Introducing the Modeling and Verification process in SysML

Marcos V. Linhares[*], Rômulo S. de Oliveira, Jean-Marie Farines
DAS – UFSC
Campus Universitário – Trindade
88040-900 Florianópolis/SC – Brazil
{marcos, romulo, farines}@das.ufsc.br

François Vernadat
LAAS – CNRS
University of Toulouse
7, avenue du Colonel Roche
31077 Toulouse – France
francois@laas.fr

Abstract

The development process of complex systems needs to take in account differents domains and aspects. SysML (Systems Modeling Language) is a new modeling language that allows a system description with various integrated diagrams (as structure, behavior and requirements diagrams), but SysML lacks formality for the requirement verification. The aim of this paper is to propose an approach to verify complex systems using SysML as a language which describes the system structure and requirements. Petri nets and temporal logic LTL are used respectively to formalize the system behavior and requirements. The benefit of such formalization is to allow an automatic formal verification. In order to demonstrate this methodology, it will be used a factory automation system, modeled by SysML and Petri nets, and verified by the TINA toolbox.

1. Introduction

With the increase of the complexity and diversity of the industrial applications, the need for collaboration appears in the development process, since it involves knowledge in the areas of software, mechanical, electrical and electronic engineering. The integration of these areas will result in a complex system product. However, each area uses different methodologies and development tools. That makes difficult the understanding of each part of the system development by project collaborators of different areas. System engineering has to use a shared methodology to integrate the products of different engineering areas.

The System Modeling Language (SysML) is a semi-formal language that intend to support the specification, analysis, design and verification of complex systems [9]. It allows to capture informations in a precise and efficient way, so as to facilitate integration and reuse in a larger context. It should also support several activities such as: to analyze and to evaluate the specified systems, to identify and to provide requirements of the system, to distribute projects and to support exchange among them; to communicate system informations, correctly and consistently among several participants of the same project (software, mechanical, electrical and other engineers).

A semi-formal modeling is easy to use and permits to quickly obtain a preliminar specification of a complex system including architecture, behavioral and requirements aspects. The counterpart of this kind of modeling is the lack of formalization. In the other hand the formal models that make a well defined behavior modeling, system properties specification, and allows verification, but requires some expertise.

An early stage in the development of any complex system is to specify its requirements. For example, several conceptual models can be used in order to understand and to organize the requirements in a systematic way. Also, there are several ways of describing the behavior of a system. The description in natural language may contain ambiguities. It is essential to insert formal or semi-formal models, that are capable of specifying the requirements, making it possible the use of system design automated methods [7]. The SysML requirements diagram provides a way to make that. But, the semi-formality of this diagram does not allow the verification task of systems. Although a *verify* stereotype is used in the SysML requirements diagram, it is not well adapted for requirement verification.

The modeling and verification process, proposed in this paper, gets the better of the two worlds (semi-formal and formal). This process is shown in the figure 1, it is based on a first semi-formal modeling task before a formal modeling and verification task. This paper intends to be more clear about this process by using a factory example.

The aim of this paper is not to give a formal semantics of various SysML diagrams but focusses on the SysML framework proposed to capture, refine and trace requirements. We use temporal logic to progressively formalize SysML requirement diagrams. We illustrate the whole approach by considering a factory automation system. As formal verification also requires a formal description of the component behaviours, we use Petri nets instead of

[*] This author is supported by CAPES/COFECUB and CNPq.

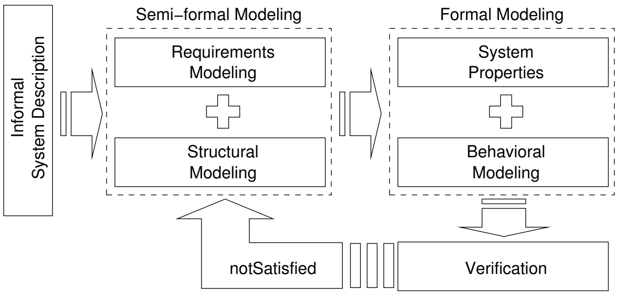


Figure 1. Modeling and Verification process.

standard activity diagrams. The global behaviour of the system is obtained by composing elementary components according the system architecture. System description, including behavioural aspects and requirements expression, is analysed using the TINA toolbox [4].

In section 2, the SysML language is briefly presented and especially, the requirement diagram is described. In section 3, some concepts of formal modeling (with Petri nets) and verification (with LTL formulas) are presented and the TINA toolbox is described. The approach joining the SysML language and Petri nets is presented and applied in a factory automation example in section 4. Section 5 presents the final considerations and future works.

