An approach to model variations of a scenario: Application to Intelligent Transport Systems

Fabien Bonnefoi[2], Lom Messan Hillah[1], Fabrice Kordon[1], and Guy Frémont[2]

[1] Université P. & M. Curie, LIP6-CNRS, 4 Place Jussieu, 75005 Paris, France
lom-messan.hillah@lip6.fr, fabrice.kordon@lip6.fr
[2] Cofiroute DSO/R&D, 6 - 10 rue Troyon, 92310 Sèvres - France
fabien.bonnefoi@cofiroute.fr, guy.fremont@cofiroute.fr

Summary. Modern distributed systems tend to integrate more and more features and components that increase their complexity and size. This often leads to the decomposition of such systems into multiple parts to overcome the complexity of their modeling and analysis. In this paper, we present a modeling methodology for systems engineering based on a modular approach. The methodology relies on the definition of components and assembling rules to model complex systems. It is founded on formal specification formalisms and tools to enable model checking. This paper proposes an example by which we apply this methodology on a complex system from the domain of Intelligent Transport Systems.

1 Introduction

Modern distributed systems tend to integrate more and more features that increase their complexity such as mobility, a variable number of components during execution or complex physical and time constrained mechanisms (i.e. braking distance or any similar complex function).

An excellent example of such systems is illustrated in *Intelligent Transport Systems* (ITS) where road operators, the infrastructure, vehicles, their drivers and other road users must cooperate for an efficient and secure system. Such systems are even more complex to analyze than previous distributed systems and require more reliability.

These distributed systems have such specific strategies that it is useless to imagine a short-term solution "in the large" that will fit numerous applications. We prefer to take into consideration the specificities of the application domain by selecting the appropriate model and designing an accurate methodology. Then, it is of interest to consider that these distributed systems are centered on the notion of "case studies" where execution scenarios are elaborated and analyzed. This approach is practiced in ITS projects [3]. Moreover, paradigms such as client/server, that allow the reuse of Object Oriented Approach, cannot scale up to the needs of such systems.

Major actors in companies or institutions dealing with critical applications acknowledge that formal methods are necessary, but that new techniques are needed to face the combinatorial explosion problem when dealing with large industrial systems [11]. Consequently, there is a need for specific methodology and tools to design and analyze them.

The purpose of this paper is to present a modeling methodology based on formal methods and tools that allow the assessment of implementation choices in distributed systems. We focus on techniques to easily change components and assembling operations to define new models, thus allowing for the reuse of components in different model architectures or case study scenarios. Thanks to these techniques, the definition of variations of scenarios within a short time, with minimum effort and maximum reusability can be performed. This methodology enables formal verification of complex systems composed of discrete and continuous events.

Our approach focuses on:

- a modular design centered on a model architecture;
- a way to integrate complex physical functions into the system specification;
- a connection to formal methods to achieve qualitative analysis.

The paper is structured as follows. The first part presents an analysis of Intelligent Transport Systems context and some related work on formal methods in Sect. 2. Section 3 presents our methodology based on formal methods. Then, Sect. 4 details a *safe insertion* case study, its architecture and scenario. In Sect. 5 we present modeled components and the assembling operations. Finally, the analysis of the system is discussed in Sect. 6.

2 Related work

In this paper we are mainly interested in *qualitative analysis* of systems and *quantitative* evaluation is not considered yet. However, these systems also have continuous characteristics we must cope with.

Here, we first present the context of intelligent Transport Systems as a good example of such modern complex systems. After which, we present a brief overview of the ongoing work around modeling and analysis using formal methods.

2.1 The context of ITS

Several recent Intelligent Transport Systems projects aim at providing assistance to drivers and deal with partially automated motorways. The community investigated first a full automated infrastructure and vehicles approach (like in the PATH [15] project) in the 1990's. This approach was then dropped in favor of a new line of research and development activities, more centered on

safety strategies in various "problem areas", such as "Lane Change and Merge Collision Avoidance", "Intersection Collision Avoidance" or "Safety Margin for Assistance Vehicles" [17].

This vision relies on *Cooperative Systems* where "road operators, infrastructure, vehicles, their drivers and other road users will cooperate to deliver the most efficient, safe, secure and comfortable journeys" [5]. Implementing these systems components then follows a peer-to-peer organization where each actor or component must fully cooperate in a time-constrained and safety-critical environment. Many different implemented features need the participation of all or some of the components and the use of complex algorithms.

Such systems are even more complex to analyze than previous distributed systems. Moreover, they require more reliability. Consequently, there is a need for specific methodology and tools to design and analyze them.

2.2 Formalisms for systems modeling and analysis

There exists a wide range of specification languages to model and analyze systems, at various levels of abstraction. For example, for sequential processes, it is possible to use transition systems or automata. When considering cooperative concurrent processes, process algebras or Petri nets are interesting choices.

We are working on the behaviors of large hierarchical distributed systems composed of cooperative systems on which we want to apply formal methods. The fundamental underlying approach is to perform formal verification of safety properties on those systems. Since we are quite familiar with Petri nets formalism which brings an extensive theory with a well developed mathematical model, we have decided to consider its modeling and analysis capabilities.

It is known that colored Petri nets [22] are suitable to formally specify the behavior of distributed systems. The proposed approach in this paper aims at verifying complex and hierarchical distributed systems in which we should cope with discrete concurrent events as well as their continuous aspects.

Therefore, the choice of a modeling formalism or a combination of modeling formalisms must take into account qualitative analysis, along with continuous characteristics. Petri Nets are suitable for such an approach by providing properties like boundness, liveness, evaluation of temporal logic formulae, etc. They were developed to allow for efficient verification techniques associated with a powerful expressivity [12].

There are other approaches combining Petri Nets with semi-formal notations, such as UML [2] or AUML [6]. Typically, in [2], systems are described by means of Statecharts and Sequence Diagrams, "emphasizing specific patterns of interactions among Statecharts". A translation to Generalized Stochastic Petri Nets (GSPNs) is then provided. Compositionality is a key concept to build the final model. Using such a technique, validation and performance evaluation are the chief objectives.

