May-Happen-in-Parallel Analysis of ESL Models using UPPAAL Model Checking

Che-Wei Chang
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA

Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA

Abstract— In this paper, we propose an approach for May-Happen-in-Parallel (MHP) analysis of electronic system level (ESL) design which models parallel discrete event simulation with concurrent automaton processes and formally identify those MHP states. Our MHP analysis utilizes formal verification by use of the UPPAAL model checker. The proposed approach converts the system model in SpecC SLDL into an UPPAAL model and generates a set of queries that automatically and completely finds all possible MHP pairs. The experimental results show our approach can report more precise MHP analysis results compared to other works at the cost of extended analysis run time.

I. Introduction

For concurrent and parallel languages, the May-Happen-in-Parallel (MHP) problem asks for two given statements whether or not there is a possibility where these two statements are executed at the same time. MHP analysis is useful as a basis for static model analysis and debugging, such as resource allocation and contention or race condition detection. In this paper, we propose an approach to abstract an UPPAAL model [1] from a system in SpecC system level description language (SLDL) and analyze the MHP statements by verifying the model with the UPPAAL verifier. In contrast to other techniques, such as [5], our approach can not only check the MHP property of two statements, but also of any number of statements and other properties.

A. MHP analysis using UPPAAL model checker

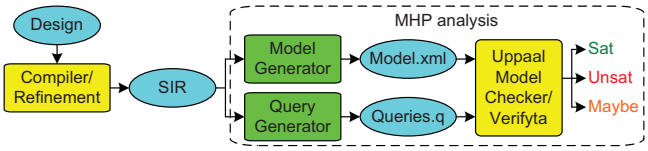


Fig. 1. MHP analysis flow with UPPAAL model checker

Fig. 1 illustrates the analysis flow and the tool chain we use in this paper to analyze MHP statements. Before the analysis, the system design is compiled into a System Internal Representation (SIR) data structure. The internal representation is then read by the Model Generator and Query Generator modules of the MHP analysis tool. The yellow blocks in the illustration are the existing tools to generate the SIR model and verify it, and the green blocks represent the tool we created to connect the design flow and the analysis. The Model Generator abstracts an automata model from the SIR structure, and the Query generator generates the queries for MHP analysis, asking if any two given scheduling points could possibly happen at the same time. According to the model and queries, the UPPAAL model checker will give one of the following answers: *satisfied*, *not satisfied*, or *maybe*. Each satisfied query represents a pair of MHP scheduling points, while an unsatisfied query means the two scheduling points will not happen at the same time. The *maybe* answer is given when the upper approximation option in the state space representation of UPPAAL verifier is enabled. In our experiments, we will use an example to demonstrate these three cases. This paper is organized as follows. Section I.B reviews the related work of MHP analysis as well as system modeling with automata. The Model and Query Generator are described in Section II and Section III. In Section IV the model optimization to shorten analysis time is introduced. Section V shows the experimental results for multimedia applications (two JPEG encoders, and MP3 decoder). Summary and future work is discussed in Section VI.

B. Related Work

MHP analysis and race condition detection in concurrent and parallel language has been broadly studied. [10] proposed an approach to find MHP statements in a untimed concurrent program with trace flow graph (TFG). [7], [8], [9] detect races and analyze non-deterministic anomalies in timed concurrent models, but in those methods simulation is required. In [5], an approach using static segment aware detection to identify MHP segment pairs is proposed. Like [5], our approach focuses on the timed model and does not require simulation. Compared to [5], our approach reports more precise results at the price of longer analysis time. Another advantage of our approach is that instead of just giving the MHP analysis results, our method can report the trace of transitions showing how statements can be executed in parallel. In addition, our approach can also identify the MHP sets of any number of statements and verify other properties like liveness (deadlock detection) and timing guarantees.

As for the system modeling, [3] and [4] propose approaches to model the behavior of SLDL designs with automata in PROMELA, and in [2] the design is modeled as a network of timed automata. The model is then analyzed with SPIN and UPPAAL model checker, respectively. Our approach also models the application with a network of automata and analyzes it with UPPAAL model checker. Compared with other works, our approach supports the modeling of richer design compositions and channel communication. More important,

978-3-9815370-4-8/DATE15/©2015 EDAA

instead of only supporting the behavior of traditional discrete event simulation (DES), the scheduler process in our model also supports *parallel* DES (PDES) [6], which is essential to our MHP analysis.

