# A State-Space Based Model-Checking Framework for Embedded System Controllers Specified Using IOPT Petri Nets

Fernando Pereira[1,2], Filipe Moutinho[1] and Luís Gomes[1]

[1]Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia – Portugal
[1] UNINOVA - CTS – Portugal
[2] ISEL, Instituto Superior de Engenharia de Lisboa- Portugal
fjp@deea.isel.ipl.pt, fcm@uninova.pt, lugo@uninova.pt

**Abstract.** This paper presents a state-space based model-checking framework to test and validate embedded system controllers specified using the IOPT Petri net formalism. The framework is composed of an automatic software code generator, a state-space generator and a query engine, used to define queries applied to the resulting state-space graphs. During state-space generation, the tools collect information required to enable the efficient implementation of hardware/software controllers, including place bounds, deadlocks and conflicts between concurrent transitions. User defined queries can check relevant system properties, as the occurrence of undesired error situations, the reachability of desired states, system liveliness and the occurrence of deadlocks and livelocks. The new tool, available online under a Web based user interface, provides a fast and efficient way to test and validate system controllers, contributing to the reduction of development time.

**Keywords:** Embedded Systems,Mmodel-Checking, Petri Nets.

## 1 Introduction

Model-based development tools offer many advantages to the design, simulation, test and rapid prototyping of embedded systems. The proposed framework uses the IOPT Petri-net [1] modeling formalism and takes advantage of the well known Petri-net mathematical properties to perform model checking and enable the fast verification of critical system properties, contributing to reduce development time and minimizing the time consumed during the test and simulation phase.

The new model-checking framework combines a state-space graph computation algorithm and an automated query system, under a Web based user interface and is the first model-checking tool that fully supports the unique nature of IOPT-Nets, containing both autonomous and non-autonomous features. The resulting information includes the detection of undesired states and error conditions, the reachability of desired states, system liveliness, deadlock and cyclic-lock detection. In addition, it also extracts information required to the automatic code generation of hardware and software controllers, including conflict detection and place bound computation.

To maximize performance, the state-space generator employs a compilation strategy, starting with the automatic creation of a parallelized C program to compute the state-space graph and execute the query engine, used to check the desired model properties. The state-space generation program uses the same Petri net semantic execution code as the final controller implementation, ensuring a high degree of behavioral consistency between the model-checking tools and the final system.

The tools offer a Web-based user interface and are currently available online. The user interface enables the insertion of IOPT Petri net model files, model visualization, automatic controller code generation, state space generation and visualization, query specification and edition and offer a query results filter page.

## 2 Contribution to Value Creation

The proposed model-checking framework offers the potential to greatly abbreviate the development time of embedded-systems, thus contributing to reduce the total system cost and faster time-to-market. Real world applications usually exhibit very complex state-space graphs with millions of states that are too complex to be visually inspected by human operators. The new query tool extends the functionality of existing state-space generation tools by automating the verification of system properties on complex state-space graphs, enabling the fast model-checking of real-world systems.

The Web based user interface with server-side data storage, enables the collaboration between multiple system designers, located at remote locations, that can simultaneously inspect system models, define new queries and inspect query results to verify relevant system properties. When applied to commercial systems, the Web interface can also be used to facilitate the communication between system suppliers and costumers, used as a collaboration tool to help refine system requirements during the system design phase and to analyze error conditions after deployment.

Finally, the storage of state-space graphs and queries on the server, constitutes an effective tool to implement automation regression tests, where all queries previously defined to inspect controller models, can be automatically applied each time a model suffers changes to detect and prevent the resurgence of old bugs.

## 3 IOPT Petri Nets

The IOPT Petri Net class, defined in [2], was specifically designed to model controller systems and the interactions between controllers and controlled systems, targeting the direct implementation of real hardware and software controller devices. IOPT Nets derive from the Place-Transition Net class, with the addition of a set of non-autonomous extensions to support automatic code generation and communication with the external world.

To achieve coherent and deterministic operation, IOPT Nets use a maximal-step execution semantics, where all enabled transitions immediately fire on the next evolution step. Conflicts between concurrent transitions (when multiple transitions are simultaneously enabled, but marking does not allow the firing of all of them), are

solved using transition priorities. To minimize conflicts and simplify modeling, IOPT nets also offer test arcs, often called read arcs, that do not change place marking.

Communication with controlled systems is performed using Input and Output Signals and Events. Signals can hold Boolean values or Integer values corresponding to Range types. Events represent instantaneous changes in Input Signals, where the Input Signal crosses a predetermined threshold in a specific direction, Up or Down.

