Verificación de Sistemas Concurrentes: Optimalidad, Escalabilidad y Aplicabilidad

Verification of Concurrent Systems: Optimality, Scalability and Applicability

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por

Miguel Isabel Márquez

Dirigida por los profesores

Elvira Albert Albiol

Miguel Gómez-Zamalloa Gil

Facultad de Informática

Universidad Complutense de Madrid

Madrid, Mayo de 2020

Verificación de Sistemas Concurrentes: Optimalidad, Escalabilidad y Aplicabilidad

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por

Miguel Isabel Márquez

Dirigida por los profesores

Elvira Albert Albiol
Miguel Gómez-Zamalloa Gil

Facultad de Informática
Universidad Complutense de Madrid
Madrid, Mayo de 2020

Verification of Concurrent Systems: Optimality, Scalability and Applicability

Ph.D. THESIS

Miguel Isabel Márquez

Advisor: Elvira Albert Albiol
Advisor: Miguel Gómez-Zamalloa Gil

Facultad de Informática
Universidad Complutense de Madrid
Madrid, May 2020

UNIVERSIDAD COMPLUTENSE MADRID

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña. Miguel Isabel Márquez,
estudiante en el Programa de Doctorado en Ingeniería Informática RD99/2011,
de la Facultad de Informática de la Universidad Complutense de Madrid, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Verificación de Sistemas Concurrentes: Optimalidad, Escalabilidad y Aplicabilidad
Verification of Concurrent Systems: Optimality, Scalability and Applicability

y dirigida por: Elvira Albert Albiol y Miguel Gómez-Zamalloa Gil

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita.

Del mismo modo, asumo frente a la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada de conformidad con el ordenamiento jurídico vigente.

En Madrid, a 11 de mayo de 2020

Firmado por ISABEL MARQUEZ MIGUEL - 02274645Z el día 13/05/2020 con un certificado emitido por AC FNMT Usuarios

Fdo.: _____

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

Financial Support. This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the MINECO/FEDER, UE project TIN2015-69175- C4-3-R, the Spanish MINECO project TIN2015-69175-C4-2-R, the Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, and by the CM project P2018/TCS-4314.

Resumen

Tanto el *testing* como la *verificación* de sistemas concurrentes requieren explorar todos los posibles entrelazados no deterministas que la ejecución concurrente puede tener, ya que cualquiera de estos entrelazados podría revelar un comportamiento erróneo del sistema. Esto introduce una explosión combinatoria en el número de estados del programa que deben ser considerados, lo que frecuentemente lleva a un problema computacionalmente intratable. El objetivo de esta tesis es el desarrollo de técnicas novedosas para el testing y la verificación de programas concurrentes que permitan reducir esta explosión combinatoria.

La reducción basada en órdenes parciales (POR) [39] es una teoría general que ayuda a mitigar esta explosión combinatoria mediante la identificación formal de clases de equivalencia de exploraciones redundantes. La teoría POR está basada en la siguiente idea: dos entrelazados pueden ser considerados equivalentes si uno puede ser obtenido desde el otro mediante el intercambio de dos pasos de ejecución que son *independientes*, consecutivos y que no están en conflicto. Tales clases de equivalencia son conocidas como trazas de Mazurkiewicz [55], y la teoría POR garantiza que es suficiente explorar un entrelazado por cada clase de equivalencia. Uno de los objetivos principales de esta tesis es el desarrollo de nuevas técnicas de reducción basada en órdenes parciales.

El pilar fundamental de la teoría POR es la noción de *independencia*, que es usada para decidir si cada par de pasos de ejecución p y t son dependientes y, como consecuencia, las ejecuciones $p \cdot t$ y $t \cdot p$ deben ser exploradas. En 2005, Flanagan y Godefroid propusieron un algoritmo dinámico basado en POR (DPOR) [32], que fue un avance fundamental en este área. Actualmente, DPOR es considerada una de las técnicas más escalables para el testing y la verificación de sistemas concurrentes. Sin embargo, este algoritmo no es *óptimo* en el sentido de que puede explorar varias ejecuciones por cada clase de equivalencia. Optimal-DPOR (ODPOR) [1] es una extensión que garantiza optimalidad. Ambos algoritmos están basados en una relación de (in)dependencia *incondicional* que determina el order parcial de cada par de transiciones, es decir, para que dos transiciones sean consideradas como independientes deben conmutar en todos los estados del programa posibles.

La noción de independencia *condicional* fue introducida en 1992 en el contexto de POR [49], donde fue demostrado que solamente una noción de independencia condicional *uniforme* puede ser utilizada correctamente, es decir, la independencia debe mantenerse a lo largo de toda la traza. El primer algoritmo que ha usado nociones de independencia condicional dentro del algoritmo DPOR clásico es conocido como *Context-Sensitive DPOR* [6] (DPOR_{cs}). Recientemente, Optimal DPOR with Observers (ODPOR^{ob}) [18] ha introducido la noción de *observabilidad*, según la cual, la dependencia entre dos pasos de ejecución p y t está condicionada a la existencia de futuros pasos (observadores) que lean las variables modificadas

por p y t. Uno de los logros principales de esta tesis es ser capaces de beneficiarse de nociones de independencia condicional. Este logro puede separarse en los siguientes retos:

i) combinar y aprovechar las nociones de independencia presentadas en DPOR$_{cs}$ y ODPOR^{ob} y estudiar sus sinergias para obtener mayores reducciones;

ii) explotar la propiedad de uniformidad que permite usar correctamente la noción de independencia condicional dentro de los algoritmos DPOR, usando *restricciones de independencia* (ICs), que garanticen la conmutatividad entre pasos de ejecución;

iii) realizar una evaluación experimental que permita medir las técnicas propuestas; y

iv) aplicar estas técnicas a un escenario realista.

Incluso tras aplicar las técnicas más avanzadas de POR para eliminar redundancias, explorar sistemáticamente las diferentes ejecuciones presenta problemas de escalabilidad. Los análisis estáticos aportan información útil acerca de los programas analizados y puede ser usado para mejorar el comportamiento del *testing*. Para ello, se han propuesto dos retos más:

v) combinar el análisis estático y el *testing* para la detección efectiva de *deadlocks*, y

vi) extender este marco de trabajo al contexto de la ejecución simbólica.

En la conferencia ISSTA'19 [9], hemos presentado Optimal Context-Sensitive DPOR with Observers que aborda el reto i). Para ello, hemos formulado Context-Sensitive DPOR sobre Optimal DPOR, dando lugar al algoritmo Optimal Context-Sensitive DPOR (ODPOR$_{cs}$). Además, hemos integrado la noción de observabilidad dentro de ODPOR$_{cs}$ y el algoritmo resultante es conocido como Optimal Context-Sensitive DPOR with Observers.

En la conferencia CAV'18 [13], hemos introducido el algoritmo Constrained DPOR (CDPOR) que aborda el objetivo ii). CDPOR está basado en condiciones suficientes – que pueden ser chequeadas dinámicamente– para explotar correctamente las ICs dentro del algoritmo DPOR. También, hemos presentado una estrategia basada en tecnología de Satisfactibilidad Módulo Teorías (SMT) para sintetizar automáticamente ICs para bloques atómicos (esto es, bloques de instrucciones ejecutados atómicamente), cuya aplicabilidad va más allá del contexto DPOR.

Esta tesis está respaldada por una evaluación experimental exhaustiva que aborda el objetivo iii), cuyos resultados pueden ser encontrados en [13, 9]. Para afrontar el reto iv), hemos aplicado nuestras técnicas para la verificación de redes definidas por software (SDN) [14], donde éstas han sido codificadas mediante el modelo de actores [5, 42] y hemos aplicado CDPOR para encontrar errores de programación en cinco casos de estudio. En la conferencia CC'16, hemos presentado SYCO [11], una herramienta para *testing* sistemático, que recoge todas estas técnicas y que puede ser encontrada online en `http://costa.fdi.ucm.es/syco`.

El reto v) ha sido abordado mediante el uso de un análisis estático de *deadlocks* [33] para guiar el *testing* hacia ejecuciones que terminen en *deadlock*. Esta técnica ha sido presentada en la conferencia iFM'16 [10]. Además, ha sido extendida al contexto de la ejecución simbólica abordando el reto vi) y presentada en la conferencia LOPSTR'17 [12].

En resumen, en esta tesis hemos propuesto soluciones para todos los retos descritos y hemos llevado a cabo una evaluación experimental para cada uno de ellos. Esta evaluación

experimental permite afirmar que el uso de nociones de independencia condicional dentro de los algoritmos DPOR mejora notablemente los resultados de las técnicas actuales. Finalmente, el uso del análisis estático para guiar el proceso del *testing* ayuda a mitigar todavía más el problema de la explosión de estados.

Abstract

Both verification and testing of concurrent systems require exploring all possible non-deterministic interleavings that the concurrent execution may have, as any of the interleavings may reveal an erroneous behavior of the system. This introduces a combinatorial explosion on the number of program states that must be considered, what leads often to a computationally intractable problem. The overall goal of this thesis is to investigate novel techniques for testing and verification of concurrent programs that reduce this combinatorial explosion.

Partial-Order Reduction (POR) [39] is a general theory that helps mitigate this combinatorial explosion by formally identifying *equivalence* classes of redundant explorations. POR is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps. Such an equivalence class is called a Mazurkiewicz trace [55], and POR guarantees that it is sufficient to explore one interleaving per equivalence class. A main objective of this Ph.D. thesis is the development of new Partial-Order Reduction techniques.

The cornerstone of the POR theory is the notion of *independence*, that is used to decide whether each pair of execution steps p and t are dependent and thus both executions $p \cdot t$ and $t \cdot p$ must be explored. The Dynamic-POR (DPOR) algorithm, introduced by Flanagan and Godefroid [32] in 2005, was a breakthrough in the area. DPOR is nowadays considered one of the most scalable techniques for concurrent software testing and verification. However, it was not *optimal* in the sense that it may explore several executions per equivalence class. Optimal-DPOR (ODPOR) [1] is an extension that guarantees optimality. Both the original DPOR and ODPOR are based on an *unconditional* dependency relation which determines the partial order of transitions, i.e., for two transitions to be considered independent they must commute in all possible program states.

Conditional independence was earlier introduced in the context of POR [49], where it was proven that only *uniform* conditional independence can be used, i.e., independence must hold along the whole trace. The first algorithm that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR (DPOR_{cs}) [6]. Recently, Optimal DPOR with Observers (ODPOR^{ob}) [18] has introduced the notion of *observability*, according to which dependencies between two execution steps p and t are conditional to the existence of future steps called observers, which read the values modified by p and t.

A main achievement of this Ph.D. thesis is to be able to exploit notions of conditional independence. This achievement is split in the next challenges:

i) combine and exploit the notions of DPOR_{cs} and ODPOR^{ob}, and study their synergies to gain further pruning;

ii) exploit the property of uniformity that allows to use soundly the notion of condi-

tion independence within DPOR algorithms using *independence constraints* (ICs), that guarantee the commutativity between execution steps;

iii) carry out a thorough experimental evaluation to compare the different extensions; and

iv) apply the techniques to a realistic setting.

Even after applying the most advanced POR techniques to eliminate redundancies, exploring systematically all different executions poses scalability problems. Static analysis provides useful information about the programs that can be used to improve the performance of testing. For this, two more research challenges are proposed:

v) combine static analysis and testing for effective deadlock detection and

vi) extend this framework to the context of symbolic execution.

At the Conference ISSTA 2019 [9], we have presented *Optimal Context-Sensitive DPOR with Observers* which addresses the challenge i). We have formulated Context-Sensitive DPOR over Optimal DPOR, which is named Optimal Context-Sensitive DPOR (ODPOR_{cs}), and it includes the extension of wake-up trees used to ensure optimality. Furthermore, we have also integrated the notion of observability into ODPOR_{cs} and the resulting algorithm is called Optimal Context-Sensitive DPOR with Observers (ODPOR_{cs}^{ob}).

At the Conference CAV 2018 [13], we have introduced *Constrained DPOR* (CDPOR) which achieves the challenge ii). CDPOR is based on sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework. Moreover, it extends the state-of-the-art DPOR algorithm with new forms of pruning (by means of expanding sleep sets and reducing backtrack sets). We have also presented an approach based on Satisfiability Modulo Theories (SMT) to synthesize ICs for *atomic blocks* (that is, blocks of instructions executed atomically), whose applicability goes beyond the DPOR context.

This thesis is backed up by a thorough experimental evaluation that addresses goal iii), whose results can be found both in [13, 9]. To address challenge iv), we have applied our techniques for the verification of Software-Defined Networks [14], where we have encoded these networks into the actors model[5, 42] and applied CDPOR to find bugs related to programming errors, e.g., forwarding loops and violation of safety policies in five case studies. Moreover, at the Conference CC 2016, we have presented SYCO [11], a tool for systematic testing that includes all the techniques presented in this thesis and it can be found online at `http://costa.fdi.ucm.es/syco`.

Finally, challenge v) has been addressed by using a state-of-the-art deadlock analysis [33] to guide the execution of testing towards paths leading to deadlock. This technique has been presented at the Conference iFM 2016 [10]. Moreover, it has been extended to the context of symbolic execution addressing challenge vi) and presented at LOPSTR 2017 [12].

To summarize, we have proposed solutions for all challenges and we have performed an experimental evaluation for each of them. Our solutions provide actual experimental evidence that using conditional independence within DPOR algorithms improves upon state-of-the-art results, since the experimental evaluations of these new approaches achieve exponential gains compared to DPOR algorithms using unconditional independence. Finally, the use of static analysis to guide the testing process can help mitigate even more the state explosion problem.

Agradecimientos

Sin duda alguna, estos cuatro años han sido una etapa muy bonita en mi vida. Ha habido muchos momentos de felicidad y orgullo, pero también difíciles, en los que he tenido ayuda de mucha gente para superarlos. En primer lugar, quería agradecer a mi directora de tesis, Elvira Albert, quien me brindó la oportunidad de comenzar en el mundo de la investigación hace seis años. Desde entonces, siempre ha estado ahí para ayudarme, aconsejarme y apoyarme. También, a mi director de tesis, Miguel Gómez-Zamalloa, a quien le quiero agradecer la confianza depositada en mí, así como su ayuda y consejo, cuando he tenido que tomar decisiones difíciles.

Gracias a todos los miembros del grupo COSTA, quienes siempre han estado dispuestos a echarme una mano cuando lo he necesitado: Samir Genaim, Jesús Correas, Guillermo Román y Enrique Martín. En especial, a Albert Rubio. Es un gusto poder trabajar y aprender de él, aunque exista peligro de irnos por las ramas, y que Elvira tenga que volver a centrar la conversación; y a Puri Arenas, por sus consejos, sus historias, y su humor cada vez que bajamos a fumar.

Los días de trabajo son mucho más llevaderos y amenos, cuando llegas al Aula 16 y te encuentras compañeros y amigos, que siempre están ahí para echarte una mano, unas risas, o unos copazos, dependiendo de la ocasión (preferiblemente, en ese orden). Gracias a Toni Calvo, con quien viví un terremoto en Japón; a Marta Caro, que siempre tiene una sonrisa y un gesto cariñoso que regalar; a Joaquín Gayoso, por siempre estar dispuesto a soltar una "joaquinada" para echarnos unas risas. A Luisma Costero y Cristina Alonso, con quienes llevo casi diez años compartiendo anécdotas y sufrimientos mientras estudiábamos tanto el doble grado como el doctorado. No puedo olvidarme de Jesús Doménech, Alicia Merayo y Pablo Gordillo, con quienes compartí una de las mejores experiencias en esta etapa: la escuela de verano en Oxford, menudas semanas de aprendizaje, diversión y cerveza. Como decía al comienzo, han sido años muy felices, con algún que otro momento difícil. En cada uno de esos momentos, siempre ha estado Pablo, con quien poder desahogarme, aconsejándome y ayudándome (y vigilando que hacía la burocracia correctamente).

Y si el día había sido duro, mi familia siempre estaba ahí para sacarme una sonrisa. Quiero agradecer a mi madre, siempre preocupada porque yo dé lo mejor de mí en todos los ámbitos de mi vida, ya que gracias a ella, soy como soy. A mis tíos y primos, por ser mis segundos padres, y los hermanos que no tuve. Y por último, a mi pareja, Cristhian, por animarme a que me fuese a Melbourne de estancia; por venirse conmigo sin dudarlo ni un momento, para vivir la experiencia más bonita de mi vida; y en definitiva, por haberme acompañado y ayudado durante estos tres años, sin duda, los más felices.

Contents

Parte I

Contents of the Thesis

Chapter 1

Introduction & State-of-the-Art

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is present everywhere in today's software applications. Most of the modern systems are designed to take advantage of as much parallelism as possible by allowing system components to execute concurrently. Such components usually share resources and data, hence the operations executed by different components can interfere with each other.

This data sharing is even more common in the context of concurrent *imperative* languages. Most of them make use of a global memory, called *heap*, to which the different components (e.g. threads, processes, tasks...) can have access. Thus, the orderings of accesses to heap memory locations performed by different components may produce possibly different final states, some of the algorithms leading to errors and/or unexpected behaviors. Most of these hazards, e.g., race conditions, data races, deadlocks, and livelocks are not present in sequential programs. For instance, a *deadlock* situation arises as a consequence of a circular dependency among components waiting for each other's resources.

To avoid these situations, a common approach is to restrict the sharing of data and/or resources by the different components. However, this restriction reduces most of the desired concurrency. Thus, programmers are usually forced to develop programs containing this sharing, but also being aware of the underlying data dependencies and, as a result, the unexpected behaviors that may appear. Indeed, some of these behaviors may not manifest until weeks or months after system release, because they might only happen for specific schedulings that are unlikely to occur.

The number of different behaviors of a program usually grows exponentially with the size of the system, making it impossible for a programmer to have control over them. Therefore, software validation techniques urge especially in the context of concurrent programming in order to detect these problems and verify the systems before being released. Unfortunately, concurrent systems are not only difficult to develop but they are also difficult to verify, debug, test and analyze.

Testing is one of the most widely-used methodologies for software validation. Several studies point out that it requires at least half of the total cost of a software project. In the last twenty years, a great deal of research has focused on testing and, as a consequence, numerous families of techniques have been developed. For instance, *white-box* testing, is an approach in which the availability of the code of the program under test is assumed, while *black-box* testing uses the requirement specifications of the program instead of the code.

Another common classification of testing is made depending on whether the program code is executed or not: in *dynamic* testing, the program is executed to validate the real output w.r.t. the expected output, whereas in *static* testing, the program code and/or its associated specification and documentation are examined without the program being run.

Traditional testing for concurrent programs is not as effective as for sequential programs, since in order not to lose any possible behavior, in principle, it must systematically explore all possible ways in which the processes or tasks can interleave. This is known as *systematic testing* [60]. This full exploration produces a *state explosion problem* that is often too time-consuming and computationally intractable.

Example 1. *Let us consider the following example: it contains three global variables* x, y *and* z *which are initialized to* 0*. The program contains three processes:* p*,* q *and* r*. Processes* p *and* q *execute two instructions: the first instruction of each process modifies the global variable* y *and* z*, respectively, whereas the second one writes the global variable* x*. Process* r *reads variable* x *and checks if it is greater than* 0*.*

Int $y = 0$; *Int* $z = 0$; *Int* $x = 0$;

process p :	*process* q :	*process* r :
$y = 1$;	$z = 2$;	*assert* $x > 0$;
$x = 5$;	$x = 4$;	

This simple example only contains five instructions and three processes, however, if the program is run, it may result in thirty different executions depending on the execution order of the instructions. Let us denote each execution by the sequence of process names following the execution order. For instance, sequence $ppqqr$ *has executed the two instructions of process* p*, followed by the two instructions of process* q *and finally, the instruction of process* r*. After the execution of sequence* $ppqqr$*, the assert holds and the final value for each global variable is* x, y, $z = 4, 1, 2$*. In sequence* $qqppr$*, the assert also holds, but the final value for each variable is* x, y, $z = 5, 1, 2$*. On the other hand, in sequence* $rppqq$*, the assert does not hold since process* r *is executed before* p *and* q *and, thus it reads the initial value of* x*, which is* 0*. Finally, let us consider the sequence* $pqpqr$*: the two first executed instructions write variables* y *and* z*. Then, the instructions involving variable* x *are executed in the same order than in sequence* $ppqqr$*:* r *again reads value* 4*, the assert holds and the execution reaches the same final value for each global variable. For this reason,* $ppqqr$ *and* $pqpqr$ *can be seen as equivalent, and systematic testing may only consider one of them to be explored.*

One of the most widely-used techniques to mitigate such state explosion problem is *Partial-Order Reduction* (POR). This theory is based on a well-known fact: many different executions are often leading to equivalent final states. POR avoids exploring those executions that are guaranteed to produce the same results. State-of-the-art POR algorithms are able to detect redundant executions dynamically during the exploration and allow generating only one execution per equivalence class, avoiding a large number of redundancies. In the next section, we will see how these algorithms alleviate the state explosion.

1.1 Partial Order Reduction

Partial-Order Reduction (POR) [39] is a general theory that helps mitigate the state explosion problem by formally identifying *equivalence* classes of redundant executions. Each equivalence class is called a Mazurkiewicz trace, and the POR theory guarantees that it is sufficient to explore one interleaving per equivalence class. POR is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps.

A cornerstone of POR is hence the notion of *independence*. Two execution steps are *independent* if at every program state (1) they do not enable/disable each other and (2) their executions commute, that is, both orderings lead to the same final state. Each execution sequence has a total order, defined by the execution order of the different processes involved in the execution sequence and also a partial order, called *happens-before*, induced by the notion of *dependency* between the processes in the sequence. Formally, each equivalence class is composed of all the execution sequences with the same partial order. Consequently, exploring only an execution per equivalence class will be enough for studying the most interesting safety properties, including race freedom, absence of assertion violations and deadlocks.

Example 2. *Let us consider again Example 1, we can see that the execution of instruction* `y = 1;` *is independent with the execution of any instruction of another process, but it is always dependent with the execution of* `x = 5;`*, since both instructions are executed by process p. On the other hand, instructions* `x = 5;` *and* `x = 4;` *are dependent, since they write a different value and thus, their executions do not commute. Finally, any of the two previous instructions that assign a value to* `x` *and the instruction* `assert x > 0;` *are dependent, since if* `x` *is not greater than* 0 *in the program state before they are executed, then they do not commute.*

1.1.1 Basics of Partial Order Reduction

In this section, we formalize the ideas described in the previous section: we introduce the notation needed to explain the POR theory and also auxiliary definitions that will be used throughout this thesis.

An *event* e of the form p_i denotes the i-th occurrence of process p in an execution sequence and $\hat{e}$ denotes process p of event e. We use $e <_E e'$ to denote that event e occurs before event e' in E, s.t. $<_E$ establishes a total order between events in E, and $E \leq E'$ to denote that sequence E is a prefix of sequence E'.

The core concept in POR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation defines a subset of the total order $<_E$, such that any two sequences with the same happens-before order are equivalent. Any linearization E' of $\rightarrow_E$ on the set of events in execution sequence E is an execution sequence with the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. Thus, $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same happens-before relation. We use $E \simeq E'$ to denote that E and E' are two executions with the same happens-before relation and $[E]_{\simeq}$ to denote the equivalence class of sequence E.

The happens-before partial order has traditionally been defined in terms of a *dependency* relation between the events in an execution sequence [39]. Two events p and q are *dependent* if there is at least one execution sequence E for which they do not commute, either because

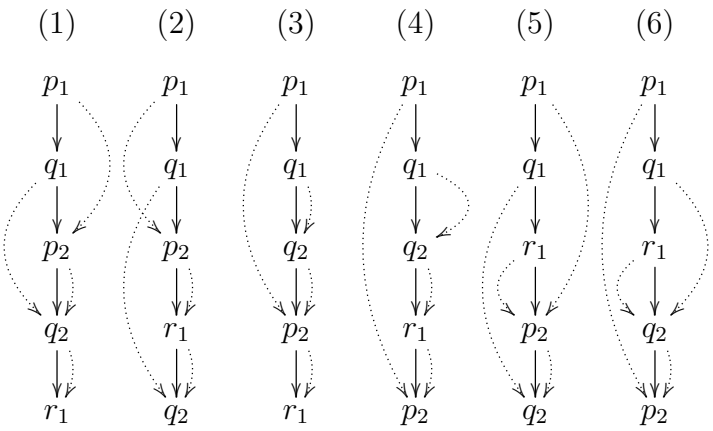


Figure 1.1: Happens-before relations for Example 1

(i) one *enable*s the other (i.e., the execution of p leads to introducing q or vice-versa), or because (ii) the final state of executions after $E.p.q$ and $E.q.p$ is not the same.

Example 3. *Figure 1.1 shows the happens-before relations for six executions of the previous example. The dotted arrows indicate a happens-before order between the events and the continuous arrows indicate the total order $<_E$ within the execution sequence E. Transitive arrows are omitted, such as the one from p_1 to r_1 in the first execution. As explained in Example 2, dotted arrows from p_1 to p_2 and from q_1 to q_2 are caused by the execution of the first event enabling the second one. The remaining dotted arrows are due to the fact that events are not commutative. For instance, sequence $E = ppqqr$ has the following happens-before pairs: $p_1 \rightarrow_E p_2$ and $q_1 \rightarrow_E q_2$, due to enabling dependencies, $p_2 \rightarrow_E q_2$, since both events modify variable* x *and $q_2 \rightarrow_E r_1$, given that event q_2 modifies the variable read by r_1. As mentioned above, transitive pairs are omitted. Each of the thirty executions of this example belongs to one and only one of these six equivalence classes:*

- *In the first equivalence class, the assert holds after reading the value* 4 *and its final state is $x, y, z = 4, 1, 2$. This class contains three executions: pqpqr, qpppr and ppqqr.*
- *In the second equivalence class, the assert holds after reading the value* 5 *and its final state is $x, y, z = 4, 1, 2$. This class contains four executions: pqprq, qpprq, pprqq and ppqrq.*
- *In the third equivalence class, the assert holds after reading the value* 5 *and its final state is $x, y, z = 5, 1, 2$. This class contains three executions: pqqpr, qpqpr and qqppr.*
- *In the fourth equivalence class, the assert holds after reading the value* 4 *and its final state is $x, y, z = 5, 1, 2$. This class contains four executions: pqqrp, qpqrp, qqrpp and qqprp.*
- *In the fifth equivalence class, the assert does not hold after reading the value* 0 *and its final state is $x, y, z = 4, 1, 2$. This class contains eight executions: rppqq, rpqpq, rqppq, prpqq, prqpq, qrppq, pqrpq and qprpq.*

- *In the sixth equivalence class, the assert does not hold after reading the value* 0 *and its final state is* $x, y, z = 5, 1, 2$. *This class contains eight executions: rqqpp, rpqqp, rqpqp, prqqp, qrqpp, qrpqp, pqrqp and qprqp.*

The happens-before relation is used for defining the concept of a *race* between two events. Event e is said to be in a race with event e' in execution E, if the events belong to different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent", i.e. there exists an *equivalent* execution sequence $E' \simeq E$ where the two events are adjacent. We write $e \precsim_E e'$ to denote that e is in a race with e' and that the race can be reversed (i.e., the events can be executed in reverse order).

