THETA: a Framework for Abstraction Refinement-Based Model Checking

Tamás Tóth[†*], Ákos Hajdu[†‡], András Vörös[†‡], Zoltán Micskei[†] and István Majzik[†]
[†]Budapest University of Technology and Economics,
Department of Measurement and Information Systems
[‡]MTA-BME Lendület Cyber-Physical Systems Research Group
Email: {totht, hajdua, vori, micskeiz, majzik}@mit.bme.hu

***Abstract*—In this paper, we present THETA, a configurable model checking framework. The goal of the framework is to support the design, execution and evaluation of abstraction refinement-based reachability analysis algorithms for models of different formalisms. It enables the definition of input formalisms, abstract domains, model interpreters, and strategies for abstraction and refinement. Currently it contains front-end support for transition systems, control flow automata and timed automata. The built-in abstract domains include predicates, explicit values, zones and their combinations, along with various refinement strategies implemented for each. The configurability of the framework allows the integration of several abstraction and refinement methods, this way supporting the evaluation of their advantages and shortcomings. We demonstrate the applicability of the framework by use cases for the safety checking of PLC, hardware, C programs and timed automata models.**

I. INTRODUCTION

Nowadays there are several model checking tools implementing algorithms for different formalisms. Most tools focus on a specific algorithm and formalism to solve a particular verification task efficiently. However, as new tasks emerge, more generic tools are also needed since the appropriate formalism and algorithm are usually not known initially.

THETA[1] is a generic, modular and configurable model checking framework, aiming to support the development and evaluation of abstraction refinement-based algorithms for the reachability analysis of different formalisms. The main distinguishing characteristic of THETA is its architecture that allows the combination of various abstract domains, interpreters, and strategies for abstraction and refinement, applied to models of various formalisms with higher level language front-ends.

THETA primarily aims to support researchers by providing a framework where new components and combinations can easily be implemented, evaluated and compared. Concrete tools were also built for the verification of transition systems, control flow automata and timed automata, combining different abstract domains (including predicates, explicit values and zones) and refinement strategies (including interpolation and unsat cores). Measurement results show strong dependency on the models and analysis components, motivating the need for a configurable framework. Furthermore, we also used THETA for education at our university, where students implemented model checkers using components from the framework.

Related tools. Abstraction refinement is a widely used approach for model checking software. Several tools, e.g. SLAM [1], BLAST [2] and SATABS [3] are based on predicate abstraction. Lazy abstraction tools like IMPACT [4] and WOLVERINE [5] use Craig interpolation to compute abstractions over the predicate domain without expensive post-image computation. Some tools apply abstraction refinement over domains other than predicates: the tool DAGGER [6] supports refinement for octagon and polyhedra domains, and the algorithm VINTA [7] applies abstraction refinement over intervals. Frameworks CPACHECKER [8] and UFO [9] support configurability by the definition of abstract domains, post operators and refinement strategies, but only targeting software models. The LTSMIN tool supports various formalisms through its Partitioned Next-State Interface (PINS) [10]. However, its main focus is on symbolic and parallel model checking algorithms. Our THETA framework aims to combine the concept of configurability with formalism independence: the core analysis algorithms can be implemented independently of the input formalisms, and relevant combinations of them can be selected to verify models of several input formalisms.

In this paper we focus on the architecture of THETA (Section II) and the use cases demonstrating the efficient use of the tools that are derived from the framework (Section III).

II. ARCHITECTURE AND IMPLEMENTATION

Figure 1 shows the architecture of THETA. The main parts of the framework are the formalism and language front-ends, the analysis back-end and the SMT solver interface.

A. Formalisms and language front-ends

One goal of the THETA framework is to enable the analysis of several formalisms. Formalisms are usually low level, mathematical representations based on first order logic expressions and graph like structures. Each formalism supports higher level languages that can be mapped to that particular formalism by a language front-end (consisting of a specific parser and possibly reductions for simplification of the model). Currently, transition systems, control flow automata and timed automata are the supported formalisms with front-ends for higher level languages as AIGER, PLC, C programs and UPPAAL XTA