II. SLDL Design to UPPAAL Model

The proposed method in this paper includes two parts: model generator and query generator. To implement the model generator, we use the approach proposed in our technical report [11] to convert an ESL design in SpecC SLDL into an UPPAAL model for formal verification purpose. The proposed approach does not only support most of the semantics in the behavioral hierarchy, but also the communication between modules such as event synchronization and most used predefined channels in SpecC semantics. Most important of all, our UPPAAL model simulates the behaviors of traditional DES and PDES, and this feature is essential for our MHP analysis. Due to space limitation, here we only briefly introduce the basic concepts of an UPPAAL model and how our approach abstracts a system design in SpecC SLDL into an UPPAAL model. For more details of the abstraction, please refer to [11].

An UPPAAL model is composed of a network of concurrent processes which are created by instantiating pre-defined automaton templates in the system description of the model. In Fig. 2(A) we provide a simple model to illustrate essential components of an UPPAAL model. This model consists of two concurrent processes `Inst1` and `Inst2` which are created through instantiating templates `TA1` and `TA2` respectively. The definition of a template clearly specifies states in the automaton, transitions between states and the expression to be evaluated on the transition, which are named **location**, **transition** and **label** respectively. Parameter **int** `a` and channel **chan** `sync` are inserted to transfer data from `Inst1` to `Inst2` and synchronize transitions X2→X4/X3→X4 and Y1→Y2.

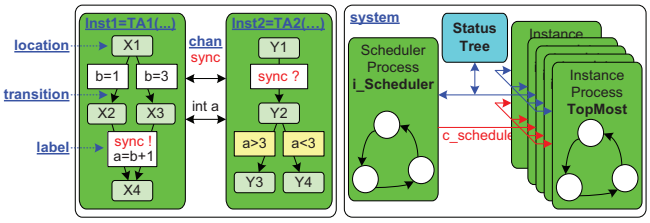


Fig. 2. SLDL Design to UPPAAL automata conversion

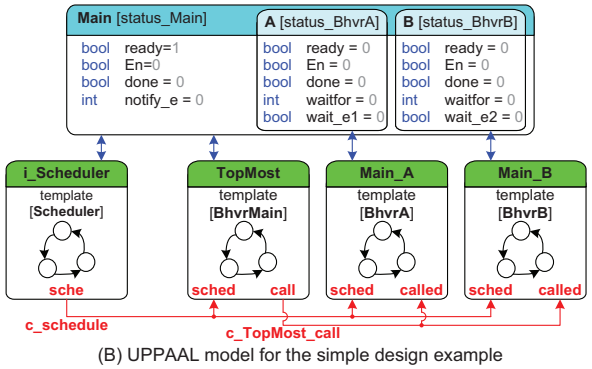
A system model is usually composed of multiple computation blocks (`behavior` in SpecC and `sc_module` in SystemC) with communication (port, channel, event synchronization) between those blocks. Fig. 2(B) shows our structure of the UPPAAL model for a system model, and an introductory example with the corresponding UPPAAL model is illustrated in Fig. 3. Our approach first defines an automaton template for each behavior in the system model. Statements and the execution flow specified in a behavior are abstracted into locations and transitions in the corresponding template. Except for the locations and transitions for statements, we also insert locations [idle], [ini], and [end] to represent the status where an instance is waiting for execution and the moment when the execution of an instance is active and finished. After the definition of behavior templates, each instance in the system model is one-to-one mapped into an instance process respectively by instantiating the corresponding behavior template in the system definition of the UPPAAL model. In the model, we also insert a scheduler process simulating the behavior of discrete event simulation to coordinate the transitions in instance processes. The structure [StatusTree] in the illustration is inserted to store the status information of all instance processes, and the scheduler process use the information to activate the idle instance processes according to the SpecC execution semantics.

```
1: int array[10] = {0, 1, 2, ..., 9};
2: int x = 0, y = 0, z = 0, w = 0 ;
3: behavior BhvrA (event e1)
4: {
5:   void main(){
6:     int i = 0;
7:     for (i=0; i<9; i++){
8:       y = x + 27 ;
9:       waitfor 1;
10:      w++ ;
11:      wait e1 ;
12:      x = array[i]*42; }}
13: } ;
```

```
14: behavior BhvrB (event e2)
15: {
16:   void main(){
17:     int i = 0;
18:     for (i=0; i<9; i++){
19:       y = y*42 + z ;
20:       waitfor 2;
21:       array[i] = array[i]*4+x++;
22:       notify e2 ;
23:       wait e2 ;
24:       z ++ ; }
25:   }
26: } ;
```

```
27: behavior Main ()
28: {
29:   event e ;
30:   BhvrA A(e) ;
31:   BhvrB B(e) ;
32:   int main() {
33:     par
34:     {
35:       A.main() ;
36:       B.main() ;
37:     }
38:   }
39: } ;
```