An important technique related to qualitative analysis we want to evaluate and improve is the encoding of complex functions in Petri nets through discretization (discussed in section 3.3). High Level Petri Nets [18] seem from the first standpoint an interesting notation to put into work. They provide much flexibility in terms of types definition as well as functions definition. However, these capabilities induce complexity in structural analysis and model checking that current tools cannot handle yet.

3 Modeling methodology

In this section we describe our modeling methodology in different steps leading to a complete set of models. This methodology is sketched in Fig. 1. As a development approach, it strongly relies on a generic model architecture gathering components of the system.

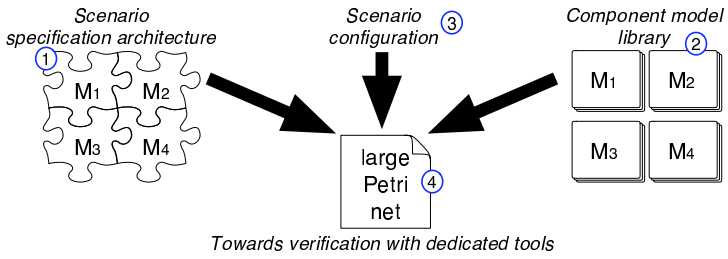

Fig. 1. Overview of the modeling and assembling methodology

3.1 Defining the case study

The main idea is that, for a situation (an ITS "case study" in [17]) the structure of the generic model architecture will not change. A new configuration, or scenario, (for example, to consider smarter vehicles, another infrastructure strategy or another part of the road network, i.e., crossroad, insertion lane...) is then obtained by replacing a modeling component by another one. Once interfaces are properly defined, it is as easy as for programs.

Then we first define the case study in terms of :

1. the main situation or set of *problematics* constitutive of the subject of the study. For example a problem area taken from the domain of ITS [17]: "Safe insertion" or "Intersection Collision Avoidance" etc.
2. the set of properties we want to verify during the study. For example the minimum distance between two vehicles or the absence of deadlocks.

3. different scenarios or configurations of the system that enable the study of the problematics and the verification of properties in the whole context. For example, considering smarter vehicles or a different part of the road network.

Then it is important to consider that each scenario leads to the definition of a specific architecture, with specific components and their configuration. It is only when a set of scenarios has been defined that the generic architecture with all components necessary for the case study analysis is obtained. Note that if we only change the initialization of components (and not components selection), we call it a variation of a scenario and not a new scenario.

Alternative descriptions of components are stored in a library and selected using a file that describes the specific scenario.

Thus, the modeling process can be defined in different steps (numbered in Fig. 1):

1. definition of the case study problematics and properties to analyze;
2. definition of different scenarios components, and the modeling of the components in the library;
3. selection and configuration of components according to a given level of abstraction in the analysis of the situation that is investigated. This is the final step in the definition of a scenario, or case study specific configuration;
4. use of the assembling mechanisms to obtain a complete model and analyzing it with an appropriate set of tools.

Steps 2 to 4 can be repeated several times, as long as the system is not fully analyzed, or there remain some variations to be analyzed, or some hypothesis made at a given level of abstraction is not ensured.

A similar approach was first experimented in the formal verification of the micro-Broker in the PolyORB middleware [16] but with almost no tools to automate the modeling part (most of it was performed using shell scripts and the Unix `sed` command as a prototyping environment).

Let us now describe the main choices we have made, as well as the tool we have designed in order to help a designer to model and analyze his system. It is important to automate the process, since this allows us to use the formal model as a basis to evaluate the non-regression of the system when strategies are explored.

3.2 Modeling the components behavior

Modeling scenarios components behaviors using colored Petri nets is the second step in the design. The first step, as stated above, consists in defining the case study.

In colored Petri nets, a color domain (a discrete data type) is associated with places and transitions. The colors of a place label tokens contained in this

place, whereas the colors of a transition define different ways of firing it. In order to specify these firings, a color function is attached to every arc which, given a color of the transition connected to the arc, determines the number of colored tokens that will be added to or removed from the corresponding place. Finally the initial marking is defined by a multi-set of colored tokens in each place.

A color domain is a cartesian product of color classes which may be viewed as primitive domains. This product is possibly empty (e.g., a place which contains neutral tokens) and may include repetitions (e.g., a transition which synchronizes two colors inside a class).

We have selected a specific class of colored Petri Nets: Well Formed Petri Nets [7]. They restrict the use of functions to : identity, cartesian product, successor and predecessor, broadcast, belongs to. Hence, they preserve some interesting properties that are useful to handle the verification of large systems using model checking techniques:

- types of token can be divided in *static subclasses* that are subsets of the type where color values have equivalent behavior all over the reachability graph; static subclasses denote *global symmetries* in the system (i.e., the identity of a process can be permuted in a critical section without changing the global behavior of the system);
- static subclasses can be divided in *dynamic subclasses* that are subsets of the type where color values have equivalent behavior in some parts of the reachability graph; dynamic subclasses capture *local symmetries* that only occur in some parts of the reachability graph;
- both static and dynamic subclasses can be computed using the structure of the specification [20, 1].

Based on these characteristics, it is possible to build the *symbolic reachability graph* where a symbolic state represents an equivalence class of states, appropriate for *Linear Time Logic* model checking [8]. The ratio between the size of the symbolic reachability graph and the reachability graph may be exponential in favorable cases.

This class of Petri nets also allows the use of structural techniques [22] such as invariants, traps or bounds. If some structural properties are not yet extended to colored nets, the unfolding operation (that transforms a colored net into a P/T one) helps in providing such properties.

3.3 Modeling complex physical functions

Well Formed Petri Nets preserve some interesting properties for verification and model checking. However, modeling is not as easy to handle as in CPNs [18] where tokens can be manipulated using any type of ML or C function.