Example 4. *Let us consider again the previous example and sequence* $E = ppqqr$, *which has the same happens-before relation than (1) in Figure 1.1. Event* p_2 *is in a reversible race with* q_2, *written* $p_2 \precsim_E q_2$, *because (1) they are dependent, (2) there exists an equivalent execution* E' *(which is* $pq\underline{pq}r$*) where they are adjacent and (3)* q_2 *can be executed before* p_2 *to reverse the race. However,* p_2 *and* r_1 *are not in a race in spite of being dependent, since there does not exist an equivalent execution* $E' \simeq E$ *where* p_2 *and* r_1 *are adjacent.*

1.1.2 State-of-the-art DPOR Algorithm

Partial-Order Reduction algorithms use the happens-before relation to reduce the number of equivalent execution sequences explored, some of them ensuring that only one execution sequence in each equivalence class is explored. Early POR algorithms [39, 28, 63] relied on static over-approximations to detect possible *future* races between events. The Dynamic-POR (DPOR) algorithm, introduced by Flanagan and Godefroid [32] in 2005, was a breakthrough in the area because it does not need to look at the future. It keeps track of the reversible races witnessed along with its execution and uses them to decide the required exploration dynamically, without the need for static approximation. DPOR is nowadays considered one of the most scalable techniques for software verification.

Algorithm 1 shows a simplification of the state-of-the-art DPOR algorithm. The algorithm carries out a depth-first exploration of the execution tree using POR receiving as parameter an execution sequence E (initially empty). Essentially, it dynamically finds reversible races and is able to backtrack with the involved process at the appropriate scheduling points to reverse them. E.g., in the case of Example 4, for the race between p_2 and q_2 in sequence $ppqqr$, the algorithm will backtrack to explore the sequence $pqqpr$, where the race is reversed.

The key of DPOR algorithms is in the dynamic construction of two types of sets at each scheduling point:

- The *backtrack set* contains processes which must be selected on backtracking in order to reverse a race previously detected.

- The *sleep set* contains processes whose exploration has been proven to be redundant (and hence should not be selected). It ensures that, given a sequence E and a process $q \in sleep(E)$, then $\forall w$ such that $E.q.w$ is an execution sequence, it is guaranteed that either $E.q.w$ has been explored by $Explore(E.q)$ or another sequence $E' \simeq E.q.w$ has been already explored.

Algorithm 1 state-of-the-art DPOR algorithm

```
1:  procedure EXPLORE(E)
2:    if ∃ p ∈ (enabled(E)\sleep(E)) then
3:      back(E) := {p};
4:      while ∃ p ∈ (back(E)\sleep(E)) do
5:        RACE_DETECTION_PHASE(E, p)
6:        sleep(E.p) := propagate(sleep(E), p);
7:        EXPLORE(E.p);
8:        add p to sleep(E);
9:  procedure RACE_DETECTION_PHASE(E, p)
10:   for all e ∈ E such that e ≾_{E.p} p do
11:     let E' = pre(E, e);
12:     let v = reversed_race(e, p, E);
13:     choose q ∈ v and add q to back(E');
```

The algorithm starts by selecting any process p that is ready to be executed in the state reached after executing sequence E (this is returned by function *enabled(E)*) and is not already in $sleep(E)$ (line 2). If it does not find any such process p, it stops the exploration of sequence E, since E is a *complete* execution. There can be two possible reasons to stop: (1) $enabled(E)$ is empty and, then execution E is completed because it cannot execute anything else, or (2) every process in $enabled(E)$ is already in $sleep(E)$, which means that it is not necessary to keep on exploring, because every possible continuation $E.w$ is redundant with other executions which have been already explored.

In case it finds a process p in line 2, it sets the backtrack set $back(E)$ to $\{p\}$. Then, it carries out a depth-first exploration of every element in $back(E)$ that is not in $sleep(E)$. The backtrack set of E might grow as the loop progresses (due to later executions of line 13).

For each such p, DPOR performs two phases:

- Race detection phase (lines 10 - 13). The race detection starts by finding all events e executed by processes in sequence E (written $e \in E$) that are in a reversible race with the last event of process p in $E.p$ (line 10). Let us notice here that, in order to simplify the notation, (1) we use process p to denote the next event performed by p after E and (2) we use $e \in E$ to denote the events executed by the processes in E.

  For each reversible race $e \precsim_{E.p} p$ detected during this phase, a process must be added to a backtrack set of a previous scheduling point in order to reverse such race. The scheduling point is the prefix E' of E just before executing the event e (such prefix is returned by function $pre(E, e)$, line 11). However, there may be situations where p is not enabled in E' and the execution of other events may be necessary before executing p. In particular, function $reversed\_race(e, p, E)$ returns the sequence of processes between E' and E such that the event executed by each of them is dependent with event p, followed by process p. Finally, $back(E')$ needs to be updated with one of the processes in v with no happens-before predecessors in such sequence (line 13).

- State exploration phase (lines 6, 7 and 8). After the race detection phase, the algorithm continues with the state exploration phase for $E.p$. Before exploring the sequence

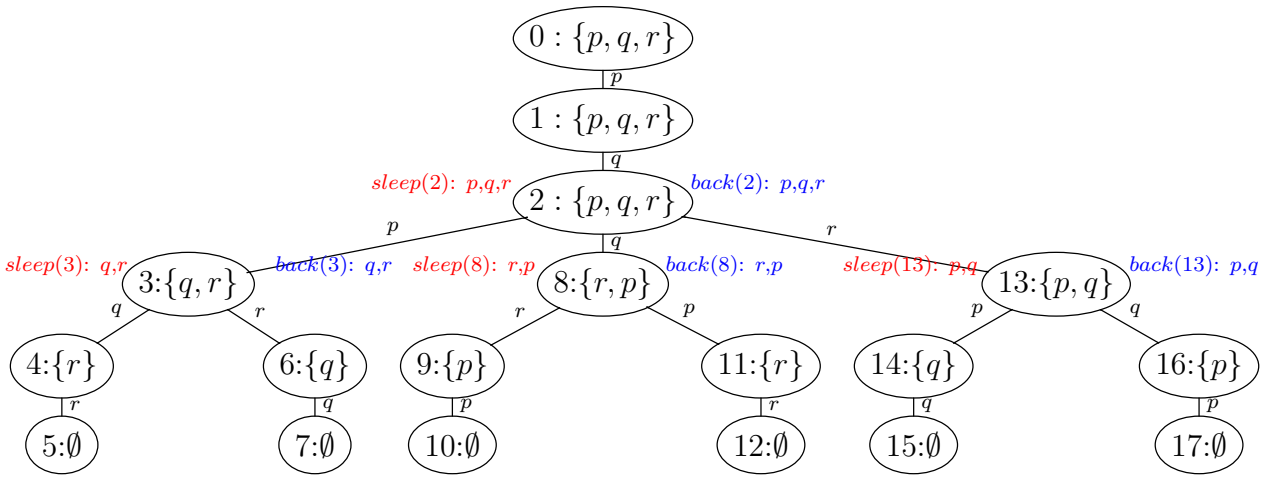


Figure 1.2: Full execution tree computed by DPOR for Example 5. Node labels: Enabled Processes. Arrow labels: scheduled process. Node labels right (in blue): backtrack set. Node labels left (in red): sleep set. Both backtrack and sleep sets are only indicated for nodes with more than one process.

$E.p$, it is important to propagate down the processes which are in $sleep(E)$ and that are independent with the event executed by process p to prevent the exploration of sequences already explored. This is performed by the function *propagate(E,p)* in line 6. Then, the algorithm explores $E.p$, and finally, it adds p to $sleep(E)$ to ensure that, when backtracking on E, p is not selected until a dependent event with it is selected.

Example 5. *Let us consider again the previous example with three processes:*

$$\textit{Int } y = 0; \quad \textit{Int } z = 0; \quad \textit{Int } x = 0;$$

process p :	*process q :*	*process r :*
$y = 1;$	$z = 2;$	$\textit{assert } x > 0;$
$x = 5;$	$x = 4;$	

The tree computed by DPOR for this example is shown in Figure 1.2. DPOR starts with the call EXPLORE*(ϵ) (state 0), that is, with the empty execution sequence ϵ. Every backtrack and sleep set is initially empty. The check in line 2 is true because $enabled(\epsilon) = \{p, q, r\}$ and $sleep(\epsilon)$ is initially $\emptyset$, thus one of these processes is chosen to be added to the backtrack set $back(\epsilon)$. Let us suppose that p is chosen. Now, the race detection phase (lines 10-13) finishes without detecting any race and it performs a call* EXPLORE*(p) (state 1).*

Again, the check in line 2 is true because $enabled(p) = \{p, q, r\}$ and $sleep(p) = \emptyset$. Thus, another process is randomly chosen. Let us suppose that q is chosen, then the race detection phase finishes without detecting races, because events p_1 and q_1 are independent, since they do not modify any shared variable read or modified by the other one. Hence, a call to EXPLORE*(pq) is performed (state 2). Similarly, p can be randomly chosen and p_2 is independent with q_1, thus no race is detected. After that, the algorithm performs a call to* EXPLORE*(pqp) (state 3). Now, q and r are both enabled after pqp. Let q be the next chosen*

process. During the race detection phase, a race $p_2 \precsim_{pqpq} q_2$ *is detected. In line 11,* E' *is set to* pq*, because it is the prefix of* $pqpq$ *before executing event* p_2*. In line 12,* v *is set to* $reversed\_race(p_2, q_2, pqpq) = q$*, since after* E' *there is not any other event which is dependent with* q_2*, thus* q *is the unique process in the sequence. Hence, it is added to the backtrack set* $back(pq)$ *(state* 2*, in blue) in line 13.*

A new call EXPLORE($pqpq$) *is performed (state* 4*). The only enabled process is* r*, then it is selected. A new race* $q_2 \precsim_{pqpqr} r_1$ *is detected. Now,* E' *and* v *are set to* pqp *and* r*, respectively. Therefore,* r *is chosen to be added to* $back(pqp)$ *(state* 3*). Finally, the call* EXPLORE*(*$pqpqr$*) is performed (state* 5*), where* $enabled(pqpqr)$ *is empty, i.e., the current execution sequence finishes. Then, DPOR backtracks to the first point where a new process has been added to the backtrack set, that is, at state* 3 *(line 4). On its way back, it adds every chosen process to its corresponding sleep set (*r *and* t_2 *to the sleep at state* 4 *and state* 3*, respectively), in line 8. Let us notice that we do not show in Figure 1.2 the sleep set for nodes whose information is irrelevant in order to simplify the figure.*

Now, process r *is chosen in line 4 and, thus, a race* $p_2 \precsim_{pqpr} r_1$ *is detected. Process* r *is added to* $back(pq)$ *(state* 2*). Once the race detection phase is over, function* propagate *in line 6 does not propagate* q *from* $sleep(pqp)$ *to* $sleep(pqpr)$ *since* q_2 *and* r_1 *are dependent. After the call* EXPLORE*(*$pqpr$*) is performed,* q *is chosen and the race* $r_1 \precsim_{pqprq} q_2$ *is detected. However,* q *is already in* $back(pqp) = \{q, r\}$*, therefore nothing new is added to the backtrack set. The execution* $pqprq$ *has been completely explored, hence the algorithm backtracks to state* 2 *adding, on its way back,* q*,* r *and* p *to the sleep sets of states* 6*,* 3 *and* 2*, respectively.*

Since events p_2*,* q_2 *and* r_1 *are dependent two by two, the DPOR algorithm must explore the six different executions shown in the execution tree. Let us notice that there are other twenty four executions that could be explored from states* 0 *and* 1 *whose exploration has been avoided by the DPOR algorithm.*

Algorithm 1 is correct in the sense that for each execution sequence E, it explores an execution sequence in $[E.v]_{\simeq}$ for some v. In particular, if E is complete, then Algorithm 1 explores an execution sequence in $[E]_{\simeq}$.

Theorem 1 (Correctness of Algorithm 1 [1])**.** *For all execution sequences* E*, Algorithm 1 explores some execution sequence* E'*, which is in* $[E.v]_{\simeq}$ *for some* v*. In particular, for all complete execution sequences* E*, Algorithm 1 explores some execution sequence* E'*, which is in* $[E]_{\simeq}$*.*

1.1.3 Optimality in DPOR Algorithms

Godefroid et al. [38] prove that DPOR algorithms are *optimal* in the sense that they do not explore two equivalent complete execution sequences thanks to the use of sleep sets (check in line 2). In 2014, Abdulla et al. propose a more precise notion of optimality for the DPOR algorithms.

Definition 2 (Optimality)**.** *A DPOR algorithm is optimal iff*

1. *it never explores two complete execution sequences that are equivalent, and*
2. *no call to* EXPLORE*(*E*) is ever stopped (in line 2) because* $enabled(E) \subseteq Sleep(E)$*.*

That is, it never starts the exploration of a sequence which is eventually stopped by a sleep set. The algorithm introduced in Section 1.1.2 is non-optimal using this notion of optimality. In [2], a new algorithm, called Optimal DPOR (ODPOR), is proposed and it guarantees the optimality of the exploration. Three major extensions are needed to achieve optimality:

- the use of a *redundancy check* between lines 12 and 13 in Algorithm 1 to detect if the backtrack set must be updated with a new process or, otherwise, if sequence v is redundant in E'. A sequence v is redundant after sequence E' if one of the events executed by v which does not have happens-before predecessors in v is already in $back(E')$. If the check is false, then the algorithm already contains a process that guarantees that a sequence equivalent to $E'.v$ will be explored.

- the use of *source* sets as backtrack sets. Godefroid et al. [38] proposed a particular backtrack set, called *persistent* set. Source sets are often smaller than persistent sets, and when a persistent set contains more elements than the corresponding source set, the additional elements will initiate sequences that are eventually stopped by a sleep set. In [2], on the way to the definition of Optimal DPOR, the authors also propose Source-DPOR (SDPOR), an algorithm using source sets, which is not yet optimal, but which behaves more effectively than the original DPOR algorithm.

- the use of *wake-up trees* to guide the initial steps in the exploration. These wake-up trees replace the source sets indicating exactly the sequence of processes that must be executed to reverse the races detected by the algorithm. These sequences are given by function *reversed_race* (in line 12), which must be also redefined. Function *reversed_race(e,p,E)* returns the sequence of processes executed after E' such that the events executed by each of them are independent with event e, followed by process p.

Another difference between the previous DPOR algorithms and ODPOR is that the latter delays the race detection phase until the current execution sequence has been completely explored. The reason for this is that the new redundancy check is accurate when function *reversed_race* takes into account all events in the entire execution to define sequence v.

Using these extensions, ODPOR guarantees that redundant explorations are never even initiated, proving optimality for *any* number of processes w.r.t. an *unconditional independence* relation.

1.1.4 On Improving the Dependency Relation

Even though optimal DPOR algorithms explore exactly one execution per equivalence class, the scalability of these algorithms can be improved. Improving the precision of the dependency relation helps induce a smaller number of equivalence classes to be considered during the execution of the DPOR algorithms.

Approximating the Unconditional Independence

The notion of independence considered in Section 1.1.1 is unconditional, i.e., it must hold for every program state. This requirement forces us to use an over-approximation of the *dependency* relation (an under-approximation of the independence relation) which results in thinner

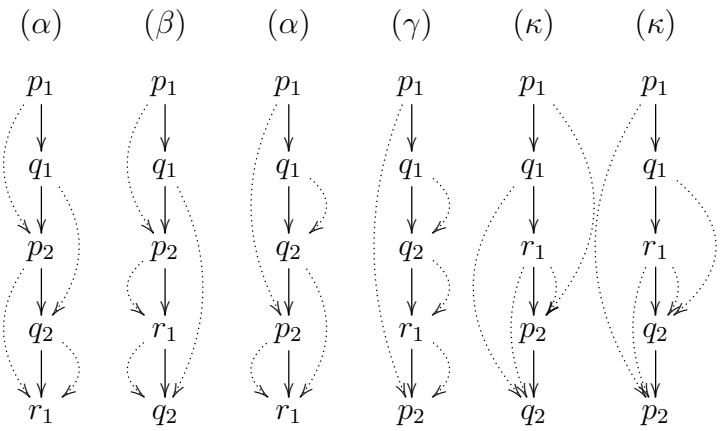


Figure 1.3: Happens-before relations for Example 6

equivalence classes in case of loss of precision. The most widely-used over-approximation is to consider two events as dependent if both access the same global variable and at least one of them is modifying it. This approximation can be imprecise in many situations and increases the number of equivalence classes that must be taken into account.

Example 6. *Let us consider a modification of the previous examples. Here, process q also writes the value* 5*:*

$$\underline{\mathit{Int}\ y = 0; \quad \mathit{Int}\ z = 0; \quad \mathit{Int}\ x = 0;}$$

process p :	*process q* :	*process r* :
$y = 1;$	$z = 2;$	$\mathsf{assert}\ x > 0;$
$x = 5;$	$x = 5;$	

Instructions $x = 5;$ *of process p and* $x = 5;$ *of process q are unconditionally independent, since they write the same value, thus they commute at every possible program state. According to the traditional over-approximation, both are dependent because they modify variable* x*. The happens-before relations for this program using the previous over-approximation coincide with the relations in Figure 1.1. Hence, the imprecision of the approximations used in the POR theory leads to exploring more redundancies.*

Let us consider now a better over-approximation that considers p_2 *and* q_2 *as unconditionally independent, since both events write the same value. Figure 1.3 shows the happens-before relation according to this new approximation. For the first execution, we can see that* p_1 *and* q_1 *happens-before* p_2 *and* q_2*, respectively, because the former events enable the latter ones. Moreover, these two last events happen-before* r_1*, since they write variable* x *that is read by* r_1*. Using this new approximation, we lose the arrows between* p_2 *and* q_2*, and vice-versa. Therefore, executions like pqpqr and pqqpr are now equivalent because, as it can be observed in the figure, they have the same happens-before relation. Consequently, there are only four different equivalence classes instead of six using the less precise approximation.*

Using Notions of Conditional Independence

As mentioned before, both the original DPOR and most of its extensions are based on an *unconditional* dependency relation (also called unconditional happens-before relation) which determines the partial order of events. Conditional independence was early introduced in the context of POR [49]. It defines the conditional dependency relation for two events at each state, instead of being defined for all possible states. For instance, even though instructions like $\mathsf{x} = 5$; and assert $\mathsf{x} > 0$; are unconditionally dependent, they are conditionally independent in states where variable x is greater than 0.

Throughout this thesis, we are going to see different extensions of this basic DPOR algorithm which improve the precision of the notions of independence considered with the objective of achieving reductions in the number of equivalence classes to be explored. The first work that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR [6]. It checks commutativity for each pair of events in a reversible race dynamically during the exploration to prune redundant exploration. However, it exploits conditional (*context-sensitive*) independence only *partially* to extend the sleep sets, but not to reduce backtrack sets. Optimal DPOR with Observers [18] introduces another kind of conditional independence based on the notion of *observability*, according to which dependencies between execution steps p and t are conditional to the existence of future steps, called observers, which read the values modified by p and t.

Example 7. *Let us consider Example 6. Using the notion of context-sensitive independence, the number of equivalence classes can be reduced, since after the execution of event p_2, events q_2 and r_1 are commutative. Using this notion, DPOR algorithms can consider sequences pqpqr and pqprq as equivalent, exploring only one of them. Similarly, sequences pqqpr and pqqrp can be also considered as equivalent. Using notions of conditional independence, DPOR algorithms only need to explore two different executions.*

To illustrate the power of the notion of observability, let us consider Example 5. Using this notion, sequences pqrpq and pqrqp are equivalent since the observer r_1 reading variable x is executed before p_2 and q_2, and thus, one does not happen-before the other one.

1.2 Deadlock-Guided Testing

In concurrent programs, *deadlocks* are one of the most common programming errors and thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and detecting deadlock executions. A deadlock situation arises as a consequence of a circular dependency among components waiting for each other's resources.

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [33, 35] and thread-based programs [54], is based on static analysis techniques. Moreover, deadlock detection has been intensively studied in the context of dynamic testing and model checking [27, 47, 16, 43]), where it sometimes combines testing with static information [47].

Static analysis and testing are two different ways of detecting deadlocks. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. This aspect of static

analysis is especially important in security assurance – security attacks try to exercise an application in unpredictable and untested ways. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a deadlock is found, state-of-the-art analysis tools [33, 36, 37] provide little (and often no) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem.

In contrast, testing consists in executing the application for either concrete input values or symbolic input values. Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings.

The primary advantage of systematic testing for deadlock detection is that it can provide the detailed deadlock trace with all information that the user needs in order to fix the problem. There are two shortcomings though: (1) Although POR techniques try to avoid redundant exploration as much as possible, the search space of systematic testing (even without redundancies) can be huge. (2) In concrete testing, there is only guarantee of deadlock freedom for finite-state terminating programs (terminating executions with concrete inputs); and in symbolic testing, one also needs to assume some termination criteria (e.g., a maximum number of iterations for each loop in the program) and, thus, it is again not possible to ensure deadlock freedom in programs that exceed the limits considered during systematic testing.

Static analysis and testing often complement each other and thus it seems quite natural to combine them. A particular case of this, which has been subject of work in this thesis is *deadlock-guided testing*, where the testing exploration is driven towards potential deadlock executions (while other executions are not explored) using the information provided by a deadlock analysis.

1.3 Main Goals and Contributions

Let us enumerate which are the main goals of this thesis. A main objective has been to be able to exploit notions of *conditional independence* –which ensure the commutativity of the considered events p and t under certain conditions that can be evaluated in the explored state– with DPOR algorithms to alleviate the combinatorial explosion problem. This achievement is split in the next challenges:

i) Combine and exploit the notions of context-sensitive independence, proposed in Context-Sensitive DPOR [6], and independence modulo observability, proposed in Optimal DPOR with Observers [18], and study their synergies to gain further pruning.

Paper 1 addresses this challenge. We have proposed a novel algorithm called Optimal Context-Sensitive DPOR with Observers, where a new kind of independence is defined to combine the previous notions. This work has been published at the Proceedings of the International Symposium on Software Testing and Analysis 2019 (ISSTA 2019).

ii) Use the notion of conditional independence within DPOR algorithms to detect less reversible races, reducing the backtrack sets and thus, improving the scalability of these algorithms. This challenge has been split in three subgoals:

ii.a) Statically synthesize independence constraints (ICs) for blocks of instructions in an automatic pre-analysis using a fully automatic SMT approach, that guarantee the commutativity of the considered events if the ICs hold.

ii.b) Propose (sufficient) conditions that ensure *uniformity* [49], a property that enables using soundly the notion of conditional independence and pruning exponentially the DPOR search state using ICs.

ii.c) integrate the notion of uniform conditional independence (which requires to look ahead) to prune the search space in a *dynamic* algorithm using ICs.

These challenges have been addressed in Paper 2, which has been published at the Proceedings of the conference Computer-Aided Verification (CAV 2018). In this work, we have proposed a new notion of independence based on the *uniform conditional independence*, that makes easier the static approximation of the uniformity. Furthermore, we have obtained exponential gains in the experimental evaluation in comparison with classical DPOR algorithms.

iii) carry out a thorough experimental evaluation to compare the different extensions.

iv) apply these techniques to a realistic setting.

All the papers presented in this thesis are supported by their corresponding experimental evaluation. We have also applied our DPOR techniques to a more realistic setting: the Software-Defined Networks (SDN). First, we have modelled several case studies of SDN and we have evaluated different network properties applying our DPOR algorithms. This work can be found in Paper 3 which is currently under revision in the Journal of Logical and Algebraic Methods. In Paper 7, we have also presented a novel tool for systematic testing, called SYCO [11], that we have used to evaluate the different experimental evaluations of each contribution in this thesis. This description of SYCO can be found at the Proceedings of Compilers Construction 2016 (CC 2016). Paper 4, also published at the Proceedings of the International Symposium on Software Testing and Analysis 2019 (ISSTA 2019), summarizes our main contributions in the field of DPOR.

Even applying the most advanced POR techniques to eliminate redundancies, systematically exploring all different equivalence classes poses scalability problems. Static analysis provides useful information about the programs that can be used to improve the performance of testing. In order to achieve further gains using a static deadlock analysis, two more research challenges are proposed:

v) combine static analysis and testing for deadlock detection and

vi) extend these results to the context of symbolic execution.

These last two challenges have been addressed in Papers 5 and 6. They have been published at the Proceedings of integrated Formal Methods 2016 (iFM 2016) and Logic-based Program Synthesis and Transformation 2017 (LOPSTR 2017), respectively. The experimental evaluation performed to evaluate both contributions shows exponential gains in comparison with systematic testing.

1.4 Organization of this Thesis

This thesis is written in the format "thesis by articles" and it consists of an introduction describing its main objectives, the state of the art, contributions and conclusions, which are presented in chapters 2, 3, 4, 5 , 6, and the set of papers which support the thesis as they appear on the corresponding formal proceedings in Chapter 7. The rest of the thesis is thus structured as follows:

- Chapter 2 overviews our work on Dynamic Partial Order Reduction addressing challenge i). In particular, it provides the basic ideas behind Context-Sensitive DPOR and Optimal DPOR with Observers, two recent algorithms that used simple notions of conditional independence, and overviews our approach to combine both notions of independence and take advantage of their synergy.

- Chapter 3 summarizes the notion of uniform conditional independence and the conditions needed to use it within DPOR algorithms.

- Chapter 4 reviews a case study of SDN networks addressing challenge iv). First, it presents the basic operations of SDN. Then, it also summarizes the actor-based concurrency model. Finally, it overviews the actor-based encoding of SDN Programs and the model checking of SDN-Actors.

- Chapter 5 first summarizes a static deadlock analysis and overviews the use of a deadlock analysis to guide the execution of testing for effective deadlock detection and its extension to symbolic execution. Moreover, it presents the tool SYCO, a systematic testing tool for concurrent objects, and its main features.