2. The SysML Language

SysML is a general purpose modeling language for system engineering applications. It establishes a description pattern for a great variety of complex systems. These systems may include hardware, software, data, methods, personal and instruments.

It was being defined based on UML 2.0, using its syntax and semantics. That will improve the communication among the several designers that participate in the development of the system, promoting interoperability among modeling tools, not only for software and hardware design but also for the other parts of the system, as electrical, mechanical ones and so on [10].

Figure 2 shows the modifications in the diagrams reused from UML 2.0 as well as the new diagrams of SysML. The specification of SysML is classified in three basic model types: the structure models, the behavior models and the requirement models. For each one there are defined constructions that are used in a specific model. Some constructions can be used together with several model types (called cross-cutting constructions).

The SysML diagrams represent the model elements such as packages, blocks and associations. They allow to reuse UML diagrams without modification. Moreover, the extensions of these diagrams aim at meeting the specific requirements of system engineering [10]. The following UML behavior diagrams are included in SysML without modifications: state machines, interactions (sequences) and use cases. Other UML diagrams such as activities, classes are reused with the addition of some extensions. Some new diagrams as requirements, parametric and allocation diagrams were also added.

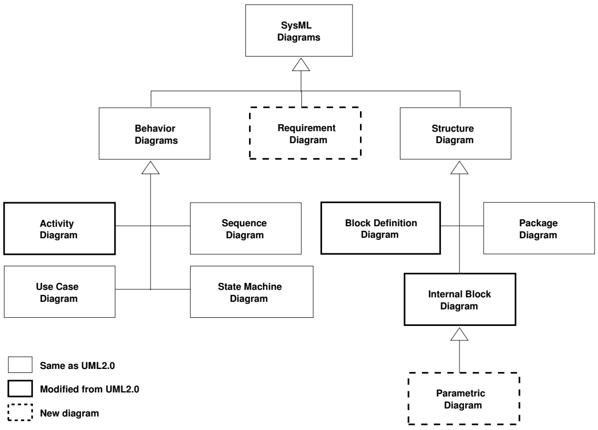


Figure 2. SysML diagrams [10]

The structural constructions define the static and structural elements used in SysML. The diagrams that include the structural constructions are: Package Diagram (to partition the system), Block Definition Diagram (to define the block features and relationships), Internal Block Diagram (to show the internal structure of a block) and the Parametric Diagram (a restricted form of internal block diagram containing constraint properties and parameters). Despite of some similarities with the UML models these contructions brought several improvements.

The behavioral constructions specify the dynamic parts used in the behavior diagrams of SysML, including: the Activity Diagram (used to describe the control flow), the Sequence Diagram, the State Machine Diagram and the Use Case Diagram, the same ones used in UML with little or none modification. The most important improvement in the behavioral models is to make possible the modeling of continuous time by the activity diagram.

Finally, another diagram which does not exist in UML, the Requirement Diagram allows the system requirement description.

2.1. SysML diagrams for the proposed approach

This approach uses SysML to perform the structural (block definition diagram and internal block diagram) and requirement modeling (requirement diagram).

The block definition diagram (BDD) and the internal block diagram (IBD) describe the external and internal system or subsystem structure using a block as its basic unit. They allow the description of which elements are interconnected and how they are interconnected. They also allow a top-down and/or a bottom-up modeling while giving an abstract/concrete system level to different system elements.

The relationships among the BDD elements (composition, inheritance, aggregation and others) and the diagram format are the same ones used in the UML class diagram. With this diagram it is possible to visualize all the parts that compose the system and the relationship among them in an abstract way. The IBD describes the internal structure of a block, its parts and the connections among them. The connections are made through the use of flow ports, where matter, energy or data may flow, and service ports that connect the services provided and/or requested by a block.

Unfortunately, to describe a complex system a better organization is necessary because the number of BDD and IBD diagrams and model elements (blocks, ports, ...) can increase very quickly, depending on the system to model, and the view to show.

The requirement diagram is developed from system engineer needs and they present an important role during the system modeling. It shows how the requirements are satisfied by the system elements. Those requirements may be provided by the user, the environment, or by the system itself. Requirements are described through sentences based on natural language. In this diagram they can be grouped in a clear way, also documenting their origin (standards or more detailed specifications) and tracing their destination. The requirements also can be used to provide useful acceptance tests before the system deployment.

However, the requirements diagram is very abstract. Requirements are inserted in the model as text using natural language. It presents the same problems of an informal specification. Moreover, the sentences in natural language can describe structural or non-structural requirements, which directly affect the system architecture and behavior characteristics. It presents a high degree of informality and consequently the verification of those requirements becomes a very hard task. Misunderstandings and inconsistencies can happen during the extraction of information from this diagram.