Transition firing can be conditioned using Input Events and Guard Functions (Boolean expressions relating Input Signals). As a result, the evolution of IOPT Nets will depend on External Signals and the Net is non-autonomous. Transition firing can also trigger Output Events, causing permanent changes to the value of Output Signals, which hold memorized values. As a result, the system state is composed by two vectors: a Net marking vector containing the marking of all Places and an Output Event Signal vector, containing the memorized values of all signals associated with Output Events. Output signals not related with Output Events hold combinatorial values and can be associated with Places, using Output Expressions.

## 4 IOPT Framework Overview

Embedded system development using the IOPT Petri Net class is supported by a tool framework resulting from the work of several authors [3][4][5]. The framework includes tools to design and edit IOPT models (SnoopyIOPT), automatic software/hardware code generator tools (PNML2C, IOPT2C and PNML2VHDL), an Animator tool to create Synoptics and animated simulations running on personal computers. The animator tool, associated with the code generators and the Animator4FPGA and IOPT2Anim4Dbg tools can also be used to create animated graphical user interfaces, including IOPT model simulators and debug screens to deploy in the final embedded devices [6].

The new tools presented in this paper complement the existing tools, adding the capability to perform model checking to analyze critical system properties and detect design flaws, automating the extraction of information from state-space graphs.

## 5 State-Space Generation

State-space generation is the first step of the model-checking architecture. As real world applications usually lead to very complex state-space graphs, with many million states, this step requires extensive computing resources, with long processing time and high memory consumption. To address this problem, a compilation strategy was employed, with the automatic generation of a dedicated software program to compute the state-space graph, optimized for each IOPT model. To take advantage of the multi-core processors available in modern personal computers, the resulting source code include OpenMP [12] parallelization directives to share the processing load among all available processing units.

The state-space generator program reuses part of the code produced by the automatic C code generator. This way, both the final controller implementations and the state-space generators share the same model semantics execution code, ensuring a high level of consistency. Automatic code generation was implemented using a set of XSL [13][14] transformations, that read the original PNML model file and create C code. The resulting code contains functions to implement the model's semantic rules, a state-space exploration algorithm, a hash-table to store the state-space database and file I/O code to store the resulting state-space graph inside a hierarchical XML file. The query processing engine, discussed in the next section, is also compiled and linked to the state-space generator program.

Although the query engine runs directly on the state-space data structures stored in RAM, maximizing performance, the final state-space graph is stored in XML format. Although XML usually leads to large disk files, the format was chosen due to the availability of an extensive set of processing tools, as XSLT, Xpath and Xquery [15]. XSL transformations enable the easy conversion from the native XML format to other file formats, as SVG for graphical display and formats used by other model-checking tools, as GML and GraphML [16], etc. The Xpath and Xquery languages can also be used to specify complex queries, considering not only the graph nodes but also the relationships between different nodes, similar to CTL [17].

The state-space exploration algorithm must account with the maximal-step execution semantics and the non-autonomous features of IOPT Nets, having to deal with external signals and events to check the compatibility between them. For example, if two transitions depend on complementary input events, then all state-space arcs where both transitions fire simultaneously must be invalidated.

## 6 The Query Engine

By default, the state-space generator program always checks several important model properties, including deadlocks where the system reaches a state without any enabled transitions, conflicts between concurrent transitions and place bounds, used to calculate the number of memory elements used to synthesize hardware controllers.

However, different models may have specific properties that must be checked on a case-by-case basis. For instance, industrial controllers requiring conformance with safety standards, must be inspected to detect states where safety rules are infringed. For example, many dangerous machines operate inside a safe area and cannot run if a door is opened. In other cases, when a system starts executing an operation, it must always reach a final state where the operation completes. In this case, the system designer must analyze the reachability of the ending state, to detect states from where the final state can no longer be reached. If a system must continuously run and always return to the original state, the reachability of the initial state must also be checked.

Although relatively small state-space graphs can be visually inspected by a human operator, large state-spaces with millions of states must be checked with automatic tools, using queries. In addition, queries can be stored and repeatedly checked whenever a model is changed, implementing a regression test mechanism.

The proposed query system is composed by a graphical user interface with an expression editor, a query processing engine that converts the user specified queries to

C source code, compiled and linked with the state-space generator program and finally a query results page with filter and sorting capabilities.