To cope with this particular issue, we have elaborated a modeling technique that allows one to specify complex functions similarily to CPNs but in a way

that enables the use of Well Formed Petri Nets and their associated properties. The principle is based on discretizing the function to be encoded.

Let us illustrate this principle using the example of Fig. 2 that represents a function $y = f(x)$ and its possible discretization.

It is possible to represent this function using one Petri net module of Fig. 2. Place **f** represents the function: its values are stored as the initial marking representing all the possible $< x, f(x) >$ couples in the considered intervals (here, `type_X` and `type_Y`). The path going from place **P1** to place **P2** via transition **T** computes y from the value of x.

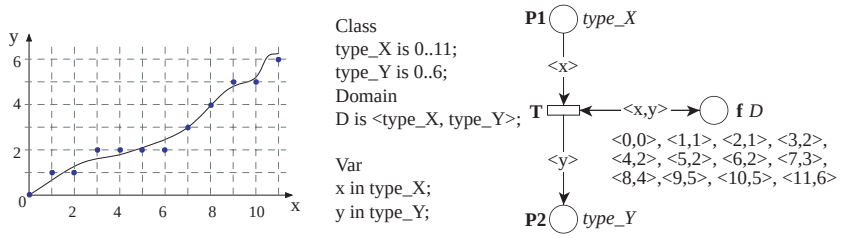

Fig. 2. Example of function, its possible discretization and the associated Petri net.

This technique can be generalized to any function $x = f(x_1, x_2, ..., x_n)$, regardless of its complexity. Non deterministic functions can also be specified the same way (for example, to model potential errors in the system). Let us note that:

- the discretization of any function becomes a modeling hypothesis and must be validated separately (to evaluate the impact of imprecision due to discretization),
- given a programmed function, it is easy to automatically generate the list of values to store in the initial marking of the place representing the function.

From the verification point of view, it is interesting to note that the large marking of places representing complex functions does not impact the size of the reachability graph if the model checking techniques mentioned in Sect. 3.2 are applied. Since the place marking never changes, it is only stored once in memory.

3.4 Assembling model components

Once a scenario architecture and components defined, the modeler has a global vision of a specification of the case study (i.e., a scenario). Behaviors of components to be modeled must then be defined, as well as their interfaces and connections.

Components interact through interfaces we have defined. Theses interfaces capture synchronizations, communications, inclusions or abstraction of a component. Operations to connect components through interfaces are also defined. When setting up the connections, the modeler must enforce some semantic rules we have also defined.

Components Interfaces

In this context, interfaces are nodes, places or transitions, through which a component interacts with others. In contrast, a local node is not an interface. Its scope is strictly within a component. Here are the relevant interfaces we use in the case study presented in Sect. 4:

- *sub-net transition.* This interface represents a component (herafter sub-net) that is to be inserted into the component (herafter super-net) that transition is an interface of.
- *synchronization transition.* Components synchronize through this interface during their execution.
- *Resource or flow sharing.* Components share resources or communicate using this type of interface. For example, an abstraction place represents another component's place.

When defining components behavior, some important choices are required:

- the level of abstraction of the analysis (it can be refined from a former analysis using the same architecture for the same case study);
- strategies to be evaluated in the system must be selected (according to the abstraction level);
- initial conditions of the system must be defined.

In such systems, multiple variations have to be investigated according to the strategies in the system, the initial conditions, or local implementation choices in components. Thus, each variation must lead to the definition of a configuration file. This configuration file is a script that gathers a specific version (selected from a library) for each component in the architecture.

For example, let us imagine that we need to formally validate the behavior of the traffic in a motorway at various saturation levels. There is one scenario and several variations that will, in this case, select more or less instances of vehicles in the system. Another variation could be the level of agressivity of each driver or the strategy of the infrastructure.

The configuration phase then corresponds to the definition of a script to assemble the selected configuration. After the PolyORB experience [16], it was obvious that some dedicated language was necessary. We then designed and implemented PetriScript [14]. Its purpose is to enable a high flexible design of Petri nets models using scripting techniques. It provides basic integers and string types, along with useful new built-in types such as lists of nodes. A key advantage of using PetriScript is the parametrization of the whole model to

be built when a particular configuration is selected for the case study. It is interesting to note that in the last release of Tina [19], a similar system has been introduced to build complex nets by composition, using place and/or transition labels (TPN).

Assembling operations

The assembling operations build a complete Petri net model from individual components. Four operations are used in our assembling scripts to assemble Well Formed Petri Nets components. Some of them are defined in [23]:

O1 *transition expansion*: it is the operation by which a sub-net is inserted into a super-net. Actually, the sub-net will replace the *sub-net transition* in the super-net. Consequently, the first and last elements of the sub-net connected to the super-net are transitions.
O2 *place fusion*: communication or abstraction places are merged between two or more components.
O3 *transition fusion*: synchronization transitions are merged between two or more nets.
O4 *Building* the net declarations, initial markings and guards. It means setting up the domains, computing the initial markings (which can be huge) and the guards according to the parameters for the selected configuration.

When assembling the complete model, rules we have defined are used to enforce its syntactic and semantic well-formedness.

4 Definition of a Case Study

In this section, we present an application of our modeling approach to a case study extracted from Intelligent Transport Systems.

4.1 Presentation of the Case Study

Let us provide a description of the system situation or *problematics*, a "Safe insertion" case study, illustrated by Fig. 3. This is a motorway with two lanes: L_1 (the rightmost one) and L_2. An entrance to the motorway, L_0, is connected to L_1. Vehicles are using the two lanes. We use the notation $V_{i,j,k}$, where i is the lane number, j is the position on the lane, and k is the identifier. $V_{0,j,k}$ vehicles are entering the motorway. We want to study a cooperative insertion of vehicles arriving in the entrance lane.