3. The Petri Net Model and Tool

Petri nets and Time Petri nets [8] are one of the most widely used model for the specification and verification of real-time systems. Time Petri nets are Petri nets in which a nonnegative real interval $I_s(t)$, with rational end-points, is associated with each transition t of the net [2]. Function I_s is called the *Static interval* function.

$\mathbf{R}^+$ and $\mathbf{Q}^+$ are the sets of nonnegative reals and rationals, respectively. Let $\mathbf{I}^+$ be the set of nonempty real intervals with nonnegative rational end-points. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left end-point, and $\uparrow i$ its right end-point (if i bounded) or ∞. For any $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ denotes the interval $\{x - \theta | x \in i \wedge x \geq \theta\}$.

Definition A *Time Petri net* (or *TPN*) is a tuple $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s\rangle$, in which $\langle P$, T, $\mathbf{Pre}$, $\mathbf{Post}$, $m_0\rangle$ is a Petri net, and $I_s : T \rightarrow \mathbf{I}^+$ is a function called the *static interval* function. Where, P is the set of *places*, T is the set of *transitions*, $\mathbf{Pre}, \mathbf{Post} : T \times P \rightarrow \mathbf{N}^+$ are the *precondition* and *postcondition* functions, $m^0 : P \rightarrow \mathbf{N}^+$ is the *initial marking*.

Time Petri nets add to Petri nets the *static interval* function I_s, that associates a temporal interval I with every Petri net transition t (with $t \in T$), where $I_s(t) \in \mathbf{I}^+$. A Time Petri net example is shown in Figure 3. This TPN represents a production worker (a worker that works in a production line) with $P = \{Wi\_idle, Wi\_work, Wi\_workExcess\}$ (places) and $T = \{Li\_startLine, Li\_endLine, Wi\_workTime\}$ (transitions). The $\mathbf{Pre}$ and $\mathbf{Post}$ functions place the arrows as it is showed in the figure 3 (for example, $\mathbf{Pre} = (Li\_endLine, Wi\_idle) = 1$ and $\mathbf{Post} = (Li\_startLine, Wi\_idle) = 1$). His initial marking ($m^0$) is Wi_idle when he is able to go to work. His working time depends on the line working time (between $Li\_startLine$ and $Li\_endLine$, more detailled in the section 4) but, if he works more than 35 minutes in a production line ($I_s(Wi\_workTime) = [35, 35]$) the transition $Wi\_workTime$ is fired and the worker go to the state $Wi\_workExcess$. The verification phase will allow to check if this property hold or not.

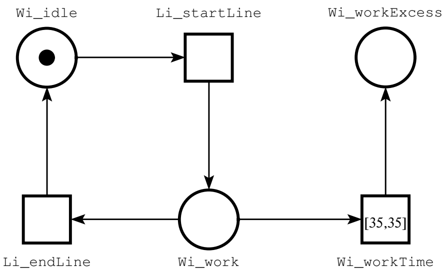


Figure 3. A Petri net example.

States, and the temporal state transition relation $\xrightarrow{t@\theta}$, are defined as follows:

Definition A *state* of a TPN is a pair (m, I) in which m is a marking ($m \in P$) and I is a function called the *interval* function. Function $I : T \rightarrow \mathbf{I}^+$ associates a temporal interval with every transition t ($t \in T$) enabled at m. We write $(m, I) \xrightarrow{t@\theta} (m', I')$ iff $\theta \in \mathbf{R}^+$ and:

1. $m \geq \mathbf{Pre}(t) \wedge \theta \geq \downarrow I(t) \wedge (\forall k \in T)(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
2. $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
3. $(\forall k \in T)(m' \geq \mathbf{Pre}(k) \Rightarrow$ $I'(k) = \textbf{if } k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k) \textbf{ then } I(k) \dot{-} \theta \textbf{ else } I_s(k))$

Definition The *state graph* of a TPN is the structure $SG = (S \xrightarrow{t@\theta} s_0)$, where:

$S = \{s | s_0 \xrightarrow{*} s\}$ is the set of states reach able from the initial state s_0

$s_0 = (m_0, I_0)$, where $I_0(t) = I_s(t)$ for any t enabled at m_0.

Many techniques for analysis of Time Petri nets proceed by building a labeled transition system (LTS) preserving the properties of interest (e.g. reachability set, deadlocks), in a first step, and then checking on this LTS the properties to be satisfied. Considering Time Petri nets, the main difficulty concerns the obtention of a finite state space.

Transitions may fire at any time in their temporal intervals, so states typically admit an infinity of successors. As with many formal models for realtime systems, the state spaces of Time Petri nets are typically infinite. Model checking Time Petri nets first requires to produce finite abstractions for their state spaces. Labeled transition systems that preserve some classes of properties of the state space, are so built.

