N-PAT: A Nested Model-Checker
(System Description)

Hadrien Bride[1], Cheng-Hao Cai[2], Jin Song Dong[1,3], Rajeev Gore[4], Zhé Hóu[1], Brendan Mahony[5] and Jim McCarthy[5]

[1] Institute for Integrated and Intelligent Systems, Griffith University, Australia
[2] School of Computer Science, University of Auckland, New Zealand
[3] School of Computing, National University of Singapore, Singapore
[4] Research School of Computer Science, The Australian National University, Australia
[5] Defence Science and Technology, Australia

Abstract. N-PAT is a new model-checking tool that supports the verification of nested-models, i.e. models whose behaviour depends on the results of verification tasks. In this paper, we describe its operation and discuss mechanisms that are tailored to the efficient verification of nested-models. Further, we motivate the advantages of N-PAT over traditional model-checking tools through a network security case study.

1 Introduction

Model-checking is the problem of formally verifying that a model of a system meets a given specification. Automated model-checking techniques have been successfully applied to find subtle errors in complex industrial designs of e.g., hardware circuits, software controllers, and communication protocols [1]. However, the adoption rate of model-checking remains low in software engineering because of the computational complexity of model-checking algorithms. The state-space explosion problem [2] makes the verification of large models intractable unless high-level abstractions are used in the development and leveraged during verification.

Nowadays, complex systems are often designed in a modular and hierarchical fashion. Hierarchical models, also called multilevel models, are abstract representations of systems that span multiple levels of abstraction. They encode the hierarchical structure of systems explicitly and therefore enable reasoning about how properties of one level reflect across multiple levels of the model [5].

In this paper, we introduce the notion of nested model and nested model-checking. The main idea is to break up a large model-checking task into a hierarchy of smaller model-checking tasks. A Nested model is a high-level model which may contain several child models nested inside; its behaviour depends on the verification results of its child models. Note that the properties to be verified in child tasks may be different from the properties to be verified in parent tasks.

We present N-PAT – a nested model-checker suited to the verification of hierarchical systems and designed to perform nested model-checking tasks. In hierarchical modelling, some verification tasks may be used to determine and lift the properties of

underlying child models to parent models. This structural abstraction provides modellers with the ability to structure the verification and guide the state-space exploration of model-checking methods, it also provides significant benefits in term of scalability for verification when compared to the traditional approach to modelling. We implement several optimisations leveraging the hierarchical structure of nested models. Also, since the time and space complexity of model-checking algorithms with respect to the size of models is super-linear, the divide-and-conquer approach employed by N-PAT significantly reduces the overall verification time. What sets N-PAT apart from existing model checkers (e.g., [7], [6], [4]) is the abstraction level of the modelling language. In our work, the modelling language of nested models has high-level primitives such as model checking and nested model instantiation.

2 Nested Model-checking

Standard model checking is the problem of verifying whether a *standard* model complies with a given property. A standard model is a static and finite-state representation of a system, which may exhibit non-deterministic and probabilistic behaviours. The semantics of a standard model can be specified as a labelled transition system or a Markov decision process. Properties that can be verified include reachability, deadlock-freeness, divergence-freeness, reachability, and LTL formulae. The result of a model checking task depends on the type of the model. When checking a property over a non-probabilistic model, the result is 0 (not satisfied) or 1 (satisfied). When checking a property over a probabilistic model, the result is the min (alt. max) probability that the property is satisfied. Note that we only consider results in natural numbers, and a probability is represented in e.g., per thousand. Formally, let M^s be the set of standard models and Φ be the set of properties. We denote by $\texttt{mc} : M^s \times \Phi \to \mathbb{N}$ the model checking function that returns the results of checking a property over a standard model.

