HYASM: a Tool to Verify Hierarchical Systems

Angelo Ferrando
University of Genoa
Genoa, Italy
angelo.ferrando@unige.it

Vadim Malvone
Telecom Paris
Palaiseau, France
vadim.malvone@telecom-paris.fr

Aniello Murano
University of Naples Federico II
Naples, Italy
aniello.murano@unina.it

Silvia Stranieri
University of Naples Federico II
Naples, Italy
silvia.stranieri@unina.it

***Abstract*—Hierarchical state machines represent a natural and useful framework to model and reason about modern systems. These machines encompass the ability to model hierarchical systems where some of the components can be reused in different contexts, e.g., by hierarchically calling subsystems. However, classical model checkers lack support to properly deal with hierarchical systems. Mostly, they treat the hierarchical calls as generic, possibly recursive, procedure calls. In this paper, we present HYASM a model checker for hierarchical systems as an extension of the tool YASM, a symbolic model-checker based on the CEGAR paradigm. Our tool uses a suitable flattening approach over hierarchical state machines, and experimental results show that our approach works very well in practice.**

***Index Terms*—Hierarchical State Machines, Model Checking, Temporal Logics**

I. Introduction

In formal system design, one of the most significant developments has been the discovery of *model checking* [17], [34]. Such a method allows to verify whether a system is correct with respect to a desired behaviour by translating the system into a labeled-state transition graph (a.k.a. *Kripke structure*) K, translating the desired behaviour into a *temporal-logic formula* φ (i.e., a temporal specification), and then checking whether K satisfies φ (formally, $K \models \varphi$) [18]. This verification method has been deeply studied in the past 40 years, implemented in tools used by industries in practice, and extended in several ways to deal with very complicated systems, including reactive and multi-agent systems [3], [26].

The translation of a system into a model K usually involves a painful blow-up, and the size of the model is typically the computational bottleneck in the model-checking algorithm. One source of such a blow-up regards coping with the procedure calls, which requires an heavy duplication of states, especially in the case of recursive calls. However, many real systems are inherently hierarchical and the procedure calls are just finite hierarchical calls to sub-components, with the possibility of calling the same component in different places [4], [6]. In [6] a formal modelling of hierarchical systems has been proposed by means of *hierarchical state machines*, that is a tuple of structures $H = (K_1, \ldots, K_n)$, where each K_i is a classical Kripke structure (e.g., a labeled-state transition graph) and, additionally, each $K_{i<n}$ is equipped with special states called *boxes* or *superstates* that correspond to nested calls to $K_{j>i}$ structures. In [6], a suitable model checking procedure for hierarchical systems has been introduced, which encompasses a "flattening" of the hierarchical state-machine model. Specifically, the procedure first transforms the model in a classical Kripke structure (i.e., with no boxes) by repeatedly substituting box-references to sub-structures with copies of these sub-structures, and then call a classical model checking procedure. Thanks to the hierarchical structures of the calls, the derived Kripke structure is finite. In [6], [7], [21], it has been shown that this approach, although exponential in the worst case (due to the possibility of having different calls to the same sub-structure), is the best one can use in case of branching-time temporal logics. Also, a closer look to the computational complexity of the algorithm shows that it performs very well in case the nesting depth of the hierarchical model is small (compared with the total number of states), which often occurs in practice [7]. Surprisingly, existing model-checking tools lack any support to deal with hierarchical state machines, and in particular flattening algorithms have never been implemented and tested in practice for temporal-logics specifications.

Our contribution. In this paper we present HYASM, a model checking tool for hierarchical systems based on an extension of the tool YASM [22], a symbolic model checker based on the CEGAR paradigm [16]. Our idea is to show that by using an ad-hoc method to handle hierarchical calls, we let the model checking to perform much better than handling the calls as in the classical recursive case. Our approach works as follows: given a C-program P with hierarchical procedure calls and a branching-time temporal logic specification φ, we first build a hierarchical state machine H of P, then we build a flat expansion F of H, and finally we call the classical YASM engine to check whether $F \models \varphi$. We give an evidence of a better performance of our approach by means of experimental results, comparing our method with the classical one provided by YASM. This paper originated from [30].