- Finally, Chapter 6 presents the conclusions of the thesis and also discusses the future work for each of the topics presented in this thesis.

- The technical details are presented in the papers which support this thesis. These papers can be found in Chapter 7.

Chapter 2

Optimal Context-Sensitive DPOR with Observers

The cornerstone of DPOR is the notion of *independence* that is used to decide whether each pair of concurrent events p and t are in a race and thus both $p \cdot t$ and $t \cdot p$ must be explored. As mentioned in Section 1.1, the classical notion used in the POR algorithms is called *unconditional independence*.

Definition 3. *(Unconditional Independence) Two events p and t are independent if for every possible state S of the program, then*

1. *if p is enabled in S and $S \xrightarrow{p} S'$, then t is enabled in S if and only if t is enabled in S'; and*

2. *if p and t are enabled in S, then there is a unique state S' such that $S \xrightarrow{t \cdot p} S'$ and $S \xrightarrow{p \cdot t} S'$.*

Let us notice that this is called unconditional independence because it must hold for every possible program state S. Any DPOR algorithm can improve its efficiency by using a more accurate notion of independence [40]. Two recent approaches – Context-Sensitive DPOR (DPOR_{cs}) [6] and Optimal-DPOR with Observers (ODPOR^{ob}) [18] – have achieved this by integrating orthogonal notions of *conditional independence* into DPOR. In this chapter, we study their combination. These notions are explained in Section 2.1 and Section 2.3, respectively. Section 2.2 introduces the reformulation of DPOR_{cs} as an extension of ODPOR. Section 2.4 explains the new notion of *context-sensitive independence modulo observability* which is illustrated using a detailed example. Finally, Section 2.5 presents our main contributions in this area and Section 2.6 overviews the related work.

2.1 Context-Sensitive DPOR

The first algorithm that has used notions of conditional independence within the state of the art DPOR algorithm is Context-Sensitive DPOR [6] (DPOR_{cs}). This algorithm exploits *context-sensitive independence*, that is, intuitively, it checks explicitly the commutativity in the current state (context) of every two events that are in race. If they commute, it

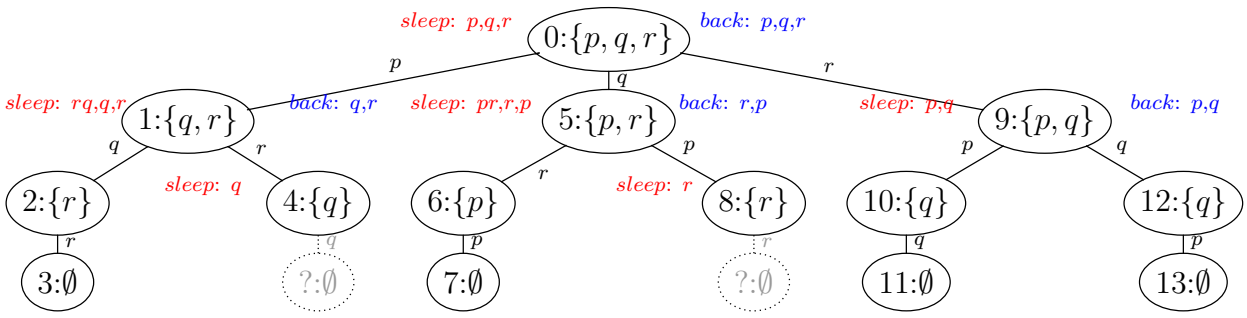


Figure 2.1: Full execution tree computed by DPOR$_{cs}$ for Example 8; dotted fragment not computed thanks to sleep sets; Arrow labels: scheduled process. Node labels right (in blue): backtrack set (only indicated for nodes with more than one process). Node labels left (in red): sleep set (only indicated for nodes with more than one sequence and sequences propagated from their parents).

prevents the exploration of the reversed race by adding additional information to the sleep set. Specifically, the main features of DPOR$_{cs}$ are:

- State equivalence check. When a race is detected, a new check is added in DPOR$_{cs}$ to detect if the sequence with the reversed race is leading to the same state as the original sequence. If the check succeeds, the appropriate backtrack set is updated as usual, but the corresponding sleep set is extended with the sequence of the reversed race to avoid its exploration.

- Sleep-set extension. Sequences of processes are more expressive than single processes, thus sleep sets are generalized to contain sequences of processes. Consequently, sequences in the sleep sets do not need to be explored because they are proven to be leading to redundant exploration thanks to the new context-sensitive independence.

Example 8. *Let us see how DPOR$_{cs}$ works for the next example:*

$$\underline{\mathsf{Int}\ x = 0;}$$

process p :	*process* q :	*process* r :
$x = 5;$	$x = 4;$	$\mathsf{assert}\ x > 0;$

Figure 2.1 shows the full exploration tree performed by DPOR$_{cs}$. Since all processes have a single event, by abuse of notation, we refer to events by their process names. Let us see the first execution. After the execution of p, q and r are in race because q is modifying variable x which is read by r. However, the state equivalence check notices that qr and rq lead to the same final state. Thus, sequence rq does not need to be explored and is added to the sleep set at state 1*, preventing the exploration of the second execution. In a similar way, r and p are in race, however they commute after q. Thus, sequence pr is added to the sleep set at state* 8*, avoiding the exploration of the fourth execution. This new check also provides an effective technique to improve the traditional over-approximation of unconditional independence. For*

instance, if process q also writes the value 5*, then p and q are considered dependent in the sequence rpq using the over-approximation since they both modify variable* x*, but the new check detects that rqp leads to the same state and thus, it does not need to be explored.*

2.2 Optimal Context-Sensitive DPOR

DPOR_{cs} was formulated in [6] over SDPOR. Thus, it did not include the extension of wake-up trees used by ODPOR to ensure optimality. We have reformulated DPOR_{cs} as an extension of Optimal DPOR, rather than of SDPOR. This yields an optimal DPOR_{cs} algorithm which we have named Optimal Context-Sensitive DPOR (ODPOR_{cs}). This reformulation has been challenging due to the fact that SDPOR (and DPOR_{cs}) performs race detection at every state. ODPOR must delay this phase until the sequence being explored is complete, since the redundancy check is accurate only if it considers the events in the entire execution to define sequence v. As a main contribution, we have proven that not all events must be taken into account by function *reversed_race* to define v and thus, we have identified the necessary events to preserve the accuracy of the redundancy check. Then, sleep sets must stop the execution not only after the race is reversed but also after these events have been also executed. Consequently, in general, sequences added to sleep sets by ODPOR_{cs} may be longer than the ones added by DPOR_{cs}. In particular, the execution tree computed by ODPOR_{cs} happens to coincide with the one in Figure 2.1.

Theorem 4 (Correctness of ODPOR_{cs})**.** *For all complete execution sequences E,* EXPLORE(ϵ) *in ODPOR_{cs} explores some execution sequence E' that either is in $[E]_{\simeq}$, or reaches an equivalent state to one in $[E]_{\simeq}$.*

2.3 Optimal DPOR with Observers

Optimal DPOR with Observers (ODPOR^{ob}) [18] has extended the traditional notion of dependency using the concept of *observability*. Intuitively, two events are *dependent modulo observability* if (1) they enable or disable each other or (2) if one of them reads a global variable and the other modifies it or (3) if both modify the same global variables and there is an observer after them which reads the new value. We use $observers(e, e', E)$ to denote the set of processes whose events observe the race between events e and e' in sequence E. There are two kinds of races: (1) if $e \precsim_E e'$ and $observers(e, e', E) = \emptyset$, then the race is produced by a read-write dependency; and (2) if $e \precsim_E e'$ and $observers(e, e', E) \neq \emptyset$, then it is caused by a write-write dependency and thus, $observers(e, e', E)$ contains the events which read the values written by e and e'. Thanks to the delay of the race detection phase, ODPOR^{ob} can decide the independence of two events by checking the existence of some observers in the whole execution.

Example 9. *Let us see how ODPOR^{ob} works for the previous example with processes p, q and r. Figure 2.2 shows the full exploration tree performed by ODPOR^{ob}. In the first execution, q and r are in a race of kind (1), because q is modifying variable* x *which is read by r, then r is added to the wake-up tree at state* 1 *to reverse the race. However, p and q are in race*

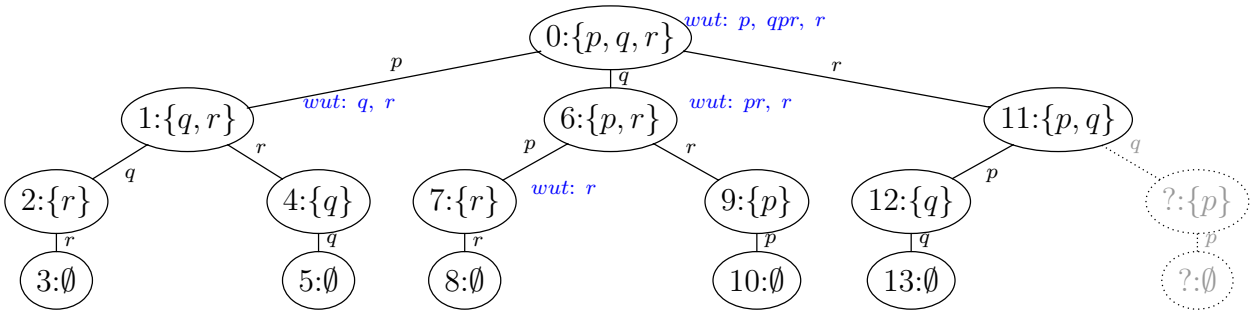


Figure 2.2: Full execution tree computed by ODPOR^{ob} for Example 9; dotted fragment not computed thanks to the notion of observability; Arrow labels: scheduled process. Node labels right (in blue): wake-up tree (only indicated for nodes with more than one sequence and sequences propagated down).

of kind (2) because both are modifying x *and there exists a process* r *in* $observers(p, q, pqr)$*, such that* r *observes the value of* x*. Then, sequence* qpr *is added to the wake-up tree at state* 0 *to ensure that the algorithm will explore a sequence where* r *observes the value written by* p *instead of* q*.*

Let us consider the fifth execution. Events p *and* q *are not in race because even though both are modifying* x*, there does not exist any observer after them which reads* x*. Therefore, in sequence* rpq*,* p *and* q *are independent modulo observability.*

2.4 Context-Sensitive DPOR with Observers

The ODPOR^{ob} and ODPOR$_{cs}$ algorithms can be easily combined to obtain a "union" algorithm which explores the intersection of each of them. We have presented a further optimization of this algorithm, called Optimal Context-Sensitive DPOR with Observers (ODPOR$^{ob}_{cs}$) that not only combines and exploits the previous powerful notions, but also takes advantage of their synergy to gain further pruning. The main point is the addition of a further and different *context-sensitive* check that compares the states *modulo observability*. E.g., in a race between p and q observed by r, instead of checking that both $p.q$ and $q.p$ are leading to the same final state, it only considers the observation in sequences $p.q.r$ and $q.p.r$ performed by the observer r. Consequently, if the second execution leads to the same observation, the sequence will be added to the appropriate sleep set to prevent its complete exploration.

Example 10. *Let us consider again the previous example. Figure 2.3 shows the full exploration tree performed by ODPOR$^{ob}_{cs}$. In Example 8, we have seen that the exploration of sequences* prq *and* qrp *are avoided thanks to the previous context-sensitive check. Moreover, sequence* rqp *is avoided thanks to the notion of observability, as we have seen in Example 9. Let us see how the exploration of* qpr *is also avoided. During the race detection phase of execution* pqr*, a race between* p *and* q *observed by* r *is detected. We can see that sequence* pq *leads to a different final state than sequence* qp*, then the previous context-sensitive check does not succeed and the sequence* qpr *would be explored. Nevertheless, considering the new context-sensitive check modulo observability,* r *observes in both* pqr *and* qpr *that* assert x > 0

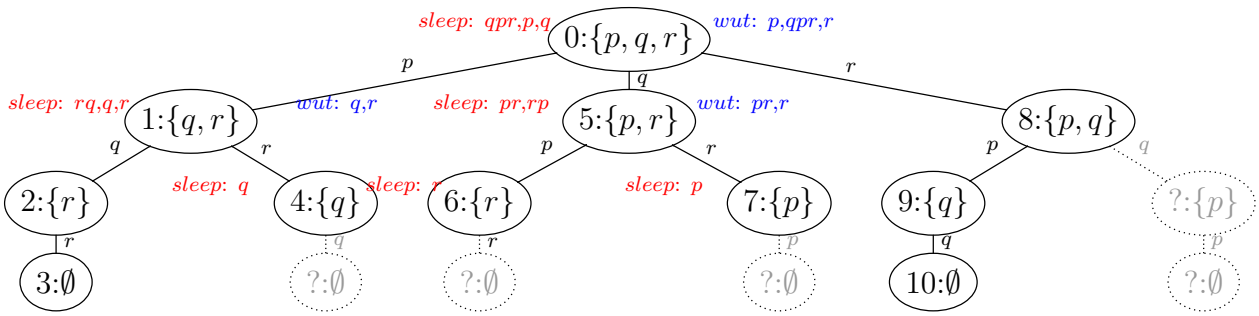


Figure 2.3: Full execution tree computed by ODPOR$^{ob}_{cs}$ for Example 10; dotted fragment not computed thanks to sleep sets; Arrow labels: scheduled process. Node labels right (in blue): wake-up tree (only indicated for nodes with more than one sequence). Node labels left (in red): sleep set (only indicated for nodes with more than one sequence and sequences propagated from their parents).

holds, and consequently, the sequence qpr can be added to the sleep set at state 0 *to be avoided by ODPOR*$^{ob}_{cs}$.

Theorem 5 (Correctness of ODPOR$^{ob}_{cs}$)**.** *For all complete execution sequences* E, EXPLORE(ϵ) *in ODPOR*$^{ob}_{cs}$ *explores some execution sequence* E' *that either is in* $[E]_{\simeq}$*, or reaches an equivalent state modulo observability to one in* $[E]_{\simeq}$.

2.5 Contributions [ISSTA'19]

This work is elaborated in detail in Paper 1 and addresses the challenges i) and iii). The major contributions achieved have been the following:

1. DPOR$_{cs}$ was formulated over Source-DPOR, but it did not include the extension of *wake-up trees* used by ODPOR to ensure optimality, and later used to handle observers. Our first contribution is the formulation of DPOR$_{cs}$ over ODPOR, which we name *Optimal Context-Sensitive DPOR* (ODPOR$_{cs}$).

2. Our second contribution is the integration of observability into ODPOR$_{cs}$. To this aim, we extend the notion of context-sensitive independence to take observability into account, hence checking equivalence for variables affected by future observers.

3. Finally, we have implemented our ODPOR$^{ob}_{cs}$ and we have performed an experimental evaluation. We have used three different sets of benchmarks, borrowed from [6], [18] and [15]. The results of ODPOR$^{ob}_{cs}$ have been compared with DPOR$_{cs}$ and ODPOR^{ob} and showed that we explore exponentially fewer sequences at least with respect to one of them in all examples considered.

2.6 Related Work

Other recent approaches have considered alternative ways of refining the detection of independence. Data-Centric DPOR [26] focuses on the read-write of variables. It defines two traces to be observationally equivalent if every read event observes the same write event in both traces. Their equivalence relation is proven to detect more traces as equivalent than the one based on Mazurkiewicz traces, which is the one used in our work and all other variants of the DPOR algorithm of [32].

Example 11. *Let us consider again the following example:*

	Int x = 0;	
process p :	*process q :*	*process r :*
x = 5;	*x = 4;*	*assert x > 0;*

According to Data-Centric DPOR, there are only three equivalence classes: (1) $\{rpq, rqp\}$ (where r reads the initialization), (2) $\{qpr, prq\}$ (where r reads the value written by p), and (3) $\{pqr, qrp\}$ (where r reads the value written by q). Instead, according to the independence modulo observability of [18], there are five different equivalence classes (see Example 9).

Let us notice that our proposal is also orthogonal to this new relation. In the above example, in case q writes x = 5;, Data-Centric DPOR must explore again three equivalence classes. However using state equivalence modulo observability, our algorithm would only need to consider two of them, namely, one where assert x > 0 holds and one where assert x > 0 does not hold.

The drawback of Data-Centric DPOR is that it is optimal only for programs with acyclic communication graphs. Recently, a new optimal DPOR algorithm [3] has been proposed based on this new notion of equivalence under sequential consistency memory semantics. This algorithm is optimal in the sense that it never explores two program executions which are observationally equivalent. Experimental results in [3] show that this new algorithm outperforms exponentially both Data-Centric DPOR and Optimal DPOR in most of the considered benchmarks. Interestingly, such notion of equivalence based on observability is still orthogonal to the notion of context-sensitive independence. It remains for future to study their combination.

Chapter 3

Constrained DPOR

As we have seen in Chapter 2, the context-sensitive DPOR algorithm is based on a notion of conditional independence. DPOR_{cs} improves the classic DPOR algorithm extending the sleep sets thanks to this notion, but it is not able to exploit it at *all* points of the algorithm where dependencies are used. We present constrained DPOR (CDPOR), an extension of the DPOR_{cs} framework which is able to use the conditional independence not only for the sleep-set extension but also for the race detection phase. This algorithm is based on the use of *independence constraints* (ICs), conditions that guarantee commutativity between two events in those program states where they are satisfied. ICs can be declared by the programmer, but importantly, we present a novel SMT-based approach to automatically synthesize ICs in a static pre-analysis. This chapter is structured as follows:

- Section 3.1 introduces the notion of conditional independence and shows that using it directly within the classic DPOR algorithm is unsound. Subsection 3.1.1 presents the *independence constraints* and Subsection 3.1.2 illustrates how to approximate the notion of uniformity within DPOR algorithms.
- Section 3.2 provides the extension from Context-Sensitive DPOR to apply independence constraints, achieving a Constrained DPOR algorithm and it is illustrated using a detailed example.
- Finally, Section 3.3 summarizes the main contributions in this area and Section 3.4 overviews the related work.

3.1 Conditional Independence within DPOR

Conditional independence consists in determining the independence of two events at a given state, instead of doing it for all possible states. The following definition captures such notion.

Definition 6. *(Conditional Independence) Two events p and t are independent at state S, then*

i1) if p is enabled in S and $S \xrightarrow{p} S'$, then t is enabled in S if and only if t is enabled in S'; and

i2) if p and t are enabled in S, then there is a unique state S' such that $S \xrightarrow{t \cdot p} S'$ and $S \xrightarrow{p \cdot t} S'$.

The next example shows that using conditional independence directly within the DPOR algorithm is unsound.

Example 12. *Let us consider now the following example, where the three processes are executed atomically. Since all processes have a single event, by abuse of notation, we refer to events by their process name.*

$$\underline{\mathit{Bool}\ b_1 = \mathit{false};\quad \mathit{Int}\ z = 5;\quad \mathit{Int}\ x = 0;}$$

process t :	*process* p :	*process* r :
$b_1 = \mathit{true};$	$x = 5;$	$\mathit{if}\ (b_1)\ z = x;$

We have the processes t, p and r enabled at the initial state with $(b_1, z, x) = (\mathit{false}, 5, 0)$. Let p and r be the first two processes to be explored by DPOR. During the race detection phase, r and p are not detected as dependent using conditional independence, because both rp and pr are leading to the same final state, that is, they commute; thus, nothing is added to the backtrack set. Now, t is explored and again, no race is detected between t and r because they are independent after p (t and r commute after p) and neither between t and p because they are independent at ϵ (in fact, they are unconditionally independent). Then, nothing is added to the backtrack set. DPOR has explored sequence prt and does not backtrack at any point, then prt is the only sequence explored. Let us notice that the final state here is $(b_1, z, x) = (\mathit{true}, 5, 5)$, however if we consider sequence trp, its final state is $(b_1, z, x) = (\mathit{true}, 0, 5)$. Thus, DPOR is not sound using conditional independence, since trp is not explored by the algorithm.

This problem was already identified by Katz and Peled [49]. Essentially, the main idea of POR is that the different linearizations of a partial order yield equivalent executions that can be obtained by swapping adjacent independent events. However, this is no longer true with conditional independence.

Example 13. *Let us see this fact in the previous example. Using conditional independence, the partial order of execution prt is empty. Then, there are six possible linearizations that represent that same equivalence class: prt, ptr, rpt, rtp, tpr and trp. The first five executions have the same final state $(b_1, z, x) = (\mathit{true}, 5, 5)$. However, as we have seen before, trp leads to $(b_1, z, x) = (\mathit{true}, 0, 5)$.*

An extra condition, called *uniformity*, is proposed in [49] to allow using conditional independence within the POR theory. Such refined conditional independence, named *uniform (conditional) independence* adds a condition $i3$) to Definition 7 to ensure that independence holds at all successor states for those events that are enabled and are *uniformly independent* with the two events whose independence is being proven.

Definition 7. *(Uniform Conditional Independence) Two events p and t are uniformly independent at state S, written unif(p, t, S), then*

i1) if p is enabled in S and $S \xrightarrow{p} S'$, then t is enabled in S if and only if t is enabled in S'; and

i2) if p and t are enabled in S, then there is a unique state S' such that $S \xrightarrow{t \cdot p} S'$ and $S \xrightarrow{p \cdot t} S'$.

i3) unif(p, t, S_r) holds for every process $r \notin \{p, t\}$ enabled in S, where S_r is defined by $S \rightarrow_r S_r$.

Example 14. *For the previous example, we can see that even though p and r commute at the initial state, the uniformity property does not hold, because (1) t and r are uniformly independent after p, (2) t and p are indeed unconditionally independent, and (3) after executing t, p and r do not commute. Thus, they are not uniformly independent and hence the partial order of sequence prt would be $\{p \rightarrow_{prt} r\}$.*

3.1.1 Independence Constraints

Instead of computing state equivalence to check condition $i2$) of conditional independence as in DPOR$_{cs}$ [6], our approach assumes precomputed *independence constraints* (ICs) for all pairs of atomic blocks in the program. An atomic block can contain just one (global statement) that affects the global state, a sequence of local statements followed by a global statement, or a block of code implemented as atomic (e.g., using locks, semaphores, etc.). ICs will be evaluated at the appropriate state to determine the independence between pairs of concurrent events executing such atomic blocks.

Definition 8 (ICs)**.** *Consider two events α and β that execute, respectively, the atomic blocks $\bar{\alpha}$ and $\bar{\beta}$. The* independence constraints $I_{\bar{\alpha},\bar{\beta}}$ *are a set of boolean expressions (constraints) on the variables accessed by α and β (including local and global variables) s.t., if some constraint C in $I_{\bar{\alpha},\bar{\beta}}$ holds at the state reached by execution sequence E, written $C(E)$, then condition (2) of conditional independence holds.*

Example 15. *Let us see the independence constraints for the previous example. This example has the following three atomic blocks, denoted with a bar over the process:*

- $\bar{t}$: $b_1 = true;$ $\quad * \ I_{\bar{t},\bar{p}} = \{true\}$
- $\bar{p}$: $x = 5;$ $\quad * \ I_{\bar{t},\bar{r}} = \{b_1,\ z == 5\}$
- $\bar{r}$: $if(b_1)\ z = x;$ $\quad * \ I_{\bar{r},\bar{p}} = \{\neg b_1,\ x == 5\}$

Regarding the independence constraints: for the first pair of blocks, we have $I_{\bar{t},\bar{p}} = \{true\}$, which means that they are (unconditionally) independent at every possible state. For $I_{\bar{t},\bar{r}} = \{b_1,\ z == 5\}$, they are independent if $b_1 == true$, because then the assignment is always executed; or if $z == 5$, since if z is already set to 5, then it does not matter if the assignment is executed. Finally, we have $I_{\bar{r},\bar{p}} = \{\neg b_1, x == 5\}$, that is, either the assignment is not executed or the value of x is already 5, thus z is always set to 5.

We have introduced a novel SMT-based approach to synthesize ICs between pairs of atomic blocks of code. Our ICs can be used within any transformation or analysis tool –beyond DPOR– which can gain accuracy or efficiency by knowing that fragments of code (conditionally) commute. The description of the inference of these ICs can be found at Paper 2.

3.1.2 Sufficient Condition for Uniformity

The challenge now is to apply the notion of uniform conditional independence, that requires to look ahead in the exploration, at all possible places in the DPOR algorithm. Condition *i1)* of Definition 7 is computed dynamically as usual during the exploration simply storing the enabling dependencies. Condition *i2)* is provided by the ICs, possibly being under-approximated in case the IC is a sufficient but not a necessary condition. Our sufficient conditions are computed by a pre-analysis, that can then be checked efficiently and dynamically during DPOR, to ensure *uniformity*. Intuitively, our sufficient condition ensures uniformity by checking that the global variables involved in the constraint C used to ensure the uniformity condition are not modified by other enabled events at the state. This check is performed statically, thus, unavoidably it can have a loss of precision that leads to considering two event dependent even though they are indeed uniformly independent. The formal definition of this condition can be found in Paper 2.

Example 16. *Let us consider again the exploration of prt as in Example 14, we have seen that events p and r are not uniformly independent. Now, let us check unif*(p, r, S_ϵ) *(where* S_ϵ *is the initial state):* $I_{\bar{r},\bar{p}} = \{\neg\ \mathsf{b}_1, \mathsf{x} == 5\}$. *Moreover,* $\neg\mathsf{b}_1$ *is* true *at the initial state and then, condition 12) is satisfied. However, there exists a process (which is t) that may modify global variable* b_1 *and, consequently, if it is executed before p and r might make the condition* $\neg\mathsf{b}_1$ *becomes* false. *Thus, condition i*3) *is not satisfied, and p and r are considered as dependent.*

Let us consider now the exploration of prt starting from the initial state $(\mathsf{b}_1, \mathsf{z}, \mathsf{x}) = (\mathsf{false}, 5, 5)$. *In this case, unif*$(p, r, S_\epsilon)$ *is satisfied, since* x *is equals to 5 at the initial state, then condition i*2) *is satisfied, and there does not exist a process that may modify global variable* x. *Hence, condition i*3) *is also satisfied, and p and r are considered as uniformly independent.*

3.2 The Constrained DPOR Algorithm

Our goal now is to introduce a DPOR algorithm which (1) provides techniques to both infer and soundly check uniform independence, and (2) is able to exploit them at all points of the DPOR algorithm where dependencies are used. The detailed algorithm can be found at Paper 2. The main points are the following:

- *Backtrack-set reduction.* The race detection is strengthened by using the new sufficient condition for uniformity. Thus, only races whose events are not uniformly independent must be reversed by CDPOR. This reduces the number of processes added to the backtrack set, hence improving the scalability of DPOR with the corresponding potential exploration reduction.

