Hierarchical Petri Net Simulator: Simulation, Design Validation, and Model Checking Tool for Hierarchical Place/Transition Petri Nets

Yojiro Harie and Katsumi Wasaki
Interdisciplinary Graduate School of Science and Technology, Shinshu University, 4-17-1, Wakasato, Nagano, Nagano, Japan

Keywords: Petri Nets, HiPS, Hierarchical Modeling, Property Analysis, Event based Verification.

Abstract: This paper introduces the Hierarchical Petri net Simulator (HiPS), a Petri net design tool implemented using C# and C++, the .NET Framework, and an interprocess communication channel. HiPS supports hierarchical modeling and can analyze the dynamic and structural properties of a Petri net by generating state spaces. The state space generation engine in HiPS provides a memory-saving technique and high-speed execution. We have devised an Extended Coverability Graph (ECG) mechanism for liveness and persistence properties to accurately maintain transition information. In this paper, we extend HiPS to include a liveness analyzer that utilizes the ECG mechanism and an on-the-fly model checker for event-based systems. We also describe an algorithm that generates state spaces by multi-threading. Furthermore, we propose priority firing estimation with on-the-fly model checking for Linear Temporal Logic (LTL).

1 INTRODUCTION

The increasing implementation of embedded systems in IT has led to communication complexity and complex architectures. A Petri net is a graphical and mathematical modeling tool that can describe many systems, such as logic circuits, data communication, and distributed cloud applications (Iakushkin et al., 2016). The set of all possible system states is called the state space. State space structures in Petri nets are presented in the form of reachability graphs. For complex hierarchical models, large simulations have high processor and memory requirements, and generating state spaces to analyze system behavior can overtax these resources. Thus, high efficiency throughput and reasonable run times are important.

Model checking (Clarke et al., 2001), which can be applied to eliminate bugs, is an automatic verification method that employs mathematical analysis. However, with large complex system models, the number of states can increase significantly. On-the-fly execution has been proposed to increase the efficiency of model verification by checking the model and generating the state space simultaneously.

This paper describes the Hierarchical Petri net Simulator (HiPS) and extensions to HiPS. HiPS is a Petri net design tool implemented in C# and C++, the .NET Framework, and an interprocess communication (IPC) channel. HiPS can support hierarchical modeling and can analyze dynamic and structural properties by generating state spaces.

Hierarchical Queuing Petri Net (HQPN) modeling tools are similar to HiPS (Falko Bause and Kemper, 1996). HiPS differs from HQPN tools depending on whether the subpage module deals with the aspect of the place. We focus on the temporal aspects and consider two classes of Petri nets, that is, timed Petri nets (TPNs) and stochastic Petri nets (SPNs) (Reisig, 1985). Colored Petri nets (CPNs) are a backward-compatible extension of the concept of Petri nets because its tokens can hold data. CPN Tools (Westergaard and Verbeek, 2016) is a modeling and analysis tool for CPNs, and PIPE 2 (Dingle and Knottenbelt, 2016), a platform independent Petri net editor, is a modeling and simulation analysis tool for TPNs and SPNs. In this paper, we do not consider CPNs; however, HiPS2 has been developed and released for modeling based on CPNs.

The state space generation engine embedded in HiPS provides a memory-saving technique and high-speed execution. The state space generation engine employs parallel execution based on Intels Threading Building Blocks (TBB), a widely used library for parallel task execution (Reinders, 2007). The engine was configured according to the operating parameters of each data structure, that is, hash table size and tree

Harie, Y. and Wasaki, K.
Hierarchical Petri Net Simulator: Simulation, Design Validation, and Model Checking Tool for Hierarchical Place/Transition Petri Nets.
DOI: 10.5220/0006463803560361
In *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2017)*, pages 356-361
ISBN: 978-989-758-265-3