We now describe the properties we want to verify in this case study. We want to let $V_{0,j,k}$ vehicles get into the main traffic without violating the following properties:

1. distance between two vehicles in the same lane must be over a minimum safe distance to let drivers react to sudden events;

2. $V_{0,j,k}$ vehicles eventually get into the motorway;
3. keep $V_{i,j,k}$ vehicles from stopping.

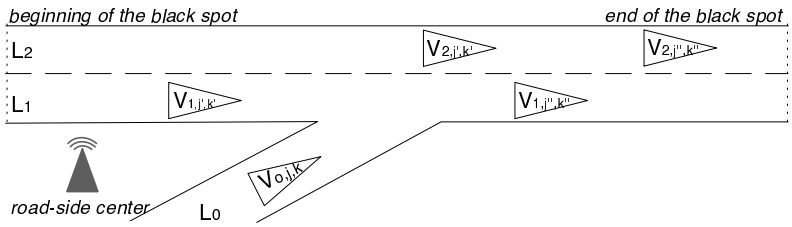

Fig. 3. Topology of the "Safe Insertion" case study

Now we describe a example scenario where the decision is mainly taken by the infrastructure. The motorway has a *road-side center* (called in the following RSC) that enables communication with vehicles and can compute commands related to safety or flow control.

Vehicles can get their positions using a satellite localization technology [4] (it may be combined with ground installations and digitized maps) and send them periodically to the infrastructure. Subsequently, the infrastructure is able to maintain a dynamic map of all vehicles in its range of communication.

The infrastructure and vehicles behaviors and interactions are split into three main steps : 1) vehicles get their positions from the context, 2) they send this information to the infrastructure and 3) when the infrastructure has all positions of vehicles, it issues commands to them according to its strategy.

We suppose in this study that all vehicles $V_{i,j,k}$ are equipped with communication devices and that the drivers follow instructions provided by the road-side center.

We also want to consider several configurations for this scenario: the density of the traffic in L_1, the existence of vehicles in L_2, and the management strategy in the road-side center (such as, trying to maintain vehicles circulating in L_1 or not, etc.).

4.2 Architecture of the model for the case study

The primary specification of the system architecture, shown in Fig. 4, is structured into eleven components.

In an infrastructure-based strategy, the range of the context we can manage is more likely to be larger than in a vehicles-centered one. Furthermore, the stress put on safety and reliability requirements for a "Safe Insertion" case study (because of the increased level of danger), leads to a strong involvement of the infrastructure in the decision process. Therefore, for this case study we

adopt an infrastructure-oriented approach and subsequently there are more components that describe the RSC and its strategy than for vehicles.

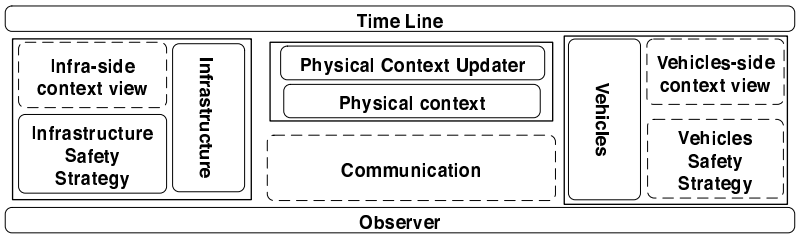

Fig. 4. Main architecture of the modeled system

This generic architecture is structured in two categories of components. The first one describes the continuous aspects of the system like the management of time and the required physical functions. The second category corresponds to the components we want to analyse like vehicles or the RSC.

The first category is composed of three components that model the discretization of the system:

- the *physical context*, which stores actual 'physical states' of vehicles (e.g., vehicles positions);
- the *physical context updater*, which implements our physical functions to update the *physical context*;
- the *time-line* that implements the time discretization of synchronizing all components at the end of each time frame. The time-line is divided in successive *time frames* that implement its discretization. It handles a "fair" execution of components in the system (i.e. no vehicle may execute more cycle than expected during a time frame).

Some components can only be executed within a time frame (e.g., vehicles or the infrastructure component), whereas others can act within or between time frames like the *observer* or the *physical context*.

The first category also comprises an *observer* which has to detect invalid behaviors of the model in terms of transitions that should not be fired or states that should not be reached.

In the second category, we have three components constitutive of the RSC, three others concerning *vehicles*, and a *communication medium* component:

- The *infrastructure* itself, describes the infrastructure behavior. This behavior is represented in terms of a chronological succession of interactions, communications or synchronizations with the components of the RSC or other components.

- The *infrastructure safety strategy* models the infrastructure decision making process, which computes commands or instructions to send to vehicles.
- The *infrastructure context view* represents what the infrastructure can see of the environment and is fed with data from communications with vehicles. It is used by the *infrastructure safety strategy* component to compute new commands. However, it may not be as accurate as the *physical context* if for example loss of data occurs during communications.
- *Vehicles* component holds vehicles behavior.
- *Vehicles context view* and *Vehicles safety strategy* act like their counterparts in the RSC.
- The communication medium component manages data exchanges or commands between vehicles and the infrastructure, and allows us to introduce loss of data.

4.3 Components selection and configuration for the first scenario

In our first scenario, we have decided to configure (or initialize) the *physical context* component so that all vehicles ($V_{i,j,k}$) are traveling along the rightmost lane (L_1 in Fig. 3) except vehicles ($V_{0,j,k}$) that are coming from the entrance lane. Initially, there is no vehicle in the second lane (L_2).

The time line is configured so as to enable random insertion of vehicles in the system with respect to the considered safety distance. Thus, vehicles are injected and removed from the system between each time frame.

We have chosen a particular *infrastructure strategy* which relies on all vehicles positions to achieve the "Safe Insertion". If a vehicle on the motorway has an invalid safety distance when the inserting vehicle arrives on the motorway, the infrastructure issues a 'change lane' command. According to that command, this vehicle must move from lane L_1 to lane L_2 where there is no vehicle. For the others, the infrastructure sends a 'nop' command. It means that they should keep their current traveling parameters.