The query expressions, specified using the graphical user interface, comprehend the following items: Place marking in state-space nodes; Output event signal values in state-space nodes; Transition firing inscriptions in state-space arcs; Literal constants; The comparative operators =, <>, <, >, <=, >=; The arithmetic operators +, -, *, / and MOD; The logical operators AND, OR and NOT; Nested expressions using parentheses; A reachability «REACH(state)» function. The graphical user interface automatically checks expression syntax, verifying the relative order of operators and operands and parentheses count.

The query editor, implemented using the Asynchronous Java-script and XML (AJAX) technology [18], stores the query expressions on the Web server as XML documents, that are later transformed using XSL transformations into C software code, integrated inside the state-space generation program.

The generated code contains a direct translation of the query expressions to C, and an iterative crawling algorithm to compute the reachability of selected states. Due to practical restrictions regarding memory usage, each query expression can only use a single *REACH(state)* function, but the results of different query expressions can be later joined and compared in the query results filter page.

## 7 Application Example

To illustrate the usage of the model-checking framework, an application example was developed and checked using the new tools, implementing a simplified controller for an electrical washing machine, as displayed in Fig. 1a.

The controller interfaces with the physical machine using several input sensors and actuator outputs: The outputs control the motor rotation, a water-entry valve, a water escape valve, a detergent entry valve and a door-lock, used to prevent the door from being opened while the machine is working. The inputs include two buttons to turn the machine on and off, a start button to initiate machine operation, two water level sensors (full and empty), a detergent level, a timer and a door-closed sensor.

Machine operation starts when the user presses the *On* button, followed by the *Start* Button and the door is closed. When these conditions are satisfied, the door lock output is enabled and a washing program is initiated. The program starts by simultaneous filling the water tank and loading detergent. Next, it performs several washing cycles where the motor runs for 7 minutes and is idle for 3 minutes. The number of cycles is defined by the initial marking of place *P_cToDo*. After the washing cycles finish, the controller opens the water escape valve and waits until the tank is empty, returning to the initial *Off* state.

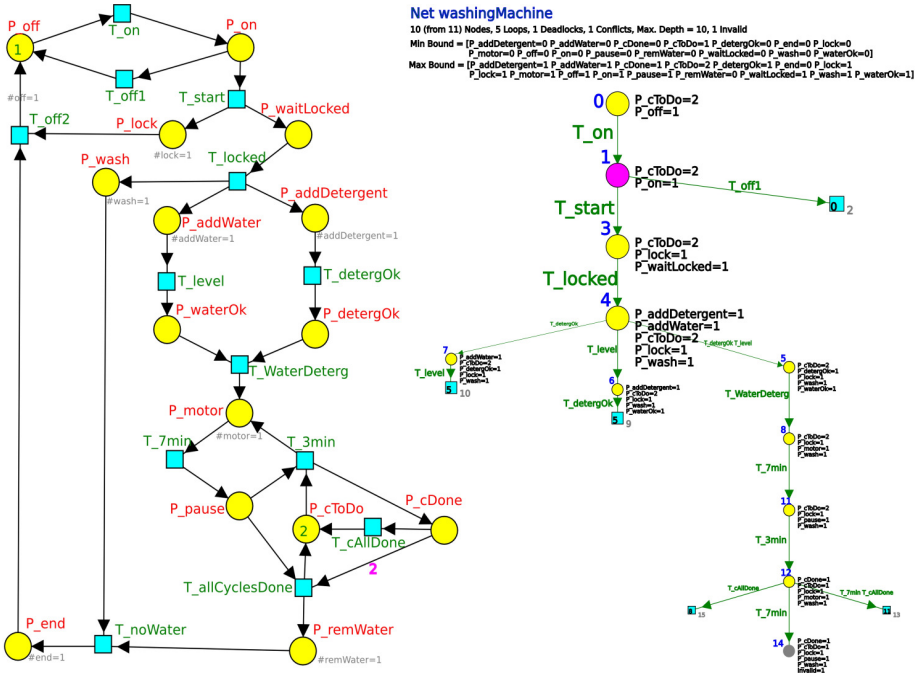


**Fig. 1.** a) Initial controller model b) Initial model state-space graph

A first model version, presented in Fig. 1a, was designed and was submitted to the IOPT Tools Web interface and a state-space graph was generated (Fig. 1b). All places exhibit a maximal bound of 1 token, except places *P_cToDo* and *P_cDone* bounded to 2 tokens, corresponding to the number of washing cycles, and places *P_end* and *P_remWater* display a maximal bound of 0, indicating the system never reaches the end of a washing cycle. The graph displays no deadlocks and one node painted with magenta denotes a conflict, which will be later solved with different transition priorities.

