Reasoning with Happens-Before Relations about Concurrent Programs in the THETA Framework

Csanád Telbisz, Levente Bajczi, Dániel Szekeres, András Vörös, István Majzik
Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering
Budapest, Hungary
Email: csanadtelbisz@edu.bme.hu, {bajczi,szekeres,vori,majzik}@mit.bme.hu

***Abstract*—The model checking of multi-threaded programs often involves reasoning about the happens-before relation of concurrent program instructions. Several algorithms exist for finding a partial order of instructions that is consistent with ordering constraints of the assumed memory model and that violates a safety property; or for proving that such partial orders do not exist. We present existing and novel bounded model checking approaches reasoning with happens-before relations of concurrent programs. These algorithms are implemented in THETA, a modular model checking framework. We also give a comparative evaluation of our THETA implementations and state-of-the-art verifiers.**

***Index Terms*—concurrency, happens-before relation, THETA**

I. Verification with Partial Orders

Due to the great number of possible thread interleavings in concurrent programs, formal verification is cardinal for discovering all possible behaviors of the program. In this paper, we focus on the bounded model checking (BMC) of multi-threaded programs with shared variables. While the techniques presented here can be generalized to weak memory models [1], the scope of this paper is limited to sequential consistency (SC). These BMC techniques assume a loop-free (or unrolled) program. The aim of verification is to prove that an unsafe state (i.e., a violation of an assertion) is not possible with any interleaving of concurrent instructions; or to reveal an execution where a violation occurs. The presented approaches represent program executions as partial orders of program instructions using a *happens-before* relation ($\prec$).

The program and the verification requirement (e.g., an assertion) are symbolically encoded to a Satisfiability Modulo Theories (SMT) formula along with some constraints regarding the happens-before relation [2], [3] (hence it is *bounded*: the whole program has to be encoded in a finite formula). Then, an SMT solver is used to decide the satisfiability of this verification formula. A satisfying assignment gives a violation of the requirement; an unsatisfiable result means that a violation is not possible, and the program is safe. In this paper, we introduce and compare the BMC techniques following the above principles implemented in the THETA model checker [4], [5] along with a novel enhancement of these techniques.

This research was partially funded by the 2024-2.1.1-EKÖP-2024-00003 University Research Scholarship Programme under project numbers EKÖP-24-2-BME-118 and EKÖP-24-3-BME-{159,213}, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; with the support provided by the Ministry of Culture and Innovation of Hungary from the NRDI Fund.

Going into more detail, an event graph is built based on the program structure where events are variable accesses, and the set of edges is the happens-before relation. Intuitively, $e_1 \prec e_2$ means that event e_1 must precede e_2 in all program executions represented by this happens-before relation $\prec$. An event is characterized by the accessed variable, a unique index per variable (for easy reference), a guard condition (the conjunction of branching conditions that lead to the respective instruction) and an access type (*read* or *write*). Assignments are encoded to constraints so that the guard of the write event implies the assignment: e.g., $y_1 \neq 1 \Rightarrow x_2 = 1 - y_2$ for the program in Figure 1. The negation of the assertion is also encoded as we aim to find a violation or to prove that none exists. For Figure 1, it is $\neg(x_5 = 1 \vee y_5 = 1)$.

The happens-before relation consists of several relations and must satisfy certain axioms. The actual set of base relations and axioms depends on the memory model. The most straightforward relation is the *program order* (po) which relates events based on the natural order of variable accesses in the program code (cf. the transitive closure of gray edges in Figure 2). The *read-from* (rf) relation relates a write event w to a read event r if r reads the value written by w. The simplest axiom is transitivity: $e_1 \prec e_2 \wedge e_2 \prec e_3 \Rightarrow e_1 \prec e_3$. Due to space limitations, we refrain from introducing other necessary relations and axioms, as they are well documented in the literature [1], [3]. The *rf*-relation is represented symbolically by creating Boolean variables for each pair of events that may be *rf*-related: $rf^x_{i,j}$ being true means that $w_{x_i} \prec_{rf} r_{x_j}$ where w_{x_i} and r_{x_j} represent the respective write and read accesses of x. Several constraints can be devised from the intended semantics of the *rf*-relation such as $rf^x_{i,j} \Rightarrow x_i = x_j \wedge guard_{x_i} \wedge guard_{x_j}$ which means that if events are *rf*-related, then their values must be equal and their guards must be true. For a complete list of *rf*-related constraints, we refer the reader to previous works [3]. These constraints are also appended to the verification formula. Some base relations and axioms are encoded in the verification formula (such as *rf* constraints), while others are guaranteed by the algorithm.

