Formal Verification of UML Sequence Diagrams in the Embedded Systems Context *

E. Cunha, M. Custódio, H. Rocha, R. Barreto

[1]Depto de Ciência da Computação – Universidade Federal do Amazonas (UFAM)
Av. Rodrigo Otávio, 3000, Aleixo, Manaus-AM-Brazil

{evertongualberto, mmc, herbert, rbarreto}@dcc.ufam.edu.br

Abstract. *This paper shows a method for translating UML sequence diagrams to Petri nets and verifying deadlock-freeness, reachability, safety and liveness properties by using a model checker. In this proposed method, the user has not to know about temporal logics to describe the property to be verified. Instead, the user may adopt a high-level properties specification interface, which is automatically translated to a suitable temporal logic. We show the application of the proposed method in an embedded control application that consists of a sensory device mounted on a motorized platform that must detect and track specific objects in the environment.*

1. Introduction

Embedded systems differ from conventional systems in the sense that they are subject to several constraints such as size, weight, height, mobility, energy consumption, and others, have high criticality, and usually have very tight time-to-market. We advocate that it is always necessary to apply methods and techniques to ensure the correctness of embedded systems. This work receives UML (Unified Modeling Language) Sequence Diagrams as input, and translates them into the mathematical formalism of Petri nets. The proposed method provides the Petri net in three different formats: APNN (Abstract Petri Net Notation), PNML (Petri Net Markup Language), and SMV (Symbolic Model Verifier). The APNN and SMV formats can be checked using the MCKit[1] and SMV[2], respectively. Such translations are the first contribution of this paper. Several types of properties can be checked, e.g., deadlock-freeness, reachability, safety and liveness properties. These properties can be expressed with so-called state formula, which is a propositional logic formula consisting of atomic propositions and logical operators. Another contribution is that the user has not to know about CTL (Computation Tree Logic) to describe reachability, safety, or liveness properties. We propose a high level properties specification interface, which is automatically translated to CTL. In this way, we avoid the user to know in depth about CTL. In order to show the practical usability of the proposed method, we show the application of the proposed method in an embedded control application that consists of a sensory device mounted on a motorized platform that must detect and track specific objects in the environment.

*The authors acknowledge the support granted by FAPESP process 08/57870-9, CAPES process BEX-3586/10-3, FAPEAM, and by CNPq processes 575696/2008-7, and 573963/2008-8.

[1]www.fmi.uni-stuttgart.de/szs/tools/mckit/

[2]http://www.cs.cmu.edu/ modelcheck/smv.html

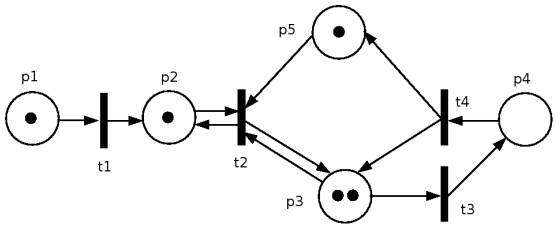

Figure 1. Petri Net Example

2. Background

2.1. UML's Sequence Diagram

The UML's sequence diagram shows object interactions arranged in time sequences. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. A sequence diagram includes time sequences but does not include object relationships. A sequence diagram can map a scenario, which is described by a use case, in order to define how objects collaborate to achieve the application's goals. There is a vertical line, usually called lifeline, that represents an object and shows all its points of interaction with other objects in events that are important to it. Lifelines start at the top of a sequence diagram and descend vertically to indicate the passage of time. Interactions between objects - messages and replies - are drawn as horizontal direction arrows connecting lifelines. In addition, boxes known as combine fragments are drawn around sets of arrows to mark alternative actions, loops, and other control structures.

2.2. Petri Nets

Petri net model [Murata 1989] is a kind of state-oriented model, specifically defined to model systems that comprise interacting concurrent tasks. The Petri net model consists of a set of places, a set of transitions, and a set of tokens. Tokens reside in places, and circulate through the Petri net by being consumed and produced whenever a transition fires. More formally, a Petri net is a quintuple $\langle P, T, F, W, u \rangle$, where $P = \{p_1, p_2, \ldots, p_m\}$ is a set of places, $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions, and P and T are disjoint. Further, the relation function F, $F \subseteq (P \times T) \cup (T \times P)$, defines arcs between places to transitions and between transitions to places. $W : F \rightarrow \mathbb{N}$ represents the weight of the flow relation F. Finally, the marking function $u : P \rightarrow \mathbb{N}$ defines the number of tokens in each place, where $\mathbb{N}$ is the set of nonnegative integers. Figure 1 shows a graphic representation of a Petri net. Note that there are five places (graphically represented as circles) and four transitions (graphically represented as solid bars) in this Petri net. The places p_2, p_3, and p_5 provide inputs to transition t_2, while p_3 and p_5 are the output places of t_2. The marking function u assigns one token to p_1, p_2 and p_5 and two tokens to p_3, as denoted by $u(p_1, p_2, p_3, p_4, p_5) = (1, 1, 2, 0, 1)$. A Petri net is executed by means of firing transitions. A transition can fire only if it is enabled – that is, if each of its input places has sufficient tokens to fire. A transition is said to have fired when it has removed all of its enabling tokens from its input places, and then deposited tokens into each output place. In Figure 1, for example, after the firing of transition t_2, the marking u will change to $(1, 1, 2, 0, 0)$. Petri nets are useful because they can effectively model a variety of system characteristics, and may be used to check several useful properties. Petri nets is used in this work because it is a well-consolidated technique for specifying concurrent systems.