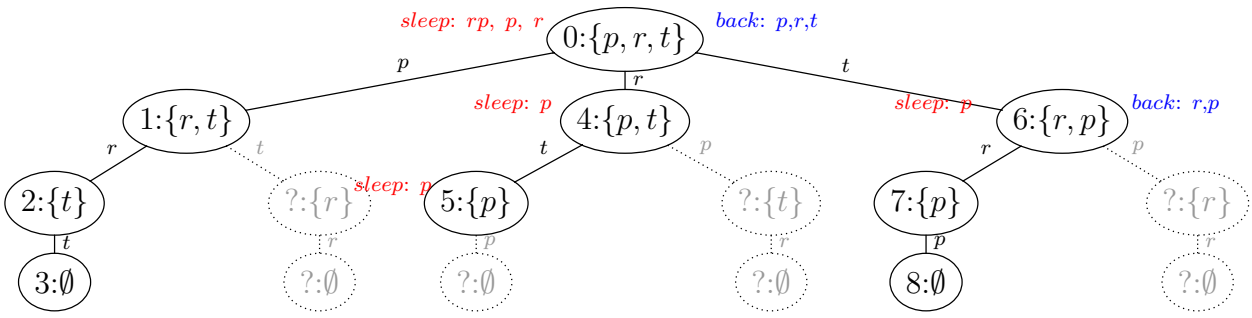


Figure 3.1: Full execution tree computed by CDPOR for Example 17; Arrow labels: scheduled process. Node labels right (in blue): backtrack set (only indicated for nodes with more than one process). Node labels left (in red): sleep set (only indicated for nodes with more than one sequence and sequences propagated from their parents).

- *Sleep-set extension.* Function *propagate(E,p)* is redefined in order to propagate down processes in $sleep(E)$ which execute events that are uniformly independent with p. Consequently, this redefinition allows potentially propagating more processes than using unconditional independence avoiding the exploration of more redundancies.
- *New state equivalence check.* We can avoid computing the alternative states to check context-sensitive independence as in DPOR_{cs}, thanks to the use of the independence constraints.

Example 17. *Let us consider again Example 12:*

$$\textit{Bool } b_1 = \textit{false}; \quad \textit{Int } z = 5; \quad \textit{Int } x = 0;$$

process t :	*process p* :	*process r* :
$b_1 = \textit{true};$	$x = 5;$	*if* (b_1) $z = x;$

Figure 3.1 shows the full exploration tree performed by CDPOR. The first execution sequence is prt. We can see that after executing p, r and t are uniformly independent, thus they are not in race and the exploration of the second execution is avoided. However, DPOR algorithms using the classical approximation must explore it even though both lead to the same final program state.

As we have seen in Example 16, p and r are dependent because the sufficient condition for uniformity does not hold, thus r must be explored from state 0*. Let us observe now the second and third points: p and r are in race because* $unif(p, r, S_\epsilon)$ *does not hold, where* S_ϵ *is the initial state. However, they are commutative, thus pr and rp are leading to the same final state. To prevent this redundancy, sequence rp is added to the sleep set at state* 0*. This new information prevents the exploration of the fourth execution. However, t can still be executed at state* 4*. Once it is executed, a race between r and t is detected to be reversed. Consequently, the fifth execution is explored.*

Theorem 9 (Correctness of CDPOR)**.** *For all complete execution sequences* E, EXPLORE(ϵ) *in CDPOR explores a complete execution sequence* E' *that reaches the same final state as* E*.*

3.3 Contributions [CAV'18]

This work is further studied in detail in Paper 2 which addresses the challenges ii.a), ii.b), ii.c) and iii). The major contributions achieved have been the following:

1. We have introduced sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework.

2. We have extended the state-of-the-art DPOR algorithm with new forms of pruning (through expanding sleep sets and reducing backtrack sets).

3. We have presented an SMT-based approach to automatically synthesize ICs for *atomic blocks*, whose applicability goes beyond the DPOR context.

4. We have proven the soundness of the overall approach.

5. Finally, we have experimentally shown the exponential gains achieved by CDPOR on some typical concurrency benchmarks used in [6]. The experimental results of CDPOR are compared against DPOR$_{cs}$ and SDPOR. For many of the benchmarks, SDPOR and DPOR$_{cs}$ time out, while CDPOR can still handle them efficiently. Moreover, the time to infer the independence constraints is negligible for most of the benchmarks.

3.4 Related Work

The work in [64, 48] generated for the first time *independence constraints* (ICs) for processes with a single instruction following some predefined patterns. This is a problem strictly simpler than our inference of ICs both in the type of IC generated (restricted to the patterns) and on the single-instruction blocks that they consider. While [41] also derives constraints directly fulfilling uniformity, we use an over-approximation to check the uniformity property. Furthermore, our approach using an AllSAT SMT solver is different from the CEGAR approach in [21]. The ICs are used in [64, 48] for SMT-based bounded model checking, an approach to model checking fundamentally different from our stateless model checking setting. As a consequence, ICs are used in a different way, in our case with no bounds on the number of processes, nor derivation lengths, but requiring a uniformity condition on independence in order to ensure soundness.

Maximal causality reduction [44] is technically quite different from CDPOR as it integrates SMT solving within the dynamic algorithm. Bounded model checking is then performed, by encoding the whole reachability problem for a bound K as an SMT formula, such that the error location is reachable in at most K steps if and only if the SMT formula is satisfiable. In this setting, ICs are used to add information to the encoding that will help the off-the-shelf SMT solver to avoid analyzing those redundant derivations with respect to independence. Therefore, both model checking approaches are radically different since we apply stateless model checking, and as a consequence ICs are used in a different way.

In [64], a reduction method that exploits conditional independence is proposed. The purpose in that work is different from ours. They aim at using DPOR *symbolically* and, as the values of variables will be unknown in general and it is necessary to be able to handle

ICs that are valid for any possible instantiation of variables. The ICs they infer are for single instructions, a problem that is strictly simpler than our inference. Besides, their method ensures that no redundancy is explored for systems with *two* threads.

Chapter 4

Application: Software-Defined Networks

Software-Defined Networks (SDN) is a relatively recent networking paradigm which is now widely used in industry, with many companies—such as Google and Facebook—using SDN to control their backbone networks and data-centers. The core principle in SDN is the separation of control and data planes—there is a centralized *controller* which operates a collection of distributed interconnected switches.

Network verification has become increasingly popular since SDN was introduced because in this new paradigm the amount of detailed information available about network events is rich enough and can be centrally gathered to check for properties, both statically and dynamically, of the network behavior. Moreover, the controller itself is a program which can be analyzed and verified before deployment. However, the distributed and concurrent nature of network behavior makes both the programming and verification tasks challenging. Some of the bugs that can be found in existing (programmable) networks are reminiscent of faults that have appeared in distributed and concurrent systems, and which have inspired much research in the verification and formal methods communities.

This chapter is organized as follows:

- Section 4.1 details how the different components of an SDN network work and the different kind of exchanged messages.
- Section 4.2 introduces the actor-based concurrency model and the language used to model the SDN programs.
- Section 4.3 describes the encoding using the actor model of the main components in SDN networks (controller, switches, hosts), and one of the most important features: OpenFlow barriers.
- Section 4.4 summarizes the main challenges of using model checking for SDN networks and the advantages of applying Constrained DPOR to several case studies.
- Finally, the main contributions of the thesis in this subject and the related work are presented in Sections 4.5 and 4.6, respectively.

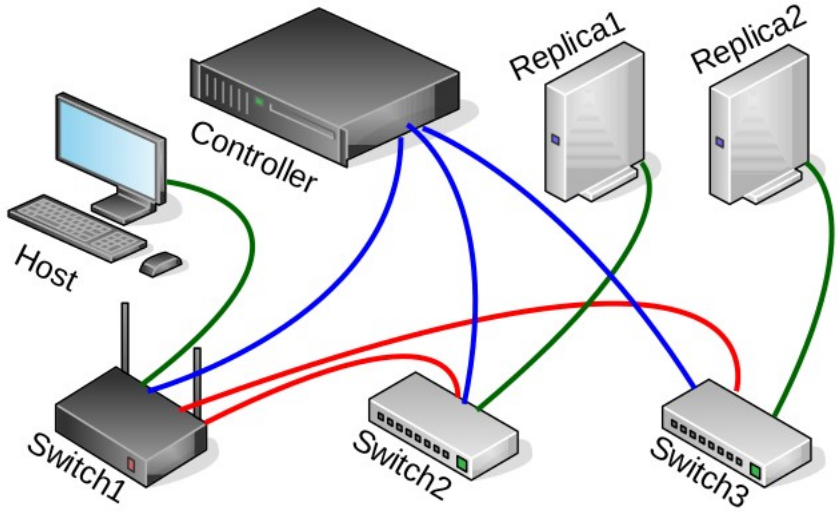


Figure 4.1: Structure of the SDN load-balancer in Example 18.

4.1 Components of Software-Defined Networks

SDN is a networking architecture where a central software *controller* operates on a collection of distributed interconnected switches. *Hosts* communicate with each other by sending packets to their *switches*. Each switch has a *flow table*, which is a collection of guarded forwarding rules to determine the route of incoming packets. When a switch receives a packet, it checks if its own flow table contains a forwarding rule to determine the route of the packet. If it does, the packet is sent to the next switch or to the final host. Otherwise, it sends a *message* to the controller via a dedicated link, in order to receive information about the destination of the packet, which is buffered until instructions from the controller arrive. Depending on its policy, the controller answers by sending several messages to the switches involved in the path of the packet to the final host and dynamically updates switches' policies depending on the observed flow of packets, which is a simple but powerful way to react to unexpected events in the network. Furthermore, in order to avoid synchronization problems, the controller may also send a special kind of message, called *barriers* [56], designed to force a switch to handle all previous control messages.

Example 18. *Figure 4.1 shows the structure of a simple load-balancer implemented in SDN and how potential bugs can easily arise due to the concurrent behavior and asynchrony of message passing. This example has been taken from [31]. Suppose we want to balance the traffic to a server by using two replicas* Replica1 *and* Replica2 *to which the controller alternates the traffic in a round-robin fashion. The host wants to communicate with the server by using* Switch1*. Even in this simple network, an incorrect implementation of the controller can lead to serious problems. Let us suppose an implementation with a naive controller, which simply instructs switches to forward packets along the shortest path to the chosen replica. This implementation ignores the potential concurrency in actions taken by switches and controller, leading to a forwarding loop between* Switch1 *and* Switch2*, that is, a situation where a packet*

is sent from Switch1 to Switch2 and vice-versa in an endless loop.

Once the host sends a packet to Switch1, Switch1 queries the route to the controller. In the first round, Replica1 is chosen. Then, the controller sends rules to Switch1 and Switch2 telling them to forward the packet to Switch2 and Replica1, respectively. Let us suppose that Switch1 forwards the packet to Switch2 before the end of the first round, i.e., before the previous rule is installed on Switch2, and this causes Switch2 to query the controller. When receiving two messages of unknown packets (first from Switch1 and then from Switch2) the controller assumes that the packet at Switch2 is a second unknown packet and therefore triggers the second round of the round-robin protocol in which the controller chooses Replica2 instead of Replica1 to balance the load. Thus, it sends instructions to install rules on Switch2, Switch1 and Switch3 to forward the packet to Switch1, Switch3 and Replica2, respectively. When the controller rules arrive at Switch1, it will have two contradictory instructions, telling it to forward the packet either to Switch2 or to Switch3. In the former, a forwarding loop between Switch1 and Switch2 happens. This issue can be avoided if the implementation uses barriers—the controller will then guarantee that Switch2 could not request for a new forwarding rule before all the previous control messages sent to it were received and processed.

Let us notice here that the control messages between the controller and the switches can be processed in arbitrary order and hence, in principle, all possible orderings must be considered in order to detect bugs of the implementation of the network. The combinatorial explosion problem in the context of SDN programs is exacerbated because all network components (switches, hosts, controllers) are distributed nodes that run in parallel and send messages and packets to each other. DPOR techniques can be used in order to alleviate this problem by identifying redundant executions. However, using unconditional independence in this context is not sufficient because every time a switch receives a message, it modifies its flow table, thus every pair of messages are considered as dependent even though many of them are not. In this framework, the use of conditional independence is vital: it allows us to declare that two accesses to the flow table are independent if they are not accessing the same entry of the flow table.

4.2 Actor-based Concurrency Model

In the actor-based paradigm [5, 42], each actor represents a processor with a memory, a procedure stack, and an unordered buffer of pending tasks. Initially, all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. When a task completes, its processor becomes idle again, chooses the next pending task, and so on. Besides reading and writing its own memory, each task can post tasks to the buffers of any processor, including its own, and synchronize with the termination of tasks. Instruction `o ! m(p)` means that the current task posts a new task `m` in the distributed component `o` with parameters `p`. The synchronization is performed by means of *future variables*. A future variable acts as a proxy for a result that remains unknown until the computation of its value by the corresponding task is completed. A future variable is bound to a task at the moment it is spawned. Instruction `Fut<Int> f = o ! m(p)` means that the future variable `f` receives an integer value returned by the execution of the spawned task `m`. When the task finishes, its future variable becomes ready, and the result can be retrieved. There are two kinds of

synchronization mechanisms: the instruction f.get is blocking because the task executing does not release the processor until the future variable f becomes ready. On the other hand, the instruction await f? is non-blocking, that is, if the future variable f is not completed, it releases the processor and other tasks of the actor can be executed. The number of actors does not need to be known a priori, they can be dynamically created using the instruction new. In order to take advantage of our DPOR-based tools, we model the SDN networks using the actor-based language ABS [45].

4.3 SDN-Actors: an Actor Based Encoding of SDN Programs

As already mentioned, verification of Software-Defined Networks is very challenging because of the combinatorial explosion on the number of situations that must be considered to detect synchronization problems, deadlocks, livelocks, and other concurrency bugs. Partial Order Reduction techniques are essential to reduce the state space to be explored. To use our tools to evaluate several SDN case studies, we need to model these networks using the actor-based paradigm. In Paper 3, we have presented the concept of *SDN-Actor*: an actor based encoding of SDN programs. The first step is to describe the creation and initialization of the actors according to the network topology:

1. A controller actor is created.
2. A switch actor is created for each switch in the topology with, at least, three fields in its memory: its identifier, the flow table, and a reference to the controller.
3. Similarly, a host actor is created for each host in the topology with, at least, two fields in its memory: its identifier and a reference to the switch actor that it is connected to.
4. Once every switch and host actor are created, the controller actor must learn the link relations among them to be able to update the switches' flow tables on demand.

The second step is to provide the encoding of the operations and communications for switch and host actors.

- Each host actor must be able to perform two different operations: (1) operation sendIn to send a packet to the switch it is connected to and (2) operation hostHandlePacket to receive a packet from the switch.
- Each switch actor must be able to perform three different operations: (1) operation switchHandlePacket to handle a packet which is received from a host. If there is an entry for the packet in the flow table, it asynchronously makes the corresponding action; otherwise, it sends a controlHandleMessage request to the controller to receive instructions to update the flow table and stores the packet in a buffer. (2) Operation sendOut to handle a message from the controller with the packet identifier to be removed from the buffer and the corresponding packet to be sent according to the information in the flow table. Finally, (3) operation switchHandleMessage to handle a message from the controller with an instruction to update the flow table.

Finally, the third step is to encode the controller actor.

- The controller actor must be able to perform, at least, two different operations: (1) operation controlHandleMessage to handle a message from the switch with a packet identifier for which it requests instructions. The current network policy is used (depending on the type of controller) to obtain a list of switch identifiers and their corresponding updating instructions to be sent by means of switchHandleMessage operations. (2) Operation addConfig with the references to switches and hosts in order to set up the initial network topology.

Altogether, our encoding provides an actor-based semantics foundation of SDN networks that follow the OpenFlow specification [56]. Proofs of soundness for the encoding can be found in Paper 3.

Furthermore, we have also encoded one of the most challenging aspects of Software-Defined Networks: *barrier messages* [56]. These messages have been designed to force a switch to handle all previous control messages before processing any new message, and thus avoid synchronization problems. Intuitively, the proposed solution consists in the controller not sending further messages to any switch on which a barrier has been activated, until this switch acknowledges that all previous control messages have been already processed. Acknowledgment messages are modeled by means of future variables. For each message sent to a switch, the controller stores the future variable related to the message. Thus, when a barrier is activated on such switch, the controller needs to ensure that all the future variables related to such switch are ready.

4.4 DPOR-based Model Checking of SDN-Actors

The state space problem for model checking SDN programs is exacerbated because all network components (switches, hosts, controllers) are distributed nodes that run in parallel and whose concurrent tasks can interact. As we have seen in Example 18, a controller message sent from a switch can change the state of another switch, and affect the route of an incoming packet. Thus, in principle, a model checker needs to explore all possible reorderings of dependent tasks, leading to a huge number of possible executions even for networks with few nodes and few packets. Besides, the space is unbounded because hosts may generate unboundedly many packets that could be simultaneously traversing the network. One of the most successful approaches to handle unbounded input is to impose a bound k on the number of packets of each type (as e.g. in [25]). Consequently, the search space is exhaustively explored for the considered bound, possibly missing bugs that only show up when more packets are considered.

When DPOR is applied to actor systems, there are inherent reductions [62] because: (i) we can atomically execute each task (without re-orderings) until a return or an await instruction are found since concurrency is non-preemptive and the active task cannot be interrupted. This avoids having to consider the reorderings at the level of instructions (as one must do in thread-based concurrency) and allows us to work at the level of tasks. (ii) Besides that, two tasks can have a dependency only if they belong to the same actor. This is because only the actor itself can modify its private memory.

When two tasks that belong to the same actor are found, in the context of DPOR techniques, unconditional independence is commonly over-approximated by requiring that

the fields of the actor that are accessed by one task are not modified by the other. In our model, all tasks posted on a given switch access its flow table, namely sendOut and switchHandlePacket read it and switchHandleMessage writes it. Thus, in principle, any task executing switchHandleMessage is considered dependent on the other two. When multiple packets are traversing the network it is usually the case that the different packets access distinct entries in the flow table. This results in the inaccurate detection of many dependencies hence producing redundant executions. Using Constrained DPOR, we alleviate this state space explosion:

1. *Entry-level independence.* We adopt a finer-grained notion of *entry-level independence* for which an access to entry i is independent from an access to j if $i \neq j$.

2. *Independence constraints.* Even when two tasks t and p access the same entry, we can pre-generate independence constraints for each pair of tasks to guarantee commutativity. Then, Constrained DPOR can use these constraints to avoid redundant explorations. For instance, executing two consecutive switchHandleMessage on the same entry might lead to the same state if the flow table contains duplicate entries.

4.5 Contributions

The main contributions achieved in our actor-based verification of SDN networks are:

1. SDN-Actors: An encoding of all basic components of an SDN network (switches, hosts, controller) into the actor-based language ABS [45] and a soundness proof of our encoding using the semantics of SDN networks.

2. Barriers: One of the most challenging aspects to encode are the OpenFlow *barrier* messages, special instructions that the controller can use to force switches to execute all their queued tasks. We provide an implementation of barriers using conditional synchronization and a soundness result.

3. Model checker: A model checker for our SDN models built on top of the SYCO tool [11] that incorporates different DPOR algorithms and visualization tools to view the exploration and execution diagram.

4. Case studies: Several case studies of SDN and properties to illustrate the versatility and potential of the approach. We were able to find bugs related to programming errors in the controller, forwarding loops, and violation of safety policies, and scale to larger networks than related techniques.

4.6 Related Work

Many static and dynamic techniques for verification closely related to our approach have been developed in the last years. Using static approaches, one has the main advantage that, when the property can be proven, it is ensured for any possible execution, while using dynamic analysis only guarantees the property for the considered inputs. As a counterpart, to cover all

possible behaviors, static analysis needs to perform abstraction, which can give a don't-know answer, and, possibly, false positives. When the behavior of the programs depends on the interaction of events, a static analyzer takes into account all possible interleavings and needs to perform abstraction that can lead to loss of precision. For properties such as congestion, it could give an overly pessimistic result (that would very unlikely to happen in practice). In [20], the work on Horn-based verification is lifted to the SDN programming paradigm, but excluding barriers. Using this kind of verification, one can prove safety invariants on the program. Using our framework, we can furthermore check liveness invariants (e.g., loop detection) by inspecting the traces computed by the model checker. Therefore, the suitability of dynamic or static approaches also depends on the property that one wants to prove, and the two approaches usually complement each other. In the line of work on the NetKat language [34, 17, 22], static algebraic techniques are used to prove properties of SDN programs. NetKat does not include primitives for concurrency and has a significantly higher level of abstraction. Therefore capturing features and scenarios we are interested in would be difficult. In [57], a particular type of attacks in the context of SDN networks has been modeled in Maude using the so-called hierarchically structured composite actor systems described in [30]. This work does not provide a general model for SDN networks and, besides, barriers are not considered. On the other hand, it applies a statistical model checker, which requires to have a given scheduler for the messages. Such scheduler determines the exact order in which messages are handled while our framework captures all possible behaviors. Hence, both their aim and their SDN model are radically different from ours.

Comparison of DPOR reductions with related work. There exist other model checkers for SDN programs that have used DPOR-based algorithms before [25, 53].

When compared with [25], there are two main differences: (1) they use state systematic testing which requires saving all states, while we have adopted a stateless algorithm that has proven to be much more scalable, (2) they consider reorderings among all possible events, what can lead to a huge search tree. The use of DPOR in their case, as their experiments show, achieves a 20% reduction of the search space. The reason for this modest reduction might be that it does not take advantage of the inherent independence of the code executed by the distributed elements of the network (switches, host, clients) nor to the fact that barriers allow removing dependencies, as our actor-based SDN model does. In contrast, the use of actors together with the Constrained DPOR algorithm in our case, instantiated with our notion of independence, allows us to avoid those reorderings that will not lead to a different network configuration. Similarly to their work, our algorithm can run symbolically.

In Kuai [53], a number of optimizations are defined to take advantage of these aspects. Such optimizations must be (1) identified and formalized in the semantics, (2) proven correct and, (3) implemented in the model checker. Instead, due to our formalization using actors, the optimizations are already implicit in the model and handled by the model checker without requiring any extension. Our approach could be adapted to apply abstractions that bound the size of buffers [53] and to consider environment messages [61], without requiring any conceptual change in the framework.

Finally, the approach of [31, 50] is fundamentally different from ours because it is based on analyzing dynamically given snapshots of the network from real executions. Instead, our approach tries to find programming errors by inspecting only the SDN program and

considering all possible execution traces, thus enabling verification at system design time.

Chapter 5

Combining Static Analysis and Testing for Deadlock Detection

One of the most common errors in concurrent programming is a *deadlock* situation. Consequently, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. A deadlock happens when a concurrent program reaches a state in which one or more tasks are waiting for each other's termination and none of them can make any progress. In this chapter, we consider the actor-based paradigm explained in Section 4.2. Let us see a simple example to illustrate what a deadlock is.

```
class A {Int B() {Fut<Int> f = this!C();
                  f.get;}
         Int C() {return 0;}
}
```

An actor executing task B spawns task C to execute in the current actor (hence uses this to refer to the actor itself) and awaits for its termination. However, instruction get does not release the processor, thus task C will never be executed and then B cannot be completed. If there are other actors awaiting for the termination of a task in the blocked actor, they will also get blocked.

The rest of this chapter is structured as follows:

- Section 5.1 introduces the state-of-the-art work in deadlock analysis and a simple example of a communication protocol between a database and several workers that will be used to illustrate the main ideas behind this chapter.
- Section 5.2 presents Deadlock-guided testing with a detailed example and our main contribution in this line of work.
- Section 5.3 summarizes the main ideas behind the symbolic execution technique.
- Section 5.4 extends the ideas explained in Section 5.2 to the context of symbolic execution. In particular, it shows the main contributions to generate initial contexts to be used during symbolic execution.
- Section 5.5 presents our tool SYCO: a systematic testing tool for actor programs.
- Finally, Section 5.6 overviews the related work in this area.

5.1 Deadlock Analysis

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. Static analysis evaluates a program by examining its code but without executing it. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a potential deadlock is detected, state-of-the-art analysis tools [33, 36, 37] provide little (and often no) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and actors), it is essential to know the task interleavings that have occurred and the actors involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem. Let us see now an example that will be used to illustrate the ideas of this chapter.

Example 19. *The code in Figure 5.1 simulates a simple communication protocol among a database and n workers. Our implementation has three classes, a* Main *class which includes the* main *method, and classes* Worker *and* DB *implementing the workers and the database, respectively. The* main *method just calls method* simulate *with the number of workers to create in its parameter (in this case* 1*). Method* simulate *creates the database and the n workers, and invokes methods* register *and* work *on each of them, respectively. The* work *method of a worker simply accesses the database (invoking asynchronously method* getData*) and then blocks until it gets the result, which is assigned to its* data *field. The* register *method of the database registers the provided worker reference adding it to its* clients' *list field. We use* add *and* contains *to refer to predefined functions for list. In case* checkOn *is true, before adding the worker, it makes sure that the worker is online. This is done by invoking asynchronously method* ping *with a concrete value and blocking until it gets the result with the same value. Method* getData *of the database returns its* data *field if the caller worker is registered, otherwise, it returns* **null**.

The deadlock analysis of [33] provides a set of abstractions of potential *deadlock cycles*. If this set is empty, then the program is deadlock-free. Otherwise, there may be a deadlock situation. Each abstract deadlock cycle has as set of nodes, the task and actor abstractions involved in the deadlock and the set of arrows must be interpreted as "the origin node is waiting for the ending node due to a synchronization primitive".