A *meta model*, also commonly referred to as a *template*, is a model of standard models. It can be viewed as a function that has a finite number of arguments and returns a standard model. Formally, a meta model is a function of the form $A_1 \times ... \times A_n \to M^s$ where $n \in \mathbb{N}$ and $A_1, ..., A_n \in \mathbb{N}$. We denote by M^m the set of meta models. In order to instantiate a meta model, every argument must be known. An instantiated meta model is a standard model and can be verified using standard model checking.

Traditionally, meta models are instantiated from values specified by the modeller. In our work, we consider the verification of meta models instantiated from values that are the result of model checking tasks. Such a meta model is called a *nested* model and denoted as M^n. Figure 1 illustrates the structure and components of a nested model. Each diamond represents a standard model. Each box represents a meta model. Verification tasks are symbolised by circles. The text within each circle is the property to be checked in the corresponding verification task. The arrows symbolise dependencies among verification tasks. In Figure 1, there are two standard models: M_2 and M_3, and two meta models: M_0 and M_1. For instance, M_1 requires the verification results from $\texttt{mc}(M_2, \phi_2)$ and $\texttt{mc}(M_3, \phi_3)$ to be instantiated. After instantiation, M_1 will become a standard model. To verify the property ϕ_0 over the nested-model M_0, we need to evaluate the following expression: $\texttt{mc}(M_0(\texttt{mc}(M_1(\texttt{mc}(M_2, \phi_2), \texttt{mc}(M_3, \phi_3)), \phi_1), \texttt{mc}(M_3, \phi_4)), \phi_0)$.

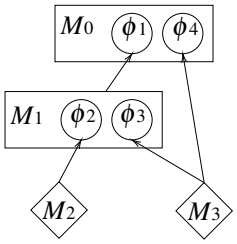


Fig. 1: Illustration of the structure of a nested model.

A nested model checking problem is an expression that can be evaluated to an integer; it is formulated in a language that has two main primitives: meta model instantiation and standard model checking. For convenience, the language of nested model checking problems is extended with integers, basic arithmetic operations and restricted scope constant definitions. Let *const* be the set of constant identifiers, the syntax of nested model checking problems is given by the grammar in Figure 2, where $[\alpha]^*$ denotes repeating α zero or more times.

$$
\begin{array}{ll}
\mathit{binding} ::= \mathit{const} \textbf{ = } \mathit{expr} & \mathit{mc} ::= \textbf{mc(}\mathit{model}\textbf{, } \Phi\textbf{)} \\
\mathit{model} ::= M^s \mid M^m\textbf{(}\mathit{binding}\ [\textbf{, } \mathit{binding}]^*\textbf{)} & \mathit{op} ::= \mathit{expr}\ (+ \mid - \mid \times \mid /)\ \mathit{expr} \\
\mathit{def} ::= \textbf{let}\ \mathit{binding}\ [\textbf{, } \mathit{binding}]^*\ \textbf{in}\ \mathit{expr} & \mathit{expr} ::= \mathbb{N} \mid \mathit{op} \mid \mathit{mc} \mid \mathit{const} \mid \mathit{def}
\end{array}
$$

Fig. 2: The BNF grammar of nested model checking problems.

We assume that the set of verification tasks related to a nested model form a directed acyclic graph (as in Figure 1), which defines the dependencies of tasks, and that the tasks and the graph are known before the verification stage. When a `let` expression introduces multiple bindings, these bindings must be independent of one another, non-recursive, and may be evaluated in parallel.