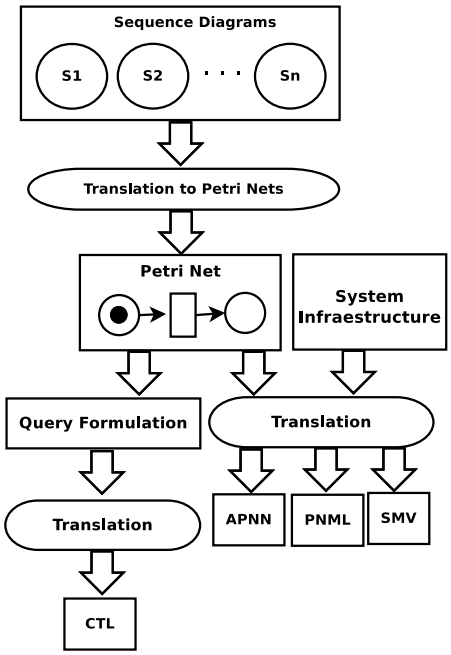

Figure 2. Proposed Method

2.3. Verification by Model Checking

This paper focuses in static formal verification using model checking formal technique, which consists of a systematically exhaustive exploration of states and transitions in a mathematical model. Model checking provides a more automated, although less general way of proving the correctness of systems. The approach requires the construction of a model of the system and the specification of its correctness properties. A model checker then automatically computes whether the model satisfies the properties or not. The power of model checkers is that they are relatively easy to use compared to manual verification techniques or theorem provers, but they also have some clear drawbacks: they suffer from the state space explosion problem, i.e., the number of states grows exponentially in the number of system components. Model checkers are capable of finding errors that are not likely to be found by simulation or test. The reason for this is that unlike simulators, which examine a relatively small set of test cases, model checkers consider all possible behaviors or executions of the system.

3. Proposed Method

3.1. Basics

Figure 2 gives an overview of the approach. The proposed method assumes that the designer is able to write the sequence diagrams, which are input to the method. Usually, it is expected that such diagrams represent, in some extent, the system's functional requirements, and that each requirement is associated with one or more tasks. This method considers that the minimum unit of computation are tasks, and these tasks are the objects that compose the sequence diagram. Sequence diagrams do not have a pre-defined domain, and they can therefore be used in the context of embedded systems. Therefore, in Section 4 we show the application of the proposed method in an embedded control application. After the diagram specification, the next step is the automatic transla-