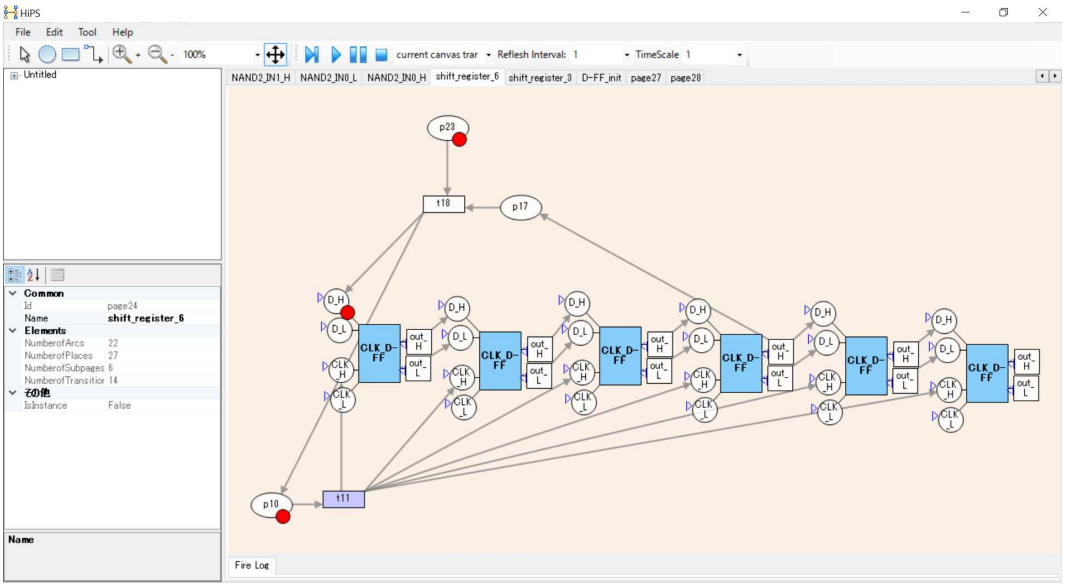

Figure 1: HiPS tool user interface (ellipses, small rectangles, plain arrows, and red circles represent places, transitions, arcs, and tokens, respectively).

depth; thus, it provides a suitable container for the structure of the model and the marking appearance. To minimize memory space, we use special limited 4-bit memory (i.e., *4BIT_ INT*). The size and structure can be varied such that we can emphasize minimizing the use of another resource; for example, by using the *4BIT_ INT* mode, we can emphasize memory efficiency rather than execution speed.

Petri nets have two sets of properties, that is, structural and dynamic. Structural properties are related to the structure of the net and dynamic properties depend on the initial marking. Note that HiPS can analyze both sets. We introduced Extended Coverability Graphs (ECGs) for liveness and persistence properties to maintain accurate information about transitions. The existing coverability graph generator cannot incorporate firing sequences without maintaining transition information. By introducing ECGs, we can perform the required fairness analysis unconditionally.

We have extended the HiPS tool to include a liveness analyzer to utilize the ECG mechanism and an on-the-fly model checker for event-based systems. In this paper, we describe an algorithm that generates state spaces by multi-threading. Furthermore, we propose the concept of priority firing estimation with on-the-fly model checking for Linear Temporal Logic (LTL).

The remainder of this paper is organized as follows. Section 2 describes the HiPS tool environment and concept. Section 3 defines state space and describes the achieved improvements. Petri net properties are discussed in Section 4, and model checking for HiPS is described in Section 5. Conclusions and suggestions for future work are given in Section 6.

2 HIPS DEVELOPMENT

HiPS has been designed to improve operability and execution speed by introducing multi-threading into a system that generates a state space. Figures 1 and 2 show an overview of the HiPS user interface and a function diagram of the HiPS tools, respectively. The architecture of the HiPS graphical user interface (GUI) framework is organized around a Petri net designer and the simulator. When implementing a verification environment, it is vital to simulate and analyze the properties and behaviors of the given system. In addition to the shown functions in Figure 2, HiPS can also analyze other properties (liveness, deadlock, etc.) of a system model. It is important that the system design such as the reactive system needs to run continuously anytime without unexpected stopping, that is, the system does not include deadlock. Property analyzer is useful for detecting potential bugs. Useful functions of the HiPS tool improve visually the display results of structural analysis such as T-invariant and the specification support function for model checking (Harie and Wasaki, 2015).