The semantics of nested model checking problems in a given *context* is defined by the *evaluation function* `eval`, given in Figure 3. A *valued* binding is a tuple $\langle c, v \rangle$ where $c \in \mathit{const}$ and $v \in \mathbb{N}$. A context is a set of valued bindings. Let Γ be a context, $e, e_1, ... \in \mathit{expr}$ be expressions, $c, c_1, ... \in \mathit{const}$ be constant identifiers, $m_s \in M^s$ be a standard model, $m_m \in M^m$ be a meta model, $m \in M^s \cup M^m$ be a model, $\phi \in \Phi$ be a property, $n, v, v_1, ... \in \mathbb{N}$ be numbers, and $op \in \{+, -, *, /\}$ be an integer operator.

$$
\begin{array}{lll}
\text{eval}(n, \Gamma) = n & \text{eval}(m_s, \Gamma) = m_s & \text{eval}(c, \Gamma) = v \text{ where } \langle c, v \rangle \in \Gamma \\
\text{eval}(e_1\ \ op\ \ e_2, \Gamma) = \text{eval}(e_1, \Gamma)\ \ op\ \ \text{eval}(e_2, \Gamma) & & \text{eval}(\text{mc}(m, \phi), \Gamma) = \text{mc}(\text{eval}(m, \Gamma), \phi)
\end{array}
$$

$$
\begin{array}{l}
\text{eval}(m_m(\{\langle c_1, e_1 \rangle, ..., \langle c_n, e_n \rangle\}), \Gamma) = \text{m}(\{\langle c_1, v_1 \rangle, ..., \langle c_n, v_n \rangle\}) \text{ where} \\
\qquad v_1 = \text{ eval}(e_1, \Gamma), ..., v_n = \text{ eval}(e_n, \Gamma)
\end{array}
$$

$$
\begin{array}{l}
\text{eval(let } \{\langle c_1, e_1 \rangle, ..., \langle c_n, e_n \rangle\} \text{ in } e) = \text{eval}(e, \Gamma') \text{ where} \\
\qquad \Gamma' = \Gamma \cup \{\langle c_1, \text{eval}(e_1, \Gamma) \rangle, ..., \langle c_n, \text{eval}(e_n, \Gamma) \rangle\}
\end{array}
$$

Fig. 3: The semantics of nested model checking problems.

3 N-PAT: Implementation

N-PAT is built on top of Process Analysis Toolkit [7] (PAT) – an industrial scale model-checker which employs an expressive modelling language called Communicating Sequential Processes with C# (CSP#) developed by Hoare [3] and others [9]. PAT features a model editor and an animated simulator using a mature and IDE-style user interface. Further, PAT facilitates new language and algorithm design and implementation as extended modules. Over the past 10 years, we have extended PAT with new verification modules for timed automata [9], real-time systems [8], and probabilistic systems [10]. We implemented N-PAT in C# as an extension of PAT. N-PAT is open-source and freely available online.[6]

Standard models are specified by (probabilistic) CSP# models and properties are specified by CSP# assertions [7]. Meta models are specified by a *meta-level* CSP# language. This language, called meta-CSP#, introduces labelled place-holders, of the form $[id]$ where $id \in const$ is a label. These labelled place-holders extend the CSP# language and can be used in place of integer constants (e.g., variable initial values, choice probabilities). Let m be a meta-CSP# model and $id_1, ..., id_n$ where $n \in \mathbb{N}$ be the set of placeholders that appears in its definition. Let $v = \{\langle id_1, v_1\rangle, ..., \langle id_n, v_n\rangle\}$ where $v_1, ..., v_n \in \mathbb{N}$ be a set of *valued* bindings. The meta-CSP# model m can be instantiated using v into a CSP# model by substituting the occurrences of $[id_i]$ by v_i for all $i \in \{1, ..., n\}$. Nested model checking problems are specified using the language described in Section 2. Model checking is performed by N-PAT through the orchestration of calls to PAT.

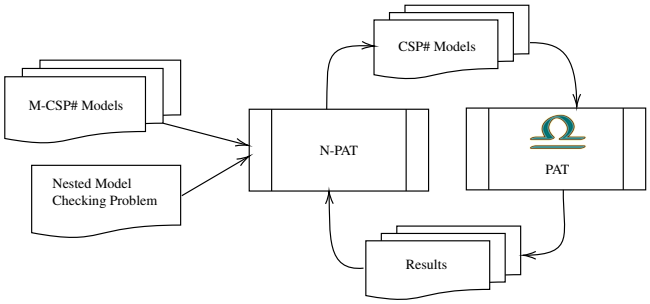


Fig. 4: N-PAT data-flow overview.