A decision procedure aims to find an assignment to the written/read values of all events and to the values of introduced

initially: $x_0 = y_0 = 0$

Thread t_1	Thread t_2
`if(`y_1`==1) `x_1`=1;`	y_3`=`x_3`;`
`else `x_2`=1-`y_2`;`	x_4`=1-`y_4`;`

finally: `assert(`x_5`==1 || `y_5`==1)`

Fig. 1: Two-threaded program with shared variables x and y

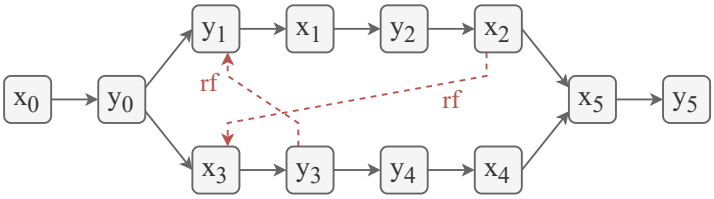

Fig. 2: Event graph with some happens-before edges

relation variables (such as *rf* variables) that is consistent with the axioms of the memory model. For this, the decision procedures use SMT solvers in different ways. In many memory models (such as sequential consistency), consistency means that the happens-before relation is acyclic (intuitively, circular precedence should be avoided). For example, the happens-before relation (with an arbitrary choice of *rf* edges) corresponding to Figure 2 is inconsistent due to the cycle formed by the two *rf* and *po* edges (or equivalently, due to the self-loop formed at any event of the cycle by transitivity).

II. Decision Procedures

This section introduces the ideas of three decision procedures for solving the satisfiability of the verification formula. While these approaches all rely on SMT solvers, the way the solvers are used is fundamentally different in the approaches.

A. Integer Difference Logic (`IDL`)

The first method encodes every constraint into the SMT formula by representing the happens-before relation as temporal precedence [2]. A Boolean variable hb_{e_1,e_2} is created for each pair of events representing the happens-before relation. A clock variable clk_e is associated with each event, meaning when that event happens. For each pair of events, the constraint $hb_{e_1,e_2} \Rightarrow clk_{e_1} < clk_{e_2}$ is added to ensure the intended semantics of the happens-before relation and clock variables.

The acyclicity of the happens-before relation is guaranteed by the acyclicity of the less-than relation on clock variables. Other constraints such as $rf^x_{i,j} \Rightarrow hb_{w_{x_i},r_{x_j}}$ (the *rf*-relation is part of the happens-before) or $hb_{e_1,e_2} \wedge hb_{e_2,e_3} \Rightarrow hb_{e_1,e_3}$ (transitivity) are also added that ensure the consistent use of relation variables and axioms. Since everything is encoded into the formula, we can simply pass it to an SMT solver to decide its satisfiability. A satisfying assignment gives an event scheduling leading to an assertion violation.

B. Refinement Step-by-Step (`RFN`)

This approach omits all happens-before constraints (that contain a hb variable). Thus, the formula may become satisfiable even if a violation is not possible. The formula is iteratively refined by querying an SMT solver: the returned assignment is analyzed if it satisfies the axioms of the happens-before relation or if it is inconsistent due to omitted constraints. In the latter case, a refinement clause is generated and added to the solver excluding this inconsistency. The same is repeated until the solver reports unsatisfiable or a real violation is found [6].

Deciding whether an assignment is a valid model for a happens-before relation is the consistency checking algorithm [3], [7]. Basically, the event graph is updated based on the values of the relation variables in the model given by the solver. Then, further edges are added (e.g., transitive closure) to make sure that all necessary axioms are satisfied. Technically, axioms can often be formulated as derivation rules with a precondition (*if certain events are related...*) and a consequence (*then some other events must also be related*). An example of a derivation rule for transitivity is: if $e_1 \prec e_2$ and $e_2 \prec e_3$, then $e_1 \prec e_3$. This is convenient for cycle detection as we only need to start from base relations (such as *po* or *rf*) and add new edges using the derivation rules until a fixpoint is reached where all axioms are satisfied. Finally, it is checked whether the event graph contains a cycle (or equivalently, a self-loop due to the transitive closure).