Example 20. *In the previous example, there are three abstract actors, m, the main actor which is not relevant for this example, W and DB, corresponding to the set of* worker *actors and the* database *actors, respectively. Furthermore, there are four relevant abstract tasks reg (method* register*), getD (method* getData*), work (method* work*) and ping (method* ping*). If the program in Figure 5.1 is analyzed by the deadlock analysis of [33], then the next abstract deadlock cycle is inferred:*

$$DB \xrightarrow{register} ping \xrightarrow{ping} W \xrightarrow{work} getD \xrightarrow{getD} DB$$

It can be interpreted as follows: a database DB can be blocked in an instance of method register *waiting for the execution of an instance of method* ping *by a worker W. At the same*

```
1 class Main{
2   main(){
3     this!simulate(1);
4     return 0;
5   }
6   simulate(int n){
7     DB db = new DB();
8     while (n > 0){
9       Worker w = new Worker();
10      db!register(w);
11      w!work(db);
12      n = n−1;
13    }
14    return 0;
15  }
16 }// end of class Main
17
18 class DB{
19   Data data = ...;
20   List<Worker> clients;// Empty list
21   Bool checkOn = true;
22   int register(Worker w){
23     if (checkOn){
24       Fut⟨int⟩ f = w!ping(5);
25       if (f.get == 5) add(clients,w);
26     } else add(clients,w);
27     return 0;
28   }
29   Data getData(Worker w){
30     if (contains(w,clients)) return data;
31     else return null;
32   }
33 }// end of class DB
34 class Worker{
35   Data data;
36   int work(DB db){
37     Fut⟨Data⟩ f = db!getData(this);
38     data = f.get;
39     return 0;
40   }
41   int ping(int n){return n;}
42 }// end of class Worker
```

Figure 5.1: Working example: Communication protocol among a DB and n workers.

time, this worker may be blocked in an instance of method work waiting for DB to complete a task getData. If a situation like this deadlock cycle happens during the execution of the program, a deadlock is found.

5.2 Deadlock-Guided Testing [iFM'16]

In order to guide the systematic execution towards paths leading to deadlock, we use these abstract deadlock cycles to generate *deadlock-cycle constraints*. These constraints must hold in all states of executions leading to the deadlock cycle. As soon as one of them is not satisfied, the exploration of this derivation is discarded because it is guaranteed not to contain any deadlock.

Deadlock-guided testing (DGT) tries all possible orderings and checks the satisfiability of these constraints every time a new state is explored. Deadlock-cycle constraints require that the current state of the execution always contains a *representative* of the deadlock cycle, that is, an instance for each of the abstract nodes of the detected cycle. As soon as one of the tasks is completely executed instead of being blocked in a synchronization point, such a task cannot be a representative. DGT checks if there can be a new representative along with the execution. In such a case, the exploration continues normally. Otherwise, the execution is stopped and other orderings must be considered.

Example 21. *Let us consider again the previous example. DGT first executes tasks* main *and* simulate*. The resulting state contains two actors, a database and a worker, with a unique task:* register *and* work*, respectively. Let us suppose that DGT chooses the database to execute* register*, then a new state is reached where the worker can execute either* work *or* ping*. If the latter is executed, DGT detects that ping has been completely executed and there cannot be another representative of the abstract task* ping*, thus it stops the exploration of such execution, because it is deadlock-free. If* register *is executed, then it detects the deadlock situation as soon as possible and it continues with the execution.*

Briefly, the main contributions of Paper 5 that achieve the goal v) of this thesis are:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made and the status of tasks.

2. We provide a formal characterization of a *deadlock state* which can be checked along with the execution and allows us to early detect deadlocks.

3. We present a new methodology to detect deadlocks which combines testing and static analysis as follows: the deadlock cycles inferred by static analysis are used to guide the testing process towards paths that might lead to a deadlock cycle while discarding deadlock-free paths. Our method can be used both for static and dynamic testing.

4. We introduce several deadlock-based testing criteria, namely to find the first deadlock trace, a representative trace for each deadlock cycle, or all deadlock traces.

5. We implement our methodology in the SYCO testing system and perform a thorough experimental evaluation of some classical examples. These experiments support our claim that testing complements deadlock analysis. We have used benchmarks that include classical concurrency patterns containing deadlocks and deadlock-free versions of some of them. We have compared the systematic testing with deadlock-guided testing and the results show that our methodology complements deadlock analysis, finding deadlock executions and discarding false positives, with a significant reduction in the number of states explored.

5.3 Initial Contexts by Symbolic Executions

Symbolic execution [19, 24, 7] is arguably the most widely used enabling technique for white-box testing. It consists in executing a program with the contents of its input arguments being symbolic variables rather than concrete values. During the course of symbolic execution, the values of the program's variables are represented as symbolic expressions over the input symbolic values and a *path condition* is maintained. Such a path condition is updated whenever a branch instruction is executed. The satisfiability of each of these branches is checked and symbolic execution stops exploring any path whose path condition becomes unsatisfiable. The result of applying symbolic execution on a program is the set of possible final states and the path conditions over the input variables that must be satisfied to reach each of them.

In the context of concurrent programs, an *initial context* is the set of initial actors and initial tasks that must be executed during symbolic execution. Even though symbolic execution is able to automatically generate different initial contexts and expose bugs and concurrency problems that would introduce a new combinatorial explosion on the initial contexts that must be explored. Then, a maximum number of initial actors and initial tasks considered must be set up to guarantee termination of symbolic execution.

5.4 Generating Deadlock Contexts for Symbolic Execution [LOPSTR'17]

One of the main challenges for symbolic execution is to automatically and systematically generate distributed contexts that give evidence of deadlock situations and then, help programmers to understand the causes of the bug. Therefore, contexts must contain not only the set of actors involved but also their interfering tasks getting blocked in the deadlock. The generation of *relevant* contexts is challenging because it must avoid (1) useless contexts that do not expose any deadlock and (2) redundant contexts that do not provide programmers with more information. Therefore, it is crucial to generate the *minimal* set of initial contexts that contains only one representative of equivalent contexts.

Example 22. *It is easy to see for the previous example which are the tasks that must necessarily be in an initial context leading to deadlock even without an automatic procedure. The initial context must contain at least two actors: a* database *and a* worker *with at least, the tasks* register *and* work*, respectively. Regarding the parameters of these tasks, they remain as variables that will be instantiated during symbolic execution. The possible aliasing between the parameter* w *of task* register *and the actor* worker *will make symbolic execution try both* $w = worker$ *and* $w \neq worker$ *during the exploration. The same situation will happen for the parameter* db *of task* work *and the actor* database*. Then, symbolic execution will be able to find a deadlock when* $w = worker$ *and* $db = database$*, and the remaining possibilities will be leading to deadlock-free executions. Consequently, the initial context must contain these two actors with task* work *and* register*, respectively.*

Let us suppose that field checkOn *is initialized to* false *and there is another method of class* DB *(called* makesTrue*) which makes such field* true*. In this situation, if we consider again the previous initial context, symbolic execution does not detect any deadlock situation using such context: during the execution of* register*, field* checkOn *is* false *and thus, no call* ping *to* worker *is performed, and there will not be any deadlock.*

In order to solve this problem, initial contexts must contain not only tasks involved in the execution but also those methods that modify those fields involved in conditions that may affect the execution of the asynchronous calls or the synchronization executions of the deadlock. For the previous example, method makesTrue must be also part of the initial state.

The major contributions of our work that address the challenge vi) of this thesis are introduced in Paper 6 and can be summarized as follows:

- We introduce the concept of a *minimal* set of initial contexts and extend a static testing framework to automatically and systematically generate them.

- We propose a new algorithm to infer which tasks produce or may produce conflicting interactions from the deadlock cycles. This information is useful to discard initial states or contexts that cannot lead to deadlock.
- We present a deadlock-guided approach to effectively generate initial contexts for deadlock detection and prove its soundness.
- We have implemented our proposal within the SYCO system [11] and performed an experimental evaluation to show its efficiency and effectiveness. These experiments show the effectiveness of our approach to generate initial contexts for deadlock detection compared to full systematic generation. Additionally, these results demonstrate the potential of the technique when it is applied within our deadlock detection framework.

5.5 SYCO: Systematic Testing for Concurrent Objects [CC'16]

We have developed a prototype tool called SYCO, a dynamic/static testing tool for the actor-based language ABS. It includes all the POR techniques described in Chapters 1, 2 and 3 to detect and avoid redundant explorations and also it includes the techniques proposed in this chapter. This tool is available for online use through a user-friendly web interface at `http://costa.fdi.ucm.es/syco`, where the code of all the benchmarks used in the experimental evaluation of this thesis can be found.

SYCO is a systematic tester for `ABS` concurrent objects. Figure 5.2 shows its main architecture. Boxes with dash lines are internal components of SYCO whereas boxes with regular lines are external components. The user interacts with SYCO through its web interface which has been built using the *EasyInterface* [29] framework. The SYCO engine receives an `ABS` program and a selection of parameters. The `ABS` *compiler* compiles the program into an abstract-syntax-tree (AST) which is then transformed into the SYCO intermediate representation (IR). The DPOR *engine* carries out the actual systematic testing process using any of the algorithms in Chapters 1, 2 and 3. The *output manager* then generates the output in the format which is required by EasyInterface, including an XML file containing all the EasyInterface commands and actions and SVG diagrams. In case deadlock-guided testing is applied, the DECO *deadlock analyzer* [33] is invoked, which returns a set of potential deadlock cycles that are then fed to the DPOR engine to guide the testing process (discarding non-deadlock executions).

5.6 Related Work

Since our method uses in conjunction static and dynamic analyses, and the individual methods can be used for multiple purposes, we need to relate it to a wide spectrum of existing techniques that we classify as follows.

Deadlock Analysis. There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs

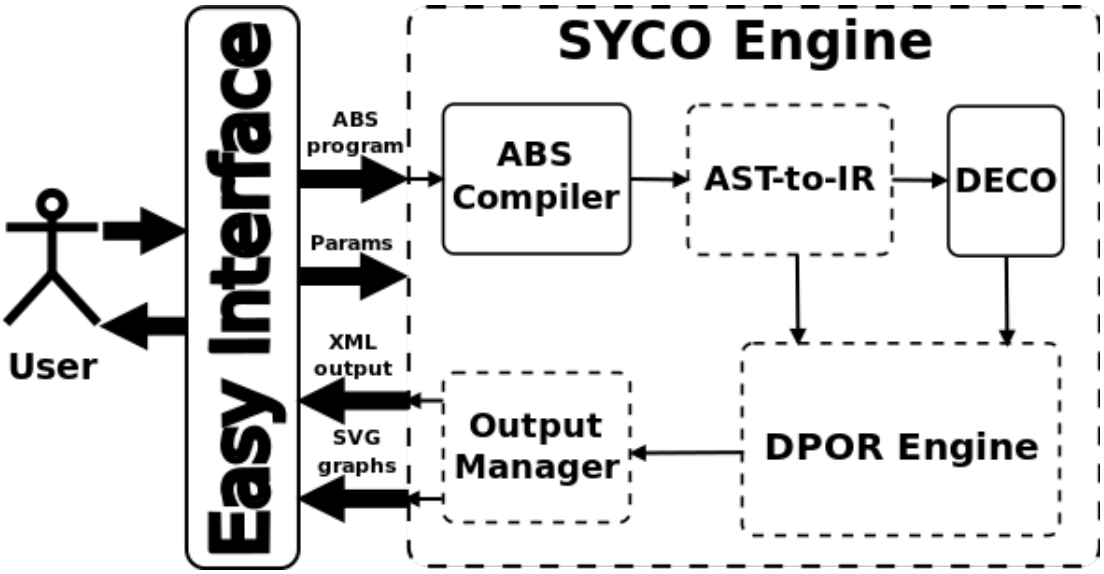


Figure 5.2: SYCO architecture

[33, 35] and thread-based programs [54], is based on static analysis techniques. Static analysis can ensure the absence of errors, however, it works on approximations (especially for pointer aliasing) which might lead to a "don't know" answer.

Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove deadlock freedom. Using our method, we try to find a deadlock by exploring the paths (possibly infinite) given by a deadlock detection algorithm that relies on the static information. Although we have used the output given by the deadlock analyzer of [33], our combined approach could use the output of other static analyzers (e.g., [35]) without requiring any conceptual change to the combined framework.

Symbolic Execution, Verification, Model Checking, Testing. By relying only on systematic testing and symbolic execution, one can do (non-guided) deadlock detection already, and besides other types of errors can also be captured (e.g., find critical states that can cause the system to crash). This is the approach taken in model checking and other verification techniques which are based on symbolic execution to automatically verify correctness properties. Indeed, deadlock detection has been intensively studied in the context of model checking (see, e.g., [58]). Both static and dynamic testing aim at finding bugs, among them deadlocks (see, e.g., [27, 47, 16, 43]). Indeed, symbolic execution is at the core of static testing systems and our symbolic execution engine is the basis for SYCO.

Hybrid Approaches. We now relate our work to hybrid approaches that use static information during testing for deadlock detection, namely [46] and [4]. As regards [46], it first performs a transformation of the program into a trace program that only keeps the instructions that are relevant for deadlock and then dynamic testing is performed on such program. The approach is fundamentally different from ours: in their case since model checking is performed on the trace program (that over-approximates the deadlock behavior), the method can detect deadlocks that do not exist in the program, while in our case this is not

possible since the testing is performed on the original program and the analysis information is only used to drive the execution. As regards [4], the information inferred from a type system is used to accelerate the detection of potential cycles. This work shares with our work that information inferred statically is used to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but works on a general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them.

Chapter 6

Conclusions and Future Work

This chapter overviews the main contributions of this thesis and the future work. It is structured as follows:

- Section 6.1 summarizes the main contributions in DPOR algorithms based on using different notions of conditional independence and it overviews the future work for this topic.
- Section 6.2 reviews the main contributions in Software-Defined Networks and the future work in this area.
- Finally, Section 6.3 sums up the contributions obtained by combining static analysis and testing to guide the executions towards deadlock situations and it also overviews its future work.

6.1 Conditional Independence in DPOR Algorithms

A main goal of this thesis has been to improve the state-of-the-art in Dynamic Partial Order Reduction algorithms by using notions of conditional independence. Our first challenge was to combine two notions of independence, namely *context-sensitive independence* and *independence modulo observability*, and study their synergy.

Firstly, we have extended the Optimal DPOR algorithm to handle context-sensitive independence. We have built on top of the Optimal DPOR algorithm, another version that allows using independence modulo observability. Furthermore, we have proposed a new check, called *state equivalence modulo observability* that allows detecting even more redundancies. Furthermore, we have performed an experimental evaluation with three sets of benchmarks that shows exponential gains compared to Context-Sensitive DPOR and Optimal DPOR with Observers.

Additionally, we have also introduced sufficient conditions that allow us to exploit independence constraints within the DPOR framework. These conditions can be checked efficiently and dynamically and allow us to extend the classic DPOR algorithms with new ways of pruning. Moreover, the experimental evaluation demonstrates huge reductions in the number of explored executions and time thanks to these prunings.

We thus argue that the techniques proposed in this thesis provide actual evidence that using conditional independence within DPOR algorithms improves upon state-of-the-art results. Moreover, our experimental evaluation shows exponential gains compared to DPOR algorithms using unconditional independence.

Recovering states of the current execution sequence. A potential hazard of using conditional independence within DPOR algorithms is that it needs to check independence in the states explored in the current execution sequence but not in the current state. It does not need to revisit states that have been completely explored and backtracked, but only those in the current execution sequence. There are several strategies to confront this challenge: *on-demand recomputing*, where all states are recomputed following the same order of events that led to them (and then no memory usage is needed); *full storage*, where all states are stored to be used until the state is backtracked; and *state caching* [59], where states are stored until the memory is approaching full utilization. Our current implementation follows the second strategy. According to our experimental results, full storage performs efficiently, since the number of stored states is limited by the number of events in each execution sequence and it remains quite low for the experiments. However, our future work includes the implementation of these strategies to study the gains obtained by each of them.

Combination of uniform conditional independence and independence modulo observability. It is on our agenda to study the combination of the constrained framework with the independence modulo observability. Independence constraints contain useful conditions about the commutativity of each pair of atomic blocks. Consequently, an improvement of these ICs with information of possible observers may enable the use of more efficient checks that replace the state equivalence modulo observability. We strongly believe that the resulting framework may achieve great gains in comparison with the previous algorithms.

Extension of Data-Centric DPOR using conditional independence. Additionally, other recent approaches have considered alternative ways of refining the detection of independence. In particular, Data-Centric DPOR [26] focused on the read-write of variables. They also use a notion of observation but different from [18] claiming that two executions are equivalent if every reading event observes the same write event in both executions. The main advantage of this approach is that it is proven to detect more executions as equivalent thanks to this notion of observation. However, it may explore several executions per equivalence class, thus this algorithm is not optimal.

Based on this new notion, [3] proposes another algorithm that guarantees optimality and, consequently, it can obtain exponential gains in comparison with Data-Centric DPOR. Nevertheless, none of these approaches considers conditional independence. We believe that both algorithms can also benefit from using a conditional variant as we have studied in this thesis. It is on our agenda to further study the combination of both techniques.

Optimality of conditional framework. Although conditional frameworks have experimentally shown to achieve exponential reductions, they have not been proven optimal with

respect to the equivalence classes induced by a conditional relation. Consequently, the Constrained DPOR algorithm sometimes initiates partial executions that get stopped by the sleep sets. Even though this partial executions must be fully explored by unconditional algorithms, our future work includes the generalization of the optimal DPOR framework to enable the use of conditional happens-before relations.

A better approximation of uniformity. One of the most important contributions of this thesis is the automatic generation of sufficient independence conditions to be able to use (uniform) conditional independence within DPOR algorithms. As the experimental evaluation shows, we can obtain exponential gains thanks to the proposed condition. However, the more precise the approximation is, the coarser the equivalence class is. Consequently, our future work includes the study of better approximations that allow achieving even more gains using conditional independence. For instance, instead of requiring that the variables involved in the constraints cannot be modified by other blocks in the state, we can pre-compute the set of blocks which keep invariant the constraints from that state on.

6.2 Model-Checking for Software-Defined Networks

Another big challenge of this thesis has been to apply this framework to a realistic setting. In this regard, we have proposed a novel actor-based framework to model and verify SDN programs. Several case studies have been developed: most of them are correct programs but some of them contain bugs. The model-checker built on top of SYCO has been able to (1) verify some properties and (2) find the bugs for the erroneous programs. Additionally, we have been able to scale up more than the state-of-the-art SDN verifiers thanks to the use of conditional independence and, in particular, to the independence constraints.

Automatic generation of ICs for SDN networks. The SMT Solver generates many redundant constraints for SDN networks. Consequently, Constrained DPOR may spend a significant amount of time and resources to check constraints. We have studied by-hand these constraints and removed the redundant ones. We plan to improve the automatic generation of these constraints as future work.

Application of other formal methods. Additionally, although it has not been explored yet in this thesis, the proposed encoding opens the door to apply a huge range of techniques other than model checking. Other tools and methods for verification of message-passing can be easily adapted [27, 51, 23, 52]. For instance, we can use the deadlock analysis proposed by [33] to detect if these networks can enter in a deadlock situation. It is also in our agenda to apply the techniques described in Chapter 4 to generate test cases for SDN networks.

6.3 Combining Static Analysis and Testing

Finally, the last challenge of this thesis has been to combine static analysis with testing and symbolic execution for deadlock detection. We have proposed a deadlock-guided framework

that uses the information provided by a deadlock analysis to guide the execution and (1) prove deadlock freedom and (2) give deadlock traces in examples containing deadlocks. We have also given a new algorithm to infer which tasks must be present in an initial context to help symbolic execution find deadlock traces. Both techniques have been also evaluated with experiments that show their efficiency and effectiveness.

Application to other properties. We are also studying the possibility of guiding the search towards paths satisfying other properties of interest for the actors' concurrency model. For instance, we can use a resource analysis [8] to guide the execution towards derivations where a particular resource is consuming more than a user-given threshold. This is a topic for future research.

Application in a thread-based concurrency model. As regards the application in a thread-based concurrency model, the fundamental difference is that even though our whole approach is defined at the level of atomic tasks, it would be adaptable to thread-based applications that rely on synchronized blocks of code, such as in monitors. As future work, we plan to investigate how our framework could be adapted to this model.

Bibliography

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Ngo Tuan Phong, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *PACMPL*, 3(OOPSLA):150:1–150:29, 2019.

[4] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, pages 191–207, 2005.

[5] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[6] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter Stuckey. Context sensitive dynamic partial order reduction. In Victor Kuncak and Rupak Majumdar, editors, *29th International Conference on Computer Aided Verification (CAV 2017)*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017.

[7] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Jose Miguel Rojas. Test case generation by symbolic execution: Basic concepts, a CLP-based instance, and actor-based concurrency. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 263–309. Springer International Publishing, 2014.

[8] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Resource analysis of distributed systems. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 33–46. Springer, 2016.

[9] Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal context-sensitive dynamic partial order reduction with observers. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19)*, ACM, 2019. To appear.

[10] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Combining static analysis and testing for deadlock detection. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2016.

[11] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 269–270. ACM, 2016.

[12] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Generation of initial contexts for effective deadlock detection. In *Logic-Based Program Synthesis and Transformation: 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 3–19. Springer International Publishing, 2018.

[13] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained dynamic partial order reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.

[14] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, Albert Rubio, Matteo Sammartino, and Alexandra Silva. Actor-based model checking for SDN networks. *CoRR*, abs/2001.10022, 2020.

[15] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. SDN-Actors: Modeling and verification of SDN programs. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 550–567, 2018.

[16] Baris Kasikci Ali Kheradmand and George Candea. Lockout: Efficient testing for deadlock bugs. Technical report, 2013. Available at `http://dslab.epfl.ch/pubs/lockout.pdf`.

[17] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126, 2014.

[18] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 229–248, 2018.

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[20] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 282–293, 2014.

[21] Kshitij Bansal, Eric Koskinen, and Omer Tripp. Commutativity condition refinement, 2015.

[22] Ryan Beckett, Michael Greenberg, and David Walker. Temporal netkat. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 386–401, 2016.

[23] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 651–662, 2015.

[24] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[25] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. A NICE way to test Openflow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 127–140, 2012.

[26] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *PACMPL*, 2(POPL):31:1–31:30, 2018.

[27] Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 154–163. IEEE Computer Society, 2013.

[28] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.

[29] Jesús Doménech, Samir Genaim, Einar Broch Johnsen, and Rudolf Schlatte. EasyInterface: A toolkit for rapid development of GUIs for research prototype tools. In *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, Lecture Notes in Computer Science, pages 379–383. Springer, 2017.

[30] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, pages 143–160, 2012.

[31] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. SDNRacer: Concurrency analysis for software-defined networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 402–415, 2016.

[32] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[33] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, June 2013.

[34] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 343–355, 2015.

[35] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, pages 394–411, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 63–77, 2014.

[37] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016.

[38] Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification,*

2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[39] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[40] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, pages 438–449, 1993.

[41] Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pages 246–265, 2017.

[42] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, February 2009.

[43] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, pages 245–264, 2000.

[44] Shiyou Huang and Jeff Huang. Speeding up maximal causality reduction with static dependency analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 16:1–16:22, 2017.

[45] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, pages 142–164, 2010.

[46] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 327–336, 2010.

[47] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 110–120, 2009.

[48] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 398–413, 2009.

[49] Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 113–126, 2012.

[51] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Basset: A tool for systematic testing of actor programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 363–364, 2010.

[52] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 385–399, 2016.

[53] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 163–170, 2014.

[54] Stephen P. Masticola and Barbara G. Ryder. A model of ada programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, USA, May 20-21, 1991*, pages 97–107, 1991.

[55] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, pages 279–324, 1986.

[56] Openflow switch specification, October 2013. Version 1.4.0.

[57] Túlio A. Pascoal, Yuri Gil Dantas, Iguatemi E. Fonseca, and Vivek Nigam. Slow TCAM exhaustion ddos attack. In *ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings*, pages 17–31, 2017.

[58] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[59] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 456–469, 2015.

[60] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[61] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 145–148, 2013.

[62] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, volume 7273 of *LNCS*, pages 219–234. Springer, 2012.

[63] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[64] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.

Parte II

Papers of the Thesis

Chapter 7

Publications

1. Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal Context-sensitive Dynamic Partial Order reduction with Observers. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19)*, pages 352–362, ACM, 2019.

2. Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained Dynamic Partial Order Reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.

3. Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, Albert Rubio, Matteo Sammartino, Alexandra Silva. Actor-Based Model Checking for SDN Networks.

4. Miguel Isabel. Conditional Dynamic Partial Order Reduction and Optimality Results. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19)*, pages 433–437, ACM, 2019.

5. Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel. Combining Static Analysis and Testing for Deadlock Detection. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2016.

6. Elvira Albert, Miguel Gómez-Zamalloa. Miguel Isabel. Generation of initial contexts for deadlock detection. In *Logic-Based Program Synthesis and Transformation: 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 3–19. Springer International Publishing, 2018.

7. Elvira Albert, Miguel Gómez-Zamalloa. Miguel Isabel. SYCO: a Systematic Testing Tool for Concurrent Objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 269–270. ACM, 2016.

Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers

Elvira Albert
Complutense University of Madrid
Spain
elvira@fdi.ucm.es

Maria Garcia de la Banda
Faculty of IT, Monash University
Australia
maria.garciadelabanda@monash.edu

Miguel Gómez-Zamalloa
Complutense University of Madrid
Spain
mzamalloa@fdi.ucm.es

Miguel Isabel
Complutense University of Madrid
Spain
miguelis@ucm.es

Peter J. Stuckey
Faculty of IT, Monash University
Australia
peter.stuckey@monash.edu

ABSTRACT

Dynamic Partial Order Reduction (DPOR) algorithms are used in stateless model checking to avoid the exploration of equivalent execution sequences. DPOR relies on the notion of *independence* between execution steps to detect equivalence. Recent progress in the area has introduced more accurate ways to detect independence: Context-Sensitive DPOR considers two steps p and t independent in the current state if the states obtained by executing $p \cdot t$ and $t \cdot p$ are the same; Optimal DPOR with Observers makes their dependency conditional to the existence of future events that observe their operations. We introduce a new algorithm, Optimal Context-Sensitive DPOR with Observers, that combines these two notions of conditional independence, and goes beyond them by exploiting their synergies. Experimental evaluation shows that our gains increase exponentially with the size of the considered inputs.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Testing, Software Verification, Model-Checking, Partial-Order Reduction.

ACM Reference Format:
Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. 2019. Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330565

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330565

1 INTRODUCTION

Partial Order Reduction (POR) considers two execution sequences equivalent if one can be obtained from the other by swapping adjacent, *independent* execution steps. Each such equivalence class is called a Mazurkiewicz [16] trace, and POR guarantees that exploring one sequence per equivalence class is sufficient to cover all. Early POR algorithms [9, 11, 19] relied on static approximations of independence. The Dynamic-POR (DPOR) algorithm [10] was a breakthrough because it uses the information witnessed during the actual execution of the sequence to decide dynamically what to explore. Thus, it often explores less sequences than approaches based on static approximations. As a result, DPOR is considered one of the most scalable techniques for software verification.

