z/EVES command checking theorems- → ***TRUE***

- *Proving operations (Precondition calculus):*

  theorem *Precondition_Coupling1*
  $\forall$*Hybrid_Car* • ***pre*** *Coupling1*
  theorem *Precondition_Coupling2*
  $\forall$*Hybrid_Car* • ***pre*** *Coupling2*

These two theorems permit to determine the preconditions of the operations schemas (the conditions which allow the operations to be performed). Therefore, coupling1 and coupling2 operations are performed if the returned preconditions are equal to the preconditions contained in the equivalent schemas. For example: if Elect_Engine is on the phase "*E_Move*" and the Therm_Engine is on the phase "*T_Start*", do them transit respectively to the phases "*Break*" and "*T_Move*" satisfying the state invariants? This question is represented by the following theorem:

$\forall$*Hybrid_Car / phaseE = E_Move* $\wedge$ *activeE = 1* $\wedge$ *speedE* $\geqslant$ *50* $\wedge$ *phaseT = T_Start* $\wedge$ *speedT = 50* $\wedge$ *activeT = 0*
pre *Coupling1*

If the answer is true, it means that these transitions respect properties of the system. Therefore their simulation is done in a coherent context.

VI. Conclusion

We have provided a multi-specification framework for modeling, verifying and validating constraints based interactive systems. In fact, the interactions can be described by the MSC, the system behavior can be captured with DEVS formalism and the functional part is well formalized with Z notation. This framework permits to improve verification and validation process by using simulation and formal verification techniques. We have presented the MSC synthesis into a DEVS model in order to validate the global behavior of the system by simulation. We have chosen scenario semantics restricted to event sequences with the notion of (iteration, alternative and sequence). Also, formal verification was used to prove formally the consistency of the system (absence of conflicts and incoherencies in system properties). Our approach permits a great automation in system analysis. In fact, once the system is modeled with MSC, our approach automatically generates equivalent DEVS model. The latter is also automatically transformed to a Z specification. In addition, our approach bridges the gap between "modeling and simulation" and "formal methods" by integrating simulation and formal proof techniques in the same framework.

References

[1] ITU, 2000. Message Sequence Charts. Recommendation Z.120. International Telecommunications Union. Telecommunication Standardisation Sector.

[2] Damm, W., Harel, D., 2001. LSCs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45--80.

[3] OMG, 2005. UML 2.0 Specification. Object Management Group. Avaibale from: http://www.omg.org [August 2005].

[4] Brian, L., Hans, E., 2004. UML 2 toolkit. Whiley Publishing OMG press

[5] Zeigler B., 1976. *Theory of Modeling and Simulation*. Krieger Publishing Company. 2nd Edition. New york.

[6] Zeigler B.P., Praehofer H., Kim T. G., "*Theory of Modeling and Simulation*." 2nd Edition, Academic Press, New York, NY 2000.

[7] Liang H., Dingel J., Diskin Z., A comparative Survey of Scenario-based to State-based Model Synthesis Approaches, SCESM'06 : International Workshop on Scenarios and State Machines: Models, Algorithms, and Tool, Shangaï, China, pp.5-12, May 2006.

[8] David, H., Kugler, H., Pnueli, A., 2005. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. Formal Methods in Software and Systems Modeling.

[9] Letier, E., Kramer, J., Magee, J., Uchitel, S., 2005. Monitoring and Control in Scenario-Based Requirements Analysis. International Conference on Software Engineering, Proceedings of the 27th international conference on Software engineering.

[10] Ziadi, T., Hélouët, L., Jézéquel, J., 2004. Revisiting Statechart Synthesis with an Algebraic Approach. Proc. of 26th International Conference on Software Engineering (ICSE), IEEE Computer Society.p. 242-251.Edinburgh, May.

[11] Damas, C., Lambeau, B., Lamsweerde A., 2006. Scenarios, Goals, and State Machines: a Win-Win Partnershi for Model Synthesis. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 197- 207. Portland, Oregon, USA.

[12] Sqali, M., Torres, L., Frydman, C., 2008. Synthèse de scénarios en DEVS, *7ème conférence internationale de Modélisation et SIMulation (MOSIM08)*. Paris, 31 mars-2 avril, .

[13] Sqali, M., Torres, L., Frydman, C., 2008. Synthetizing scenarios to DEVS models. *SpringSim08*. Ottawa, Canada.

[14] B. F. Potter, J. E. Sinclair, and D. Till. 1991. An Introduction to Formal Specification and Z. Prentice Hall International Series in Computer Science.

[15] Jonathan Bowen.Formal Specification and Documentation Using Z: A case study approach.Revised. 2003

[16] Jacky, J. 1997. The way of Z: Practical Programming with formal methods. Cambridge University Press.

[17] Peschanski, F., and D. Julien. 2003.When Concurrent Control Meets Functional Requirements, or Z+Petri-Nets." ZB 2003: Formal Specification and Development in Z and B, Springer Berlin / Heidelberg. 79-97.

[18] Trojet, M. W., A. Hamri, and C.Fydman. 2008. Logical analysis of DEVS models using Z." Proceedings of International Simulation Multi-conference ISMc'08.

[19] Felipe Cantal de Sousa, Nabor C. Mendonça, Sebastian Uchitel, Jeff Kramer "Detecting Implied Scenarios from Execution Traces", *Proceedings of the 14th Working Conference on Reverse Engineering,* pages: 50-59 Washington, DC, USA,2007.

[20] Muccini H., "Detecting Implied Scenarios analyzing non-local Branching Choices", *Proc. Int. Conf. on Fundamental Approaches to Software Engineering*, ETAPS2003, Warsaw, Poland, April 2003.

[21] M. E.-A. Hamri, G. Zacharewicz, "LSIS DME: An Environment for Modeling and Simulation of DEVS Specifications", in: AIS-CMS International modeling and simulation multiconference, pp. 55-60, Buenos Aires - Argentina, February 8-10 2007. ISBN 978-2-9520712-6-0.