Figure 4 depicts the overall data-flow of N-PAT. The input of N-PAT is a set of standard CSP# and meta-CSP# models and a nested model checking problem. N-PAT evaluates the result of the nested model checking problem similarly to how a dynamic interpreter evaluates an expression. The nested model checking problem is first parsed, and a corresponding abstract syntactic tree is built. This step is implemented using parser combinators (i.e., using recursive descent parsing). The resulting abstract syntactic tree is then recursively evaluated in a bottom-up fashion.

[6] `https://formal-analysis.com/research/npat/index.html`

N-PAT exploits the hierarchical nature of nested models and provides improved verification scalability when compared to traditional model checkers that operate on flattened models. First, since CSP# models are static (i.e., they are not modified during execution), N-PAT applies stage-wise partial evaluation of verification sub-tasks to optimise the verification phase of nested CSP# models. Second, given a nested CSP# model, we assume that its verification sub-tasks are independent and can be computed concurrently. Thus N-PAT uses parallelism to speed up verification on modern architectures with multiple cores. This parallelism manifests itself in three places: bindings, operands, and the evaluation of meta model arguments.

4 Case Study and Experiment

We introduce a network security case study to illustrate the modelling and scalability advantages of nested model-checking. The case study is concerned with computing the probabilistic security level of a network. It is a simplified version of a real-life example which is studied by Australian Defence. The problem is hierarchical by nature, which illustrates code-reuse and modularisation of the proposed modelling approach. Another nice property of this example is that we can create models of different sizes to test the scalability of the verification, as will be shown in the experiment.

The details of the example are as follows: suppose there is a cluster of computation nodes, and each node can be in one of the three states: *safe*, *compromised*, and *isolated*. Initially, each node is safe, but it has a chance to be *hacked*, which changes the state of the node to compromised. When a node is compromised, it can either be *patched*, which will make the node safe again, or be *isolated*, which will disconnect the node from the cluster. If a node is isolated, then it loses the connection to other nodes and thus cannot contribute to the computational power of the cluster. When the node is isolated, it has a chance to be *recovered*, which will lead the node to the safe state again. Otherwise, the node stays isolated, in which case we increase `down_nodes_counter`, i.e., the number of nodes that are offline, by 1. For simplicity, we assume that the hacking of each node is independent. We model two types of nodes: *normal* node and *premium* node, where the latter has a higher chance to be patched when it is compromised and a higher chance to be recovered when it is isolated.

Traditional Method: We shall formalise the above as a Markov chain in CSP#, and we have to define the state transitions for both types of nodes. Next, we model a `cluster manager`, which iterates through each node in the cluster and checks whether the node is offline. The manager will report that the cluster is in critical condition if at least half of the nodes in the cluster are offline. Consider an example where there are only two nodes in the network: the first node is a normal node, and the second node is a premium node. We show the overall Markov chain in Figure 5. We annotate the event and its probability on the arrows. The upper part of the figure describes the process of checking the normal node, which can be either safe or isolated. The lower part of the figure splits into two cases when checking the premium node. The `Down = x` line in each circle indicates the `down_nodes_counter`. The premium node in Figure 5 has a higher chance to be hacked than because in our hypothetical scenario, the hacker is more likely to target a premium node.