We first consider a level of abstraction where communications are idealistic, thus there is no loss of data. Hence, the *communication medium* component is not selected. We also consider at this level of abstraction that the *infrastructure context view* is refreshed at the same rate as the *physical context*. Thus, the *infrastructure* can directly use the *physical context* since it does not need its own context component. Consequently, the *physical context* component is in fact the same as the infrastructure's . The same assumption was made for *vehicles context view*. Finally, in this scenario, we also consider that vehicles are fully cooperative and execute immediately commands they receive. Hence, no smart behavior is selected. In Fig. 4, selected components are represented with a continuous line.

This scenario, and its corresponding infrastructure strategy are simple as this is a way to assess our methodology.

5 Modeling and assembling components for the scenario

5.1 Modeling the components

We now have defined the generic architecture of the model for the case study. Then we have chosen some components from the library and have configured them according to the first scenario architecture. We are now describing each specific behavior in this section, along with the interactions between the components.

In the Petri net models of the presented components, sub-nets interfaces are depicted by squares. while synchronization interfaces are depicted by longer transitions. Communication interfaces are represented by bigger circles. Normal notation is used for conventional (i.e., local) places and transitions.

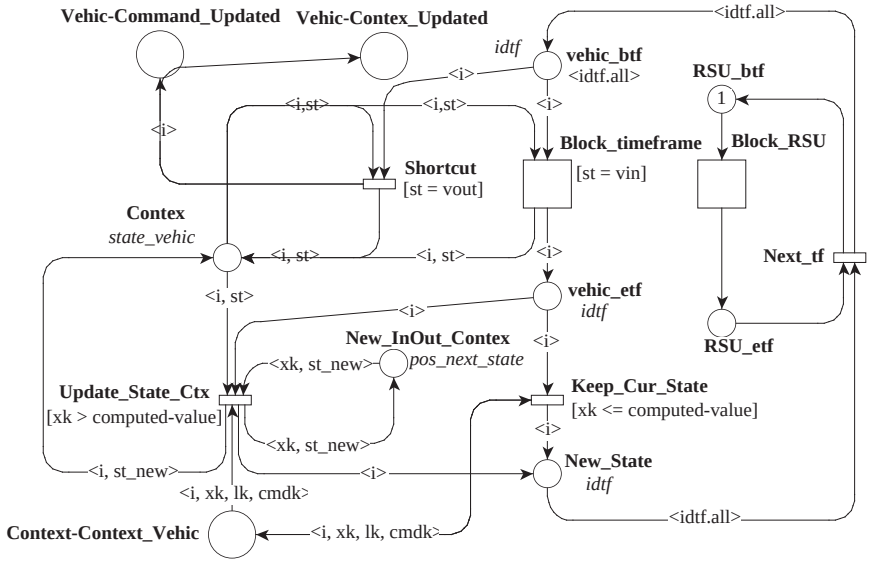

Fig. 5. One Petri Net component of the time line

Time line component

The time line, shown in figure 5, provides two essential functions:

- the first one implements elapsing of time and components synchronization in the system;
- the other one implements vehicles injection and removal of the system between each time frame.

The purpose of the first function is to handle a fair execution step for all the other components in terms of discretized time. It thus synchronizes the infrastructure and vehicles within each time frame simultaneously. The Petri net model representing the *infrastructure* component is inserted into transition *Block_RSU* and the one representing *vehicles* into transition *Block_timeframe* (see section 5.2 for more details about this operation). At the end of each time frame, the time line waits for all synchronized components at transition *Next_tf* before the next time frame can begin.

Between two time frames, the time line component is able to remove or add vehicles to the motorway using *Update_State_Ctx*, *New_InOut_Context*, *New_State*, *Contex* and *Shortcut* transitions and places. For instance, as shown in figure 5, vehicles that are simply taken out of the motorway are tagged with "vout" color and effectively removed with transition *Shortcut*.

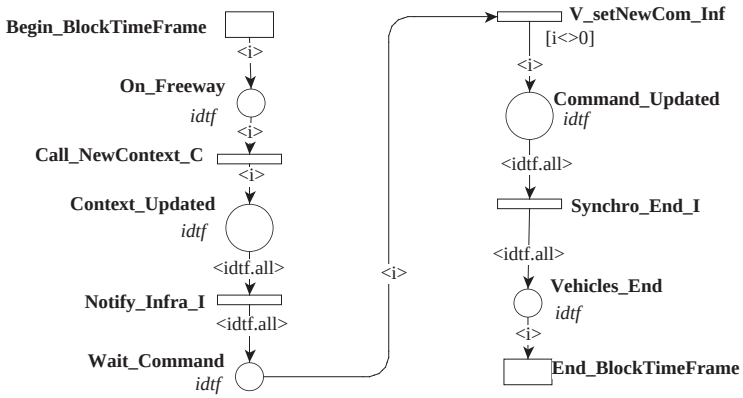

Fig. 6. One Petri Net component for vehicle

Vehicles component

This component, shown in Fig. 6, is included in transition *Block_timeframe* of the *time line*. Concretely, transition *Block_timeframe* is replaced by the Petri net model of vehicles component (details about this operation are provided in section 5.2).

Vehicles perform three main different operations within each time frame.

- First of all, they "get" their positions. This is implemented by the *physical context updater* updating the *context*. To do so, we merge transition *Call_NewContext_C* of vehicles component with the corresponding one of the *physical context updater*.
- Afterwards, they all "send" their positions to the RSC. In fact, the RSC is notified that their positions are ready to be retrieved. This operation

is performed by merging transition *Notify_Infra_I* with the corresponding one of the *infrastructure* component.

- Finally, they "wait" for the RSC's decisions at transition *V_setNewCom_Inf*, which is merged with transition *S_getNewCom_Inf* of the *infrastructure safety strategy* component (see Fig. 7). Vehicles acknowledge reception of commands by merging transition *Synchro_End* with the corresponding one of the infrastructure.