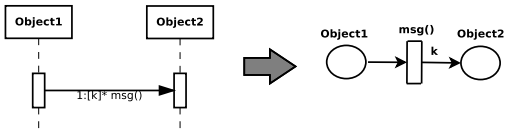

Figure 3. First Translation Rule

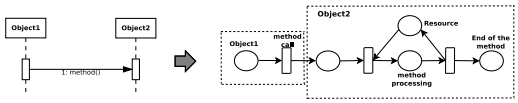

Figure 4. Second Translation Rule

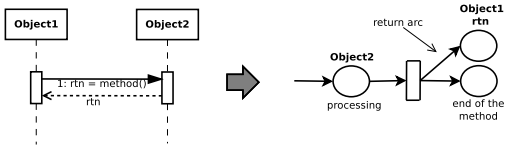

Figure 5. Third Translation Rule

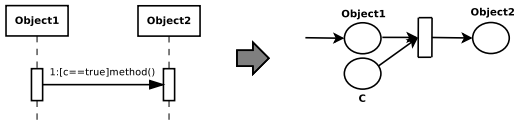

Figure 6. Fourth Translation Rule

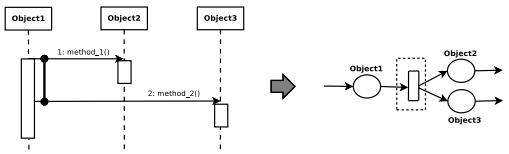

Figure 7. Fifth Translation Rule

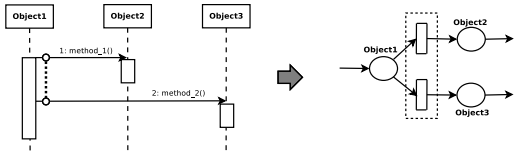

Figure 8. Sixth Translation Rule

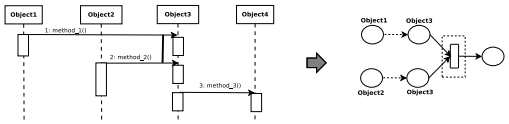

Figure 9. Seventh Translation Rule

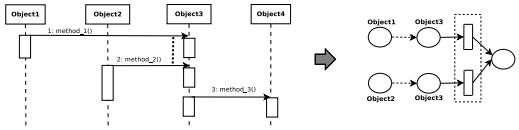

Figure 10. Eighth Translation Rule

tion to the formal model of Petri Nets. This part is based on the method proposed by Jeng [Jeng and Lu 2002]. The idea is that the Petri net can be used as input to model checkers. In this case, the Petri net format may differ depending on the specific model checker to be adopted. As we can see in the Section 3.4, we provide the Petri net in three different formats.

Another feature of the proposed method is that it provides a mechanism for specifying the properties to be verified in the models at high level and using natural language. After this, such specification is automatically translated into a CTL formula, suitable to be run in several model checkers. The modeled system can consider systems distributed across multiple processors, where tasks may communicate via a shared bus. The result of applying the proposed method can be considered as the junction of the best of both worlds: (i) UML, which is considered an industry standard, it is a semi-formal model, and it is comprehensive approach; and (ii) Petri nets, a mathematical model and suitable for formal verification.

3.2. Translations Rules

The translations rules considered in this work are based on an extended sequence diagram as proposed by Jeng and Lu [Jeng and Lu 2002]. The extension has the aim to improve the behavior of the systems, including concurrency, choice, synchronization, and confluence. In the paper of [Jeng and Lu 2002] they proposed eight translations rules. (1) **send message**: The sending message is transformed in a transition that connects both objects (Fig 3). The number (k) inside the brackets defines the amount of interactions, and it is

represented by a weight of the arc. (2) **method call**: A method call is transformed in a building block as depicted in Fig. 4. As we can see, the user can define resources to be used by the specific method. (3) **method return**: A method return is transformed in a place (*rtn*) (Fig. 5), where when the execution of a method is completed, this means that the place *rtn* receives a token. (4) **conditions**: Each condition is transformed in a place which contains a token needed to fire a transition, as illustrated in Fig. 6. (5) **concurrency**: Fig. 7 shows the concurrency, which is denoted by a solid line with hollow endpoints at the arrows, where all messages are considered as being concurrently executed. (6) **choice**: Fig. 8 shows the choice, which is denoted by a dashed line with hollow endpoints at the arrows, where in each time only one message can be triggered to execute. (7) **synchronization**: Fig. 9 shows the synchronization, which is denoted by a solid line at the arrows. In this case, the execution of the next step can only be triggered after all messages in the synchronization group finish their executions. (8) **confluence**: Fig. 10 presents the confluence, which is denoted by a dashed line at the arrows. Differently from the synchronization, the confluence allows that only one message that finishes its execution to trigger the execution of the next step.

3.3. High Level Properties Specification

The proposed method provides a high-level property specification to be verified in the model. The proposal is to translate the high level specification to a temporal logic in such a way that it can be verified by a model checker. The main aim of this feature is to maintain the high level of abstraction in both (i) the construction of system models, and (ii) verifying system's properties. At the same time, the designer gets rid of the learning curve of temporal logics, and besides that it makes the process a little bit more agile. Therefore, we believe that this interface makes it easy the use of formal methods for people with little background in the formal methods area.

The specification is divided into 4 parts: The *first part* of the specification is relative to "path quantifiers" and "linear-time operators". There are six options: (1) for any execution path of the system, the property will be true in all future states, which is translated into CTL as AGp (All and Globally); (2) for any execution path of the system, the property will be true in at least one state in the future, which is translated into CTL for AFp (All and Finally (or Eventually)); (3) there is an execution path of the system where the property will be true in all states future, which is translated into operators EGp (Exists, and Globally); and (4) there is an execution path of the system where the property will be true in at least one state in the future, which is translated into operators EFp (Exists and Finally (or Eventually)). (5) for any execution path of the system, the property will be true in the next state, which is translated into CTL as AXp (All and neXt); (6) there is an execution path of the system where the property will be true in the next state, which is translated into CTL as EXp (Exists and neXt). The *second part* of the specification is the property to be verified. Each clause of the property is related to a task, which is an object of the sequence diagram; and it is possible to join several clauses using logical conectives such as AND, and OR. It is also possible to apply to each clause path quantifiers and linear-time operators. The *third part* of the specification is about the consequent. We may choose between (i) *run in sequence*, which means *implication*, (ii) *until*, or (iii) no consequent. The *fourth part* of the specification is the property to be verified in the consequent. In the same way as the second part of the specification we may also use logical