In this particular example, the resulting state space graph is small and can be visually inspected to verify important system properties. For example, observing the the bottom branch, the ending nodes do not show any links to the original state 0, indicating that the system may be not reversible. This means the system will no longer be able to return to the original state and the machine cannot execute more washing programs. As there are no deadlocks, the graph must contain cyclic locks.

For complex state-space graphs, where reversibility cannot be easily detected by visual inspection, the query «NOT REACH(0)» can be used. Using this query, the state-space generator program will perform a recursive search through the entire graph, to detect all states that cannot reach the original state number 0. Results from this query indicate that states numbered from 3 to 12 cannot reach state 0, constituting a cyclic lock. Observing the end of the selected branch, it is possible to observe that the transitions T_3min, T_7min and T_cAllDone continue to fire cyclically and the

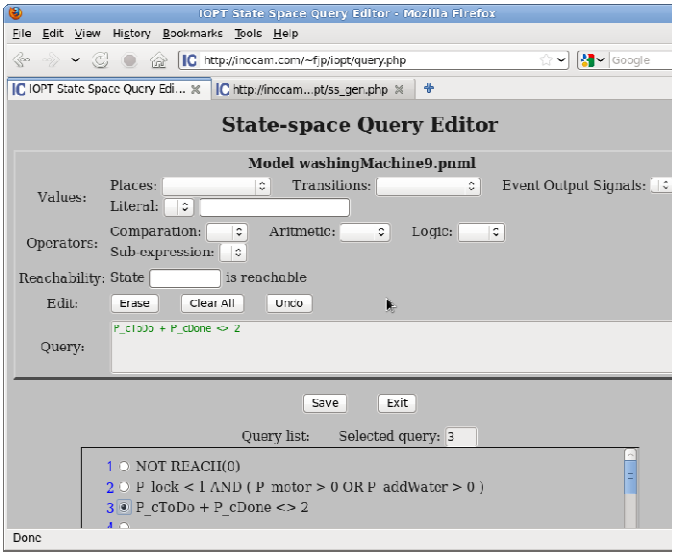


**Fig. 2.** Query Editor User Interface

system is failing to count the number of washing cycles. To solve this problem, the model was changed and transition t_cAllDone was removed.

However, the state-space of the second controller version has one deadlock corresponding to the marking *P_cDone*=1, *P_pause*=1 and *P_wash*=1. In fact, this version of the controller correctly completes the first washing program, but enters in a deadlock before the end of a second washing program. This happens because the arc from transition *T_allCyclesDone* to *P_cToDo* is failing to reinitialize the number of washing cycles to 2 when a program finishes. To solve this problem, the inscription in this arc must also be equal to the number of washing cycles.

To analyze other system properties, more queries must be specified. For example, for safety reasons it should never be possible to open the door while the motor is running. The digital outputs that lock the door and enable the motor are associated with places P_lock and P_motor respectively. Hence, an error situation corresponds to states where the marking of place P_motor has tokens and the marking of P_lock has no tokes. This can be specified using the query *«P_motor > 0 AND P_lock < 1»*. This rule can also be extended to the water loading, that should never occur with the door opened: *«P_lock < 1 AND (P_motor>0 OR P_addWater>0)»*.

As the errors found in the original model were related with wash cycle counting, it is important to define rules check the marking of the complementary places *P_cToDo* and *P_cDone*, representing respectively the number of washing cycles not yet executed and the number of cycles already performed. The sum of tokens on both places should remain equal to the number of cycles: *«P_cToDo + P_cDone <> 2»*.

Fortunately, the final model does not exhibit any states obeying to any of the previous rules, meaning the system is safe for operation. Fig. 2 shows the query editor and the queries used in this example. These queries are automatically checked

whenever the state-space is calculated, working as a regression test mechanism to verify future model changes.

## 8 Related Work

The new model-checking framework presented combines a unique set of capabilities required to analyze controllers modeled with IOPT Petri Nets. Although many state-space generators and model-checking tools are available [7][8][9], none of the existing tools supports all the necessary features.

While most existing model-checking tools only support autonomous systems [7][8], IOPT Nets are non-autonomous. The addition of input and output signals and events has a direct impact on state-space generation. Contrary to autonomous Petri Net classes, the IOPT state-vector contains the value of all output signals related to output-events, along with the place marking. The influence of input signals and events on the firing of transitions must also be accounted during state-space computation. For example, transitions with exclusive input-events cannot fire simultaneously and enabled transitions controlled by the same event should not fire independently. The same applies to the input signals used in transition guard functions.