If a cycle is found, a conflict clause is generated. A conflict clause is the conjunction of the reasons for the edges of a cycle, trivial edges omitted (e.g., *po* edges). The reason for an edge is either the Boolean variable in case of a base relation (e.g., an *rf* variable) or the precondition of the derivation rule that added the edge to the event graph. The refinement clause added to the solver is the negation of the conflict clause. For example, the consistency check finds the cycle in Figure 2 when the solver returns a model where the *rf* variables corresponding to the red edges are set to *true*. Then, the following refinement clause would be added to the solver to avoid this inconsistency: $\neg(rf^y_{3,1} \wedge rf^x_{2,3})$.

C. SMT Theory with User Propagator (`PROP`)

The third method incorporates consistency checking into the SMT solver as a theory solver [3]. SMT solvers gradually build a partial variable assignment to find a satisfying complete variable assignment. Each time a new variable gets a value in the partial assignment, all relevant theory solvers (e.g., a linear arithmetic theory solver) are queried to confirm if this new partial assignment is still consistent with their theory. If a theory solver finds an inconsistency, it propagates a conflict clause, and the SMT solver backtracks and tries to extend the partial assignment with different values.

The ordering consistency theory solver (that realizes the main logic of this decision procedure) subscribes to change notifications for relation variables (e.g., *rf* variables) and guard conditions: the SMT solver calls the theory solver whenever a registered expression gets a value (*true* or *false*) in the partial assignment during the search space exploration [8]. Upon such

	Solved	Time(s)
IDL	538	34153
RFN	544	9989
PROP	549	10183

TABLE I: Decision procedures

	IDL	RFN
Z3	538	544
cvc5	537	529
MathSAT	307	528

TABLE II: Solved tasks by solver

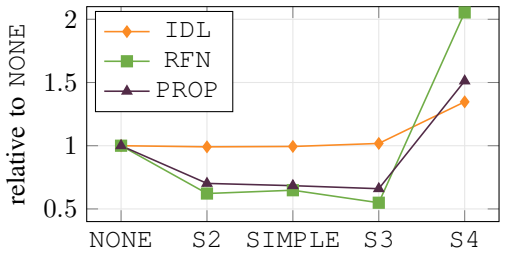

Fig. 3: Times with conflict avoidance

notifications, a consistency check is performed using the same method as in Section II-B to update the event graph and check for cycles [7]. When an inconsistency (that is, a cycle) is found, a clause is generated similarly to Section II-B, and it is propagated as a conflict clause.

From a theoretical point of view, the main difference between this decision procedure and the previous approach is that RFN performs consistency checking on a complete variable assignment, while this approach works on a partial model. The advantage of the propagator approach is that inconsistencies are found earlier during the SMT solving as there is no need to find a satisfying assignment of all variables to detect a conflict. On the other hand, consistency checking is performed more frequently by the propagator (each time a relevant expression is assigned). Section IV reveals the performance trade-off between the two methods.

III. Automatic Conflict Avoidance

The presented methods put most of the reasoning into the SMT solver or the consistency checking algorithm: initial constraints are simple, and excluding inconsistencies is left to the decision procedures. We apply a novel optimization before starting the decision procedure.

Potential conflicts can be found by analyzing the program structure. A potential conflict is a cycle in the event graph that may arise at some point in the decision procedure. Our algorithm looks for cycles in the potential happens-before relation, the so-called *may-set* of the happens-before relation, an over-approximation of all possible actual happens-before relations [9]. Conflict clauses are generated from potential cycles similarly to how it is done in the RFN and PROP decision procedures presented in Section II. These clauses (their negations) are added to the verification formula. For example, the same clause as at the end of Section II-B can be found and added to the solver by our algorithm before starting the decision procedures. Since these clauses represent inconsistencies, the satisfiability of the formula does not change by adding these extra clauses. Thus, the final verification verdict is not affected either, so our optimization is sound. However, it can greatly reduce the solver search space.