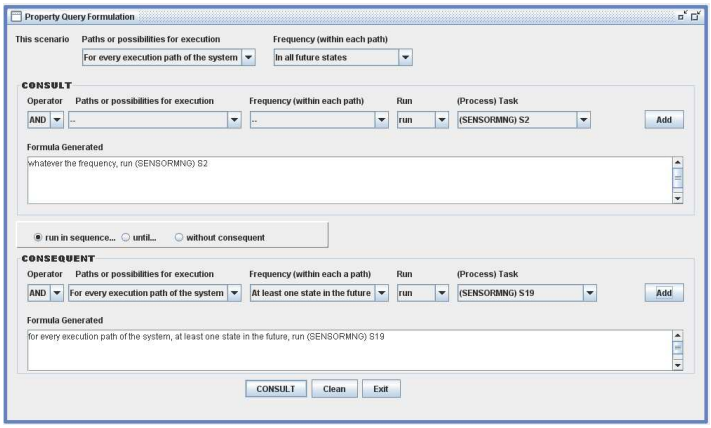

Figure 11. High level properties specification screenshot

conectives, path quantifiers and linear-time operators.

3.4. Tools

The proposed method was implemented, and can be free downloaded at the link *http://sourceforge.net/projects/foreve*. The tool consider that the extended sequence diagram is specified in the XML format file. It is an ongoing work to develop a graphical tool to assist the designers in this process. Another extension, is to provide a domain-specific language for specifying such extended sequence diagrams. From now, the user may either write the XML file by himself, or adopt the UML Editor proposed by Jeng and Lu [Jeng and Lu 2002].

The tool receives the sequence diagram and generates the respective Petri net in three formats: (i) Abstract Petri Net Notation (APNN), which can be used as input to the Model Checking Kit (MCK); (ii) Petri Net Markup Language (PNML), which is a format adopted in several existing verification tools; or (iii) Specific state machine models to be the input of the SMV model checker, which is the format used as input in SMV (Symbolic Model Checker). In the experimental evaluation (see Section 4) we adopted SMV to perform queries of properties in the FOREVER tool. In order to generate the input file to the SMV, it is necessary information about the system's infrastructure, such as processors, processes and tasks. The infrastructure is provided as another XML file. In the link `http://sourceforge.net/projects/foreve` you may find information about XML formats, and examples shown on the case study conducted in the experimental evaluation.

The proposal is to translate the high level specification to a CTL (Computation Tree Logic) so that it can be verified by both MCKit and SMV model checkers. Figure 11 shows a screenshot of an example of specification of properties. In the first part we use a high-level language to specify the path quantifier *Always*, and the linear-time operator *Globally*. In the second part we specify the task S2. In the third part we specify that the previous properties will *imply* in the consequent, which is defined by the path quantifier *Always*, and the linear-time operator *Exists*. The fourth part specifies that the task is S19. The properties specification is then automatically translated to CTL. In the case of the properties of Figure 11 the result is `AG (S2 -> AF S19)`.

4. Experimental Evaluation

4.1. Description

The scenario considered in the experiments is an *embedded control application*, originally described in [DiNatale and Stankovic 1994]. It is basically a process model P_1, P_2, ..., P_n, where each process is divided into a sequence of tasks T_1, T_2, ..., T_m. In this context, in terms of scheduling, the tasks are indivisible computational units. The system physically consists of a sensory device mounted on a motorized platform that must detect and track specific objects in the environment. The system is controlled by four processors connected by a network. The first processor controls the sensor and runs two processes: SENSOR and SENSOR-MNG; the second processor controls the actuators and runs the processes ACT-CONTRL and ACTUATOR-MNG. Both of them communicate with another processor running the main control activity MAIN CONTROL and are responsible for the detection of significant events in the environment. The last processor runs the process SIGNAL, and it is responsible for the management of the significant events that are detected in the environment (e.g., alarm generation).