To conclude, we want to summarize the strengths and weaknesses of YASM that have influenced our decision to extend it for handling hierarchical model checking. On the one hand, YASM takes input programs with procedures, making it easy to represent them as hierarchical models, that is a model for each procedure. In addition, since YASM employs an abstraction-refinement method, it incorporates the concept of abstract state, that is closely related to the box state. On the other hand, YASM requires the help of a domain expert to define the right abstraction, impacting its success in recent years. However, with the introduction of IA, we believe that this tool will strengthen its position.

A. Related Work

Hierarchical system verification has received a lot of attention from the formal-verification community, in the past 25 years [4], [9], [11], [13], [14], [19], [20], [25], [27], [31], [33]. Mainly, the work has concentrated on understanding in which situations it is better not to flatten the hierarchical model (using an ad-hoc hierarchical verification procedure) or, conversely, state that one cannot gain much with respect to the flattening approach. Notably, most of the works only report a theoretical study.

In [4], [6], it is shown that for LTL model checking, it is much better not to flatten the hierarchical model (saving an exponential blow-up), whereas for CTL the exponential blow-up is unavoidable, except for very restricted cases. The authors in [7], [21] reached the same conclusion for μ-calculus specifications, showing that one can trade for an exponential blow-up in the (often much smaller) size of the formula and the maximal number of exits of sub-structures. Slightly more efficient model-checking algorithms have been obtained by considering hierarchical state machines in which also boxes (in addition to regular states) are labeled with atomic propositions [28], [29].

In [2], [5], a verification tool for hierarchical systems called *HERMES* is presented. However HERMES just deals with reachability specifications and, contrarily to our approach, it works directly on the hierarchical representation of the hierarchical system, by exploiting its modularity.

In [35] a flattening technique has been also considered and implemented in a tool named *SCOPE* - a code generator targeting constrained embedded systems. However, the flattening approach used there is substantially different from the expansion to a product machine we use. In fact, the flat procedure used in SCOPE is a predecessor language of statecharts: a set of traditional Mealy machines operating concurrently.

The extension of hierarchical state machines with recursive calls have been studied in [1]. Recursive machines are equivalent to pushdown systems [10]. Formal verification of pushdown systems is also an important field of research (e.g., [8], [12], [24]) with recent promising developments in the multi-agent setting [15], [32]. Note however that flattening a recursive machine may produce an infinite state model.

II. Hierarchical Systems

Hierarchical systems can be modeled by means of *hierarchical state machines*, i.e., hierarchical labeled-state transition graphs, where states can be ordinary states (as in classical Kripke structures) or *superstates* (also named *Boxes*) representing hierarchical calls to submodels. Hierarchical state machines have become popular not only because the superstates allow to model the intrinsic modularity of hierarchical systems, but also because hierarchical models allow a sharing of preexisting model components, to be used in different contexts [4].

As an example, in Figure 1, we provide a hierarchical state machine for a digital clock [29]. The model $K1$ is made of 24 superstates, one for every hour of the day. Each of them, in turn, calls a machine $K2$, which is composed of 60 superstates, one for every minute, and each superstate, in turn, calls a machine $K3$. Finally, machine K3 is made of 60 ordinary states, one for every second, in fact, $K3$ is a classical Kripke structure. Remarkably, as shown in the figure, each superstate of the model $K1$ refers to the same model $K2$, and each superstate in $K2$ refers to the same model $K3$, but in different contexts. The hierarchical approach, in this case, considerably reduces the number of states used to specify the behavior of the system, allowing the reuse of the same structure in several contexts. In the following, we are going to formally define a hierarchical state machine (formally, a hierarchical Kripke structure) and the corresponding flattened structure.

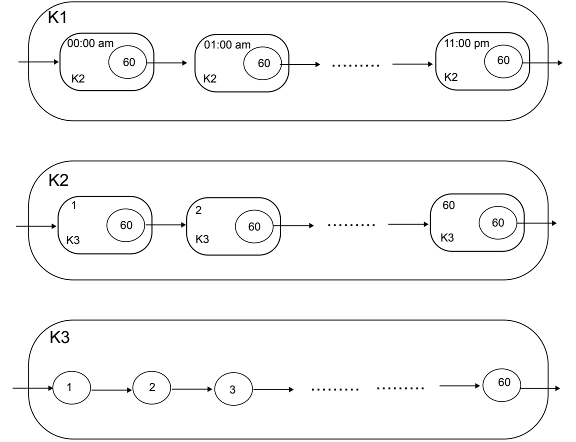

Fig. 1: Hierarchical model of a digital clock.