The way of searching for such cycles is based on the relation variables collected for encoding the verification formula, and on axioms of the assumed memory model. The same idea is used as in the consistency checking algorithm of the decision procedures, the only difference being that an actual happens-before relation is used there, while this optimization only has the potential happens-before relation. Since the potential happens-before relation contains all possible related pairs, there is an impractically large number of cycles. Therefore, we bound the size of these cycles. To have a measure for the size of cycles, the size is defined to be the number of non-trivial base relations (not *po*) plus the number of non-trivial derivation rule applications (not transitivity) that create the cycle. For example, the size of the cycle in Figure 2 is 2.

IV. Evaluation and Conclusions

We evaluate the presented techniques implemented in Theta by comparing the methods to each other and to state-of-the-art verifiers. Theta is a configurable formal verification framework that supports several input formalisms and several model checking backends [4], [5]. Our implementation of the IDL and RFN methods can use any SMT solver via SMT-LIB: specifically, we tested Z3 (4.13.0) [10], cvc5 (1.0.8) [11], and MathSAT5 (5.6.10) [12]. Our PROP algorithm can only use Z3 via JavaSMT [13] being the only solver that allows the easy integration of a custom SMT theory via its user propagator interface [8]. We evaluated on the 725 SV-COMP concurrent reachability benchmark programs [14]. The experiments were executed in the cloud platform of our university with a 15-minute timeout and a 15 GB memory limit per task.

First, we compare the performance of the three decision procedures implemented in Theta (using Z3 in all cases). The number of solved tasks and the total CPU times for the 525 commonly solved tasks are listed in Table I. The methods performing consistency checking on event graphs are more efficient than a direct encoding to IDL based on the results. Even though the PROP algorithm scales slightly better (inasmuch as solving more tasks before timeout without the conflict avoidance optimization), the average solving time is lower for RFN.

The winner in terms of SMT solver is Z3 (see Table II). With RFN, cvc5 solves 3 extra tasks and MathSAT solves 1 extra task that Z3 cannot solve; with IDL, the two solvers solve 5 and 6 additional problems compared to Z3, respectively. Regarding CPU time, Z3 dominates by far.

For evaluating the automatic conflict avoidance algorithm, we test the following options with each decision procedure: NONE (disabled), S2, SIMPLE, S3, S4 where Sn means

	THETA (complete)			DARTAGNAN	
	IDL	RFN	PROP	Eager	Lazy
Solved/	398	409	410	456	457
filtered	398	409	410	434	433
Time (s)	26000	4150	5770	11200	6370

	THETA (bounded)			DEAGLE
	IDL	RFN	PROP	
Solved/	538	554	553	623
filtered	538	554	553	577
Time (s)	35000	6020	8690	2020

TABLE III: Comparison with state-of-the-art verifiers (using conflict avoidance for THETA)

that cycles of size $\leq n$ are encoded, and SIMPLE allows 2 (non-trivial) base relations plus 1 (non-trivial) derivation rule application for deriving a cycle (so SIMPLE is between S2 and S3). Figure 3 shows the total CPU times for commonly solved tasks relative to the NONE configuration by decision procedure. Clearly, adding conflict clauses of small cycles to the verification formula improves verification performance. As the allowed size of cycles increases, the gain of trimming the search space is negatively compensated by the overhead of finding and handling the large number of cycles (see S4). The optimization does not improve the IDL procedure, but it considerably accelerates the other two methods. The most solved tasks are achieved by the S2 and SIMPLE options (554 for for RFN and 553 for PROP).

We also compare THETA to state-of-the-art verifiers: DARTAGNAN (winner of SV-COMP'24 concurrency category [14]) that implements an eager encoding (similar to our IDL) and a lazy decision procedure [1], [9]. Since DARTAGNAN only accepts safe results when complete loop unrolling is possible, we apply the same strategy for this comparison (complete). We also compare to DEAGLE (SV-COMP'25 concurrency winner) that follows the propagator approach [15]. DEAGLE reports safe if no violation is found even if only a bounded loop unrolling is applied: for a fair comparison, we apply the same strategy (bounded). Table III lists the total number of solved tasks, the number of solved tasks filtered on the 605 programs THETA can parse, and the total CPU times for commonly solved tasks. While the other tools solve more tasks, the RFN and PROP procedures of THETA are faster than DARTAGNAN. Though THETA is also disadvantaged due to front-end limitations, the comparison shows the potential of our algorithms.