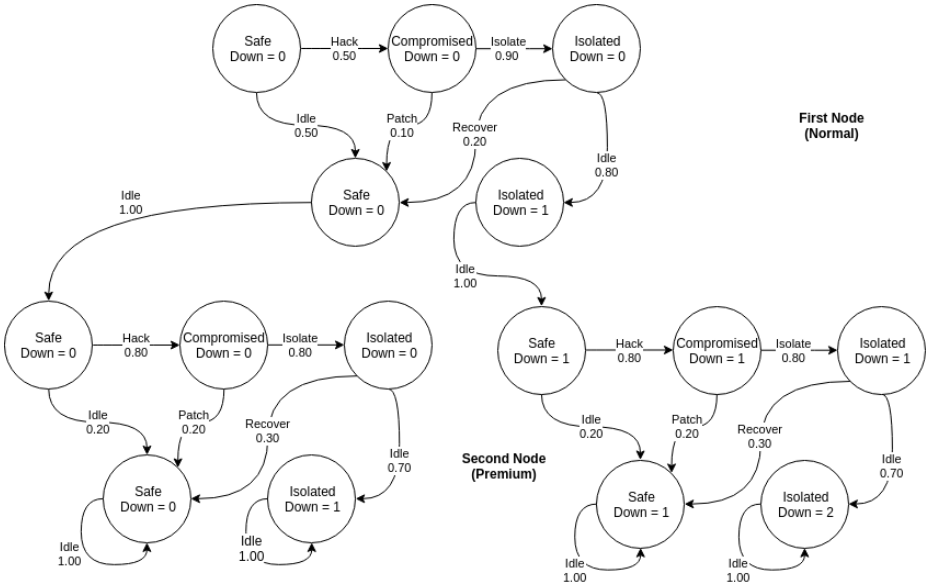
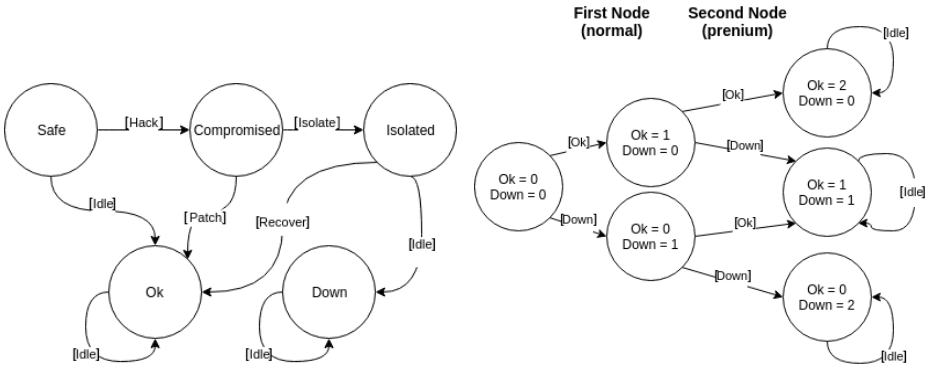


Fig. 5: The Markov chain of the traditional model, exemplified with two nodes.

Nested Model Checking Method: We break the model in Figure 5 down into two levels of abstraction: the node level and the cluster level. The idea is to create more modular models so that the model for a node can be used for both normal nodes and for premium nodes, and potentially can be used in future developments of other models. At the node level, we study the common properties of the two types of nodes, and try to generalise the model so that the model-checking result is exactly what we need at the cluster level. As the cluster manager, we only need to know whether a node is offline or not. Therefore, besides safe, isolated, and compromised, we give a node two additional states: `ok` and `down`. Like in the traditional model, each node is initialised to be in the safe state. Since we do not consider cluster manager operations at the node level, we do not use the counter for offline nodes. Instead, when the node stays isolated, we change its state to `down`, and when it is not hacked/patched/recovered, we change its state to `ok`. The state transition diagram is shown in Figure 6a.

At the cluster level, we abstract the notion of a node into only two states: `ok` and `down`. The probability of going to these two states will be given by the model-checking results from the node level. If a node is down, then we increment `down_nodes_counter`. We then model the cluster manager similarly as in the traditional model. The (partial) Markov chain for checking nodes is illustrated in Figure 6b, where we only show two nodes. Note that places holders for probabilities in Figure 6a, such as `[Isolated]`, will be instantiated with specific values, depending on the type of the node, at run-time. Place holders in Figure 6b will be instantiated with the results of node-level verification at run-time. Compared to the traditional model (cf. Figure 5), the nested model is much simpler, and we will show that this leads to significant improvement in scalability.