Secondly, IOPT Nets use another feature not supported by most tools: maximal-step semantics. As a consequence, state-space computation must calculate all possible combinations of enabled transition firings, leading to state-space graphs incompatible with the other execution semantics.

Finally, the compilation strategy used to compute the state-space graph and execute queries, starting with the automatic creation of an optimized C program that will subsequently perform both tasks, offers a very high level of performance. Although other tools have already resorted to compilation strategies [10][11], the new tool automatically generates parallelized C code, taking advantage of multi-processor and multi-core computers. This way, the new tool can be applied to real-world applications, enabling the fast analysis of complex systems with millions of states. State space generation speed depends on the characteristics of each model, but for a given example, the state-space generator calculated approximately 50 million states in 15 seconds (1.1Million independent states and 49 Million duplicated), using an Intel Core i7 920 with 8 virtual cores, plus 2 minutes to save the resulting 4.2Gb XML file.

## 9 Conclusions

The new tools offer a very effective way to check and validate embedded system controllers designed using the IOPT modeling formalism. Development time can be reduced, as system properties can be automatically checked during the early design phases, before the prototype implementations. More importantly, state-space based model-checking will cover error situations caused by low probability event sequences that could not be detected during simulation and prototype tests.

Finally, the user friendly Web user interface stores a model and a query database, enabling the reuse of previously defined queries whenever a model suffers changes, implementing an automatic regression test mechanism.

**Acknowledgment.** The second author's work is supported by a Portuguese FCT grant ref. SFRH/BD /62171/2009.

## References

1. Reisig, W.: Petri nets: an introduction. NY, USA: Springer Verlag, New York, Inc., (1985)
2. Gomes, L., Barros, J., Costa, A., Nunes, R.: The Input-Output Place-Transition Petri Net Class and Associated Tools, in Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN'07), Vienna, Austria, (2007)
3. Gomes, L., Costa, A., Barros, J., Lima, P.: From Petri net models to VHDL implementation of digital controllers, in Proceedings of the IECON2007 - The 33rd Annual Conference of the IEEE Industrial Electronics Society, Taipei, Taiwan, (2007)
4. Gomes, L., Rebelo, R., Barros, J., Costa, A., Pais, R.: From Petri net models to C implementation of digital controllers, in Proceedings of the ISIE2010 - IEEE International Symposium on Industrial Electronics, Bari, Italy, (2010)
5. Gomes, L., Lourenco, J.: Rapid Prototyping of Graphical User Interfaces for Petri-Net-Based Controllers, in IEEE Transactions on Industrial Electronics, vol. 57, pp. 1806--1813, (2010)
6. Pereira, F., Gomes, L., Moutinho, F.: Automatic generation of run-time monitoring capabilities to Petri nets based Controllers with Graphical User Interfaces, in Proceedings of DoCEIS'11 - Technological Innovation for Sustainability, IFIP AICT 349, Springer, pp. 246--255, Costa da Caparica, Portugal, (2011)
7. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, vol. 1 Basic Concepts. Berlin. Germany. Springer Verlag, (1997)
8. Schmidt, K.: LoLa, a Low Level Petri net Analyzer, Institute Für Informatic, Humboldt Universität zu Berlin, 10099 Berlin, (2000)
9. Wolf, K.: Generating Petri net state spaces, In Proceedings of the 28th Int. conference on Applications and theory of Petri nets and other models of concurrency (ICATPN'07), Jetty Kleijn and Alex Yakovlev (Eds.). Springer Verlag, Heidelberg, (2007)
10. Varpaaniemi, K., et al.: Prod Reference Manual, Helsinky University of Technology, August (1995)
11. Roch, S., Starke, P.H.: INA Integrated Net Analyzer, Version 2.2 Manual, Humboldt-Universität, Berlin, http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/
12. The OpenMP API specification for parallel programming page, http://openmp.org/wp/.
13. Tidwell, D.: XSLT, O'Reilly, (2001)
14. XSL Transformations (XSLT), version 2.0, W3C Recommendation, 23 January 2007 http://www.w3.org/TR/xslt20/
15. XQuery 1.0 and XPath 2.0 Formal Semantics, Second Edition, W3C Recommendation 14 Dec. 2010, http://www.w3.org/TR/xquery-semantics/
16. The GraphML File Format, Available at http://graphml.graphdrawing.org/.

17. Emerson, E.A.: Temporal and Modal Logic. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, vol. B, pp. 996--1072. Elsevier Science Publishers, (1990)
18. AJAX Tutorial, http://www.xul.fr/en-xml-ajax.html