(A) SLDL source code for a simple design example



(B) UPPAAL model for the simple design example

Fig. 3. SLDL source code for an introductory design example

III. Queries for May-Happen-in-Parallel Analysis

In this section we introduce our idea of analyzing a MHP pair of statements by asking the model checker whether a corresponding query is satisfiable, as well as how the query generator creates a set of queries for MHP analysis.

A. Query in UPPAAL Model Checker

In the UPPAAL verifier, a query is described in the UPPAAL requirement specification language which supports five types of properties, namely *Possibly* (`E<>`), *Invariantly* (`A[]`), *Potentially always* (`E[]`), *Eventually* (`A<>`) and *Lead to* (`-->`). In our apporach, since the query is "whether or not two statements can possibly happen in parallel", we use the *Possibly* property `E<>p` which tests if there is a reachable state where property `p` is satisfied. We skip the detail descriptions of other properties, for they are not used in our approach.

The next step is to decide what property `p` should be. In the UPPAAL requirement specification language, it is possible to test whether or not a certain process is at a given location with the query of the form `process.location`. Since in the model generator we have mapped instances into processes, statements into locations, and execution of statements into transitions, the query "*for two given statements, Stmnt1 in instance Inst1 and*

Stmnt2 in instance Inst2, whether or not they can possibly happen in parallel" can be created in the following expression:

```
E <> Proc_Inst1.Loc_Stmnt1 and Proc_Inst2.Loc_Stmnt2
```

Here `and` is used to state "happen in parallel" in the query.

B. Queries for MHP Analysis

The most intuitive approach to generate a set of queries for MHP analysis for any two given statements is to generate a query for each possible combination of statement pair. This approach certainly will do the work, but the number of queries may be tedious even for a simple model. To reduce the number of queries, we apply three approaches to generate a compact set of queries for MHP analysis.

First, instead of generate query for all possible statement pairs, we only identify the suitable scheduling points that may happen in parallel. The scheduling points mark the moments when instances are activated or woken by the scheduler. According to the semantics, the statements between two scheduling points share the same simulation clock and delta cycle. Therefore, by checking the MHP pairs of these scheduling points, we actually check the MHP pairs of all statements in the design. The scheduling points in our model include location [Initial] of all instances, and location [end] of wait and waitfor statements.

Query	MHP
E<> Main_A.BHVR_INI and Main_B.BHVR_INI	sat
E<> Main_A.BHVR_INI and Main_B.WAITFOR_20_END	unsat
E<> Main_A.BHVR_INI and Main_B.WAIT_23_END	unsat
E<> Main_A.WAITFOR_9_END and Main_B.BHVR_INI	unsat
E<> Main_A.WAITFOR_9_END and Main_B.WAITFOR_20_END	unsat
E<> Main_A.WAITFOR_9_END and Main_B.WAIT_23_END	unsat
E<> Main_A.WAIT_11_END and Main_B.BHVR_INI	unsat
E<> Main_A.WAIT_11_END and Main_B.WAITFOR_20_END	unsat
E<> Main_A.WAIT_11_END and Main_B.WAIT_23_END	sat

Fig. 4. Queries for the Fig. 3 example for MHP analysis

The second approach is that we only generate queries for leaf instances intead of all instance. The reason is the computation and communication statements only exist in leaf instances. Since at this point the main purpose of our MHP analysis is to analyze the concurrent computation, we only need the queries for leaf instances. Note that in order to simplify the analysis, we take the end locations of the channel function calls as scheduling points instead of generating queries for the wait statements inlined for the communication method.