Different state class constructions have been proposed and are available in TINA, preserving different families of properties of the state space. *State class graph* construction [3] preserves markings of the TPN and all its properties one can express in linear time temporal logics like LTL. [5] presents alternatives preserving states and bisimilarity with the state graph.

3.1. Model-Checking

We use $State/Event - LTL$ [6], a linear time temporal logic supporting both state and transition properties. The modeling framework consists of labeled Kripke structures (the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. Formulae Φ of $State/Event - LTL$ are defined according the following grammar (here p ranges over P and a ranges over Σ):

$$\Phi \quad ::= \quad p \quad | \quad a \quad | \quad \neg\Phi \quad | \quad \Phi \quad \vee \quad \Phi \mid \bigcirc \ \Phi \mid \Box \, \Phi \mid \Diamond \, \Phi \mid \Phi \; U \; \Phi$$

Example Example of $State/Event - LTL$ formulae :

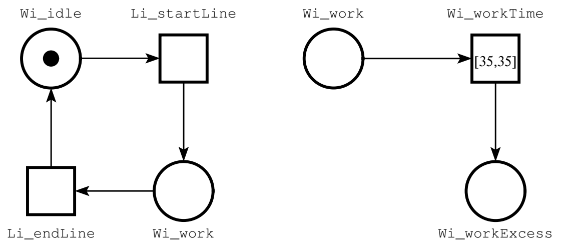
	(For all paths)
P	P holds at the beginning of the path,
$\bigcirc P$	P holds at the next step,
$\Box P$	P globally holds,
$\Diamond P$	P holds in a future step,
$P \; U \; Q$	P holds until a step is reached where Q holds

3.2. Tina toolbox for Time Petri Nets Verification

TINA (TIme Petri Net Analyzer[1]) is a software environment to edit and analyze Petri nets and Time Petri nets. This paper overviews its capabilities, architecture, and main applications. More details can be found in [4].

In addition to the usual editing and analysis facilities of similar environments, TINA offers various abstract state space constructions that preserve specific classes of properties of the state spaces of nets, like absence of deadlocks, linear time temporal properties, or bisimilarity. For untimed systems, abstract state spaces helps to prevent combinatorial explosion. For timed systems, abstractions are mandatory as their state spaces are typically infinite, TINA implements various abstractions based on state classes.

The TINA toolbox offers the capacity to construct complex specification in a compositional manner. Each transition or place may be labelled and a Petri net composition is performed according the transitions or places merging with the same label. This facility allows to a modular specification for each part of the system. The figures 4(a) and 4(b) show the Petri nets used to compound the Petri net ilustrated before in figure 3, it can be observed a place merging labeled as *Wi_work*.



(a) (b)

Figure 4. Petri nets composition.

The different tools constituting the environment can be used alone or together:

- *nd* (*NetDraw*): is an editing tool for automata and time Petri nets, under a textual or graphical form. It integrates a "step by step" time Petri nets simulator.
- *tina*: this tool builds the state space of a Petri nets, timed or not. TINA can perform classical constructs (marking graphs, covering trees) and also allows abstract state space construction, based on partial order techniques.
- *selt*: to check more specific properties than the general ones such as boundedness, deadlocks, pseudo liveness and liveness already checked by TINA. The *selt* tool is a *model-checker* for temporal logic extension formulae (State/Event LTL - *seltl*) [6].

Realtime properties, like those expressed in Timed Computation Tree Logic $TCTL$ [1] could be checked by using the standard technique of observers. The technique is applicable to a large class of realtime properties and can be used to analyze most of the "timeliness" requirements found in practice. This facility is used later in order to allow the verification of deadline properties. For example, looking to the figure 3 it is necessary to know if there is a moment that the production worker works more than 35 minutes; in this case, it is possible to use the following LTL formula: $\Box\neg Wi\_workExcess$. In case of non satisfiability, `selt` is able to build a readable counter-example sequence which presents an execution violation of the requirement.

[1] http://www.laas.fr/tina

4. A Modelling and Verification Example

We used a simple factory plant example to show how to join these two modeling languages in a single approach that allows the high level modeling with both semi-formal and formal modeling language. The aim is to use part of SysML language to model structure and requirements, together with Petri net to describe the behavioral part. After that, the formal requirement verification was performed.

4.1. Structural modeling

The example is based on a simple factory plant. The factory plant objective is to manufacture products and, for that, workers and machines are available. The products (P_x) are made by the production lines (L_x) that put the machines (M_x) in a specific production order. The workers are divided in two types: a) the production workers (W_x), that use the machines and work in the production lines and; b) the technicians (T_x), that make the machine maintenance.