*This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

[1]http://theta.inf.mit.bme.hu

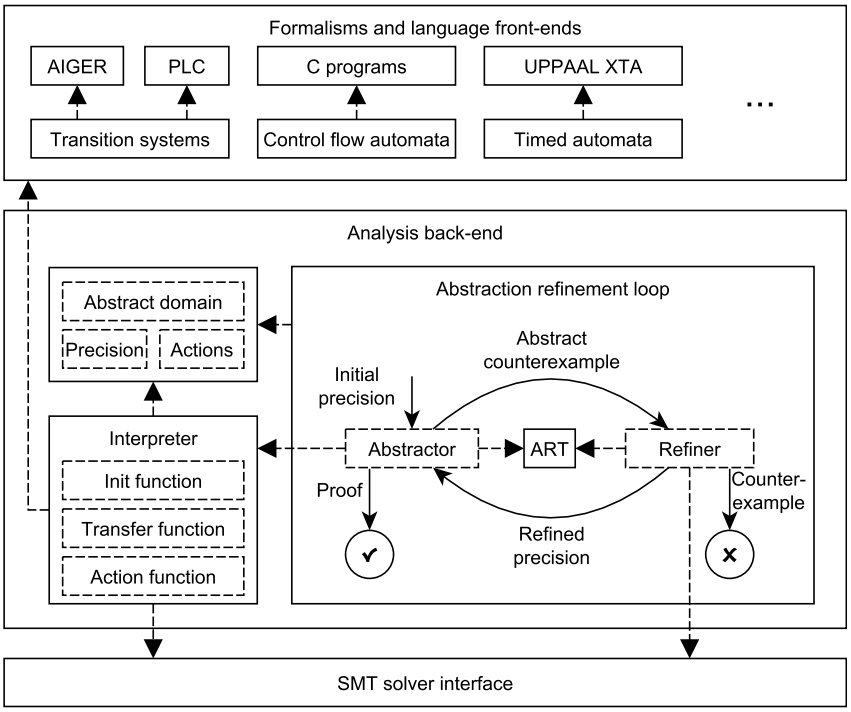

Fig. 1. Architecture of the THETA framework.

models. Section III describes instantiations of the framework for each of these formalisms.

B. Analysis back-end

The analysis back-end consists of three main parts: the abstract domain, the interpreter and the abstraction refinement loop for reachability analysis, with the interpreter being dependent on the formalisms. The basis of the analysis is an *abstract domain* with a set of *abstract states*, its bottom element and a partial order over the states. The accuracy of a given analysis is represented by an element of a set of precisions. Moreover, the formalism for which the analysis is performed defines a set of actions. Given a precision, an *interpreter* defines an abstract operational semantics over the abstract domain and set of actions. The abstract initial states are given by an *init function*. For an action, the abstract successors of a state are computed by a *transfer function*. An *action function* determines for an abstract state a set of actions that are enabled from that state.

The reachability analysis is performed by the *abstraction refinement loop*. As usual for lazy abstraction methods [4], its central data structure is an *abstract reachability tree* (ART), with nodes annotated with abstract states that represent over-approximations of reachable states along a given path, and edges annotated with actions. The ART is manipulated by the two main components of the loop. Using an interpreter, the *abstractor* constructs the ART w.r.t the current precision and an abstraction strategy, i.e. when to expand a node or cover it by an other node. If no target (i.e., unsafe) nodes are encountered, the constructed ART serves as an evidence for the safety of the input model. Otherwise, given a target node, the *refiner* is invoked to analyze the abstract path for feasibility. If the path is feasible, it is a counterexample to safety. Otherwise, the refiner carries out its refinement strategy to ensure that the analysis can continue without encountering the same spurious counterexample again (refinement progress). This can typically be achieved by pruning nodes and computing a new analysis precision (overapproximation-driven approach), or by uncovering nodes and strengthening labels (underapproximation-driven approach), both of which includes partial deconstruction of the ART.

Currently, built-in domains in THETA include predicates, explicit values, zones and their combinations. There are also interpreters provided for transition systems, control flow automata and timed automata. A default abstractor implementation is built-in that relies on the domain and the interpreter, also parameterizable with a search strategy. Interpolation and unsat core-based refinement strategies are provided for formalisms that are described with first order logic expressions.

C. SMT solver interface

The framework provides a general SMT solver interface that supports incremental solving, unsat cores, and the generation of binary and sequence interpolants. The solver interface can be used by the analysis components. Typically, the partial order over states and the transfer function are implemented in terms of queries to an SMT solver. A refiner component may use the interface to check feasibility of an abstract path and to generate interpolants or unsat cores for abstraction refinement. Currently, the interface is implemented by the SMT solver Z3 [11], but it can easily be extended with new solvers.

D. Extending and instantiating the framework

The framework can easily be extended with new formalisms and analyses. As an example, suppose that one wants to add support for the reachability checking of Petri nets [12]. First, the formalism has to be implemented, which is a collection of simple classes representing places, transitions and arcs of Petri nets. A possible language front-end could be the standard PNML format for Petri nets.

In order to perform reachability checking, the analysis back-end has to be extended as well. Petri nets can be described with first order logic formulas, for example by representing places (marked with tokens) with integer variables and transitions as FOL expressions adding/subtracting from places. Therefore, some abstract domains (such as predicates and explicit values) along with abstraction and refinement strategies (such as interpolation) work out of the box if the interpreter is implemented. An action of a Petri net can be implemented as the expression describing a transition and the action function as the collection of all transitions. The init and transfer functions also work out of the box for the abstract domains mentioned before.