Definition 1 (Hierarchical Kripke Structure): A hierarchical Kripke structure $\mathcal{K}$, over a set of atomic propositions AP, is a tuple of structures $\langle K_1, \ldots K_n \rangle$, where each $K_i = \langle N_i, B_i, n_0^i, O_i, X_i, Y_i, E_i \rangle$, such that:

- N_i is the finite set of states or nodes;
- B_i is the finite set of *superstates* or boxes;
- $n_0^i \in N_i$ is the starting node;
- $O_i \subset N_i$ is the set of exit nodes;
- $X_i : N_i \to 2^{AP}$ is an evaluation function associating the subset of atomic propositions holding in each node;
- $Y_i : B_i \to \{i+1, \ldots, n\}$ is a function associating an index to each box. Such an index leads to one of the underlying structures to which the box refers to;
- $E_i \subseteq (N_i \cup B_i) \times (N_i \cup B_i)$ is the set of edges. Precisely, in the case of edges (b, v) having a box as the source node, we can write such an edge as the pair $((b, o), v)$, where $b \in B_i$, and $o \in O_j$ with $j = Y_i(b)$.

Notice that K_n does not contain any box. In fact, it is a classical Kripke structure.

A hierarchical Kripke structure can be "flattened" into a standard Kripke structure (that is, without boxes) by recursively replacing each box (*superstate*) with the corresponding associated structure as described below. Note that the flattening needs to remember the context in which each substructure

is called (see [6] for more details and proofs of semantic equivalence).

Definition 2 (Flattened Structure): Given a Hierarchical Kripke Structure $\mathcal{K} = \langle K_1, \dots K_n \rangle$, its flattened structure is denoted by $\mathcal{K}^F$ and it is computed by flattening each K_i of $\mathcal{K}$ from n to 1. Formally, $\mathcal{K}^F = K_1^F$ where, for each i, the flattened structure $K_i^F = \langle S_i, in_i, \delta_i, \lambda_i \rangle$ is as follows:

- $S_i = N_i \cup \{(u, v) \mid u \in B_i \wedge v \in S_j, \text{with } j = Y_i(u)\}$ is the set of states made of (i) all the ordinary nodes of K_i, and (ii) the pairs of *superstates* and nodes of the flattened Kripke structure they refer to.
- $in_i = n_0^i$ is the starting state, which is as in K_i.
- $\delta_i = \{(u, v) \in E_i \mid v \in N_i\} \cup \{(u, (v, in_j)) \mid (u, v) \in E_i, v \in B_i \wedge Y_i(v) = j\} \cup \{((w, u), (w, v)) \mid w \in B_i \wedge Y_i(w) = j \wedge (u, v) \in \delta_j\}$ is the transition relation s.t.:
  - Each pair $(u, v) \in E_i$, such that the target v is an ordinary node, is added to δ_i;
  - Each pair $(u, v) \in E_i$, such that the target v is a *superstate*, a transition from the source node u to the initial state of the structure to which the box refers to is added to δ_i;
  - For each *superstate* w in K_i referring to the Kripke structure K_j and for each transition (u, v) in K_j^F, a transition $((w, u), (w, v))$ is added to δ_i.
- $\lambda_i(w) = \begin{cases} X_i(w) & \textit{if } w \in N_i \\ \lambda_j(v) & \textit{if } w = (u, v) \wedge u \in B_i \wedge Y_i(u) = j \end{cases}$ is an evaluation function, corresponding to the evaluation function of the ordinary nodes in K_i, or, in case of a *superstate*, to the evaluation function of the associated nodes in the flattened Kripke structure.

III. YASM: A SYMBOLIC MODEL CHECKER

In this section, we present the model cheker YASM, along with its main features. We also provide a running example that will be used in the rest of the paper.

YASM [22] is a symbolic model checker based on the Counter-Example Guided Abstraction Refinement (CEGAR) framework [16]. A typical YASM execution requires the following phases:

1) Abstractions are represented through Boolean programs, by approximating weakest precondition of program statements.
2) The model checking is performed through a BDD-based symbolic model checker.
3) The model checker provides counterexamples for inconclusive properties.
4) When the verification result is not defined, a refinement phase can be performed to determine a more precise abstraction.

The main strength of the tool is the determination of three possible states [22]:

- States from which the error is unavoidable.
- States from which the error is unreachable.
- States that have paths leading to the error.

Only the third set of states may lead to undefined results, requiring the refinement phase.