Figure 5 illustrates the SysML block definition diagram for the factory plant. It shows the factory structure in an abstract way. Looking to this figure, it is possible to know the elements that compose the factory and their relationship with each other without too many details. For example, a factory is composed by one or more machines (composition relationship); the production lines use the machines and the production workers in the product manufacturing task (dependency relationship).

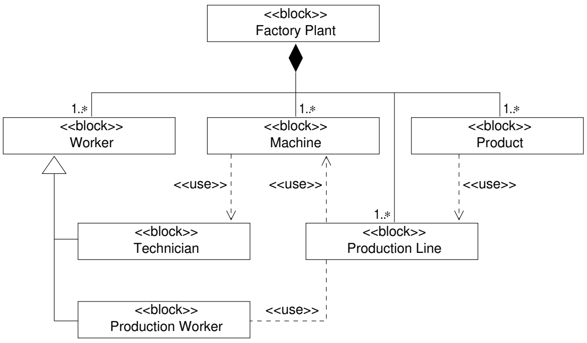


Figure 5. Factory structure.

Figure 6 shows the internal block diagram, in this case the internal factory structure. It can be noticed the interconnection points among all factory elements. Those interconnection points are performed by flow ports. Like a block definition diagram, that is based on a high level structure specification, the internal block diagram shows a more detailed structure specification. The factory manufactures a product (P_1) and uses two production lines (L_1 and L_2) to perform it. There are four machines (M_1, M_2, M_3 and M_4) and each production line uses three machines in its production task, respectively $L_1(M_1, M_2, M_3)$ and $L_2(M_2, M_3, M_4)$. The production worker W_1 works on L_1 and the W_2 works on L_2. The technician T_1 makes the maintenace on all machines.

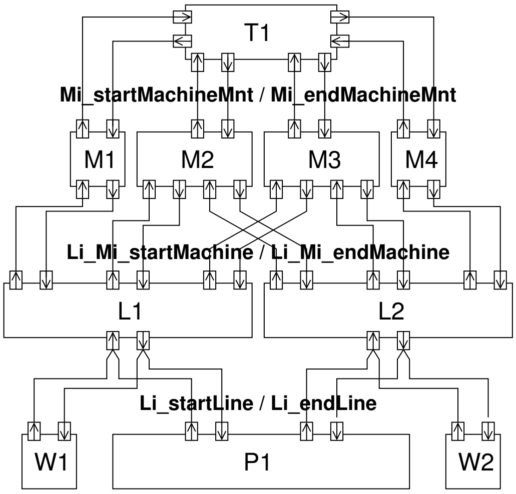

T1
Mi_startMachineMnt / Mi_endMachineMnt
M1
M2
M3
M4
Li_Mi_startMachine / Li_Mi_endMachine
L1
L2
Li_startLine / Li_endLine
W1
P1
W2

Figure 6. Factory internal structure.

4.2. Requirements modeling

There are some requirements that should be satisfied by the factory. The SysML requirement diagram describes the requirements hierarchy and creates a requirement traceability matrix. Figure 7 shows the requirement diagram with the properties that we need to verify in the system. For example, it is important the norms and the work legislation to be satisfied by the factory. The work legislation requires that the production workers have a pause after a continuous work and any worker does not work more than 35 minutes on a production line. The factory norms require that the machine maintenance is ensured during the manufacturing process.

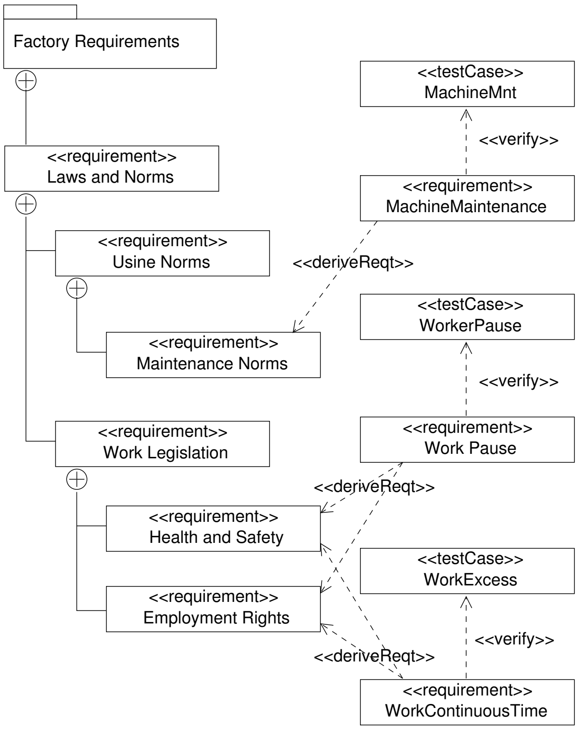


Figure 7. Factory requirements diagram.