The cornerstone of DPOR is the notion of *(in)dependence*, which is used to decide if two concurrent execution steps p and t (do not) interfere with each other and, thus, both $p \cdot t$ and $t \cdot p$ sequences must (not) be explored. To guarantee soundness, DPOR approximates independence and, thus, can lose precision if it treats execution steps as interfering when they are not. Optimal DPOR (ODPOR) [2] ensures optimality (never explores equivalent execution sequences), but only w.r.t. *unconditional* independence, which requires execution steps to be independent in any possible state. In practice, syntactic approximations are used to detect unconditional independence: typically, two execution steps are considered dependent if both access the same variable and at least one modifies it.

Any DPOR algorithm can thus improve its efficiency by using a more accurate independence notion [12]. Two recent approaches – DPOR$_{cs}$ (Context-Sensitive DPOR) [3] and ODPOR^{ob} (Optimal-DPOR with Observers) [7] – have achieved this by integrating orthogonal notions of *conditional independence* into DPOR:

- *DPOR$_{cs}$*: introduced the notion of *context-sensitive* independence, which only requires execution steps p and t be independent in the state S where they appear. This is determined by executing sequences $p \cdot t$ and $t \cdot p$ in S, and checking if the two states reached are equal. Consider, for example, the three concurrent processes p, q and r below, and the execution tree in Fig. 2(a), which will be explained throughout the paper.

```
int x = 0;
p: x = 1;
q: x = 2;
r: assert (x < 3);
```

Assume p is scheduled first and we reach state 1, where x==1. Executing either $q \cdot r$ or $r \cdot q$ in state 1 yields the same final state: x==2 and the assertion holds. Therefore, DPOR$_{cs}$ considers r and q independent in the context of state 1.

- *ODPOR^{ob}*: introduced the notion of *observability*, where dependencies between execution steps p and t are conditional to the existence of future steps, called observers, which read the values modified by p and t. Consider again the three concurrent processes p, q and r above, and the execution tree in Fig. 2(a). Assume r is scheduled first reaching state 9, where x==0 and the assertion holds. ODPOR^{ob} considers q and p independent since, while their interleaved execution leads to different final states, variable x is not observed later.

DPOR$_{cs}$ and ODPOR^{ob} modified the DPOR algorithm to exploit their notions of independence. We present a further modification of DPOR, called Optimal Context-Sensitive DPOR with Observers (ODPOR$^{ob}_{cs}$), that not only combines and exploits these two powerful notions, but also takes advantage of their synergy to gain further pruning. Let us consider the leftmost branch of the execution tree in Fig. 2(a). DPOR$_{cs}$ does not consider p and q independent, as they give different values to variable x. ODPOR^{ob} does not consider them independent either, as r observes the different values they give to x. However, ODPOR$^{ob}_{cs}$ does consider them as independent, as the assertion of observer r evaluates to *true* after executing either $p \cdot q$ or $q \cdot p$. Two major contributions are needed for this:

(1) DPOR$_{cs}$ was formulated over Source-DPOR [1]. Thus, it did not include the extension of *wakeup trees* used by ODPOR to ensure optimality, and later used to handle observers. Our first contribution is the formulation of DPOR$_{cs}$ over ODPOR, which we name *Optimal Context-Sensitive DPOR* (ODPOR$_{cs}$).
(2) Our second contribution is to integrate observability into ODPOR$_{cs}$, obtaining ODPOR$^{ob}_{cs}$. For this, we modify context-sensitive independence to be *modulo observability*, which only requires equivalence for variables affected by future observers.

We have implemented ODPOR$^{ob}_{cs}$ and experimentally evaluated it with benchmarks from [3], [6] and [7]. Our experimental results show ODPOR$^{ob}_{cs}$ can explore exponentially less sequences than either DPOR$_{cs}$ or ODPOR^{ob}.

2 PRELIMINARIES

2.1 Basics of DPOR and ODPOR

An *event* (p, i) of execution sequence E represents the i-th occurrence of process p in E. We use $e <_E e'$ to denote that event e occurs before event e' in sequence E, s.t. $<_E$ establishes a total order between events in E.

The core concept in ODPOR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation is used to define a subset of the $<_E$ total order, such that any two sequences with the same happens-before order are equivalent. Let $dom(E)$ denote the set of events in E. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence with the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. Thus, $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same happens-before relation. We use $E \simeq E'$ to denote that E and E' are equivalent.

The happens-before relation is also used for defining the notion of *race*. Event e is said to be in race with event e' in execution E, written $e \lessdot_E e'$, if the events belong to different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent" ($\exists E'$ s.t. $E' \simeq E$ and the two events are adjacent in E'). We write $e \precsim_E e'$ to denote that e is in a reversible race with e', i.e., e is in a race with e' and the two can be reversed ($\forall E'$ s.t. $E' \simeq E$ and e appears immediately before e', e' is not blocked).

Optimality in ODPOR is achieved through the use of *wakeup trees*. A wakeup tree is an *ordered tree* $\langle B, \prec \rangle$, where the set of nodes B is a finite prefix-closed set of sequences of processes, with the empty sequence ϵ at the root. The children of node w, of form $w.p$ for some set of processes p, are ordered by $\prec$. Intuitively, a wakeup tree of sequence E, written $wut(E)$, is composed of partial execution sequences that must be explored from E, as they (a) reverse the order of detected races, and (b) are provably not equivalent. As a result, ODPOR does not even initiate equivalent explorations, achieving exponential reductions over earlier DPOR algorithms.

To ensure an execution sequence v does not lead to equivalent explorations if inserted in $wut(E)$, *sleep* and *weak initials* sets [1] are used. The sleep set of execution E, $Sleep(E)$, contains the processes that should not be explored from E, as they lead to equivalent executions. The weak initials set of sequence w from execution E, $WI_{[E]}(w)$, contains any process with no "happens-before" predecessors in $dom_{[E]}(w)$, where $dom_{[E]}(w)$ denotes the subset of events in execution sequence $E.w$ that are in w, i.e., $dom(E.w) \backslash dom(E)$. Then, v is known to lead to equivalent explorations from E if $Sleep(E) \cap WI_{[E]}(v) \neq \emptyset$.

Other notation we use includes: $\hat{e}$, denoting the process of event e; $s_{[E]}$, the state after executing sequence E; *enabled(s)*, the set of processes that can perform an execution step from state s; $pre^+(E, e)$ and $pre(E, e)$, the prefix of sequence E up to, including and not including e, respectively; $insert_{[E]}(v, wut(E))$, the extension of $wut(E)$ with new sequence v; and $subtree(\langle B, \prec \rangle, p)$, the subtree of wakeup tree $\langle B, \prec \rangle$ rooted at process $p \in B$, i.e., the tree $\langle B', \prec' \rangle$, where $B' = \{w | p.w \in B\}$ and $\prec'$ is the restriction of $\prec$ to B'.

2.2 ODPOR with Observers

The notion of dependency we use in this paper extends the traditional one by using the concept of *observer* introduced in [7].

Definition 2.1 (observers(e, e', E)[7]). Given an execution sequence E, for all events $e, e' \in dom(E)$ where $e \lessdot_E e'$, there exists a set $O = observers(e, e', E) \subseteq dom(E)$ such that:

(1) For all $o \in O$, it holds that $e \rightarrow_E o$, $o \neq e'$, and $o \nrightarrow_E e'$.
(2) For all $o, o' \in O$, it holds that $o \nrightarrow_E o'$.
(3) If $E' \simeq E$, then $observers(e, e', E') = O$.
(4) For every prefix E' of E such that $e, e' \in dom(E')$:
   - If O is empty, then $e \rightarrow_{E'} e'$.
   - If O is nonempty, then $e \rightarrow_{E'} e'$ iff $dom(E') \cap O \neq \emptyset$.
(5) If $e \precsim_E e'$, for all sequences w s.t. $E.w$ is a sequence, and all events $e'' \in dom(E)$:

```
1:  procedure RaceDetection(E)
2:    for all e, e' ∈ dom(E) such that e ≾_E e' do
3:      let E' = pre(E, e);
4:      if observers(e, e', E) ≠ ∅ then
5:        let o = max_E(observers(e, e', E));
6:        let v = notdep*(e, e', E).ê'.ê.(notobs*(e, e', E)\ê').ô;
7:      else
8:        let v = notdep*(e, e', E).ê';
9:    if v ∉ redundant(E', done) then
10:     wut(E') := insert_[E'](v, wut(E'));
```

Figure 1: RaceDetection of ODPOR^{ob} [7]

- If $e \nrightarrow_E e''$, then $e \not\lessdot_{E.w} e''$.
- If $e'' \nrightarrow_E e'$, then $e'' \not\lessdot_{E.w} e'$.

(6) For all $e'' \in dom(E)$ such that $e' \rightarrow_E e''$, it holds that $O \cap observers(e', e'', E) = \emptyset$.

(7) If $O = \{o\}$ and $E = E'.\hat{o}$ for some o and E', then for any $E'' \simeq E'$, either $e \rightarrow_{E''.\hat{o}} e'$ or $e' \rightarrow_{E''.\hat{o}} e$.

Intuitively, in the usual particular case, $observers(e, e', E)$ is the set of other events in E, independent of each other (by Property 2), that read the value written by e (e') for any variable also written by e' (e) (by Property 4). By an abuse of notation, we will sometimes treat this set as a sequence.

Thus, the happens-before relation used in this paper is the one defined in [7], based on the notion of observability discussed previously (except from Sec. 3, which uses the relation in [2]). Intuitively, two processes p and q are *dependent modulo observability* in execution E if either *enable*s the other (i.e., executing $E.p$ introduces q, or vice versa), or (ii) $s_{[E'.p.q]} \neq s_{[E'.q.p]}$, where $E' < E$, and there exists in E at least one observer reading the variable written by both of them.

The code in black of procedure *Explore* of Algorithm 1 (lines 11-33, excluding underlined blue parts) and the code of procedure *RaceDetection* of Fig. 1, corresponds to the ODPOR^{ob} algorithm [7], which extends the original ODPOR [2] with the notion of observers, and is our starting point. ODPOR^{ob} carries out a depth-first exploration of the execution tree from execution sequence E (initially empty) using DPOR. Essentially, it dynamically finds reversible races and is able to backtrack at the appropriate scheduling points to reverse them. For this purpose, it keeps two sets at every prefix E' of E: the usual wakeup tree $wut(E')$, with the execution sequences that must be explored from E', and the set $done(E')$ of processes that have already been explored from E'.

ODPOR^{ob} starts by selecting (line 24) the leftmost process p in the wakeup tree, according to its order $\prec$, that is enabled by state $s_{[E]}$ (due to line 14). If there is such a process, it sets WuT' as the subtree of $wut(E)$ with root p (line 28), and recursively explores every sequence in WuT' from $E.p$ (line 31). Note that $wut(E)$ might grow as this recursion progresses, due to later executions of line 10. After the recursion finishes, it adds p to $done(E)$, removes from $wut(E)$ all sequences that start from p, and iterates selecting a new p. Once a *complete* sequence E has been explored (E is said to be complete if $enabled(s_{[E]}) = \emptyset$), the algorithm performs the race detection phase (line 14). This starts by finding all pairs of events e and e' in $dom(E)$ such that $e \precsim_E e'$. For each such pair, it sets E' to $pre(E, e)$ and checks if the race between e and e' is *observed* (line 4).

If the race is not observed, v is set to $notdep^*(e, e', E).\hat{e'}$ (line 8), where $notdep^*(e, e', E)$[1] is the subsequence of processes $\hat{e''}$ of E such that events e'' hold $e <_E e''$ and $e \nrightarrow_E e''$. If it is observed, the race must be reversed and observed by the same observers. Thus, the last (max_E) observer o executed in E is selected (line 5) and used to compute v (line 6), where $notobs^*(e, e', E)$[1] denotes the subsequence of E containing any process $\hat{e''}$ such that $e \rightarrow_E e''$, but e'' does not observe the race $e \precsim_E e'$, and $o' \nrightarrow_E e''$ for any observer o' of the race. There is a small change in line 5 with respect to [7]: we select o as the last (rather than an arbitrary) observer from $observers(e, e', E)$. The reason for this will be clear in Sec. 4.1. Finally, if v is not redundant for E' (line 9), it is inserted into $wut(E')$ (line 10). To detect if v is redundant from E', ODPOR^{ob} cannot use sleep sets because they are not sufficiently precise, in the presence of observers, for avoiding redundant explorations without missing non-redundant ones [7]. Instead, ODPOR^{ob} uses the set $done$: $v \in$

Algorithm 1 ODPOR$_{cs}$ algorithm

```
11: procedure explore(E,WuT,DnD)
12:   dnd(E) := DnD;
13:   done(E) := ∅;
14:   if enabled(s_[E]) = ∅ then RaceDetection(E);
15:   else if WuT ≠ ⟨{ϵ}, ∅⟩ then
16:     wut(E) := WuT;
17:   else if enabled(s_[E])\dnd(E) = ∅ then
18:     for each p ∈ dnd(E) such that |p| = 1 :
19:       RaceDetection(E.p);
20:   else
21:     choose p ∈ enabled(s_[E])\dnd(E);
22:     wut(E) := ⟨{ϵ, p}, {(p, ϵ)}⟩;
23:   while ∃p ∈ wut(E) do
24:     let p = min_≺{p ∈ wut(E)};
25:     if p ∈ dnd(E) then
26:       RaceDetection(E.p);
27:     else
28:       let WuT' = subtree(wut(E), p);
29:       let DnD' = {v | v ∈ dnd(E), p ∉ v, E ⊨ p ⋄ v}
30:                  ∪ {v | (p.v) ∈ dnd(E)};
31:       Explore(E.p, WuT', DnD');
32:       add p to done(E);
33:     remove all sequences of form p.w from wut(E);
34: procedure RaceDetection(E)
35:   for all e, e' ∈ dom(E) such that e ≾_E e' do
36:     let E' = pre(E, e); let dont = ϵ;
37:     let v = notdep*(e, e', E).ê'; v := v.I_fut(E', v, E);
38:     if s_[pre+(E,e')] = s_[E'.(v.suc(e,E))_{≤E e'}] then
39:       dont := v.ê;
40:     if v ∉ redundant(E', done) then
41:       wut(E') := insert_[E'](v, wut(E'));
42:       add dont to dnd(E');
```

[1] The mark * in functions $notdep^*$ and $notobs^*$ indicates they will be redefined later. Function $notdep^*(e, e', E)$ does not use parameter e', it will be used once redefined.

$redundant(E', done)$ iff $E'.v$ is an execution sequence and there is a partitioning $E' = w.w'$ such that $done(w) \cap WI_{[w]}(w'.v) \neq \emptyset$.

Example 2.2. Consider again the processes p, q and r in Fig. 2(a). Since they all have a single event, by abuse of notation, we will refer to events by their process name. The algorithm starts at state 0 in Fig. 2(a), with both E and WuT empty. The execution first chooses p, and explores sequence p with an empty $done$ set to state 1. The execution then chooses q and explores sequence $p.q$ with an empty $done$ set to state 2. Since now only r can be chosen, the execution explores sequence $p.q.r$ to state 3. Now, the race detection phase detects an observed race for p and q, as they both write variable x and are observed by r (line 4). It then creates sequence $q.p.r$ in line 6, which will later lead to sequence $q.p$ and will thus make r observe the value written by p. The created sequence is added to $wut(\epsilon)$ of state 0 in line 10. A race between q and r is also detected and r is added to $wut(p)$. Execution then backtracks to state 1, adding q to $done(p)$ on the way. Next, it chooses r and continues, exploring the first five executions in Fig. 2(a). Once the fifth one is completed, it checks if p is in an observed race with q. Since it is not, as there is no observer after them, the sixth execution is not even started. Thus, ODPOR^{ob} explores one sequence less than ODPOR.

3 OPTIMAL CONTEXT-SENSITIVE DPOR

The happens-before relation used in this section is the one in [2], which does not consider observability. Essentially, DPOR$_{cs}$ works as follows: when a reversible race $e \precsim_E e'$ is detected, it not only updates the appropriate structures to ensure the race is reversed on backtracking, but also checks whether events e and e' are independent in the current context E, that is, whether $s_{[E.\hat{e}.\hat{e'}]} = s_{[E.\hat{e'}.\hat{e}]}$. If they are, it stores a sequence in a new *don't-do* set (in the original DPOR$_{cs}$ it was stored in the sleep set) at every prefix E' of E, indicating that this sequence must not be explored in full when backtracking to E'. Consider the working example of Fig. 2(a). When DPOR$_{cs}$ reaches state 3 (execution sequence $p.q.r$), it realizes q and r can be regarded as independent in context p, as $s_{[p.q.r]} = s_{[p.r.q]}$ even though they are dependent according to the happens-before relation in [2] with the usual syntactic approximation, since q writes global variable x and r reads it. Hence, it adds $r.q$ to the don't-do set of state 1. Once r is explored, q is not executed because it is in the don't-do set of state 4, which prevents the full exploration of $p.r.q$.

As mentioned before, our first contribution is the reformulation of DPOR$_{cs}$ as an extension of ODPOR, rather than of Source-DPOR. This yields an optimal DPOR$_{cs}$ algorithm (see below), referred to as ODPOR$_{cs}$, which makes it easier to integrate the notion of observers (as done in Sec. 4). Reformulating DPOR$_{cs}$ in terms of ODPOR is challenging due to two main problems:

- Problem I: While Source-DPOR performs race detection at every state, ODPOR must delay race detections until the sequence being explored is complete.
- Problem II: As shown in [3], the effectiveness of DPOR$_{cs}$ is highly dependent on exploring don't-do sequences as soon as possible. Indeed, DPOR$_{cs}$ uses these sequences to guide the selection of the next process to be explored. However, the wakeup trees of ODPOR fix part of these decisions, which can affect guidance.

The ODPOR$_{cs}$ algorithm corresponds to the code of Algorithm 1. It is discussed in detail in Secs. 3.1 and 3.2, which explain how problems I and II, respectively, have been overcome. Finally, Sec. 3.3 discusses its correctness and optimality.

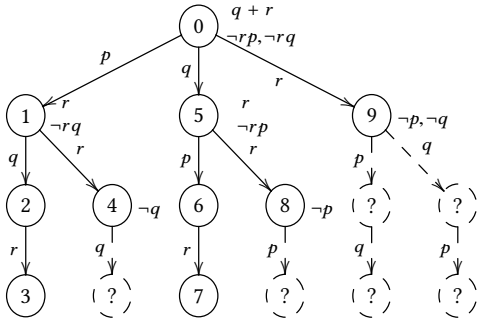
3.1 Overcoming Problem I

Delaying race detections until the entire sequence is explored, complicates the implementation of the context-sensitive checks, as they need access to intermediate states. One could recover these states by, for example, re-executing the sequence of events to reach them, or storing them, either in full or by means of incremental state updates, to be undone on backtracking. One could also perform (part of) the checks on the fly during the exploration, instead of at the end, thus reducing the number of intermediate states needed. The preferred strategy will depend on the available memory and the concrete language features. In any case, the following assumes access to all states of the current sequence.

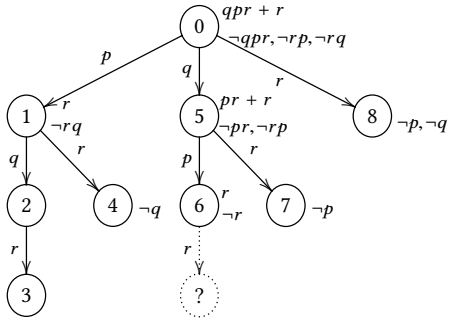
The new context-sensitive check corresponds to the underlined blue code in line 38 of Algorithm 1 (for now, we use the black code for v in line 37; it will be redefined in Sec. 3.2). Recall that the black code of Algorithm 1 is common to both ODPOR and ODPOR^{ob}, and was explained in Sec. 2.2. Intuitively, given a reversible race $e \precsim_E e'$ for events e and e', the check succeeds if the state right after the race, $s_{[pre^+(E,e')]}$, is the same as that obtained when the race is reversed, $s_{[E'.(v.suc(e,E))_{\leq_E^{e'}}]}$, where $suc(e, E)$ is the subsequence w of E that starts with $\hat{e}$ and contains all $\hat{e''}$ s.t. $e \rightarrow_E e''$, and $w_{\leq_E^{e'}}$ is the subsequence of w in E of processes that execute events up to, and including, e' (i.e., keeps $\hat{e''}$ only if $e'' \leq_E e'$). As a result, the sequence $E'.(v.suc(e, E))_{\leq_E^{e'}}$ executes the same events as $pre^+(E, e')$ but with the race reversed. Assuming we have access to $s_{[pre^+(E,e)]}$ and $s_{[E']}$, we only need to compute the state after the sequence $(v.suc(e, E))_{\leq_E^{e'}}$ from $s_{[E']}$. If the check succeeds, sequence $v.\hat{e}$ is added to the don't-do set $dnd(E')$ (line 42). Note that, unlike in the original DPOR$_{cs}$, v contains the processes of events executed after e' in E, that are independent of e and, thus, also independent of e'. This issue is further discussed in Sec. 3.2.

As in the original DPOR$_{cs}$, if a sequence w is added to the don't-do set of state s, w can be inherited down once we backtrack to s, possibly being reduced until it eventually becomes a unitary sequence and the exploration stops. In that case, race detection must be forced explicitly. This is the task of the new *if* statement in lines 25 and 26. Similarly, if every process enabled in $s_{[E]}$ is also in $dnd(E)$ for sequence E, then the exploration of E stops and race detection is forced explicitly, in this case for every unitary sequence in $dnd(E)$ (lines 17, 18 and 19). The support to inherit down don't-do sequences is the same as in the original DPOR$_{cs}$, corresponding to lines 29 and 30. Essentially, $E.p$ inherits each sequence v where $p.v \in dnd(E)$ (line 30), and where every process in v (line 29) is independent of p in E (denoted as $E \models p \diamond v$), i.e, where the event in $dom_{[E]}(p)$ does not happen-before any event in $dom_{[E.p]}(v)$.

Example 3.1. Let us explain the exploration performed by ODPOR$_{cs}$ on our running example in Fig. 2(a). The algorithm first explores sequence $p.q.r$ and then performs race detection. For the reversible race between q and r, the check (line 38) $s_{[p.q.r]} = s_{[p.r.q]}$ succeeds and, hence, $r.q$ is added to $dnd(p)$. The algorithm also finds a



(a) Full tree computed by ODPOR; dashed fragment not computed by DPOR$_{cs}$ (nor by ODPOR$_{cs}$); Arrow labels: scheduled process. Upper Node label: wakeup trees ($v + w$ is a tree with two traces). Lower node labels: don't-do (dnd).



(b) Full tree computed by ODPOR$_{cs}$; dotted fragment not computed by ODPOR$^{ob}_{cs}$. Labels are as in 2(a).

Figure 2: Execution trees computed by DPOR algorithms for our running example starting from x==0

reversible race between p and q, but this time $s_{[p.q]} \neq s_{[q.p]}$ and, thus, nothing is added to $dnd(\epsilon)$. After backtracking to state 1 with r, sequence $r.q$ is inherited down to state 4 as q (line 30). Hence, this exploration is stopped at state 4 and race-detection is explicitly invoked (lines 25 and 26). For the reversible race between p and r, the check $s_{[p.r]} = s_{[r.p]}$ also succeeds adding $r.p$ to $dnd(\epsilon)$. At this point $wut(\epsilon)$ contains q and r. After backtracking to state 0 with q, p and r can be executed. Let us suppose that $q.p.r$ is fully explored. This exploration is analogous to that of $p.q.r$. Therefore, $r.p$ will be added to $dnd(q)$, stopping the exploration at state 8 in Fig. 2(a). Due to the reversible race between q and r, the algorithm checks $s_{[q.r]} = s_{[r.q]}$, which succeeds adding $r.q$ to $dnd(\epsilon)$. Finally, after backtracking to state 0 with r, sequences $r.p$ and $r.q$ are inherited down to state 9, as p and q, respectively (line 30). Hence, the exploration stops at state 9.

3.2 Overcoming Problem II

Consider the processes p and q from our running example, and the initial exploration $E_1 = t.t'.p.q$, where t is a process defined as $t : y = 1$; and t' is another instance of the same process t. Let us assume, for now, that ODPOR$_{cs}$ uses the original definition of sequence v (line 37), that is, $v = notdep^*(e, e', E).\hat{e'}$. For the reversible race between p and q, ODPOR$_{cs}$ adds q to $wut(t.t')$. Hence, upon backtracking to $t.t'$, it will explore $E_2 = t.t'.q.p$. For the reversible race between t and t', ODPOR$_{cs}$ sets v to $p.q.t'$ and adds it to $wut(\epsilon)$. Also, since $s_{[t.t']} = s_{[t'.t]}$, it adds $p.q.t'.t$ to $dnd(\epsilon)$. Later, when backtracking to ϵ and exploring $p.q.t'$, sequence $t'.t$ is inherited down to $dnd(p.q)$. Hence, t is inherited down to $dnd(p.q.t')$, causing the exploration to stop and the race-detection phase to start (line 26) for $p.q.t'.t$. This detects a race between p and q, causing the exploration of $t'.t.q.p$, which is redundant to E_2 (as $s_{[t.t']} = s_{[t'.t]}$).

Such a redundant trace would not have been explored by DPOR$_{cs}$. This is because DPOR$_{cs}$ (as well as Source-DPOR and the original DPOR) does not record the sequence to be explored upon backtracking but, rather, an initial event to explore plus the sequences that should not be selected (by means of the so called *backtrack-set* and *sleep set*). This allows using don't-do sequences to guide DPOR$_{cs}$ decisions regarding what to explore, achieving earlier and more effective context-sensitive prunings. However, wakeup trees are essential for ODPOR to achieve optimality. Therefore, the challenge is to determine whether it is possible to keep optimality, while at the same time being able to exploit don't-do sequences at least as effectively as DPOR$_{cs}$.

In order to reverse race $e \precsim_E e'$, it suffices to have all ancestors of e' before it. Let us then re-define $notdep^*(e, e', E)$ as $ance(e, e', E)$, the subsequence of E containing the processes whose events occurs after e and happen-before $\hat{e'}$ (and thus, independent with e). This solves the problem in the above example: for the race between t and t' in E_1, the sequences added to $wut(\epsilon)$ and $dnd(\epsilon)$ would be t' and $t'.t$, respectively. This is however not enough since, in order to achieve optimality, v needs to include part of the processes of E whose corresponding events are independent with the ones in v, thus being detected as redundant in line 38. Let us define the set of *future initials*, written $I_{\text{fut}}(E', v, E)$, that contains any process with no "happens-before" predecessors in $dom_{[E'.v]}(w)$ (i.e., $WI_{[E']}(w) \setminus v$), where $E = E'.w$. Intuitively, every event executed in w is dependent with one in $v.I_{\text{fut}}(E', v, E)$ (i.e., $\forall \hat{e} \in w, \exists \hat{e'} \in v.I_{\text{fut}}(E', v, E)$ such that $e' \rightarrow_E e$). Indeed, the future initials are also required in sequence v, so that when an exploration is stopped by a don't-do sequence (line 26), the corresponding race detection phase has enough information to build the appropriate sequences for each detected new race. As a result, we redefine v as $notdep^*(e', e', E).\hat{e'}.I_{\text{fut}}(E', v, E)$ (line 37) with $notdep^*(e, e', E) = ance(e, e', E)$. In the example above, for the race between t and t' in E_1, the new sequences added to $wut(\epsilon)$ and $dnd(\epsilon)$ are $t.p$ and $t.p.t'$, respectively.