In the following, we explain the main features of the tool and how we modify them to manage a hierarchical model checking, starting with a running example. We present a simple program in pseudo-code (Algorithm 1) that will be used in the sequel to show the behavior of the tool. The resulting C program is what we verify with YASM. In order to do so, we define a CTL formula, such as $\phi = EF(pc = END)$, meaning that we are asking whether, for any path, eventually the program terminates.

Algorithm 1 Example

(a) Main
```
y ← 10
y ← y + 1
if y == 11 then
    y ← PROCEDURE(y)
else
    y ← y + 1
    y ← PROCEDURE(8)
end if
y ← PROCEDURE1(8)
```

(b) PROCEDURE
```
Require: y
  if y == 10 then
      y ← PROCEDURE1(y)
      return y
  else
      return 11
  end if
```

(c) PROCEDURE1
```
Require: z
  if z > 5 then
      return 35
  else
      return 53
  end if
```

A. YASM Predicate Program

The predicate program or Boolean program represents the first (and most important) transformation of the program to verify [23]. From now on, this will be called pProgram, for short. Each node of the pProgram is an object of the class PStmt having the following features:

- a string specifying the label of the statement;
- a connection with the syntactic successor statement (next), represented with a solid arrow;
- a connection with the control-flow successor statement (dest), represented with a dashed arrow.

We illustrate in Figure 2 the resulting pProgram for the running example.

IV. HYASM A HIERARCHICAL MODEL CHECKER

The main goal of this work is to provide a hierarchical model to perform model checking operations with YASM. In order to do so, we adapted some transformations of the YASM modules. Here, we will focus on the Hierarchical pProgram. The full package of HYASM, the hierarchical extension of YASM, can be found here[1].

The verification is made through the flattened structure shown in Section II. Moreover, we provide a comparison between the original YASM tool and HYASM, not only in terms of coherence of the results but also from the performances point of view. The execution flow is shown in Figure 3.

A. Hierarchical and flattened pProgram

A hierarchical pProgram is created by splitting a flat pProgram. Each pProgram is a function of the input program and is not related to any of the other programs, i.e. it represents a separated entity.

[1] https://github.com/AngeloFerrando/hierarchical_model_checking

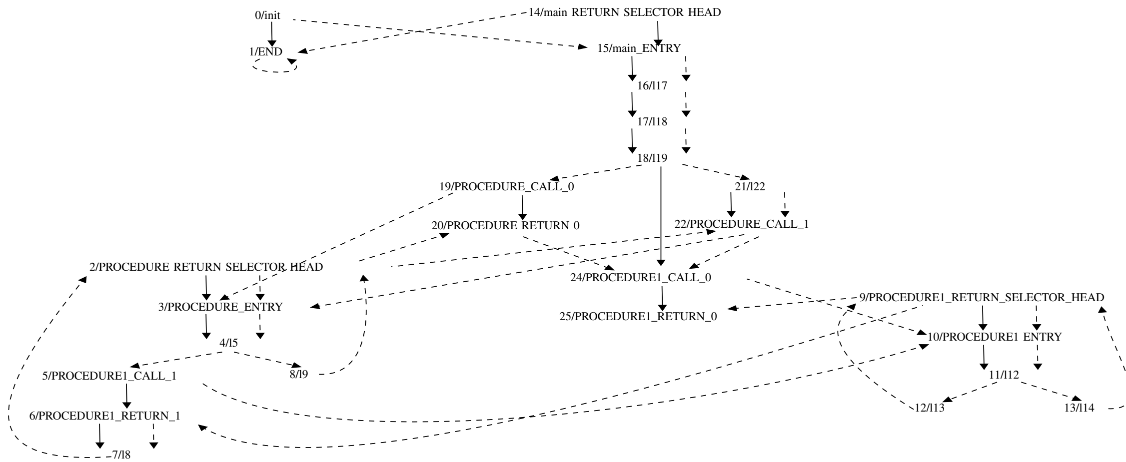

Fig. 2: pProgram of the running example.

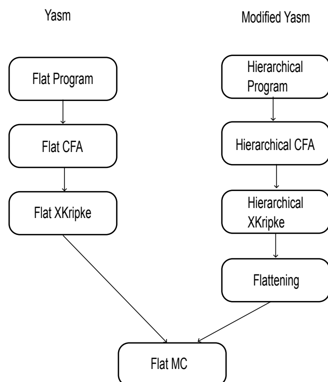

Fig. 3: YASM tools flow execution.