Last, we use static analysis to rule out statement pairs which are executed sequentially for sure and generate queries for pairs that may happen in parallel. This analysis includes two steps. The first step is to generate all possible MHP pairs of instances. In this step all combinations of any two leaf instances are generated, except combinations where two instances share the same parent with sequential or FSM composition in their hierarchy, since with these composition the execution of the child instances cannot overlap. The second step is to generate queries for all combinations of the scheduling points in each MHP pair of instances. Take the introductory design as the example. The red arrows in Fig. 3(A) mark the scheduling points in both leaf instances. Fig. 4 shows the queries generated by our tool as well as their satisfiability. According to the result, the statement set at lines 6∼8 and statement set at lines 17∼19 are MHP statements. Also statements at line 12 or 8 and statement at line 24 or 19 are MHP statements as well.

In the end, for any two leaf instance processes which are potentially activated by the scheduler at the same time, the query generator create a set of queries for all combinations of scheduling points in these two processes, and let UPPAAL model checker verify the satisfiability of these queries.

IV. Model Optimization

In this paper, we apply two main methods from different aspects to shorten the run time of MHP analysis. The first one is to generate a compact set of queries to shorten the analysis time, which has been described in the previous section. Another method is to trim the redundant search space in the model so that the solver can still give the identical result with less search time.

In order to analyze the MHP statements in the design, our model supports PDES and activates as many instances as possible in parallel. The downside of this method is that for most of the steps in the trace there are multiple enabled transitions and therefore the search space is much larger than regular DES. To address this, we give certain locations in the model higher priority than others and use the priority to remove the redundant sequences of transitions resulting in the same state. In UPPAAL model, different priority can be given to a location by specifying the type of the location. There are three types of locations supported in UPPAAL: *committed*, *urgent*, and regular. The committed location has the highest priority. If any automaton is in a committed location, the next transition must depart from one of the committed locations, i.e., it blocks the transitions with lower priority.

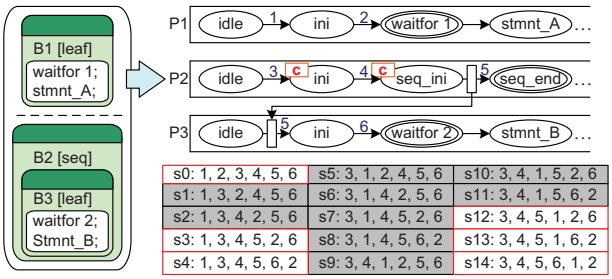


Fig. 5. Optimization with location prioritization

Let's use a UPPAAL model of a system with three instances in Fig. 5 to illustrate how the prioritized locations can trim the search space. In this example, instance B1 and B2 are executed concurrently and B3 is a child intance of B2. Processes P1, P2, and P3 are created for these three instances in the UPPAAL model. If now the query is "whether or not stmnt_A and stmnt_B can happen in parallel", the solver shall try all possible transitions listed in Fig. 5 to determine the satisfiability since this query is not satisfiable (in fact, stmnt_A and stmnt_B are executed at different simulation time). Fig. 5 lists all 15 possible sequences of transitions before P1 wakes up and moves into location for stmnt_A. Here, we give P2.ini and P2.seq_ini the *committed* priority to reduce the number of possible sequences. After the location prioritization, transition 4 and 5 must occur right after transition 3, and thus the number

of possible sequences is reduced from 15 to 6 (s0, s3, s4, s12, s13, s14).

In the optimized model, we assign the following locations with the *committed* priority: [Ready] and [Scheduling] in the scheduler automaton, [Initial] and [End] of processes for hierarchical instances, and [seq_ini] and [fsm_ini]. Note that the prioritization trims the search space significantly without violating the execution semantics. The remaining possible sequences still keep the concurrency between transitions in the bodies of leaf instances (for example, transition 2 and 6 in Fig. 5). Our experimental results show that our assumption is valid. Table I in Section V shows one example demonstrating the analysis run time for the model before optimization and after. The analysis results for both models are identical, but the difference in runtime and memory requirements is tremendous.

V. Experiments and Results

We now show experimental results of running MHP analysis on the introductory example and three in-house models of embedded applications, a greyscale JPEG encoder, color JPEG encoder, and MP3 decoder. The generation of the models and queries is very quick. The analysis, however, takes time to verify the satisfiability of the queries.