3.3 Correctness, Optimality and Final Remarks

The correctness of the ODPOR$_{cs}$ algorithm follows from the correctness of ODPOR, and the fact that context-sensitive checks only remove equivalent Mazurkiewicz traces. The optimality of ODPOR$_{cs}$

with respect to the Mazurkiewicz traces based on any conditional independence is not guaranteed, since it only detects certain cases of context-sensitive independence. However, it has similar optimality results as [2] (i.e., for the Mazurkiewicz traces based on unconditional independence): if ODPOR$_{cs}$ explores a sleep set blocked execution E, then ODPOR explores completely an execution with the same happens-before relation than E. We do not compute sleep sets but they can be obtained from the *dnd* and *done* sets. Furthermore, ODPOR$_{cs}$ never explores more traces than ODPOR.

Definition 3.2 (Sleep set and Sleep set blocked execution [2]). Given an execution sequence E and $dnd(E)$ set and a $done(E')$ set for each prefix $E' \leq E$, we define $Sleep(E)$ as the set of processes $\{p \mid p \in dnd(E)$ such that $\exists E' \leq E, p \in done(E')\}$. A call to $Explore(E, WuT, DnD)$ is sleep set blocked during the execution of Algorithm 1 if $enabled(s_{[E]}) \subseteq Sleep(E)$.

Note that this section focuses on the correctness and optimality theorems of ODPOR$_{cs}$ and, thus, the original check [2] is used ($sleep(E') \cap WI_{[E']}(v) \neq \emptyset$). However, a similar reasoning can be done for the check in [7]: $v \in redundant(E, done)$. Proofs for the theorems in the paper can be found online in a technical report at costa.fdi.ucm.es/papers/costa/issta19-proofs.pdf.

Lemma 3.3. *If Algorithm 1 discovers that $s_{[pre^+(E',e')]} = s_{[E_0.(v.suc(e,E))_{\leq_E e'}]}$, then for any complete sequence E of the form $E = E_0.v.\hat{e}.u'.w'$ that contains a race $e' \precsim_E e$, there is a complete sequence $E' = pre^+(E', e').w$ that defines a different Mazurkiewicz trace $T' =\!\!\rightarrow_{E'}$ and leads to an identical final state.*

Theorem 3.4 (Soundness of ODPOR$_{cs}$). *For each Mazurkiewicz trace T defined by the happens-before relation, $Explore(\epsilon, \langle\{\epsilon\}, \emptyset\rangle, \emptyset)$ of Algorithm 1 explores a complete execution sequence that either implements T, or reaches an identical state to one that implements T.*

Let us claim now the optimality of Algorithm 1.

Lemma 3.5. *Let $E'.v.u$ be a complete execution sequence such that $v \in wut(E)$, $v.u \in dnd(E)$ and v' is the sequence created to reverse a race found in $E'' < E'.v.u$. For all w, such that $E'.v.u.w$, let v_w be the corresponding sequence to reverse the same race in $E'' < E'.v.u.w$. Then:*

$$Sleep(E'') \cap WI_{[E'']}(v') \neq \emptyset \Leftrightarrow Sleep(E'') \cap WI_{[E'']}(v_w) \neq \emptyset$$

Theorem 3.6 (Optimality of ODPOR$_{cs}$). *Algorithm 1 never explores two complete execution sequences that are equivalent and never initiates sleep set blocked executions.*

Finally, let us conclude this section by noting that both DPOR$_{cs}$ and ODPOR$_{cs}$ are likely to be highly beneficial for programs with large atomic code sections (e.g., monitors, concurrent objects, and message-passing systems), where the usual approximation of dependence can be rather imprecise. It is also likely to be beneficial for programs with assertions, as these only result in two possibly (local) states: either the assertion holds or it does not. Hence, the context-sensitive independence check is more likely to succeed.

4 OPTIMAL CONTEXT-SENSITIVE DPOR WITH OBSERVERS

The ODPOR^{ob} and ODPOR$_{cs}$ algorithms of Secs. 2.2 and 3 can be combined simply by joining their codes together. This also requires using the happens-before relation based on the notion of observability of [7] throughout the algorithm. The exploration performed by such a "union" algorithm would be the intersection of the explorations of ODPOR^{ob} and ODPOR$_{cs}$, and its prunings the union of the ODPOR^{ob} and ODPOR$_{cs}$ prunings.

This section goes beyond the union of the algorithms and proposes in Secs. 4.1 and 4.2 two enhancements that exploit the combination and the synergy between the notions of context-sensitive independence and observability. The resulting algorithm ODPOR$^{ob}_{cs}$ is presented in Algorithm 2. Finally, Sec. 4.3 studies the soundness of ODPOR$^{ob}_{cs}$.

Algorithm 2 ODPOR$^{ob}_{cs}$ algorithm

```
43: procedure EXPLORE(E,WuT,DnD)
44:   dnd(E) := DnD;
45:   done(E) := ∅;
46:   if enabled(s_[E]) = ∅ then RaceDetection(E);
47:   else if WuT ≠ ⟨{ϵ}, ∅⟩ then
48:     wut(E) := WuT;
49:   else if enabled(s_[E])\dnd(E) = ∅ then
50:     for each p ∈ dnd(E) such that |p| = 1 :
51:       RaceDetection(E.p);
52:   else
53:     choose p ∈ enabled(s_[E])\dnd(E);
54:     wut(E) := ⟨{ϵ, p}, {(p, ϵ)}⟩;
55:   while ∃p ∈ wut(E) do
56:     let p = min_≺{p ∈ wut(E)};
57:     if p ∈ dnd(E) then
58:       RaceDetection(E.p);
59:     else
60:       let WuT' = subtree(wut(E), p);
61:       let DnD' = {v | v ∈ dnd(E), p ∉ v, E ⊨ p ⋄ v}
62:           ∪ {(u.v) | (u.p.v) ∈ dnd(E), E ⊨_{u.p.v} p ⋄ u};
63:       Explore(E.p, WuT', DnD');
64:       add p to done(E);
65:     remove all sequences of form p.w from wut(E);
66: procedure RACEDETECTION(E)
67:   for all e, e' ∈ dom(E) such that e ≾_E e' do
68:     let E' = pre(E, e); let dont = ϵ;
69:     if observers(e, e', E) ≠ ∅ then
70:       let o = max_E(observers(e, e', E));
71:       let v = notdep*(e, e', E).ê'.ê.(notobs*(e, e', E)\ê').ô;
72:       let o_s = observers(e, e', E); v := v.I_fut(E', v, E));
73:       if ⋀_{o'∈o_s} s_[pre+(E,o')] =^{e,e'}_{o'} s_[E'.v_{≤^o_E}.(ô_s\ô)] then
74:         dont := v.(ô_s\ô);
75:     else
76:       let v = notdep*(e, e', E).ê'; v := v.I_fut(E', v, E);
77:       if s_[pre+(E,e')] = s_[E'.(v.suc(e,E))_{≤_E e'}] then
78:         dont := v.ê;
79:     if v ∉ redundant(E', done) then
80:       wut(E') := insert_[E'](v, wut(E'));
81:       add dont to dnd(E');
```

4.1 Refining the Context-Sensitive Check for Write-Write Races

Consider again the race detection phase on our running example of Fig. 2(b), after exploring the sequence $p.q.r$. The "union" algorithm still finds a reversible race $p \precsim_E q$ observed by r. After setting v to $q.p.r$ in line 71, the check $s_{[p.q]} = s_{[q.p]}$ in line 77 fails (recall this line is temporarily assumed to be out of the **else**). Hence, nothing is added to $dnd(\epsilon)$ and $q.p.r$ is added to $wut(\epsilon)$. Interestingly, sequence $q.p.r$ is equivalent to the already explored $p.q.r$, from the point of view of the observer, i.e., while the value of variable x is different, the assert in r holds in both cases.

It thus seems natural to perform a further and different context-sensitive check that compares the states *modulo observability*, e.g., compares $s_{[p.q.r]}$ and $s_{[q.p.r]}$ only considering the observation performed by r. Since the observed value in both cases makes the assert hold, $q.p.r$ could be added to $dnd(\epsilon)$, thus stopping the exploration of the third sequence at state 6. More precisely, given a race $e \precsim_E e'$ observed by o, we say that two states s and s' are *equivalent modulo observability*, written $s =_o^{e,e'} s'$, if the effect produced by observing e or e' is the same. If o is an assertion, the effect is the same if after both events o evaluates to the same Boolean value. Similarly, if o is an assignment to a variable y, where there is at least a variable which is read on its right-hand side and modified by both e and e', then the effect is the same if the value of y in s and s' is the same.

The implementation of the refined check corresponds to the underlined red code in lines 72 and 73 of Algorithm 2. First, it sets o_s to the subsequence of observer processes $observers(e, e', E)$. Then, after extending v with the required processes (see explanation below), it checks that for every observer process o' in o_s (this time treated as a set), the state after executing o' is equivalent modulo observability to the state obtained by the alternative sequence $E'.v_{\leq_E^o}.(\hat{o}_s \setminus \hat{o})$, which contains the reversed race, followed by observer o and the remaining observers.

As with the original context sensitive check (see Sec. 3.2), in order to be effective, it is important not to include in v unnecessary processes before the reversed race, while at the same time including at the end those that are necessary to keep optimality. The solution is analogous to that of Sec. 3.2. First, unnecessary processes are taken away from sequence v (line 71). In particular, $notdep^*$ inherits the redefinition of Sec. 3.2, and $notobs^*(e, e', E)$ is redefined as the subsequence of processes of E, excluding the occurrence e, whose events happen-before those in $observers(e, e', E)$. Finally, to ensure optimality, v is extended with $I_{\text{fut}}(E', v, E)$ (line 71), to ensure it has enough information to detect redundancies.

Note that all the predecessors of other observers are in $v_{\leq_E^o}$, thanks to the choice of o as $max_E(observers(e, e', E))$. Thus, we can execute $\hat{o}_s \setminus \hat{o}$ ($\hat{o}$ is already in v) without problem after $E'.v_{\leq_E^o}$. Note also that we cannot use $s_{[pre^+(E,o)]}$ to perform all the checks because, after every $o' \in o_s$ has been executed, there may be another event $e'' < o \in E$ such that $o' \rightarrow_E e''$, which would invalidate the check by modifying the value of the variables used in the check. That is why we use $s_{[pre^+(E,o')]}$ for each $o' \in o_s$ to perform each check in line 73. Another possibility, which could be more efficient in certain contexts (and does not require accessing these intermediate states), would be to perform all the checks with the state $s_{[notdep^*(e,e',E).\hat{e}.\hat{e'}.(notobs^*(e,e',E)\setminus\{\hat{e}\}).\hat{o}_s]}$, (where the race between e and e' has not been reversed).

The following provides the intuition behind the need to consider every observer $o' \in observers(e, e', E)$ for the new check, rather than just the selected one o. Consider our running example extended with one more observer $r' : assert(x < 2)$; and an initial exploration of the sequence $E = p.q.r.r'$. For the race between p and q we have that $observers(p, q, E) = \{r, r'\}$. Let us assume the algorithm selects $o := r$. If the new check only considers o (instead of every $o' \in o_s$), the check succeeds (the assert holds in both cases) and, hence, $q.p.r$ is added to $dnd(\epsilon)$. This prevents the exploration of sequence $q.p.r.r'$(where the assert of r' does not hold) which is not equivalent to any previously explored sequence. In this concrete example, this does not cause losing any different final result (the assert of r' also fails in other combinations). However, this would not be the case in an example where the only possibility for the assert of r' to fail would be to execute it after $q.p$.

Note that this new check is only applied in the case of write-write races followed by an observer (i.e. when the algorithm enters the **if** of line 69) and that it can only be finer than the original check. That is why in the final algorithm, the blue code of lines 76, 77 and 78 goes within the **else** scope, hence replacing the original check for the case of write-write races. For those races that are not observed, the original check is still applied in line 77.

Example 4.1. Let us extend our running example with a process r_2 := `assert(x < 2);` which is enabled only after executing r and let us suppose that $E = p.q.r.r_2$ is the first exploration explored by Algorithm 2. We detect a race between p and q because $observers(p, q, E) = \{r\}$, so the race is observed by r. Now, the check in line 69 is true, $p.q.r$ is equivalent modulo observer r to $q.p.r$, so $q.p.r$ is added to $dnd(\epsilon)$. However, $q.p.r.r_2$ and $p.q.r.r_2$ have a different effect in r_2 (let us notice that r_2 is not an observer for these executions). We also detect a race between q and r in E, so r and $r.q$ are added to $wut(p)$ and $dnd(p)$, respectively. Now, $p.r.r_2.q$ is also explored. Let us notice that the effect of r is always true for any possible execution and the effect of r_2 is true (in $p.r.r_2.q$) or false (in $p.q.r.r_2$) depending on the execution.

4.2 Refining the Inheritance of Don't-Do Sequences

One could think that whenever a sequence w is added to a $dnd(E')$ set of sequence E' due to the new refined check, then a prefix of w is also added to $wut(E')$. Indeed, if the refined check of line 73 succeeds, the sequence $v.(\hat{o}_s \setminus \hat{o})$ is added to $dnd(E')$, and the sequence v is inserted to $wut(E')$. However, it is possible for the sequence not to be added to $wut(E')$ if it already contains an equivalent sequence (which had been added before). In such cases, the dnd sequence might not be propagated successfully during the exploration of the corresponding sequence in $wut(E')$, resulting in unnecessary exploration.

Example 4.2. Let us consider our running example but replacing process r by $r : o = x;$, and first exploring sequence $E_1 = p.p'.q.r$, where p' is another instance of the same process p. For the race between p and q, the refined check builds the alternative sequence

$p'.q.p.r$ (note that p' happens-before q in E_1). The obtained observation is $o == 1$, whereas in the original E_1 it was $o == 2$, hence $p'.q.p.r$ is added to $wut(\epsilon)$ but not to $dnd(\epsilon)$. The algorithm explores four more sequences before backtracking to the root, including sequence $E_2 = p.q.p'.r$. In this case, for the race between p and p', the refined check builds the alternative sequence $q.p'.p.r$ (note that q happens-before p' in E_2). The obtained observation both in E_2 and $q.p'.p.r$ is $o == 1$. Hence, $q.p'.p.r$ is added to $dnd(\epsilon)$ but not to $wut(\epsilon)$, since it is equivalent to $p'.q.p.r$, which was added before. The propagation of *dnd* sequences in Algorithm 1 (underlined blue code of lines 29 and 30) is not able to propagate down $q.p'.p.r$ when exploring $p'.q.p.r$, even though they are equivalent sequences.

The refined propagation allows to generalize the previous propagation of *dnd* sequences, which can be seen in the underlined red code of line 62 of Algorithm 2. Essentially, a sequence $u.p.v$ will now be propagated as $u.v$, if p is independent of all processes in u. In addition, the new case can take advantage of observability using the information of the trace $E'.u.p.v$. We define $E \models_{u.q.p.v} q \diamond p$ if $E.u \models q \diamond p$, (i.e., they are unconditional independent), or $\exists \hat{w} \in v$, such that the set of variables written both by p and q is overwritten by w and $\forall \hat{r} \in v$ that observes any of these variables, $w <_{E.u.q.p.v} r$. Intuitively, this refined propagation allows transitively propagating equivalences between the *dnd* set and the WuT of a state.

In the case of Example 4.2, when backtracking to the root to explore $p'.q.p.r$, the sequence $q.p'.p.r$ in $dnd(\epsilon)$ is propagated down to $dnd(p')$ as $q.p.r$. This allows detecting $p'.q.p.r$ as redundant. Indeed, $\epsilon \models_{qp'pr} q \diamond p'$ in $q.p'.p.r$ since r is not observing their effect (it observes the subsequent write p), whereas they would be dependent with the traditional notion of dependency.

Let us finally point out that this refinement is also applicable to the ODPOR$_{cs}$ algorithm of Sec. 3, and also to the original DPOR$_{cs}$ algorithm of [3], although in these contexts it would be much less likely to be applied.

4.3 Correctness and Optimality

The theorem for ODPOR$^{ob}_{cs}$ is analogous to the one in Sec. 3.3, but using the definition of *equivalence modulo observability*, introduced in Sec. 4.1. As in [7], the optimality used in this theorem (based on not exploring redundant complete execution sequences) is weaker than the one in Sec. 3.3 (based on not exploring sleep set blocked executions). This is because, as we have mentioned before, *sleeps sets* cannot be used with observers to achieve the stronger optimality.

Lemma 4.3. *If Algorithm 2 discovers that* $s_{[pre^+(E',o')]} =^{e,e'}_{o'} s_{[E_0.v_{\leq^o_E}.(\hat{o}_s \backslash \hat{o})]}$ $\forall o' \in o_s$, *for any complete sequence* E *of the form* $E = E_0.v.(\hat{o}_s \backslash \hat{o}).w'$ *that contains a race* $e' \precsim_E e$ *observed by* $o_s = observers(e, e', E)$ *and* $o = max_E(o_s)$, *there is a complete sequence* $E' = pre^+(E', o).w$ *that defines a different Mazurkiewicz trace* $T' = \rightarrow_{E'}$ *and leads to an identical final state modulo observability.*

Lemma 4.4 (soundness of new inheritance). *Let* E' *be an execution such that* $p.q.u \in wut(E')$, $q.p.u.v \in dnd(E')$, *and* $E' \models_{q.p.u.v} p \diamond q$. *If* $E = E'.p.q.u.v$ *and* $E'' = E'.q.p.u.v$, *then* $s_{[E]} = s_{[E'']}$ *modulo observability.*

Theorem 4.5 (Soundness of ODPOR$^{ob}_{cs}$). *For each Mazurkiewicz trace* T *defined by the happens-before relation,* *Explore*$(\epsilon, \langle\{\epsilon\}, \emptyset\rangle, \emptyset)$ *of Algorithm 2 explores a complete execution sequence that either implements* T, *or reaches an equivalent state modulo observability as one that implements* T.

Let us claim now the optimality of Algorithm 2.

Theorem 4.6 (Optimality of ODPOR$^{ob}_{cs}$). *Algorithm 2 never explores two complete execution sequences that are equivalent.*

5 EXPERIMENTS

This section reports on an experimental comparison of the performance of DPOR$_{cs}$ [3], ODPOR^{ob} [7] and our proposed ODPOR$^{ob}_{cs}$. We have implemented and experimentally evaluated our method within the SYCO tool [4], a systematic testing tool for message-passing concurrent programs. SYCO can be used online through its web interface available at http://costa.fdi.ucm.es/syco. We have used three sets of benchmarks: The first one is a subset of the synthetic programs used in [7] to compare ODPOR and ODPOR^{ob}. Benchmarks FR, FR-a, LW, and abs are similar to our running example, while Lam is a mutual exclusion protocol. We have not included apr_1, an Apache library written in C, because translating it to our language is very complex. Similarly, we have excluded the second set of benchmarks used in [7], because they are written in Erlang and exploit the notion of observability inherent to a `receive` synchronization primitive that is not supported by our language [13]. Our second set of benchmarks is a subset of the classical concurrent programs used in [3] to compare Source-DPOR and DPOR$_{cs}$. They feature typical distributed and concurrent algorithmic patterns, in which computations are split into smaller atomic subcomputations that concurrently interleave their executions, and work on shared data. Our set includes two concurrent sorting algorithms, QS and MS, concurrent Fibonacci, Fib, a database protocol, DBP, and a consumer producer interaction, BB. We excluded Pi, PSort and Reg, because they were already optimal in DPOR$_{cs}$ and behave as Fib and MS. Our third set of benchmarks include two larger programs: MapRed, an implementation of a map-reduce model developed by a company (440 lines of code); and SDN [6], a model of a software-defined network featuring a safety policy violation (490 lines).

We have executed each benchmark with 4 size increasing input parameters and a timeout of 120 seconds. When reached, we write $>X$ to indicate that, for the corresponding measure, we encountered X units at timeout (i.e., it is at least X). Table 1 shows the results of the executions. An exception for this is Lam, for which we only show one input (corresponding to two processes trying to access the critical section), since it is not tractable for more than two processes in our implementation (in the implementation of [7] it becomes intractable for more than three processes). Column E shows the number of execution sequences, S the number of states explored, and T the time in seconds needed to compute them. Times are obtained on an Intel(R) Core(TM) i7 CPU at 2.5Ghz with 8GB of RAM (Linux Kernel 5.4.0). Columns G^{cs}_T and G^{ob}_T show the time speedup of ODPOR$^{ob}_{cs}$ over DPOR$_{cs}$ and ODPOR^{ob}, respectively, computed by dividing their respective times by that of ODPOR$^{ob}_{cs}$. To measure memory requirements, we compute for each explored trace, the sum of the cardinality of all its *dnd* sets, and show in $\mathbf{M_D}$ the maximum of these sums. In addition, $\mathbf{M_S}$ shows the maximum

Table 1: Experimental evaluation results

	DPOR_{cs}			ODPOR^{ob}			ODPOR^{ob}_{cs}					Speed-up	
Bench.	**E**	**S**	**T**	**E**	**S**	**T**	**E**	**S**	**T**	$\mathbf{M_D}$	$\mathbf{M_S}$	$\mathbf{G^{cs}_T}$	$\mathbf{G^{ob}_T}$
FR(3)	17	36	0.02	13	29	0.03	8	27	0.04	8	6	0.5x	0.7x
FR(5)	416	865	0.35	81	247	0.17	29	141	0.14	43	8	2.7x	1.3x
FR(7)	21k	42k	34.21	449	2k	2.28	68	456	0.82	128	10	42.1x	2.9x
FR(9)	>51k	>104k	120.00	3k	12k	29.92	129	2k	5.18	328	12	>23.2x	5.8x
FR-a(4)	24	107	0.05	33	86	0.05	1	40	0.04	17	7	1.3x	1.6x
FR-a(6)	720	4k	2.17	193	680	0.88	1	119	0.25	51	9	9.0x	3.7x
FR-a(8)	>20k	>88k	120.00	2k	5k	10.79	1	270	1.18	116	11	>101.9x	9.2x
FR-a(10)	>18k	>79k	120.00	>5k	>26k	120.00	1	517	5.09	224	13	>23.6x	>23.6x
LW(3)	6	17	0.01	3	10	0.01	1	10	0.01	3	6	0.9x	0.7x
LW(5)	120	327	0.16	5	21	0.02	1	21	0.03	8	8	7.8x	0.9x
LW(7)	6k	14k	8.96	7	36	0.06	1	36	0.07	15	10	137.8x	0.9x
LW(10)	>32k	>85k	120.00	10	66	0.32	1	66	0.31	29	13	>396.0x	1.1x
abs(2)	4	28	0.02	2	15	0.01	1	12	0.01	6	9	2.0x	1.2x
abs(3)	54	577	0.30	2	26	0.02	1	23	0.02	15	12	19.8x	1.3x
abs(4)	2k	33k	29.34	2	44	0.04	1	40	0.05	25	15	598.6x	0.8x
abs(5)	>4k	>109k	120.00	2	58	0.09	1	54	0.10	37	18	>1250.0x	0.9x
Lam(2)	37	605	0.30	30	470	0.59	26	456	0.46	12	29	0.7x	1.3x
Fib(3)	1	18	0.01	6	22	0.01	1	18	0.01	2	10	0.8x	1.0x
Fib(4)	1	43	0.02	90	250	0.18	1	43	0.03	5	18	0.7x	8.2x
Fib(5)	1	99	0.04	4k	11k	22.36	1	99	0.09	10	30	0.5x	266.2x
Fib(6)	1	228	0.13	>2k	>6k	120.00	1	228	0.63	20	50	0.2x	>192.6x
QS(8)	1	763	0.31	4k	12k	23.67	1	309	0.19	7	30	1.7x	130.8x
QS(10)	1	4k	1.47	>6k	>31k	120.00	1	607	0.50	9	38	3.0x	>243.4x
QS(13)	1	25k	14.83	>2k	>15k	120.00	1	2k	1.68	12	50	8.9x	>71.6x
QS(15)	1	99k	72.95	>826	>10k	120.00	1	3k	3.56	14	58	20.6x	>33.8x
MS(7)	1	70	0.03	2k	4k	5.12	1	68	0.06	6	26	0.5x	91.4x
MS(9)	1	121	0.06	14k	37k	116.75	1	107	0.14	8	34	0.4x	877.8x
MS(11)	1	172	0.09	>4k	>13k	120.00	1	166	0.33	15	42	0.3x	>372.7x
MS(14)	1	254	0.14	>2k	>5k	120.00	1	224	1.08	14	54	0.2x	>111.5x
DBP(5)	361	5k	2.89	32	210	0.24	4	65	0.09	5	32	35.2x	2.9x
DBP(6)	2k	21k	66.59	64	451	0.73	4	73	0.14	6	38	479.1x	5.3x
DBP(7)	>3k	>26k	120.00	128	964	2.23	5	109	0.28	7	44	>431.7x	8.0x
DBP(8)	>3k	>27k	120.00	256	3k	6.78	5	117	0.44	8	50	>275.9x	15.6x
BB(3)	11	38	0.02	20	49	0.03	5	23	0.02	5	7	1.0x	2.0x
BB(5)	80	326	0.13	252	671	0.56	17	103	0.09	9	11	1.5x	6.7x
BB(7)	580	3k	1.18	4k	10k	18.02	65	459	0.82	13	15	1.5x	22.2x
BB(8)	5k	21k	12.49	>10k	>28k	120.00	257	3k	15.87	17	19	0.8x	>7.6x
MapRed	9	162	114	118	856	2961	9	162	185	24	26	0.6x	16.0x
SDN	22	242	83	58	287	229	16	83	52	20	14	1.6x	4.4x

number of states stored, which corresponds to the number of states of the longest explored trace.