A Petri net is a particular type of bipartite directed graph consisting of two types of nodes (i.e., places and transitions) (Murata, 1989). Arcs are either from a place to a transition or from a transition to a place.

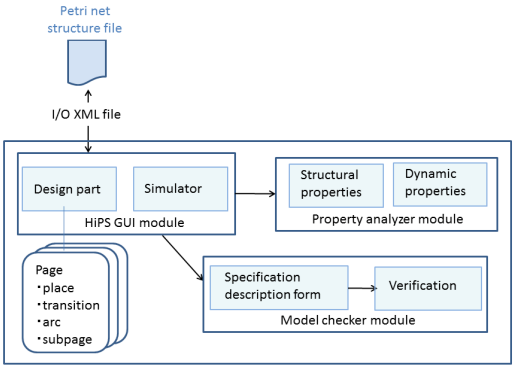

Figure 2: HiPS tool function diagram.

Petri net design can be performed intuitively to connect components on the main screen, such as transitions, places, and arcs. HiPS can observe the behaviors of the design model by simulating random walks from the initial marking. The Petri net data structure in HiPS comprises several pages. The page instance has many elements, for example, place, transition, arc, and subpage. The HiPS GUI uses the .NET Framework, which has a container implemented in C#.

The main HiPS window includes standard menu panels to load/save files, edit models, call the Petri net analyzer, and help users (Figure 1). The GUI buttons to create models are shown in Figure 3. By clicking on the buttons (e.g., arc, transition, and place), the applications operation mode can be switched to edit mode. After selecting these buttons, the user can add a new responding element by clicking on the canvas. After clicking the "Select" button, the user can select elements on the canvas. A place is represented by an empty ellipse. When positioning tokens, we edit the place attribute information. Note that fireable transitions are represented by small blue rectangles.

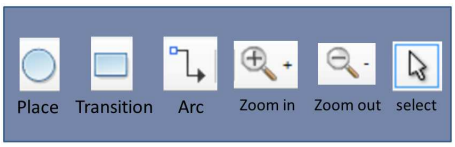

Figure 3: HiPS model design buttons.

We exploit a hierarchy to adjust logic synthesis in order to describe the logic circuits in an embedded system. The tool can generate a hierarchical and timed Petri net design by considering a subpage that describes the Petri net model relative to a given object. HiPS can apply hierarchical modeling to connect lower subpage instances, as shown in Figure 4 where subpage instances are represented by large blue rectangles. A subpage instance possesses transition ports or place ports.

Figure 5 shows the system construction of the composition framework in HiPS. The Petri net design and simulator can be constructed on top of the interface. When performing verifications, such as dynamic analyses, the HiPS GUI loads the state space generation engine in the background using the "wrapper-engine" wrapper application. The core portion of the tool is implemented in C#. HiPS is also implemented in the C++ language for state space generation because C++ has a rich parallel collection and good execution speed. Note that we use the Intel C++ compiler, which is an optimizing compiler, and Intel TBB, which is a C++ parallelization library.

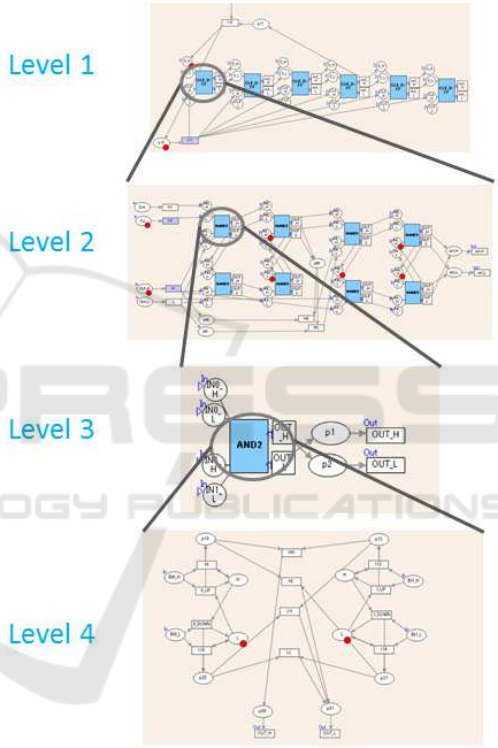

Figure 4: Example hierarchical structure with subpage instances.