Infrastructure safety strategy component

To compute decisions, *safety strategy* needs to know the current context of inserting vehicles ($V_{0,j,k}$). This context is then retrieved using the synchronization on transition *S_getVOSt_Inf* and the communication place *V0_State* (see Fig. 7).

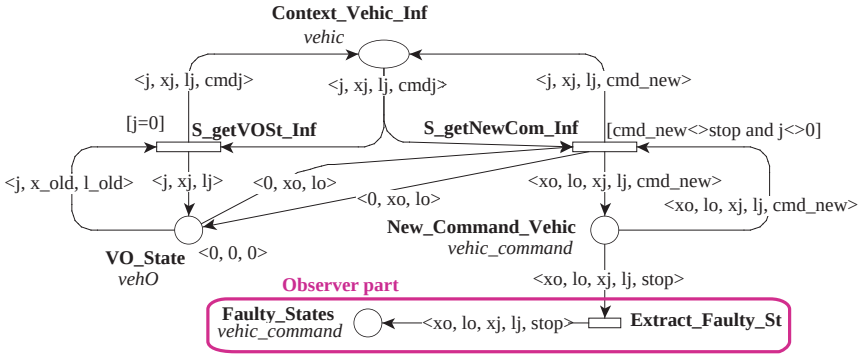

Fig. 7. One Petri Net component of the infrastructure

From that point and in full cooperation with the *context* component, each vehicle context is retrieved and compared to that of the inserting vehicles $V_{0,j,k}$ and the appropriate command issued to that vehicle. That command is either *nop*, or *chglane*.

Since this component implements the decision functions of the RSC, it also follows the design technique introduced in section 3.3 and illustrated by figure 2, Consequently, the marking of place *New_Command_Vehic* is automatically generated using PetriScript.

Physical context updater and physical context components

Context updater implements physical functions to update vehicles physical context, using the technique depicted in section 3.3.

These physical functions are in initial markings of the component. They list the position values of vehicles according to physical parameters, such as velocity, road conditions, etc. These markings are generated using PetriScript.

Physical context component stores current 'physical states' of vehicles (i.e., for instance, vehicles positions) in place *Context_Vehic_Inf*.

The observer component

The observer captures invalid behaviors we are interested in by looking up states that point them out. The key advantage of using an observer is that it is not intrusive for the modeled system. Therefore, it does not affect the behavior framework of the system.

In Fig. 7, the observer is represented by transition *Extract_Faulty_St* and place *Faulty_States*. If the implemented *safety strategy* is valid, w.r.t. rules defined in sect. 4. then this place is supposed to have no marking in the system state space.

5.2 Assembling a configuration

The assembling is performed in four main steps, as expressed in section3. We use the assembling operations defined in section 3.4.

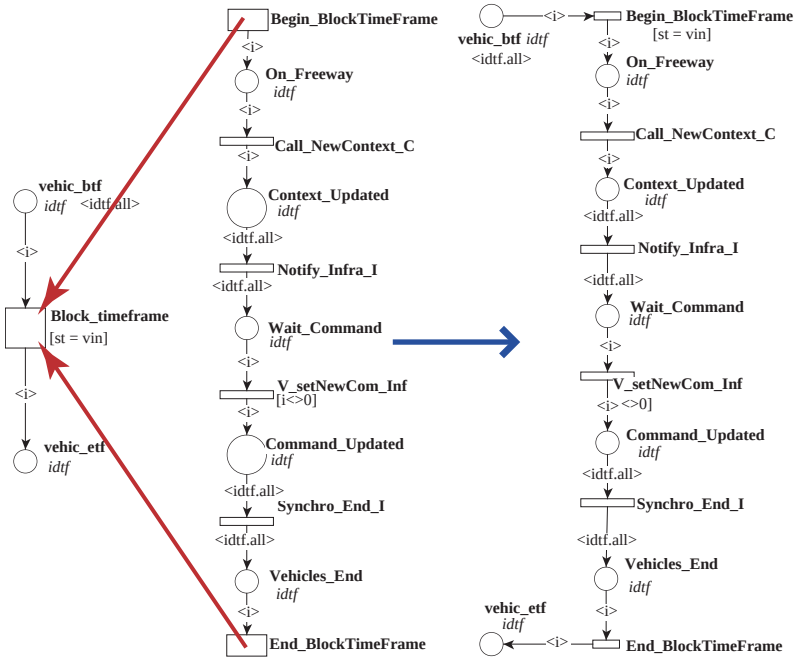

Fig. 8. Insertion of vehicles component into the time line

1. The first step corresponds to the scenario specification architecture depicted in Fig. 1. This first step is completed through six actions:
   a) *Inserting* the infrastructure and vehicles components as *sub-nets* into the time line, applying operation O1. Vehicles component is inserted into transition *Block_timeframe*, as depicted by Fig. 8, and the infrastructure component into *Block_RSU*. Actually, *Block_timeframe* and *Block_RSU* are replaced by the new inserted components. The previous links are redirected onto and from the new inserted components.
   b) *Connecting* the physical context and infrastructure safety strategy components by applying O2. According to the configuration described in Sect. 4.3, the infrastructure's vision of the context is based on the physical context component.
   c) Applying O3 to synchronize vehicles component with the physical context and the physical context updater in order for them to have their context updated at each execution step. This synchronization is set upon transitions *Call_NewContex_C* (vehicles side) and *C_getNewContext_V* (physical context side).
   d) We also synchronize vehicles component with the infrastructure safety strategy component, upon transitions *V_setNewCom_Inf*, *S_getNewCom_Inf* and *Call_NewCom_SV*. This enable commands issuance from the infrastructure to vehicles, based on their physical context with respect to the current position of V_0.
   e) Vehicles notify the infrastructure with their updated context through the synchronization of transitions *Notify_Infra_I* (vehicles side) and *Notify_Infra_V* (infrastructure side).
   f) For the sake of fairness of processing all vehicles within each execution step, they also synchronize with the infrastructure when all commands have been issued. This synchronization is set upon transitions *Synchro_End_I* (vehicles side) and *Synchro_End_V* (infrastructure side).
2. During the second step, we define or we select the system components relevant for the elaborated scenario, according to the architecture. These components specify behaviors we want to analyze, for example the safety strategy coupled with both cooperative and non-cooperative vehicles. From this point of view, we consider that:
   - either the components library (Fig. 1) is already populated with the different versions of the components that we need and in which case we just select the relevant components,
   - either we do not have them yet and in which case we specify them. We may reuse existing components to design new variants, the ground behaviors of which are similar to the previous ones. For example, non-cooperative smart vehicles would differ from cooperative ones by having no synchronization to receive commands from the infrastructure.
3. The third main step corresponds to the scenario configuration, illustrated in Fig. 1.