For each function invocation, a new statement is created as:

$$\text{Box_}\mathit{function\text{-}name_invocation\text{-}number} \tag{1}$$

where the parameters are the function name and a specific index identifying each function invocation, respectively. The invocation number is defined to uniquely identify each invocation procedure.

In order to create those pProgram, first we have to modify the compiling phase. Indeed, in the original YASM tool, invoking a procedure whose body is not in the file would have generated an error. Instead, in HYASM, a situation like the one just explained needs to be handled, since we have to create several pPrograms including fragments of the same C code. We also produce the new Box statement, as anticipated. Precisely, for each procedure invocation, we replace each CALL/RETURN statement with the Box one, having the same incoming edges of the CALL and the same outgoing ones of the RETURN.

In Figures 4 and 5, we show the hierarchical pPrograms associated to the functions *PROCEDURE* and *main* of the running example (the pProgram for *PROCEDURE1* is left out since not hierarchical and trivial).

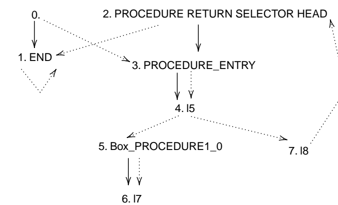

Fig. 4: Hierarchical pProgram of the function *PROCEDURE*.

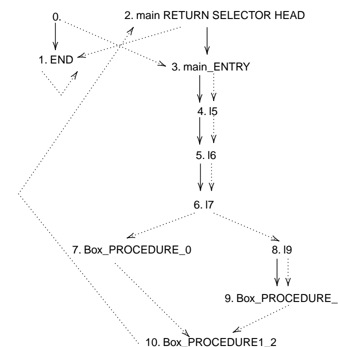

Fig. 5: Hierarchical pProgram of the *main* function.

Given the three hierarchical pPrograms and by applying the flattening rules shown in Section II, we can generate a flat pProgram that can be passed in input for the YASM tool.

V. Experiments

We tested our prototype on a machine with the following specifications: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 4 cores 8 threads, 16 GB RAM DDR4. We carried out different kinds of experiments. In Table I we report benchmarks we performed on our prototype. In more detail,

we focused on three aspects: (i) number of different function invocations in the C program, (ii) number of times a unique function is called in the C program, (iii) number of nested function invocations in the C program. Each set of experiments was aiming at showing the time performance of HYASM w.r.t. the classic YASM tool.

The first set of experiments (i) considers a scenario where the C programs analysed contains multiple function invocations. In (i), each function invocation corresponds to a unique function, that is, each function invocation in the program has a corresponding function definition (the same function is never called twice). The second set of experiments (ii) is a variation of (i) where the function invocations do not refer to different functions, but to the same one. That is, a unique function is defined, but called multiple times in the C program. The third set of experiments (iii) is another variation of (i) and (ii), where the functions are called in a nested way. That is, a single function is called in the main of the C program, and such function invokes another function, which in turn calls another function. This is repeated a certain number of times (to obtain the required number of function invocations).

In each experiment, the property analysed is the CTL formula $EF\ pc{=}END$. Intuitively, the formula checks whether there exists (E) an execution s.t. eventually (F) the program terminates ($pc = END$). Naturally, other properties can be verified but the previous one covers an interesting scenario concerning program termination. Moreover, the END atom corresponds to a YASM annotation, thus can be inserted in other places of interest of the generated programs.

All the experiments have been carried out on an increasing number of function invocations (from 10 to 100). To put this in terms of C lines of code, in the worst case scenario the files had around 1000 lines of code. The columns in Table I report the execution time required by HYASM and YASM in each scenario. As we can see, our prototype obtained really good results in comparison with standard YASM. Indeed, in most of the cases, our prototype even required less execution time to perform the verification of the C program. This can be easily observed in Figure 6, where we draw the results reported in Table I. Note that, in Table I, we focused on the time complexity of our approach (resp., standard YASM); nonetheless, we also considered the memory consumption. However, both our approach, and standard YASM, consumed at most 300 KB of memory. This is a promising result that may enable the use of YASM even in bigger programs.