Each requirement has a specification written in textual form. The high level requirement is that the laws and the norms will be respected. This requirement can be subdivided in:

- Work Legislation:
  – Employment Rights: the employer will provide a reasonable support and a safe working environment.
  – Health and Safety: employers will be responsible for ensuring physical and psychological health, safety and welfare of employees at work.
  – Work Pause: a pause (5 minutes in our case) will be guaranteed after a continuous work.
  – Work Continuous Time: the worker will not work more than 35 minutes in a continuous work.
- Factory Norms:
  – Maintenance Norms: the maintenance norms will be responsible for ensuring the factory functioning without failure.
  – Machine Maintenance: the machine maintenance will be ensured during the manufacturing process.

The requirement diagram also shows the requirement hierarchy and derivation, the hierarchy is made by a composition relationship that ilustrates a requirement that contains anothers requirements (for example, *Work Legislation* is compounded by the *Employment Rights* and *Health and Safety*). The derivation relationship (*deriveReqt*) shows a requirement that is derived from one or more requirements, the objective is to know a requirement origin (such as *Work Pause* that is derived from *Employment Rights* and from *Healt and Safety*). It also shows a relationship with a test case (performed by *verify*) that will work as a verification case, as explained later.

4.3. Behavioral modeling

After the requirement and structural modeling is ended, it is necessary to perform the system behavioral modeling. In the SysML specification it is possible to associate a behavior for each block. It is usually represented by an UML activity diagram or an UML state diagram. The UML 2.0 activity diagram includes some modifications that allow to use it as a Petri net diagram. In our case we used only the activity diagram features that allow the construction of a Petri net. Our objective is the formalization of the behavior because we want to verify some properties on it. The TINA toolbox was used to build the Petri net system representation and to verify its properties (or requirements).

The system behavioral modeling starts with the product manufacturing behavior (figure 8). It shows the production line sequence to manufacture the product P_1. It can be noted its interconnection points with the production lines: *Li_startLine* and *Li_endLine*.

The second step was to create the behavior modeling of the production line, ilustrated in figure 9. It is possible to observe the interconnection points *Li_startLine* and *Li_endLine* between production workers and product manufacturing task, and *Li_Mi_startMachine* and *Li_Mi_endMachine* between machines, as represented in the IBD diagram of factory internal structure (figure 6).

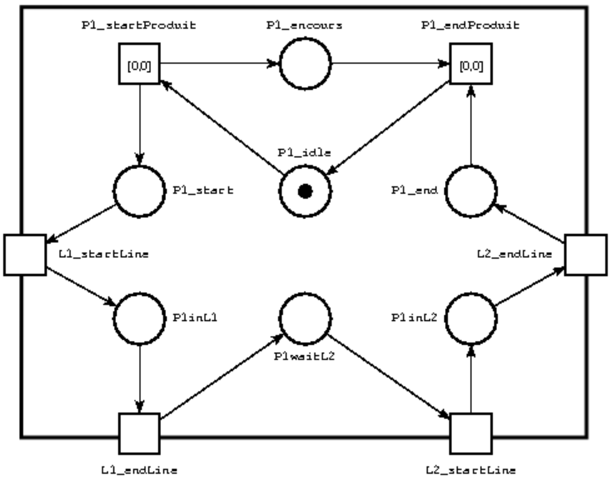


Figure 8. Product manufacturing behavior.

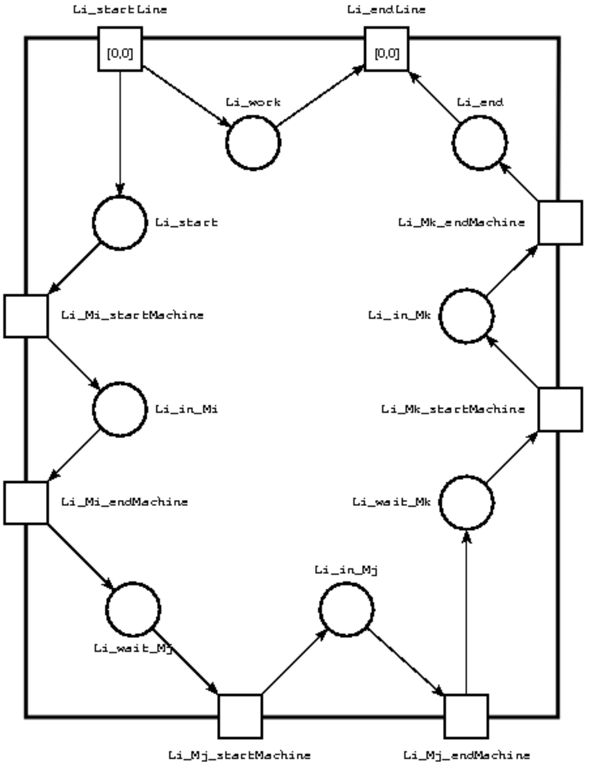


Figure 9. Production lines behavior.