a) The time line component has the ability to remove or add a vehicle to the motorway, as stated in its description in Sect. 5.1. To enable this behavior, it must have access to vehicles physical context between each execution step. Therefore, operation O2 is applied to connect place *Context_Vehic* (physical context side) to transition *Update_State_Ctx* (time line side).
b) The observer is connected to the safety strategy component of the infrastructure, by applying O3 (see Fig. 7).
c) By operation O4, we compute for one variant of the scenario declarations and initial markings from parameters such as the length of the black spot, the number of vehicles on or out of the motorway, the position at which the entrance lane joins the main motorway, etc. Guards are set to the transitions where needed, for example enabling the observer to detect invalid behaviors of the system by setting a guard on *S_getNewCom_Inf* and *Extract_Faulty_St* (see Fig. 7).

4. The last main step corresponds to the generation of the assembled large Petri net model (Fig. 1), by putting the complete specification encoded in PetriScript into action.

The following PetriScript example performs the insertion of vehicles component into the time line, presented in Fig. 8.

```
-- ------------------------------------------------------------------
-- inserting Vehicles module into Time Line
-- ------------------------------------------------------------------
delete (place "vehic_btf" , transition "Block_timeframe");
connect "<i>" place "vehic_btf" to transition "Begin_BlockTimeFrame";

delete (transition "Block_timeframe" , place "vehic_etf");
connect "<i>" transition "End_BlockTimeFrame" to place "vehic_etf";

delete (place "Contex" , transition "Block_timeframe");
connect "<i, st>" place "Contex" to transition "Begin_BlockTimeFrame";

delete (transition "Block_timeframe" , place "Contex");
connect "<i, st>" transition "Begin_BlockTimeFrame" to place "Contex";

set transition "Begin_BlockTimeFrame" to guard "[st=vin]";

delete transition "Block_timeframe";
```

6 Analysis of the assembled specification for the case study

For the configuration depicted in Sect. 4.3, we have quite simple modules, as their statistics shows in the table below.

Component name	number of places	number of transitions
time line	7	6
vehicles	5	6
infrastructure	4	6
physical context	1	0
context updater	1	1
safety strategy	3	2
observer	1	1
assembled model	21	14 (63 arcs)

The assembled model has a reasonable size. So far, the variations we experimented for this scenario consist in modifying the number of vehicles in the system to evaluate how it behaves when the traffic becomes dense. For example, it is of interest to evaluate when the infrastructure may decide to reduce the entrance speed in the black spot.

However, despite this reasonable size, there is a clear progress when we consider the time we spent for building it, with respect to the complexity of the system: a few days to understand the specification, elaborate the architecture, build the Petri net components and write the assembling script. Moreover, the architecture remains for later analysis of variations in the system.

Analysis of the assembled model was performed with CPN-AMI [13]. The following properties were validated:

- Structural bound analysis for places marking (by unfolding the Petri net into an equivalent P/T net) showed that the net is bounded. It is of interest to note that the unfolding tool allowed us to detect a bad arc marking in the infrastructure component.
- Model checking was performed using small color domains using PROD [10] and GreatSPN [9] tools (as they are integrated in CPN-AMI). With some difficulties due to the extreme complexity of the system, we could prove that our system has no deadlock.

There is an issue for the analysis of such models. In particular, the technique adopted to model complex physical function introduces:

- very large markings that are not appropriately handled by the investigated tools,
- local asymmetries that partially disable the use of static classes in GreatSPN (this corresponds to dynamic subclasses).

However, places representing complex functions have a stable marking, thus, a simple optimization in the model checker will allow for representing these places only once in the memory (instead of representing it for each state in the system). Important memory space could then be saved. Such an optimization can be achieved by encoding the state space using decision diagrams.

Moreover, handling of dynamic symmetries should provide nice results for this type of systems. This enforces the choice for Well Formed Petri Nets we

motivated in Sect. 3.2. Combined with data decision diagrams to encode the symbolic reachability graph, another order can be gained [21], leading to the analysis of very large systems. Some experiments have provided good results and this emerging approach needs to be implemented.

A new model checker exploiting these techniques is under development at LIP6. Theoretical aspects are already defined and partially experimented in cooperation with the developers of GreatSPN [1]. Our purpose is to be able to perform reachability analysis (e.g., is the minimum safety distance between two vehicles respected), as well as the evaluation of temporal logic formulae (e.g., if a vehicle gets into the entrance lane, will it eventually get into the motorway).

7 Conclusion

In this paper, we presented a modeling methodology dedicated to large systems. Similarly to programming, the idea is to build a model architecture and then to fill the gaps, with possibly several alternative solutions. This allows to analyze several variations of a design.

A first implementation of our methodology is integrated into CPN-AMI by means of recent tools. The PetriScript interpretor allows us to write assembling scripts easily. A new optimized Petri net unfolder allows us to handle large models and take benefits of some structural tools like the computation of structural bounds in the system.

Such an approach is particularly appropriate for Intelligent Transport Systems (ITS) where very complex situations have to be analyzed. In such projects, the analysis is driven by case studies so as to handle more efficiently the complexity of identified issues in the domain.