The control flow of the application is the following: The process MAIN CONTROL communicates remotely to the SENSOR-MNG process the request for a new activity. The SENSOR-MNG locally asks for sensory data from the SENSOR process. When sensory data returns, the SENSOR-MNG process extracts the significant features of the object to be tracked and communicates back the results to the MAIN CONTROL process. The MAIN CONTROL sends the position information to the ACTUATOR-MNG process that communicates with the ACT-CONTRL process to perform the tracking of the sensed object and keep it in the visual range of the sensors. At the same time the MAIN CONTROL process checks for the presence of significant events in the system and sends a report to the processor running the SIGNAL process, responsible for the detection and

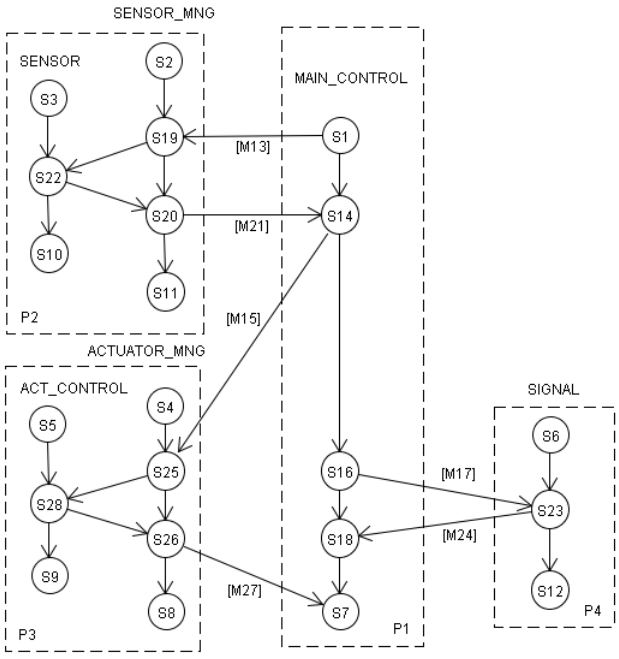

Figure 12. Precedence graph of the Embedded Control Application

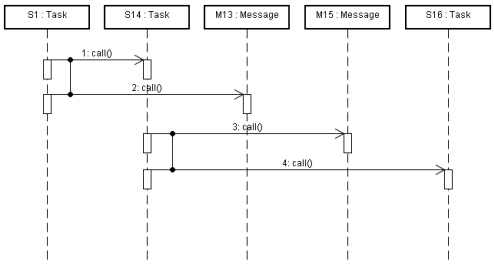

Figure 13. Seq.Diag.: Part 1 of 6

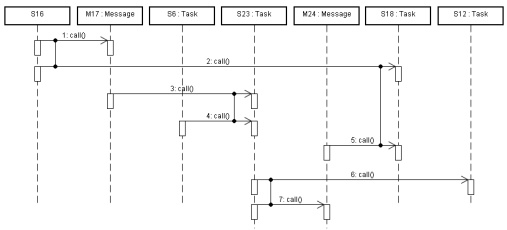

Figure 14. Seq.Diag.: Part 2 of 6

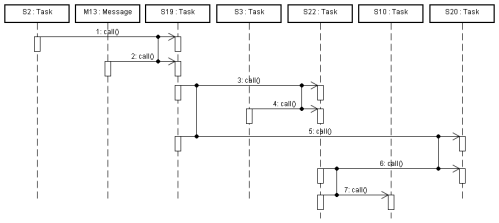

Figure 15. Seq.Diag.: Part 3 of 6

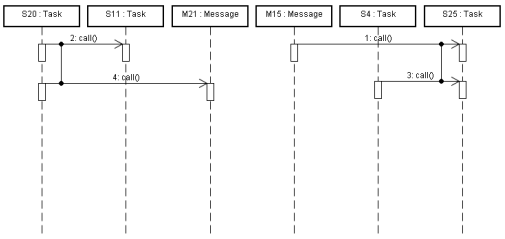

Figure 16. Seq.Diag.: Part 4 of 6

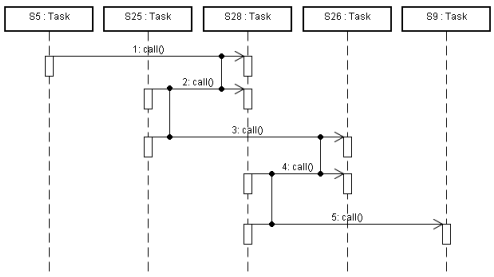

Figure 17. Seq.Diag.: Part 5 of 6

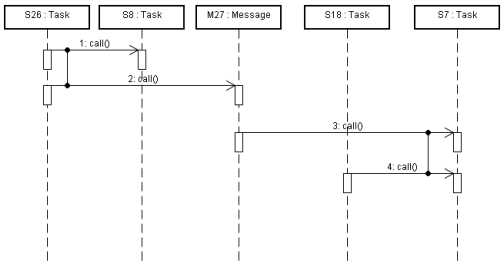

Figure 18. Seq.Diag.: Part 6 of 6

activation of alarms and warnings. Figure 12 shows a precedence graph with the processes and tasks divided among the four processors that make up the system.