Next, the production worker behaviors can be created. Figure 10(a) shows a production worker behavior when he has not rights to a pause. He stays in an idle state Wi_idle waiting the line to start production; then he goes to a work state Wi_work and after a work he returns to an idle state. But, looking to the requirements again (figure 7) it is possible to see a structural requirement constraint for the production worker (*Work Pause*). Then, it is necessary to change that behavior to a new behavior that includes a pause state (Wi_pause) after a work. This new behavior is shown in figure 10(b).

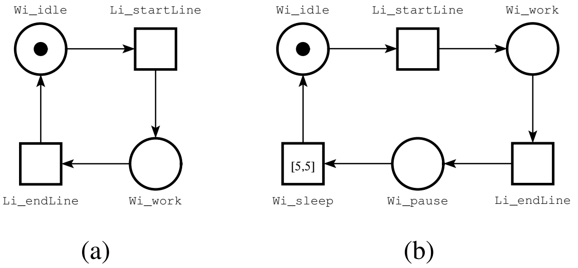


Figure 10. Production worker behavior: (a)without pause and (b)with pause.

Figure 11 shows the technician behavior. Like the production worker, he stays in an idle state Ti_idle waiting for the machine maintenance cycle. When the machine is ready to start the maintenance, the technician goes to a work state Ti_work and the machine is stopped. For the maintenance task the technician does not need more than five minutes (represented by transition *Mi_endMachineMnt*).

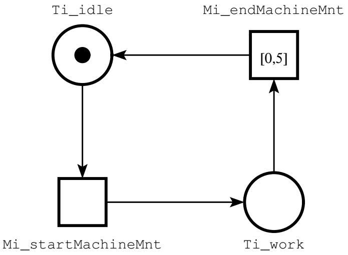


Figure 11. Technician behavior.

Finally, figure 12 shows the machine behavior. Each machine starts from an idle state (Mi_idle) and go to a work state (Mi_work), staying between 5 and 10 minutes working. When this task ends, it returns to the idle state. Looking again to the requirement diagram (figure 7), it can be observed another structural requirement constraint for the machine maintenance task. For that, it was included a cycle counter. Each time that the cycle counter (Mi_cycle) reaches fifteen cycles, the machine is stopped to begin its maintenance task. The machine maintenance state (Mi_mnt) is guaranteed by an inhibition arc that does not permit the machine working state when the machine is ready to go to maintenance.

The composition of these behaviors, as defined by the system structure of figure 6, performs the complete factory behavior, including product lines, machines, workers and technicians. The aim of this paper is to verify that requirements are correctly satisfied by the system. This composition is made by TINA toolbox that merges the places or transitions, with the same label, in a complete behavioral model. The complete model has 47 places and 28 transitions.

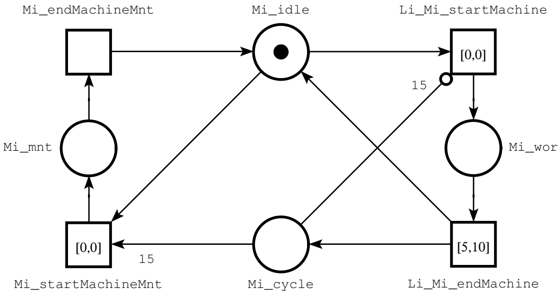


Figure 12. Machines behavior.

4.4. Requirements verification

The SysML requirement diagram recommends the use of the UML Testing Profile to allow the requirements testing. This recommendation is represented by an arrow stereotyped by *verify* that connects a requirement to a test case. Test and verification are used in different moments of the system development cycle and correspond to different goals. Test is realized on the implemented system. Furthermore, the test case, following the profile definition, is a behavior performed against the SUT (System Under Test). This behavior corresponds to a set of possible inputs that can put the system in trouble; the objective consists to determine the possible system behavior when a specific system property is tested.

“SysML Test” only allows to ensure that specific scenarios are accepted by the system implementation while formal verification allows to be sure that the expected requirement are fullfilled by the system specification.