We thus applied this methodology to an ITS case study as an assessment of our modeling process. The result is satisfactory. Both the approach and the PetriScript language, developed after a previous similar experiment on the PolyORB middleware, appear to be appropriate for our purpose.

Moreover, as mentioned in the paper, this work raises several interesting open issues we will investigate in future work:

1. if the complexity of the system can be handled thanks to a smoother modeling process, there is a problem of analyzing such very large specifications. Involved modeling techniques must be studied carefully to extract the relevant optimizations (i.e., the ones that are often activated). Then, it is necessary to consider both the modeling and the analysis phases as a whole process to identify and implement appropriate optimizations in the analysis with regards to the selected modeling techniques.
2. some elements of the demonstration, such as the correctness of complex functions' discretization must be investigated separately. Our methodology will have to consider this and provide hints to perform this task.

3. if several scenarios are investigated with different levels of abstraction, there is a need to ensure that a more precise level of abstraction is an appropriate refinement of the previous one. For example, the analyzed configuration relies on a "perfect network". If a new configuration includes the modeling of the network, it is important to verify that a correspondence between the two configurations is maintained. Links with algebraic specification should be interesting to investigate for that purpose.

The use of Petri Nets has given us the ability to make formal verification, and our modular approach helped us to obtain an easy changeable, reusable and modular model.

The most critical problem faced in this study is first the integration of the time-constrained and physical context environment. It is handled by a component that synchronizes other components between time frames and a complex function integration technique. Then the definition of components was based on their roles and interfaces with other components. To achieve formal analysis we were forced to update our tools.

Finally, it seems that ITS provide numerous interesting situations to be analyzed. Moreover, several projects in this area are investigating scenarios and solutions to be evaluated. This is a challenge for formal methods to be able to handle such problems.

References

1. S. Baarir, S. Haddad, and J-M. Ilié. Exploiting Partial Symmetries in Well-formed nets for the Reachability and the Linear Time Model Checking Problems. In *Proc. of WODES'04 - IFAC Workshop on Discrete Event Systems, part of 7th CAAP*, Reims - France, 2004. Springer Verlag.
2. S. Bernardi, S. Donatelli, and J. Merseguer. From uml sequence diagrams and statecharts to analysable petri net models. In *proceedings of the 3rd international workshop on Software and performance*, pages 35–45, New York, NY, USA, 2002. ACM Press.
3. R. Bishop. Intelligent Vehicle R&D: a review and contrast of programs worldwide and emerging trends. In Jacques Ehrlich, editor, *Annals of Telecommunications - Intelligent Transportation Systems*, volume 60, pages 228–263. GET-Lavoisier, March-April 2005.
4. J-M. Blosseville. Driving assistance systems and road safety: State-of-the-art and outlook. In Jacques Ehrlich, editor, *Annals of Telecommunications - Intelligent Transportation Systems*, volume 60, pages 281–298. GET-Lavoisier, March-April 2005.
5. P. Bly. e-safety - co-operative systems for road transport (ist work progamme 2005-2006). Technical report, European Commission, 2004.
6. L. Cabac and D. Moldt. Formal semantics for auml agent interaction protocol diagrams. In J. Odell, P. Giorgini, and J. P. Müller, editors, *5th International Workshop on Agent-Oriented Software Engineering*, number 3382 in LNCS, pages 47–61, 2004.

7. G. Chiola, C. Dutheillet, G. Franceschini, and S. Haddad. On Well-Formed Coloured Nets and their Symbolic Reachability Graph. *High-Level Petri Nets. Theory and Application, LNCS*, 1991.
8. C. Dutheillet, I. Vernier-Mounier, J-M. Ilié, and D. Poitrenaud. *State-space-based methods and model checking*, chapter 14, pages 201–276. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.
9. GreatSPN: Graphical Editor, Analyzer for Timed, and Stochastic Petri Nets. `http://www.di.unito.it/ greatspn`.
10. PROD: An Advanced Tool for Efficient Reachability Analysis. `http://www.tcs.hut.fi/Software/prod`.
11. J. Gogen and Luqi. Formal methods: Promises and problems. *IEEE Software*, 14(1):75–85, 1997.
12. S. Haddad. *Issues in verification*, chapter 13, pages 183–200. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.
13. A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. New features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. In *6^{th} International Conference on Application of Concurrency to System Design (ACSD'06)*, Turku, Finland, June to be published in 2006. IEEE Computer Society.
14. A. Hamez and X. Renault. *PetriScript Reference Manual*. LIP6, www-src.lip6.fr/logiciels/mars/CPNAMI/MANUAL_SERV.
15. R. Horowitz and P. Varaiya. Control design of an automated highway system. In *IEEE Proceedings on Special Issue on Hybrid Systems*, volume 88, pages 913–925, July 2000.
16. J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume ENTCS 133, pages 139 – 157. Elsevier, 2004.
17. Intelligent Vehicle Initiative. Saving lives through advanced vehicle safety technology, September 2005.
18. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer Verlag, 2nd edition, 1997.
19. LAAS. TINA: TIme petri Net Analyzer (version 2.8.0), `http://www2.laas.fr/tina/description.php`.
20. Y. Thierry-Mieg, C. Dutheillet, and I. Mounier. Automatic symmetry detection in well-formed nets. In W. M. P. van der Aalst and E. Best, editors, *24th International Conference on Applications and Theory of Petri Nets 2003*, volume 2679 of *LNCS*, pages 82–101. Springer Verlag, 2003.
21. Y. Thierry-Mieg, J-M. Ilié, and D. Poitrenaud. A symbolic symbolic state space representation. In D. de Frutos-Escrig and M. Núñez, editors, *FORTE*, volume 3235 of *LNCS*, pages 276–291. Springer Verlag, 2004.
22. R. Valk. *Basic definitions*, chapter 4, pages 41–51. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.
23. M. Voorhoeve. *Techniques*, chapter 9, pages 106–117. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.