Figure 12 also presents the following *communication tasks*: M13, M21, M15, M17, M24 e M27. These tasks are not part of processes, are just connectors that enable the flow of the action of system processes.

4.2. Sequence Diagram Input

UML sequence diagrams, as discussed in Section 2.1, capture the dynamic aspects of the system, emphasizing the temporal order of events in it. The parts of the diagrams from Fig.13 to Fig. 18, model these aspects in the context of the embedded control application. It is worth saying that the sequence diagrams have to be provided in the specific format, in accordance with the method detailed in Section 3.4. In this experiment, in particular, there are no choice or confluence situations. The translations of the sequence diagrams to Petri net is depicted in Figure 19.

4.3. Properties Verification

In order to show the high-level properties specification interface, we will show two examples of properties of the embedded control application. To view the complete list of prop-

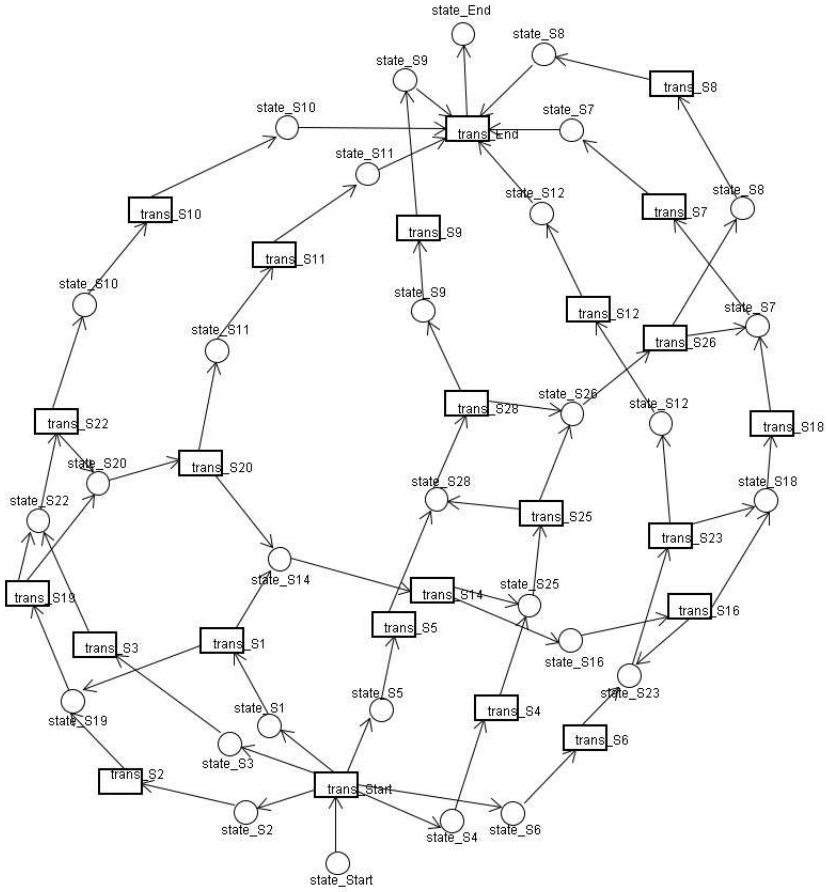

Figure 19. Petri net of the embedded control application

erties verified, please visit: *http://www.sourcefourge.net/projects/foreve*. Is worth noting that to use the interface the user has to understand the concepts of *execution paths*, which consist of several possibilities of states, where each state is is characterized by different variable values, where, in this context, each such variable is equivalent to a task.

First Property Verification: The first question is: *In all execution paths, and in all future states, i.e., continuously from beginning to end, all tasks of MAINCONTROL processes, namely, S1, S14, S16, S18, S7, will run in parallel and, therefore, be activated in all instants?* Figure 20 shows the tool screenshot. After pressing the CONSULT button the following CTL formula was automatically generated: `AG (S1 & S14 & S16 & S18 & S7)`. Figure 21 shows the screenshot of the SMV log of Query I and the resources allocated. The result is FALSE, according to the SMV log. This shows that is not true that in all times the process MAINCONTROL will have all their tasks running in parallel at the same time.