We now comment the obtained results a little bit further. We start with the first set of experiments (i), which are shown in Figure 6. In such a scenario, we can observe how our approach performs better almost in all cases. Nonetheless, the improvement is not too marked. Indeed, the two approaches almost require the same amount of time (even though for smaller models our technique seems to perform better). Moving on with the second set of experiments (ii), which are also shown in Figure 6, we obtain far better results w.r.t. (i). This is reasonable, because even though we have multiple function invocations (as before), each one relates to the same function.

# fun. calls	Different calls		Same calls		Nested calls	
	HYASM	**YASM**	**HYASM**	**YASM**	**HYASM**	**YASM**
10	377.77	639.09	313.37	589.56	379.41	639.48
20	452.45	721.81	297.89	599.52	427.21	682.48
30	517.87	802.77	316.26	612.88	519.22	720.34
40	618.74	853.83	329.83	634.50	625.63	781.18
50	692.50	935.67	360.28	636.51	708.14	834.01
60	769.99	747.42	365.11	650.33	788.69	802.94
70	849.77	836.28	387.57	639.39	843.62	827.64
80	885.71	873.26	415.82	657.51	942.35	849.94
90	986.97	1009.85	443.59	673.90	1024.65	892.26
100	1062.84	1106.57	475.35	678.70	1106.95	832.73

TABLE I: Benchmarks of HYASM and YASM (time is in ms).

Finally, we have the third set of experiments (iii), also shown in Figure 6. This is the only case where our approach performs worse than YASM (for bigger models). However, this is only due to the fact that YASM does not work on such set of experiments. Indeed, YASM raises exceptions on such C programs. For this reason, the execution time is less than ours; since YASM does not actually complete the verification because of the failure. Anyway, we opted for reporting such a set of experiments because it allows us to point out further how our solution extends and improves YASM.

Note that, even though the experiments may seem restrictive, they actually cover a large set of programming patterns. One could claim that calling the same function multiple times may not be common, but it serves us as a stressed case scenario. Any program will have some *different* function calls, some calls to the *same* function, and some functions with *nested* calls. Thus, our experiments have to be seen more as general execution patterns, rather than actual programs. Of course, from the viewpoint of the model checker, they are programs to be verified indeed; but, the information we extract from experimenting on them is not limited to such programs.

VI. Conclusion and Future works

The main objective of this work was to extend the model checker YASM to verify hierarchical systems through an ad-hoc procedure. First, the hierarchical model of the system was produced then a flattened structure was provided to YASM for the model checking step. As highlighted along the paper, our empirically results demonstrate that the recursive approach for hierarchical models performs worse than the one we propose. This helped us to show the highly applicability of the flattening approach for performing model checking on hierarchical models.

As far as we are aware of, HYASM is the first model checking tool that implements a hierarchical procedure (via a flat expansion) and shows that it is advantageous in practice for branching-time temporal logics. This opens for several interesting questions for future work, mainly in two directions. On the one hand, one may further explore ad-hoc solutions for hierarchical model checking to speed up the evaluation process in YASM. On the other hand, one can extend all conceived methods to other existing model checkers.

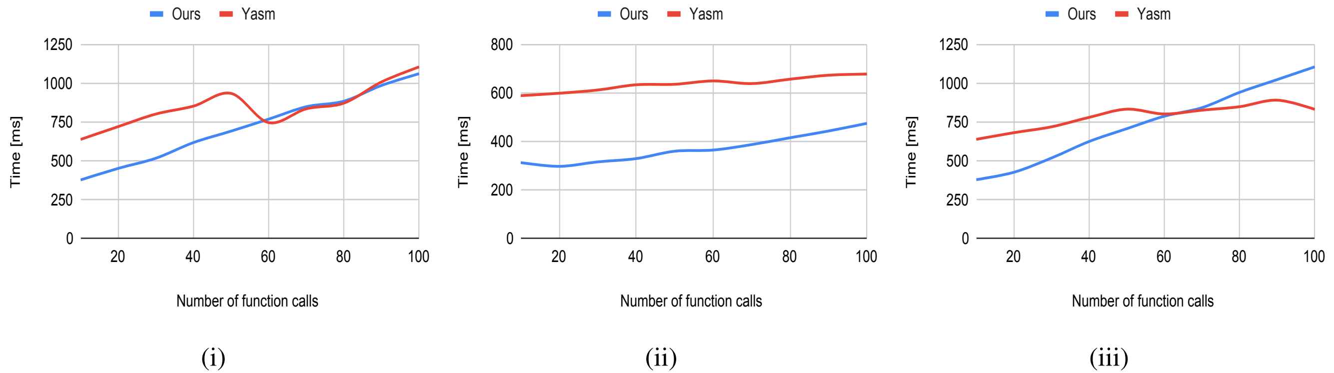

Fig. 6: Different (i), same (ii), and nested (iii) function invocations.