In our case, the profile associated with the requirement diagram is not sufficient because our objective is to verify the system before its implementation. Moreover, test cases don’t provide the necessary informations to represent the system properties. But, some little modifications can facilitate the verification task. The aim of our proposal is not the change of the SysML requirements diagram but the introduction of few modifications which allow the system verification. In this case the requirement under verification is connected by the same arrow stereotyped with *verify* and the same test case structure. The difference is that in the case of verification, its attributes hold LTL logic formulas and the behavior associated to each one, when necessary, is performed by a Petri net as an observer.

Then, a test case, in this proposal, can be understood as a property verification performed against the SUV (System Under Verification). It is composed by a LTL logic formula which is a formalization of a property which was expressed in an informal way in the requirements diagram. The LTL formula specifies a property (or a system requirement) that the user must know if it is satisfied or not by the system.

The informal requirements of figure 7 are now translated in LTL formulas. *WorkerPause* and *MachineMaintenance* are qualitative requirements. They admit a direct translation, like:

WorkerPause : $\Box(Li\_endLine \Rightarrow \Diamond Wi\_pause)$, "Always, the worker Wi has a pause after a continuous work in a line".

MachineMnt : $\Box\Diamond Mi\_mnt$, "Always, there is a moment that the machine Mi is under maintenance".

WorkExcess is a quantitative requirement. Its verification needs the use of an observer (see figure 4(b)). *WorkExcess* holds if event $Wi\_workExcess$ never occurs (the objective is to guarantee that it is not performed by the system), then:

WorkExcess : $\Box\neg Wi\_workExcess$, "The worker Wi never works in excess".

The LTL formulas can be verified using the TINA toolbox. For this example, all refined requirements are verified and satisfied by the system. Looking to the requirements diagram (figure 7), it is possible to see that the high level requirements (Laws and Norms) are satisfied by the system. The reachability graph has 635 states and 787 transitions.

If any requirement is not satisfied by the system it is necessary to search in the structural or behavioral diagrams the cause of that (see figure 1). For example, if a production worker works in excess more than 35 minutes), probably, the production line spends too much time because it has many machines (a production line with 5 machines in the worst case works for 50 minutes). One possible solution is to break the line in two other lines with less machines. Other solution is to buy machines that perform their work faster than before (a production line with 5 machines, that perform their work in 5 minutes, works for 25 minutes). The solutions are not the aim of this paper, but at this moment it is important to notice that the system is not yet developed, only specified and that it is already possible to know if this system satisfies or not the requirements.

5. Conclusions and Future Work

In this paper was presented a preliminar approach to introduce a modeling and verification process using SysML, Petri nets and LTL formulas. The objective was to allow the requirements verification before the system implementation. The SysML language was introduced to perform a semi-formal system structural and requirement modeling, the Petri nets was used to perform the formal system behavior modeling and the formal verification was made by LTL formulas.

A semi-formal approach allows a rapid modeling but lacks in formality and encapsulates some ambiguity. For this reason, it is not possible to make formal verifications. However, a formal approach allows the formal verification but lacks to perform a more understandable system structure view and the requirements traceability from more abstract requirements.

In order to allow the system requirements verification a mixing approach was chosen. A modular modeling was used to connect the behavior to the structure models and the requirements model was refined until that the requirements could be expressed by means of temporal logical formulas. The TINA toolbox was used to perform the formal behavioral system modeling with Petri nets and also the formal verification using LTL formulas.

The paper aim was not modified the SysML initial specification but allows the system verification, for that reason it was used the UML Testing Profile in the verification context. But, testing is different from verification and in future work, it will be necessary to define a new profile that performs the verification.

Finally, we will conclude that the use of both semi-formal and formal models brings powerful to the modeling and verification tasks. The objective was to ensure a well defined and understable system modeling to the formal system requirements verification.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar. 1991.

[3] B. Berthomieu and M. Menasche. A state enumeration approach for analyzing time Petri nets. In *Proc. Applications and Theory of Petri Nets (ATPN'82), Como, Italy*, pages 27–56, 1982.

[4] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time P etri nets. *International Journal of Production Research*, 42(14):2741–2756, 15 July 2004.

[5] B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2 003), Warsaw, Poland, Springer LNCS 2619*, pages 442–457, 2003.

[6] S. Chaki, M. E, Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *4th International Conference on Integrated Formal Methods (IFM'04), Springer LNCS 2999*, pages 128–147, apr 2004.

[7] S. Edwards, L. Lavagno, E. Lee, , and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation and synthesis. *Proceedings of the IEEE*, March 1997.

[8] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Comm.*, 24(9):1036–1043, Sept. 1976.

[9] SEDSIG. Systems Engineering Domain Special Interest Group. http://syseng.omg.org/, March 2006.

[10] SysML Merge Team (SMT). *Systems Modeling Language (SysML) Specification*. version 1.0. Object Manegement Group (OMG), April 2006.