Second Property Verification: The second question is: *In at least one state in the future, S23 (SIGNAL process) will be executed considering that tasks S6 (SIGNAL process) and S16 (MAINCONTROL process) run in all paths?* Figure 22 shows the tool screenshot. After pressing the CONSULT button the following CTL formula was automatically generated: `AG (S6 & S16 -> AF S23)`. Figure 23 shows the screenshot of the SMV log of Query II and the resources allocated. The result is TRUE, according

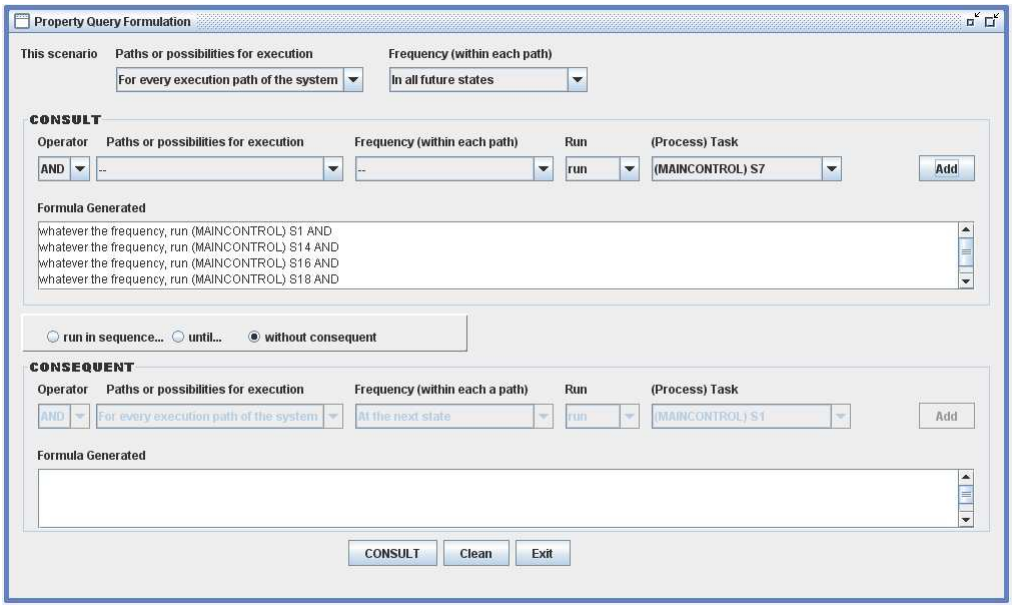

Figure 20. First property verification screenshot

```
Aplication: "Embedded Control Application"
Properties: AG "( (S1 & S14 & S16 & S18 & S7) )"
Result: The property is FALSE
```

Figure 21. SMV log of the first property verification

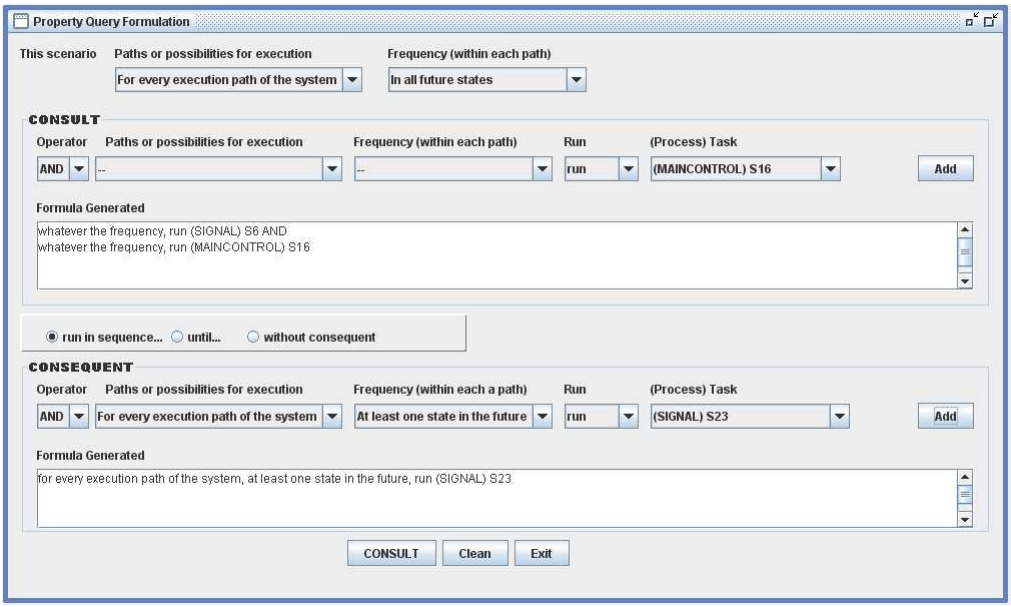

Figure 22. Second property verification screenshot

to the SMV log. This shows that this property demonstrates the proper communication through the bus between the processes MAINCONTROL and SIGNAL.

5. Related Work

Jeng and Lu [Jeng and Lu 2002] proposed a translator of UML extended sequence diagrams to place/transition Petri nets through the composition of building blocks. The ex-

```
Aplication: "Embedded Control Application"
Properties: AG "( (S6 & S16 ) ->  (AF S23) )"
Result: The property is TRUE
```

Figure 23. SMV log of the second property verification

tended sequence diagram is increased with some descriptive elements that aim to increase the power of the notation, thus allowing the presence of structures such as synchronization, confluence, choice, etc. Their proposal was used to prove the efficiency of a slicing algorithm for scheduling tasks and, moreover, no formal verification of properties was performed.