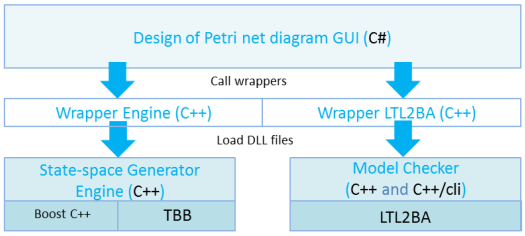

Figure 5: HiPS composition framework.

3 EXHAUSTIVE STATE SPACE GENERATION BY PARALLEL THREADS

The state space generator presents all possible behaviors of the entire system as a labeled transition system (LTS), which comprises the set of all states reachable from the initial state (Ohta and Wasaki, 2013). State spaces are expressed by the LTS and are output in Aldebaran automaton format, which is a file format of the LTS. The LTS labels the transitions between states and describes the system behavior based on an event.

To speed up generation of the state space, we implement multi-threading in the generator engine. Figure 6 shows the flow of the algorithm for the exhaustive state space generation using parallel threads. The generation algorithm of the reachable graph is shown (Murata, 1989). Since the firing evaluation of the marking can be determined, generation of the next marking is finished in a finite step. The choice of marking from the next marking list is nondeterministic and state space does not depend in the order of marking selection. Therefore, it is possible to parallelize the loops for selecting markings from the next marking in the generation algorithm.

By adopting TBB container, we have implemented state space generator by multi-threading in HiPS tool. Newly generated marking(s) in state space would be contained in "*concurrent_buf*", and the generated state space is stored in "*concurrent_map*" Each container is used to be implemented thread-safe container by TBB. The parallel executions operate a series of processes, that is, the firing sequence estimation, inserting and searching a new marking, and obtaining transition relations. The generating process adds a marking to the state space if it finds a new marking from the "*concurrent_map*".

4 DYNAMIC BEHAVIOR AND STRUCTURAL ANALYSIS

4.1 Dynamic Behavior Properties

Examples of dynamic property analysis include reachability graph analysis, (extended) coverability graph analysis, deadlock analysis, k-bounded analysis, reversibility analysis, and synchronic distance and fairness analysis. The dynamic properties depend on the initial marking and are in fact analyzed by generating the set of reachable state spaces.

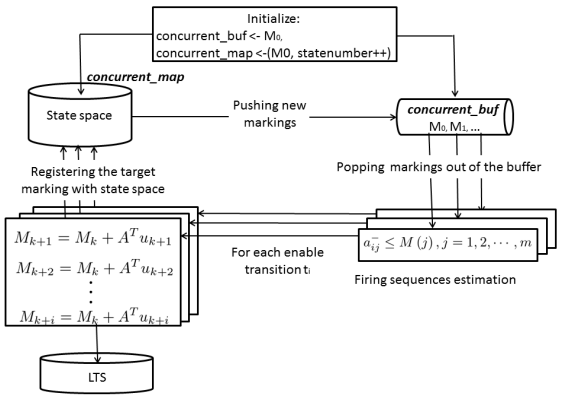

Figure 6: Algorithm flow for exhaustive state space generation by parallel threads.

4.1.1 Reachable Analysis

A marking M_n is said to be reachable from marking M_0 if there exists the firing sequence σ s.t. $M_0 \xrightarrow{\sigma} M_n$. For a net (N, M_0), the set of all markings reachable from M_0 is denoted $R(N, M_0)$. Reachability analysis in HiPS enumerates all reachable markings from the initial marking and the transition relations between markings.

4.1.2 Coverability Graph Analysis

A marking is coverable if there exists M_1 in $R(N, M_0)$ s.t. $M_1(p) \le M(p)$ for all places p in the given net. The coverability analyzer constructs a finite graph structure to represent a reachable marking as a tree node and the firing transition as a tree connection by introducing ω to cut off an unbounded net. If a marking reachable from the initial marking is detected, the transition between the markings is listed.