Threat to validity: The experiments were executed on a cloud computing platform, so the loads on other virtual machines of the same host might have influenced the results. However, repeated executions of the experiments consistently yielded the reported results, so the fluctuation is not significant compared to the differences between different configurations.

Conclusion: There is a clear difference between different decision procedures deciding the satisfiability of the verification formula: choosing the more sophisticated RFN or PROP methods yields a significant performance increase. The proposed automatic conflict avoidance algorithm also considerably accelerates verification. The differences are more accentuated in the CPU time results, which is probably due to having a non-linearly increasing complexity among the used benchmark problems. In future work, we plan to extend the presented methods in THETA to allow weak memory models.

Data Availability: THETA is an open-source formal verification framework [4], [5]. Version 6.11.10 used for the evaluation of this paper is available at [16].

REFERENCES

[1] T. Haas, R. Meyer, and H. P. de León, "CAAT: consistency as a theory," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 114–144, 2022. [Online]. Available: https://doi.org/10.1145/3563292

[2] J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *CAV*, ser. LNCS, vol. 8044. Springer, 2013, pp. 141–157. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_9

[3] F. He, Z. Sun, and H. Fan, "Satisfiability modulo ordering consistency theory for multi-threaded program verification," in *PLDI*. ACM, 2021, pp. 1264–1279. [Online]. Available: https://doi.org/10.1145/3453483.3454108

[4] T. Tóth, A. Hajdu, A. Vörös, Z. Micskei, and I. Majzik, "THETA: a Framework for Abstraction Refinement-Based Model Checking," D. Stewart and G. Weissenbacher, Eds., 2017, pp. 176–179. [Online]. Available: https://doi.org/10.23919/FMCAD.2017.8102257

[5] C. Telbisz, L. Bajczi, D. Szekeres, and A. Vörös, "Theta: Various Approaches for Concurrent Program Verification (Competition Contribution)," in *TACAS*, ser. LNCS. Springer, 2025. [Online]. Available: https://ftsrg.mit.bme.hu/paper-tacas25-svcomp/theta.pdf

[6] L. Yin, W. Dong, W. Liu, and J. Wang, "On scheduling constraint abstraction for multi-threaded program verification," *IEEE TSE*, vol. 46, no. 5, pp. 549–565, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2864122

[7] L. Bajczi, C. Telbisz, D. Szekeres, and A. Vörös, "On Stability in a Happens-Before Propagator for Concurrent Programs (Reproducibility Study)," in *TACAS*, ser. LNCS. Springer, 2025. [Online]. Available: https://ftsrg.mit.bme.hu/paper-tacas25-ocfix/paper.pdf

[8] N. S. Bjørner, C. Eisenhofer, and L. Kovács, "Satisfiability modulo custom theories in Z3," in *VMCAI*, ser. LNCS. Springer, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-24950-1_5

[9] N. Gavrilenko, H. P. de León, F. Furbach, K. Heljanko, and R. Meyer, "BMC for weak memory models: Relation analysis for compact SMT encodings," in *CAV*, ser. LNCS, vol. 11561. Springer, 2019, pp. 355–365. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_19

[10] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[11] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *TACAS*, ser. LNCS. Springer, 2022, pp. 415–442.

[12] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS*, ser. LNCS. Springer, 2013. [Online]. Available: http://doi.org/10.1007/978-3-642-36742-7_7

[13] D. Baier, D. Beyer, and K. Friedberger, "JavaSMT 3: Interacting with SMT Solvers in Java," in *CAV*, ser. LNCS. Springer, 2021, pp. 195–208.

[14] D. Beyer, "State of the art in software verification and witness validation: SV-COMP 2024," in *TACAS*, ser. LNCS. Springer, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_15

[15] F. He, Z. Sun, and H. Fan, "Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution)," in *TACAS*, ser. LNCS, vol. 13244. Springer, 2022, pp. 424–428. [Online]. Available: https://doi.org/10.1007/978-3-030-99527-0_25

[16] C. Telbisz, L. Bajczi, D. Szekeres, A. Vörös, and I. Majzik, "Artifact archive," Mar 2025. [Online]. Available: http://doi.org/10.5281/zenodo.15090897