Instantiating an executable tool from the framework (see examples in Section III) is also straightforward. A (command line or GUI) application has to be written that takes the parameters (path of the input model, domain, abstraction and refinement strategies, etc.), parses the input model using the language front-ends and instantiates and runs the analysis.

III. Use cases

A. Theta for transition systems

The tool Theta-STS is an instantiation of the Theta framework for reachability analysis of (symbolic) transition systems, based on an earlier, preliminary version [13]. As input language, the tool supports the AIGER format (also used in the Hardware Model Checking Competition [14]) and an intermediate language for describing PLC models [15]. The tool relies on the built-in predicate and explicit value domains and refinement strategies based on binary interpolation, sequence interpolation and formulas from unsat cores. Some additional utilities are also implemented, for example inferring the initial precision and simplifying the input system.

Figure 2 (from [16]) shows a heatmap of the execution time of 20 analysis configurations on 12 hardware (hw) and 6 PLC models. White squares correspond to a timeout. Configurations are abbreviated with the first letter of the domain (predicate, explicit), the refinement strategy (binary interpolation, sequence interpolation, unsat cores), the initial precision (empty, property-based) and the exploration strategy (DFS, BFS). The heatmap shows that no single configuration can verify all models and the execution time is very diverse, motivating the need for a configurable framework.

B. Theta for control flow automata

The tool Theta-CFA is an instantiation of the Theta framework for the reachability analysis of control flow automata. As input language, the tool supports a subset of C, enhanced by various size reduction techniques such as compiler optimizations and program slicing methods [17]. This tool uses the same built-in abstract domains and refinement strategies as the Theta-STS tool, only the interpreter differs.

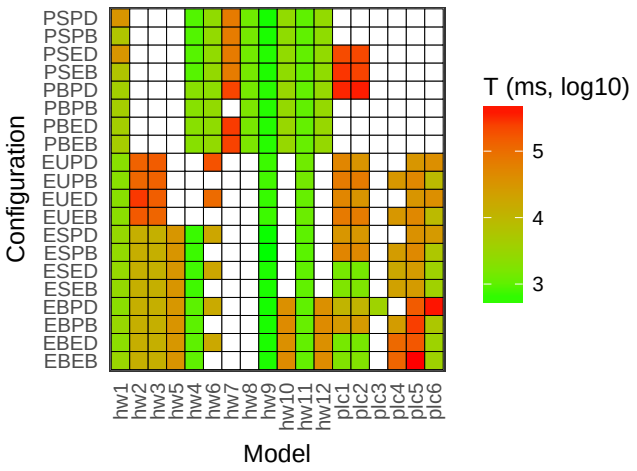

Fig. 2. Heatmap of execution time for transition systems (millisec., log. scale)

Figure 3 (from [17]) presents a heatmap of the verification time of 16 analysis configurations on 9 models from SV-COMP [18], selected from those categories that are currently supported by our C frontend. Configurations are abbreviated with the first letter of the slicing method (none, backward, value, thin), the compiler optimizations (true, false) and the exploration strategy (DFS, BFS). Similarly to transition systems, different configurations are more suitable for different input models.

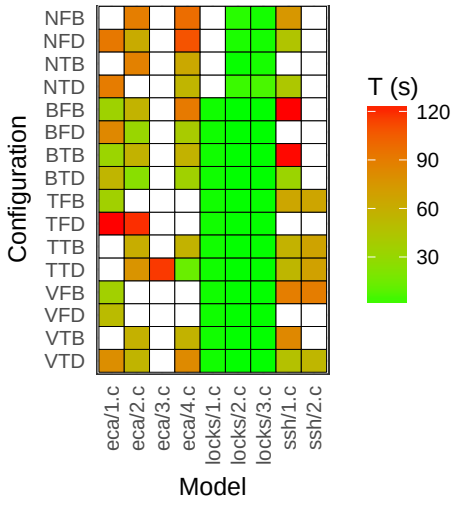

Fig. 3. Heatmap of execution time for C programs (in seconds)

C. Theta for timed automata

The tool Theta-XTA is an instantiation of the Theta framework for reachability checking of timed automata. As input language, the tool supports a subset of the Uppaal

4.x XTA format[2]. The tool implements two lazy abstraction algorithms based on zone abstraction: a variant of $\langle \mathbf{a}_{\preccurlyeq LU}, \mathsf{disabled} \rangle$ [19], a non-convex lazy abstraction algorithm based on LU-bounds, and an algorithm based on interpolation for zones [20] with two different refinement strategies (BIN and SEQ). Table I (from [20]) presents some measurement results for the tool. Column *time* is the total execution time in ms, and *passed* is the number of expanded nodes in the ART. Models come from the PAT benchmarks[3].