Note that the current coverability graph generator cannot introduce firing sequences without maintaining information about transitions because the ω notation would lose information about specific behaviors only to manage information about more than one token. Therefore, we introduce ECGs and perform unconditional fair analysis. Note that an ECG is expressed by three special symbols, that is, N_u, N_c, and N_d, rather than ω.

4.2 Structural Properties

Structural properties depend on the topological structure of the Petri nets. These properties are characterized by incidence matrices and homogeneous expressions related to these matrices. Structural properties are independent of the initial marking in that they maintain their properties for any initial marking

or are concerned with the existence of certain firing sequences from some initial marking. Note that we assume all Petri nets are pure when analyzing structural properties.

HiPS can check for seven characteristics, that is, structurally bounded, (partially) conservative, (partially) repetitive, (partially) consistent, structurally unbounded, unconservative, and inconsistent characteristics.

When A is the incidence matrices of the Petri net and x is the firing count vector, the integer solution x of equation $A^T x = 0$ is considered T-invariant, and when A is the incidence matrices of the Petri net and y is the firing count vector, the integer solution y of $Ay = 0$ is considered S-invariant. The partially conservative and partially repetitive properties are present if and only if there exists such an invariant solution. Thus, the properties analyzer can also calculate the invariant.

5 ON-THE-FLY MODEL CHECKING

5.1 Fluent LTL On-the-fly Model Checking in HiPS

Automatic model checking methods are based on state space exploration. However, exploring an entire state space incurs significant memory and time costs. On-the-fly checking reduces the effort required to generate and search the state space (Schwoon and Esparza, 2005). In on-the-fly checking, the search process operates concurrently with the state space generation process. In addition, on-the-fly checking can terminate searching to generate a state space earlier rather than constructing the overall state space when an acceptable sequence is detected.

In association with CADP tools (VASY/INRIA, 2015), HiPS can perform model checking. However, currently, it is essential to construct the full state space for model checking. We have addressed this requirement by implementing on-the-fly model checking to HiPS(Harie and Wasaki, 2015). Of the many formal languages available, we introduce LTL, which is a well-known language used in SPIN (Bell Labs, 2016). To apply LTL model checking to event-based systems, fluent, which is a truth-value predicate defined by events, and fluent LTL (FLTL) have been proposed (Giannakopoulou and Magee, 2003). Note that the FLTL specification does not differ from LTL. We attempted to implement an FLTL on-the-fly model checker.

To achieve on-the-fly model checking, we implemented a verification process for parallel execution of the state space generator. The IPC channel (Stevens et al., 2003) is defined as a communication service within the same machine for remoting in .NET Framework. By using the IPC channel, the state space generator can be expanded to include an LTS transfer function. The state space generation process and the verification process can transmit and receive such data using a remote object to transfer LTS data.

5.2 Priority Estimation for State Space Generation

We aim to improve the efficiency of the implementation of on-the-fly model checking. We implemented on-the-fly model checker by using the Nested-DFS algorithm (Harie and Wasaki, 2016). By operating the implemented model checker, we state the following results. IPC information transfer creates overhead; thus, one approach is to improve the transfer method. Considering the data transfer overhead, it is desirable to transfer data collectively rather than sequentially. Recently, we use an LTS, in which transferred data are represented by a data structure that uses vector classes. We seek to redefine this format to its data format in order to generate more compact data (e.g., unsigned integers).

Here, we describe our state space generation approach. The Nested-DFS algorithm searches acceptable sequences in synchronization automaton. So a Path to acceptable sequences depends on specification, it is desirable to generate state space on specifications. We consider a strategy in which the transition in the given specification is considered high priority. Note that a high priority transition is fired preferentially. Figure 7 shows the state space generation flow with the priority estimation strategy.