TABLE I. Run time and memory requirement for optimized model

Optimization (JPEG)	# of queries	# of mhp pairs	total runtime	memory req.
Before	143	51	1h:44m:41s	310MB
After	143	51	42s	26MB

TABLE II. MHP Analysis of SLDL design using UPPAAL model checker

Application	lines of codes	# of queries	# of mhp pairs	total runtime
Intro	39	9	2	$<$1s
Intro-M	39	9	-	∞
Intro-M*	39	9	2*	3s*
Mono-JPEG	1.5k	143	51	42s
Color-JPEG	2.5k	210	25	16m:18s
MP3 Decoder	7k	141	24	21h:32m:36s

Table I shows the comparison before and after the search space optimization described in Section II.F. Note that the memory requirements listed here are obtained by running the verification on an unsatisfiable query. Table II first shows the result for the introductory example of Fig. 3. The verification takes less than one second and reports two out of nine MHP pairs of scheduling points are true. Compared to [5] in which four out of nine are reported true, our approach is more precise. Intro-M is a modified introductory example where the loop is replaced by a while loop. For a design with unbounded loop transitions, the verification tool keeps expanding the search space and tries to find a trace to satisfy the query. For satisfiable properties like the first and last query in Fig. 4 the verification is still quick, but for unsatisfiable properties the verification will not terminate. To deal with this situation, we either set an upper bound for the loop or use the under approximation option provided by the verifier. The results listed in Intro-M* row are obtained with this approximation option. Instead of keeping searching until running out of memory, the verifier replies `"MAY NOT be satisfied"` for the unsatisfiable queries.

The fourth and fifth experiment are MHP analysis of greyscale and color JPEG encoder. We can see that the number of MHP queries for color JPEG is more than greyscale JPEG because there are more leaf instances and scheduling points in the color JPEG encoder. The true MHP pairs in the greyscale encoder, on the other hand, are more than the MHP pairs in the color encoder. The reason is that in the color encoder the communication between modules are implemented with double handshake channels, while the communication in greyscale encoder are implemented with queues. Given the medium size design, the analysis run time is acceptable.

The MP3 decoder is a large example with three times as many instances as the greyscale encoder (34 and 12 respectively). The left and right channel are decoded in parallel and each channel contains multiple instances with children below them. The search space is much larger than the JPEG encoder and it takes much more time to verify the satisfiability. Giving the complexity of formal verification, the run time of less than a day is still reasonable.

VI. Conclusions and Future Work

In this paper, we have described a new approach to identify the MHP statements in a system. Our approach includes the abstraction of the automata network from a design and the generation of queries for MHP analysis. The satisfiability of MHP queries is verified using UPPAAL model checker. Compared to state-of-the-art other work [5], our results are more precise, but take longer to compute. In future work, we plan to shorten the analysis time by introducing more static analysis and generating a more compact set of queries as well as further exploit the priorities of locations to trim the search space.

References

[1] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P.Pettersson, W. Wi, and M. Hendrikes, *"UPPAAL 4.0,"* in Proc. QEST, 2006, pp. 125-126.

[2] P. Herber, and S. Glesner, *"A HW/SW co-Verification framework for SystemC,"* in ACM Trans. Embed Comput, Syst. 12, 1s Article 61, 2013.

[3] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, *"A SystemC/TLM Semantics in PROMELA and its Possible Application, "* in Proc. SPIN, LNCS, 2007, pp 204-222

[4] A.Cimatti. I. Narasamdya, and M. Roveri, *"Software Model Checking SystemC,"* in IEEE Trans. on CAD of Integrated Circuits and Systems 32(5): 774-787 (2013).

[5] W. Chen, X. Han, and R. Dömer, *"May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models,"* in DATE '14 European Design and Automation Association.

[6] W. Chen, X. Han, and R. Dömer, *"Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment,"* in IEEE Design and Test of Computer, vol. 28, pp. 20-31, 2011.

[7] C. Schumacher, J. Weinstock, R. Leupers, and G. Ascheid, *"Scandal: SystemC Analysis for Nondeterminism Anomalies,"* in Forum on Specification and Design Languages, 2012

[8] A. Sen, V. Ogale, and M. S. Abadir, *"Predictive Runtime Verification of Multi-processor SoCs in SystemC,"* in DAC '08 ACM

[9] N. BLANC, E. Zurich, and D. Kroening, *"Race Analysis for SystemC using Model Checking,"* in ACM Trans. Des. Autom. Electron. Syst. 15, 3, Article 21

[10] G. Naumovich and G.S. Avrunin, *"A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel,"* in ACM SIGSOFT on Software Engineering Notes,vol.23,pp.24-34,1998

[11] C. Chang, R. Dömer, "Abstracting ESL Designs to UPPAAL System Models", Center for Embedded Cyber-Physical Systems, Technical Report 14-13, November 2014.