TABLE I
COMPARISON OF ALGORITHMS FOR TIMED AUTOMATA IN THETA-XTA

Model	$\mathbf{a}_{\preccurlyeq LU}$		BIN		SEQ	
	time	*passed*	*time*	*passed*	*time*	*passed*
Critical 3	1.8	4923	1.6	3213	1.6	3157
Critical 4	65.0	130779	78.2	83686	75.2	78252
CSMA 9	6.6	30476	7.3	30476	7.9	30476
CSMA 10	21.3	78605	21.0	78605	22.8	78605
CSMA 11	61.4	198670	58.9	198670	63.8	198670
CSMA 12	167.2	493583	168.7	493583	179.1	493583
FDDI 50	1.4	402	2.0	402	2.0	402
FDDI 70	2.9	562	3.5	562	3.7	562
FDDI 90	5.9	722	6.8	722	7.1	722
FDDI 120	12.9	962	15.0	962	15.4	962
Fischer 7	1.9	7737	2.8	7737	2.8	7737
Fischer 8	5.1	25080	7.7	25080	8.7	25080
Fischer 9	21.3	81035	29.0	81035	32.4	81035
Fischer 10	94.4	260998	133.2	260998	149.7	260998
Lynch 7	2.6	9977	3.6	9977	4.0	9977
Lynch 8	7.7	30200	12.2	30200	13.9	30200
Lynch 9	32.8	92555	45.2	92555	54.2	92555

As can be seen from the data, in general, the $\mathbf{a}_{\preccurlyeq LU}$-based algorithm performs better in terms of execution time, but the interpolation based algorithms might construct a significantly smaller ART, thus easy configurability of the tool pays off.

IV. CONCLUSIONS

In this paper we introduced THETA, a configurable model checking framework for abstraction refinement-based reachability analysis for different formalisms. We described the architecture that helps to implement, evaluate and combine various algorithms in a modular way for different formalisms. We also demonstrated the applicability of the framework by use cases for the verification of hardware, PLC, software and timed automata models. Results of the evaluation with configuring and combining different analysis modules support the need for a generic framework, such as THETA.

Future work. At the moment the framework focuses on flexibility rather than performance (hence it is not yet intended to be competitive with highly optimized implementations). We are currently extending both the supported formalisms and the algorithms. We are working on supporting a wider set of elements in the C programming language and on defining hierarchical statecharts in THETA along with an interpreter. We are also working on increasing the number of input models in our experiments in order to reach stronger conclusions. This would also allow us to address the problem of selecting the most suitable configuration for a given verification task. Moreover, we also plan to experiment with novel, state-of-the-art algorithms, e.g., abstractions over data variables for timed automata.

[2]See the web help on http://www.uppaal.org for a language reference

[3]http://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html

REFERENCES

[1] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer Aided Verification*, ser. LNCS, vol. 2102. Springer, 2001, pp. 260–264.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, 2007.

[3] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: Sat-based predicate abstraction for ansi-c," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.

[4] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.

[5] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with WOLVERINE," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 573–578.

[6] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Automatically refining abstract interpretations," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2008, pp. 443–458.

[7] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Craig interpretation," in *Static Analysis*, ser. LNCS, vol. 7460. Springer, 2012, pp. 300–316.

[8] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190.

[9] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "UFO: A framework for abstraction- and interpolation-based software verification," in *Computer Aided Verification*, ser. LNCS, vol. 7358. Springer, 2012, pp. 672–678.

[10] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "LTSMIN:: High-performance language-independent model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 9035. Springer, 2015, pp. 692–707.

[11] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[12] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[13] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A configurable CEGAR framework with interpolation-based refinements," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS, vol. 9688. Springer, 2016, pp. 158–174.

[14] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, K. Heljanko, and J. Baumgartner, "Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 135–172, 2016.

[15] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[16] A. Hajdu and Z. Micskei, "Exploratory analysis of the performance of a configurable CEGAR framework," in *Proceedings of the 24th PhD Mini-Symposium*. BUTE DMIS, 2017, pp. 34–37.

[17] G. Sallai, A. Hajdu, T. Tóth, and Z. Micskei, "Towards evaluating size reduction techniques for software model checking," in *Verification and Program Transformation*, ser. EPTCS. Open Publishing Association, 2017, (Accepted).

[18] D. Beyer, "Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016)," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2016, vol. 9636, pp. 887–904.

[19] F. Herbreteau, B. Srivathsan, and I. Walukiewicz, "Lazy abstractions for timed automata," in *Computer Aided Verification*, ser. LNCS, vol. 8044. Springer, 2013, pp. 990–1005.

[20] T. Tóth and I. Majzik, "Lazy reachability checking for timed automata using interpolants," in *Formal Modeling and Analysis of Timed Systems*, ser. LNCS, vol. 10419. Springer, 2017, (Accepted).