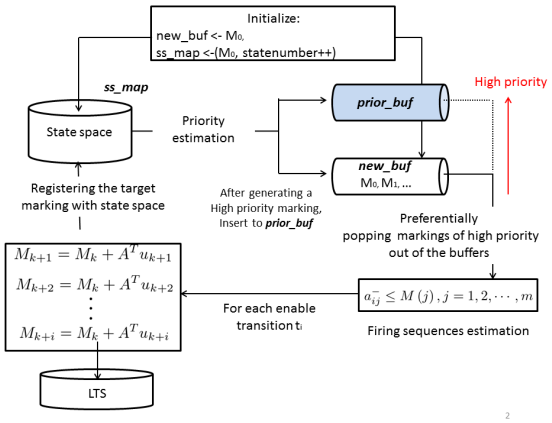

Figure 7: Algorithm flow of state space generation with the priority estimation strategy.

Note that information about transitions with higher priority can be obtained before the state space is generated. Currently, in state space generation, a single buffer is used to store new markings. Therefore, when a new marking is generated, the firing possibility is evaluated from the marking, and the highest priority marking is selected if a given condition applies. If a priority marking is selected, it is inserted into the priority generation buffer; otherwise, it is inserted into the normal buffer. After generating a high priority marking, the algorithm inserts it into the beginning of the normal buffer.

Here, we consider methods to prioritize transitions. One approach is to set priorities by analyzing a specification automaton in advance. The rating analysis for a specification automaton is expected to be calculated relatively quickly because the number of states in the specification automaton is much smaller than that of the state space. Another approach is to update priority relative to transition appearance frequency in the set of priority transitions when generating the state space.

6 CONCLUSION

In this paper, we have shown how the HiPS tool can be used to simulate hierarchical modeling of Petri nets and analyze the structural and dynamic properties of Petri nets. The current release supports ECGs with synchronic distance. In addition, the HiPS tool incorporates on-the-fly model checking to describe LTL (FLTL) and accomplish specification description support. The technology described in this paper has potential as an integrative execution system. Future work includes support for model correction and model checking by simulating obtained counter examples. SPEC PATTERNS have been proposed to describe property specifications for finite-state verification (SAnToS laboratory, 2015).

REFERENCES

Bell Labs (2016). Verifying Multi-threaded Software with Spin. http://spinroot.com/spin/whatispin.html.

Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

Dingle, N. and Knottenbelt, W. (2016). QPN-Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets. http://pipe2.sourceforge.net/.

Falko Bause, P. B. and Kemper, P. (1996). Platform Independent Petri net Editor 2. http://ls4-www.cs.tu-dortmund.de/QPN/QPN-TOOL_ article/article/article.html.

Giannakopoulou, D. and Magee, J. (2003). Fluent model checking for event-based systems. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 , 2003*, pages 257–266. ACM.

Harie, Y. and Wasaki, K. (2015). On-the-fly LTL Model Checker on the Petri Net Design Tool : HiPS. In *14th Forum on Information Technology, FIT2015*, pages 139–142.

Harie, Y. and Wasaki, K. (2016). Formal Verification of the Safety Testing for Remote Controlled Consumer Electronics Using the Petri Net Design and Tool: HiPS. In *5th IEEE Global Conference on Consumer Electronics, GCCE2016*, pages 290–294.

Iakushkin, O., Shichkina, Y., and Sedova, O. (2016). *Petri Nets for Modelling of Message Passing Middleware in Cloud Computing Environments*, pages 390–402. Springer International Publishing, Cham.

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

Ohta, I. and Wasaki, K. (2013). Model Designing using A Petri Net Tool and State Space Generation Algorithm for Post-Verification Tool. In *12th Forum on Information Technology, FIT2013*, pages 171–174.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

Reisig, W. (1985). *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer.

SAnToS laboratory (2015). Spec Patterns. http://patterns.projects.cis.ksu.edu/.

Schwoon, S. and Esparza, J. (2005). A note on on-the-fly verification algorithms. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, pages 174–190.

Stevens, W. R., Fenner, B., and Rudoff, A. M. (2003). *UNIX Network Programming, Vol. 1*. Pearson Education, 3 edition.

VASY/INRIA (2015). CADP toolbox. http://cadp.inria.fr/.

Westergaard, M. and Verbeek, H. E. (2016). CPN tools. http://cpntools.org/.