ACKNOWLEDGMENT

This research has been supported by PRIN project RIPER (20203FFYLK), PNRR MUR project PE0000013-FAIR, and InDAM project "Strategic Reasoning in Mechanism Design"

REFERENCES

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[2] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *International Conference on Computer Aided Verification*, pages 280–295. Springer, 2000.

[3] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[4] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *International Colloquium on Automata, Languages, and Programming*, pages 169–178. Springer, 1999.

[5] R. Alur, M. McDougall, and Z. Yang. Exploiting behavioral hierarchy for efficient model checking. In *International Conference on Computer Aided Verification*, pages 338–342. Springer, 2002.

[6] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):273–303, 2001.

[7] B. Aminof, O. Kupferman, and A. Murano. Improved model checking of hierarchical systems. *Information and computation*, 210:68–86, 2012.

[8] B. Aminof, A. Legay, A. Murano, O. Serre, and M. Y. Vardi. Pushdown module checking with imperfect information. *Inf. Comput.*, 223:1–17, 2013.

[9] B. Aminof, F. Mogavero, and A. Murano. Synthesis of hierarchical systems. *Sci. Comput. Program.*, 83:56–79, 2014.

[10] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Int. Conf. on Concurrency Theory*, pages 135–150. Springer, 1997.

[11] L. Bozzelli, A. Murano, G. Perelli, and L. Sorrentino. Hierarchical cost-parity games. *Theor. Comput. Sci.*, 847:147–174, 2020.

[12] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. *Formal Methods Syst. Des.*, 36(1):65–95, 2010.

[13] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[14] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Workshop on Software Model Checking, SoftMC 2003*, volume 89 of *ENTCS*, pages 417–432. Elsevier, 2003.

[15] T. Chen, F. Song, and Z. Wu. Global model checking on pushdown multi-agent systems. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[17] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131, pages 52–71, 1981.

[18] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[19] R. Faran and O. Kupferman. A parametrized analysis of algorithms on hierarchical graphs. *International Journal of Foundations of Computer Science*, 30(06n07):979–1003, 2019.

[20] P. Garoche, T. Kahsai, and X. Thirioux. Hierarchical state machines as modular horn clauses. *arXiv preprint arXiv:1607.04457*, 2016.

[21] S. Göller and M. Lohrey. Fixpoint logics on hierarchical structures. In *Int. Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 483–494. Springer, 2005.

[22] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In *International Conference on Computer Aided Verification*, pages 170–174. Springer, 2006.

[23] K. Ku. *Software model-checking: Benchmarking and techniques for buffer overflow analysis*. Citeseer, 2008.

[24] O. Kupferman, N. Piterman, and M. Y. Vardi. Pushdown specifications. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 262–277. Springer, 2002.

[25] O. Kupferman and T. Tamir. Hierarchical network formation games. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–246. Springer, 2017.

[26] O. Kupferman, M.Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.

[27] S. La Torre, M. Napoli, M. Parente, and G. Parlato. Hierarchical and recursive state machines with context-dependent properties. In *International Colloquium on Automata, Languages, and Programming*, pages 776–789. Springer, 2003.

[28] S. La Torre, M. Napoli, M. Parente, and G. Parlato. Verification of succinct hierarchical state machines. 2007.

[29] S. La Torre, M. Napoli, M. Parente, and G. Parlato. Verification of scope-dependent hierarchical state machines. *Information and Computation*, 206(9-10):1161–1177, 2008.

[30] V. Malvone. Implementazione di un algoritmo di verifica formale per programmi gerarchici nel tool yasm. Bachelor's thesis, University of Naples Federico II, Italy, 2010.

[31] A. Murano, M. Napoli, and M. Parente. Program complexity in hierarchical module checking. In *Int. Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 318–332. Springer, 2008.

[32] A. Murano and G. Perelli. Pushdown multi-agent system verification. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[33] A. Petrenko, O. N. Timo, and S. Ramesh. Model-based testing of automotive software: Some challenges and solutions. In *Proc. of 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[34] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. volume 137, pages 337–351, 1982.

[35] A. Wasowski. Flattening statecharts without explosions. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 257–266, 2004.