(a) Node Level (b) Cluster Level

Fig. 6: Modular models of Figure 5 for nested model checking.

Experimental comparison: We compare the performance of both modelling approaches by evaluating the overall security of network of different sizes. In each case, the number of normal nodes are $4/5$ of the total number. All experiments were carried on a desktop with Core i7-7700 quad-core processor at 3.6GHz and 32GB RAM. We verify multiple instances of the above models, starting with 8 nodes in a cluster, and increase the number of nodes by 2 at a time, and maintain the number of normal nodes at $(4 \times num\_of\_nodes)/5$. We then observe the time used in model checking as the number of nodes increases. We run each test 5 times and compute the average time spent to obtain the results.

Number of Nodes	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**	**30**	**32**	**34**
Runtime Traditional (ms)	248	306	569	1771	7411	35K	279K	Out of memory						
Runtime N-PAT (ms)	427	430	430	430	430	431	445	438	442	461	458	469	465	476

Table 1: Experiment of traditional (probabilistic) model-checking compared with nested model checking.

Discussion: As seen from the results in Table 1, the run-time of traditional model-checking grows rapidly as the size of the model (the number of nodes) increases. On the other hand, the run-time growth of nested model checking is moderate, and it solves instances up to 34 nodes less than 0.5 seconds. N-PAT also uses very little memory compared to PAT which uses up to 26.6GB memory when running the 20 nodes instance. For small examples, the traditional modelling approach may be faster because the verification of nested models involves several calls to PAT which incur marginal overhead. However, the nested model checking approach scales better. The source code of this experiment, i.e. both the traditional model and the hierarchical model, can be found online.[7]

[7] `https://formal-analysis.com/research/npat/examples.html`

5 Conclusion and Future Work

We presented N-PAT – a high-level model checker that enables the verification of models that relies on the results of other verification tasks. We demonstrated in a case study in network security that this tool permits the use of high-level abstraction mechanisms and can therefore significantly improve the time and memory efficiency of verification tasks. These results indicate that nested model checking provides a novel modelling approach that can in some cases scale better than traditional model-checking.

In future work, we intend to provide more modelling flexibility by allowing dynamic calls to verification tasks that are not known a priori. We also planned to apply dynamic language optimisation techniques such as memoisation to speed up verification. Finally, we planned on supporting a fully reflective modelling language that permits inspection and modification of the behaviour and structure of models at verification-time.

References

1. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
2. Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.
3. Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
4. Jesús J López-Fernández, Esther Guerra, and Juan De Lara. Meta-model validation and verification with metabest. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 831–834. ACM, 2014.
5. Ovidiu Pârvu and David Gilbert. A novel method to verify multilevel computational models of biological systems using multiscale spatio-temporal meta model checking. *PloS one*, 11(5):e0154847, 2016.
6. Bernhard Steffen and Alnis Murtovi. M3c: modal meta model checking. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 223–241. Springer, 2018.
7. Jun Sun, Yang Liu, and Jin Song Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In *International symposium on leveraging applications of formal methods, verification and validation*, pages 307–322. Springer, 2008.
8. Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Trans. Softw. Eng. Methodol.*, 22(1):3:1–3:29, March 2013.
9. Jun Sun, Yang Liu, Jin Song Dong, and Xian Zhang. Verifying stateful timed CSP using implicit clocks and zone abstraction. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering*, pages 581–600, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
10. Jun Sun, Songzheng Song, and Yang Liu. Model checking hierarchical probabilistic systems. In *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering*, ICFEM'10, pages 388–403, Berlin, Heidelberg, 2010. Springer-Verlag.