Fernandes *et.al.* [Fernandes et al. 2007] presents an approach to translate given UML descriptions, in this case Use Cases and Sequence Diagrams, into a Coloured Petri Net (CPN) model. The main aim is to improve the tool support for modeling using Petri nets. They apply their proposed method in the specification of an elevator controller. Yao and Shatz [Yao and Shatz 2006] consider translation of UML sequence diagrams and statecharts to Extended Colored Petri Net (ECPN). The ECPN model is used to formally describe state transitions of individual objects and interactions among objects. Our proposed method adopts ordinary Petri nets, not high level nets, such as Coloured Petri Nets.

Eshuis [Eshuis 2006] presents a two translations from activity diagrams to the input language of SMV. The first translator is called requirements-level translation, which defines a state machine that can be efficiently verified, but are a bit unrealistic since they assume the perfect synchrony hypothesis with their environment, i.e., they respond immediately and infinitely fast to new input events. The second translator is called implementation-level translation, which defines state machines that cannot be verified so efficiently, but that are more realistic since they do not use the perfect synchrony hypothesis, i.e., they use input queues that shield them from their environment. Our proposed method does not translate from activity diagrams, but sequence diagrams.

Amorim *et. al.* [Amorim et al. 2005] show an approach for software synthesis in embedded hard real-time systems that starts from Live Sequence Charts (LSC) scenarios as specification language. LSCs specify liveness, i.e., things that must happen. Therefore, this method allows the distinction between possible and necessary behavior as well as the specification of possible anti-scenarios. Starting from LSC scenarios, they use a time Petri net (TPN) formalism for system modeling in order to find feasible pre-runtime schedules, and for synthesizing predictable and timely scheduled code. In this work time Petri net is used for find a pre-runtime schedule. In our proposed method, we may verify safety, liveness, and reachability properties.

6. Conclusions and Future Work

The development process of embedded systems becomes more difficult than for desktop systems due to several constraints such as power consumption, timing, size of memory, processing power, among others. Moreover, in case of failures, such systems can cause damage to human life or economic loss. Therefore, we argue that it is necessary to apply formal methods for the verification of systems at the early phases of development, in order to confirm that they were correctly modeled. Nevertheless, it is also necessary to provide agile methods in order to be intuitive and easy to use by developers, mainly because of

increasing time-to-market pressure for delivering new products as soon as possible. In this sense, this paper proposed a formal verification of embedded systems. But the input to the method is not a formal model, but processes and tasks described using UML sequence diagrams, where such diagrams are well known in the developer community.

A major advantage of the proposed method is that it frees the designer to know about mathematical formalisms where, in this case, the learning curve might be high. In order to ensure correctness, the proposed method showed several translation rules from sequence diagrams to the Petri net formalism, and then the use of the formal verification technique called model checking.

We also present a tool that implements the proposed method. The tool is available for free download. The tool provides the model of Petri nets in three diferent formats: PNML, APNN, and SMV. An important contribution of this work was an user-friendly high-level interface to collect properties to be verified, and the subsequent translation of this high-level notation to the Computation Tree Logic temporal logic, which is suitable to be used by model checkers. We also show the practical use of the method through a case study of an embedded control application, which consists of a sensory device mounted on a motorized platform that must detect and track specific objects in the environment, in a 4-processors platform. We have shown two properties verification using the proposed high level interface, and using the SMV model checker.

As future work we will consider other UML diagrams in order to make a broader modeling, for allowing the system specification under several views, and to compose these multiple views.

References

Amorim, L., Barreto, R., Maciel, P. R. M., Tavares, E., Jr., M. O., Bessa, A., and Lima, R. M. F. (2005). A methodology for software synthesis of embedded real-time systems based on TPN and LSC. In *ICESS*, pages 50–62.

DiNatale, M. and Stankovic, J. A. (1994). Dynamic end-to-end guarantees in distributed realtime systems. In *IEEE Real-Time Systems Symposium*, pages 216–227.

Eshuis, R. (2006). Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38.

Fernandes, J. M., Tjell, S., Jorgensen, J. B., and Ribeiro, O. (2007). Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *International Workshop on Scenarios and State Machines*, pages 2–11. IEEE.

Jeng, M. D. and Lu, W. Z. (2002). Extension of uml and its conversion to petri nets for semiconductor manufacturing modeling. In *IEEE International Conference on Robotics and Automation (ICRA'02)*, pages 3175–3180.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Yao, S. and Shatz, S. M. (2006). Consistency checking of uml dynamic models based on petri net techniques. In *International Conference on Computing*, pages 289–297.