The results from the first set of benchmarks show that ODPOR^{ob}_{cs} can explore exponentially less sequences than DPOR_{cs} and ODPOR^{ob}. In most cases we obtain speedups with respect to both methods, although when the reduction in sequences is small, the overhead of the more complex context-sensitive checks of ODPOR^{ob}_{cs} does not pay off. For FR and FR-a, ODPOR^{ob}_{cs} obtains gains over both algorithms, scaling by several orders of magnitude. For LW(n), ODPOR^{ob} behaves very well, only exploring n sequences. Thus, ODPOR^{ob}_{cs} obtains similar results and the overhead is small. The same happens for abs. When compared with DPOR_{cs}, we achieve reductions of up to 4 orders of magnitude. Since most examples reach the timeout, the gains can be bigger than the ones shown.

In the second set of benchmarks DPOR_{cs} is already optimal for Fib and MS. Hence, the addition of observers has no benefit and the slower context-sensitive checks introduce a slowdown. For DBP, observers achieve important gains and the combination with context sensitivity gives further benefits. For QS, we obtain significant gains over both algorithms, although those over ODPOR^{ob} do not scale. In most benchmarks, we have been able to identify which of the extensions proposed in the paper are leading to the gains. In particular, the gains in QS are achieved due to the extension of Sec. 3. The refined context-sensitive check is fundamental for the gains achieved in FR and FR-a. Finally, the new way of inheriting the dnd sets leads to the gains of abs and DBP.

The results from the third set of benchmarks give evidence of the potential of our algorithm when applied over larger programs. For MapRed, both ODPOR$^{ob}_{cs}$ and DPOR$_{cs}$ explore 9 executions in 185 ms. and 114 ms. respectively, while ODPOR^{ob} explores 118 executions and takes almost 3 seconds, since there is no gain in using observers in this case. For SDN, ODPOR$^{ob}_{cs}$ explores 16 executions in 52 ms., whereas DPOR$_{cs}$ explores 22 executions in 83 ms. and ODPOR^{ob} 58 in 229 ms.

Regarding the memory requirements of ODPOR$^{ob}_{cs}$, the results show that $\mathbf{M_S}$ (i.e., the maximal length of the explored traces) remains low in all examples and grows linearly with the input size. The same holds for $\mathbf{M_D}$ except for FR and FR-a, where an exponential growth can be observed. As already mentioned in Sec. 3 (and also discussed in [18] for a different algorithm), there are a number of strategies to reduce the amount of information to be stored.

In summary, our experimental results show the exponential reduction that can be achieved by ODPOR$^{ob}_{cs}$, as our gains increase exponentially at least w.r.t. one of the algorithms in all examples we have considered.

6 CONCLUSIONS AND RELATED WORK

DPOR is one of the most scalable techniques used in the verification of concurrent systems. Recent work has introduced orthogonal notions of conditional independence into DPOR: DPOR$_{cs}$ [3] proposes a context-sensitive check in the current state to detect more accurately independence among processes, ODPOR^{ob} [7] proposes a finer notion of independence which is conditional to the existence of observers that read the values written by the processes. We propose a seamless integration of DPOR$_{cs}$ and ODPOR^{ob}, via two major technical extensions to DPOR$_{cs}$: (1) incorporating (and using effectively) the notion of *wakeup tree* used by ODPOR^{ob}, and (2) refining the context-sensitive check (and the sequences computed with it) to take observers into account. As shown in our experimental evaluation, the resulting algorithm achieves prunings that go beyond the combination of the individual algorithms.

Other recent approaches have considered alternative ways of refining the detection of independence. Data-Centric DPOR [8] focuses on the read-write of variables. It defines two traces to be observationally equivalent if every read event observes the same write event in both traces. In contrast, we use the notion of observability introduced by [7], which is based on observing interference of operations, not just individual writes. The equivalence relation used by Data-Centric is proven in [8] to see more traces as equivalent than the one based on Mazurkiewicz traces, which is the one used in our work and in all other variants of the DPOR algorithm of [10]. The drawback of Data-Centric is that it is optimal only for programs with acyclic communication graphs. Instead, our work is an extension of an optimal algorithm [7].

Another approach is to generate *independence constraints* (ICs), which ensure the independence of each pair of processes in the program. The work in [14, 20] generated for the first time ICs for processes with a single instruction following some predefined patterns. Recently, Constrained DPOR [5] proposed to generate ICs in a pre-phase, using an SMT solver. It later used the generated ICs within DPOR in a similar way to how our context-sensitive checks are used. In addition, it can perform another type of pruning using the notion of *transitive uniform* conditional independence –which ensures the ICs hold along the whole execution trace (and ensures uniformity as defined in [12, 15]). The extension of Constrained DPOR with observers, to the best of our knowledge, has not been studied yet. We believe the integration of wakeup trees could be done similarly to our proposal in Sec. 3, and the enhancements in Sec. 4 would be applicable also in the Constrained DPOR framework. Still, the combination of transitive uniformity and observability remains to be investigated.

An orthogonal approach to increase scalability, introduced in Quasi-Optimal POR [17], is to approximate the optimal exploration using a provided constant k. In essence, by using approximation, alternatives are computed in polynomial time, rather than making an NP-complete exploration, as in ODPOR. Another orthogonal improvement is to inspect dependencies over event chains [18].

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the Spanish MINECO project TIN2015-69175-C4-2-R, the Spanish MICINN/FEDER, UE project RTI2018-094403-B-C31, the CM project S2018/TCS-4314 and the Australian ARC project DP180100151.

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014.

[3] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter Stuckey. Context Sensitive Dynamic Partial Order Reduction. In Victor Kuncak and Rupak Majumdar, editors, *29th International Conference on Computer Aided Verification (CAV 2017)*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017.

[4] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Syco: A Systematic Testing Tool for Concurrent Objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 269–270. ACM, 2016.

[5] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained Dynamic Partial Order Reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.

[6] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. Sdn-actors: Modeling and verification of SDN programs. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 550–567, 2018.

[7] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 229–248, 2018.

[8] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *PACMPL*, 2(POPL):31:1–31:30, 2018.

[9] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.

[10] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[11] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[12] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV*, pages 438–449, 1993.

[13] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[14] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, pages 398–413, 2009.

[15] Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *TCS*, 101(2):337–359, 1992.

[16] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, pages 279–324, 1986.

[17] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2018.

[18] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 456–469, 2015.

[19] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[20] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.

Constrained Dynamic Partial Order Reduction

Elvira Albert[1], Miguel Gómez-Zamalloa[1(✉)], Miguel Isabel[1], and Albert Rubio[2]

[1] Complutense University of Madrid, Madrid, Spain
mzamalloa@fdi.ucm.es
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. The cornerstone of dynamic partial order reduction (DPOR) is the notion of *independence* that is used to decide whether each pair of concurrent events p and t are in a race and thus both $p \cdot t$ and $t \cdot p$ must be explored. We present *constrained* dynamic partial order reduction (CDPOR), an extension of the DPOR framework which is able to avoid redundant explorations based on the notion of *conditional independence*—the execution of p and t commutes only when certain *independence constraints* (ICs) are satisfied. ICs can be declared by the programmer, but importantly, we present a novel SMT-based approach to automatically synthesize ICs in a static pre-analysis. A unique feature of our approach is that we have succeeded to exploit ICs within the state-of-the-art DPOR algorithm, achieving *exponential* reductions over existing implementations.

1 Introduction

Partial Order Reduction (POR) is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps. Such equivalence class is called a Mazurkiewicz trace, and POR guarantees that it is sufficient to explore one interleaving per equivalence class. Early POR algorithms [8,10,20] relied on static over-approximations to detect possible *future* conflicts. The Dynamic-POR (DPOR) algorithm, introduced by Godefroid [9] in 2005, was a breakthrough in the area because it does not need to look at the future. It keeps track of the independence races witnessed along its execution and uses them to decide the required exploration dynamically, without the need of static approximation. DPOR is nowadays considered one of the most scalable techniques for

This work was funded partially by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish MINECO projects TIN2015-69175-C4-2-R and TIN2015-69175-C4-3-R, and by the CM project S2013/ICE-3006.

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10982, pp. 392–410, 2018.
https://doi.org/10.1007/978-3-319-96142-2_24

software verification. The key of DPOR algorithms is in the dynamic construction of two types of sets at each scheduling point: the *sleep set* that contains processes whose exploration has been proved to be redundant (and hence should not be selected), and the *backtrack set* that contains the processes that have not been proved independent with previously explored steps (and hence need to be explored). Source-DPOR (SDPOR) [1,2] improves the precision to compute backtrack sets (named *source* sets), proving optimality of the resulting algorithm for *any* number of processes w.r.t. an *unconditional independence* relation.

Challenge. When considering (S)DPOR with unconditional independence, if a pair of events is not independent in all possible executions, they are treated as potentially dependent and their interleavings explored. Unnecessary exploration can be avoided using conditional independence. E.g., two processes executing respectively the atomic instructions `if(z ≥ 0) z = x;` and `x = x + 1;` would be considered dependent even if $z \leq -1$—this is indeed an *independence constraint* (IC) for these two instructions. Conditional independence was early introduced in the context of POR [11,15]. The first algorithm that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR (CSDPOR) [3]. However, CSDPOR does not use ICs (it rather checks state equivalence dynamically during the exploration) and exploits conditional (context-sensitive) independence only *partially* to extend the sleep sets. Our challenge is twofold: (i) extend the DPOR framework to exploit ICs during the exploration in order to both reduce the backtrack sets and expand the sleep sets as much as possible, (ii) statically synthesize ICs in an automatic pre-analysis.

Contributions. The main contributions of this work can be summarized as:

1. We introduce sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework.
2. We extend the state-of-the-art DPOR algorithm with new forms of pruning (by means of expanding sleep sets and reducing backtrack sets).
3. We present an SMT-based approach to automatically synthesize ICs for *atomic blocks*, whose applicability goes beyond the DPOR context.
4. We experimentally show the exponential gains achieved by CDPOR on some typical concurrency benchmarks used in the DPOR literature before.

2 Background

In this section we introduce some notations, the basic notions on the POR theory and the state-of-the-art DPOR algorithm that we will extend in Sect. 3.

Our work is formalized for a general model of concurrent systems, in which a program is composed of *atomic blocks* of code. An atomic block can contain just one (global) statement that affects the global state, a sequence of local statements (that only read and write the local state of the process) followed by a global statement, or a block of code with possibly several global statements

but whose execution cannot interleave with other processes because it has been implemented as atomic (e.g., using locks, semaphores, etc.). Each atomic block in the program is given a unique block identifier. We use $spawn(P[ini])$ to create a new process. Depending on the programming language, P can be the name of a method and $[ini]$ initial values for the parameters, or P can be the identifier of the initial block to execute and $[ini]$ the initialization instructions, etc., in every case with mechanisms to continue the execution from one block to the following one. Notice that the use of atomic blocks in our formalization generalizes the particular case of considering atomicity at the level of single instructions.

As previous work on DPOR [1–3], we assume the state space does not contain cycles, executions have finite unbounded length and processes are deterministic (i.e., at a given time there is at most one event a process can execute). Let Σ be the set of states of the system. There is a unique initial state $s_0 \in \Sigma$. The execution of a process p is represented as a partial function $execute_p : \Sigma \mapsto \Sigma$ that moves the system from one state to a subsequent state. Each application of the function $execute_p$ represents the execution of an *atomic block* of the code that p is running, denoted as *event* (or execution step) of process p. An *execution sequence* E (also called *derivation*) of a system is a finite sequence of events of its processes starting from s_0, and it is uniquely characterized by the sequence of processes that perform steps of E. For instance, $p \cdot q \cdot q$ denotes the execution sequence that first performs one step in p, followed by two steps in q. We use ϵ to denote the empty sequence. The state of the system after E is denoted by $s_{[E]}$. The set of processes enabled in state s (i.e., that can perform an execution step from s) is denoted by $enabled(s)$.

2.1 Basics of Partial Order Reduction

An *event* e of the form (p, i) denotes the i-th occurrence of process p in an execution sequence, and $\hat{e}$ denotes the process p of event e, which is extended to sequences of events in the natural way. We write $\bar{e}$ to refer to the identifier of the atomic block of code the event e is executing. The set of events in execution sequence E is denoted by $dom(E)$. We use $e <_E e'$ to denote that event e occurs before event e' in E, s.t. $<_E$ establishes a total order between events in E, and $E \leq E'$ to denote that sequence E is a prefix of sequence E'. Let $dom_{[E]}(w)$ denote the set of events in execution sequence $E.w$ that are in sequence w, i.e., $dom(E.w) \backslash dom(E)$. If w is a single process p, we use $next_{[E]}(p)$ to denote the single event in $dom_{[E]}(p)$. If P is a set of processes, $next_{[E]}(P)$ denotes the set of $next_{[E]}(p)$ for all $p \in P$. The core concept in POR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation defines a subset of the $<_E$ total order, such that any two sequences with the same happens-before order are equivalent. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence with exactly the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. Thus, $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same happens-before relation. We use $E \simeq E'$ to denote that E and E' are linearizations of the same happens-before relation. The happens-before partial order has traditionally been defined in terms of a *dependency* relation between

Algorithm 1. (Source+Context-sensitive)+Constrained DPOR algorithm

```
 1: procedure EXPLORE(E)
 2:   if (∃p ∈ (enabled(s_[E]) \ sleep(E))) then
 3:     back(E) := {p};
 4:     while (∃p ∈ (back(E) \ sleep(E))) do
 5:       let n = next_[E](p);
 6:       for all (e ∈ dom(E) such that e ≾_{E.p} n) do
 7:         let E' = pre(E, e);
 8:         let u = dep(E, e, n);
 9:         if (¬(U⇒(I_{ē,n̄}, e, n, s_[E'.û]))) then
10:           updateBack(E, E', e, p);
11:           if C(s_[E'.û]) for some C ∈ I_{ē,n̄} then
12:             add û.p.ê to sleep(E');
13:           else
14:             updateSleepCS(E, E', e, p);
15:       sleep(E.p) := {x | x ∈ sleep(E), E ⊨ p ⋄ x}
16:                   ∪ {x | p.x ∈ sleep(E)}
17:                   ∪ {x | x ∈ sleep(E), |x| = 1, m = next_[E](x), U⇒(I_{n̄,m̄}, n, m, s_[E]))};
18:       EXPLORE(E.p);
19:       sleep(E) := sleep(E) ∪ {p};
```

the execution steps associated to those events [10]. Intuitively, two steps p and q are *dependent* if there is at least one execution sequence E for which they do not commute, either because (i) one *enables* the other (i.e., the execution of p leads to introducing q, or viceversa), or because (ii) $s_{[E.p.q]} \neq s_{[E.q.p]}$. We define $dep(E, e, n)$ as the subsequence containing all events e' in E that occur after e and happen-before n in $E.p$ (i.e., $e <_E e'$ and $e' \rightarrow_{E.p} n$). The unconditional dependency relation is used for defining the concept of a *race* between two events. Event e is said to be in race with event e' in execution E, if the events belong to different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent", i.e. there exists an equivalent execution sequence $E' \simeq E$ where the two events are adjacent. We write $e \precsim_E e'$ to denote that e is in race with e' and that the race can be reversed (i.e., the events can be executed in reverse order). POR algorithms use this relation to reduce the number of equivalent execution sequences explored, with SDPOR ensuring that only one execution sequence in each equivalence class is explored.

2.2 State-of-the-Art DPOR with Unconditional Independence

Algorithm 1 shows the state-of-the-art DPOR algorithm –based on the SDPOR algorithm of [1,2],[1] which in turn is based on the original DPOR algorithm of [9]. We refer to this algorithm as DPOR in what follows. The context-sensitive extension of CSDPOR [3] (lines 14 and 16) and our extension highlighted in blue

[1] The extension to support *wake-up trees* [2] is deliberately not included to simplify the presentation.

(lines 8–10, 11–13 and 17) should be ignored by now and will be described in Sect. 3.

The algorithm carries out a depth-first exploration of the execution tree using POR receiving as parameter a derivation E (initially empty). Essentially, it dynamically finds reversible races and is able to backtrack at the appropriate scheduling points to reverse them. For this purpose, it keeps two sets at every prefix E' of E: $back(E')$ with the set of processes that must be explored from E', and, $sleep(E')$ with the set of sequences of processes that previous executions have determined do not need to be explored from E'. Note that in the original DPOR the sleep set contained only single processes, but in later improvements sequences of processes are added, so our description considers this general case. The algorithm starts by selecting any process p that is enabled by the state reached after executing E and is not already in $sleep(E)$. If it does not find any such process p, it stops. Otherwise, after setting $back(E) = \{p\}$ to start the search, it explores every element in $back(E)$ that is not in $sleep(E)$. The backtrack set of E might grow as the loop progresses (due to later executions of line 10). For each such p, DPOR performs two phases: race detection (lines 6, 7 and 10) and state exploration (lines 15, 18 and 19). The race detection starts by finding all events e in $dom(E)$ such that $e \precsim_{E.p} n$, where n is the event being selected (see line 5). For each such e, it sets E' to $pre(E, e)$, i.e., to be the prefix of E up to, but not including e. Procedure *updateBack* modifies $back(E')$ in order to ensure that the race between e and n is reversed. The source-set extension of [1,2] detects cases where there is no need to modify $back(E')$ –this is done within procedure *updateBack* whose code is not shown because it is not affected by our extension. After this, the algorithm continues with the state exploration phase for $E.p$, by retaining in its sleep set any element x in $sleep(E)$ whose events in $E.p$ are independent of the next event of p in E (denoted as $E \models p \diamond x$), i.e., any x such that $next_{[E]}(p)$ would not happen-before any event in $dom(E.p.x) \backslash dom(E.p)$. Then, the algorithm explores $E.p$, and finally it adds p to $sleep(E)$ to ensure that, when backtracking on E, p is not selected until a dependent event with it is selected. All versions of the DPOR algorithm (except [3]) rely on the unconditional (or context-insensitive) dependency relation. This relation has to be over-approximated, usually by requiring that global variables accessed by one execution step are not modified by the other.

Example 1. Consider the example in Fig. 1 with 3 processes p, q, r containing a single atomic block. Since all processes have a single event, by abuse of notation, we refer to events by their process name throughout all examples in the paper. Relying on the usual over-approximation of dependency all three pairs of events are dependent. Therefore, starting with one instance per process, the algorithm has to explore 6 execution sequences, each with a different happens-before relation. The tree, including the dotted and dashed fragments, shows the exploration from the initial state $z = -2$, $x = -2$. The value of variable z is shown in brackets at each state. Essentially, in all states of the form $E.e$, the algorithm always finds a reversible race between the next event of the current selected process (p, q or r) and e, and adds it to $back(E)$. Also, when backtracking on E, none

of the elements in $sleep(E)$ is propagated down, since all events are considered dependent. In the best case, considering an exact (yet unconditional) dependency relation which realizes that events p and r are independent, the algorithm will make the following reductions. In state 6, p and r will not be in race and hence p will not be added to $back(q)$. This avoids exploring the sequence $p.r$ from 5. When backtracking on state 0 with r, where $sleep(\epsilon) = \{p, q\}$, p will be propagated down to $sleep(r)$ since $\epsilon \models r \diamond p$, hence avoiding the exploration of $p.q$ from 8. Thus, the algorithm will explore 4 sequences.

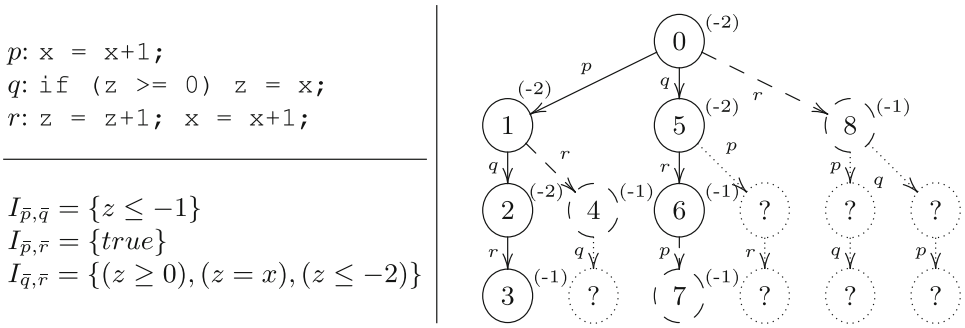


Fig. 1. Left: code of working example (up) and ICs (down). Right: execution tree starting from $z = -2$, $x = -2$. Full tree computed by SDPOR, dotted fragment not computed by CSDPOR, and, dashed+dotted fragment not computed by CDPOR.

3 DPOR with Conditional Independence

Our aim in CDPOR is twofold: (1) provide techniques to both infer and soundly check conditional independence, and (2) be able to exploit them at *all* points of the DPOR algorithm where dependencies are used. Section 3.1 reviews the notions of conditional independence and ICs, and introduces a first type of check where ICs can be directly used in the DPOR algorithm. Section 3.2 illustrates why ICs cannot be used at the remaining independence check points in the algorithm, and introduces sufficient conditions to soundly exploit them at those points. Finally, Sect. 3.3 presents the CDPOR algorithm that includes all types of checks.

3.1 Using Precomputed ICs Directly Within DPOR

Conditional independence consists in checking independence at the given state.

Definition 1 (conditional independence). *Two events α and β are* independent *in state S, written $indep(\alpha, \beta, S)$ if (i1) none of them* enables *the other from S; and, (i2) if they are both enabled in S, then $S \xrightarrow{\alpha\cdot\beta} S'$ and $S \xrightarrow{\beta\cdot\alpha} S'$.*

The use of conditional independence in the POR theory was firstly studied in [15], and it has been partially applied within the DPOR algorithm in CSDPOR [3]. Function *updateSleepCS* at line 14 and the modification of *sleep* at 16 encapsulate this partial application of CSDPOR (the code of *updateSleepCS* is not shown because it is not affected by our extension). Intuitively, *updateSleepCS* works as follows: when a reversible race is found in the current sequence being explored, it builds an *alternative* sequence which corresponds to the reverse race, and then checks whether the states reached after running the two sequences are the same. If they are, it adds the alternative sequence to the corresponding *sleep* set so that this sequence is not fully explored when backtracking. Therefore, sleep sets can contain *sequences* of events which can be propagated down via the rule of line 16 (i.e., if the event being explored is the head of a sequence in the sleep set, then the tail of the sequence is propagated down). In essence, the technique to check $(i2)$ in Definition 1 in CSDPOR consists in checking state equivalence with an alternative sequence in the current state (hence it is conditional) and, if the check succeeds, it is exploited in the *sleep* set only (and not in the *backtrack* set).

Example 2. Let us explain the intuition behind the reductions that CSDPOR is able to achieve w.r.t. unconditional independence-based DPOR on the example. In state 1, when the algorithm selects q and detects the reversible race between q and p, it computes the alternative sequence $q.p$ and realizes that $s_{[p.q]} = s_{[q.p]}$, and hence adds $p.q$ to $sleep(\epsilon)$. Similarly, in state 2, it computes $p.r.q$ and realizes that $s_{[p.q.r]} = s_{[p.r.q]}$ adding $r.q$ to $sleep(p)$. Besides these two alternative sequences, it computes two more. Overall, CSDPOR explores 2 complete sequences ($p.q.r$ and $q.r.p$) and 13 states (the 9 states shown, plus 4 additional states to compute the alternative sequences).

Instead of computing state equivalence to check $(i2)$ as in [3], our approach assumes precomputed *independence constraints* (ICs) for all pairs of atomic blocks in the program. ICs will be evaluated at the appropriate state to determine the independence between pairs of concurrent events executing such atomic blocks.

Definition 2 (ICs). *Consider two events α and β that execute, respectively, the atomic blocks $\bar{\alpha}$ and $\bar{\beta}$. The* independence constraints $I_{\bar{\alpha},\bar{\beta}}$ *are a set of boolean expressions (constraints) on the variables accessed by α and β (including local and global variables) s.t., if some constraint C in $I_{\bar{\alpha},\bar{\beta}}$ holds in state S, written $C(S)$, then condition (i2) of $indep(\alpha, \beta, S)$ holds.*

Our first contribution is in lines 11–13 where ICs are used within DPOR as follows. Before executing *updateSleepCS* at line 14, we check if some constraint in $I_{\bar{e},\bar{n}}$ holds in the state $s_{[E'.\hat{u}]}$, by building the sequence $E'.\hat{u}$, where $u = dep(E, e, n)$. Only if our check fails we proceed to execute *updateSleepCS*. The advantages of our check w.r.t. *updateSleepCS* are: (1) the alternative execution sequence built by *updateSleepCS* is strictly longer than ours and hence more states will be explored, and (2) *updateSleepCS* must check state equivalence

while we evaluate boolean expressions. Yet, because our IC is an approximation, if we fail to prove independence we can still use *updateSleepCS*.

Example 3. Consider the ICs in Fig. 1 (down left), which provide the constraints ensuring the independence of each pair of atomic blocks, and whose synthesis is explained in Sect. 4.1. In the exploration of the example, when the algorithm detects the reversible race between q and p in state 1, instead of computing $q.p$ and then comparing $s_{[p.q]} = s_{[q.p]}$ as in CSDPOR, we would just check the constraint in $I_{\bar{p},\bar{q}}$ at state ϵ, i.e., in $z = -2$ (line 11), and since it succeeds, $q.p$ is added to $sleep(\epsilon)$. The same happens at states 2, again at 1 (when backtracking with r), and 5. This way we avoid the exploration of the additional 4 states due to the computation of the alternative sequences in Example 2 (namely $q.p$, $r.p$ and $r.q$ from state 0, and $r.q$ from 1). The algorithm is however still exploring many redundant derivations, namely states 4, 5, 6, 7 and 8.

3.2 Transitive Uniformity: How to Further Exploit ICs Within DPOR

The challenge now is to use ICs, and therefore conditional independence, at the remaining dependency checks performed by the DPOR algorithm, and most importantly, for the race detection (line 6). In the example, that would avoid the addition of q and r to $back(\epsilon)$ and r to $back(p)$, and hence would make the algorithm only explore the sequence $p.q.r$. Although that can be done in our example, it is unsound in general as the following counter-example illustrates.

Example 4. Consider the same example but starting from the initial state $z = -1$, $x = -2$. During the exploration of the first sequence $p.q.r$, the algorithm will not find any race since p and q are independent in $z = -1$, q and r are independent in $z = x = -1$, and, p and r are always independent. Therefore, no more sequences than $p.q.r$ with final result $z = 0$ will be explored. There is however a non-equivalent sequence, $r.q.p$, which leads to a different final state $z = -1$.

The problem of using conditional independence within the POR theory was already identified by Katz and Peled [15]. Essentially, the main idea of POR is that the different linearizations of a partial order yield equivalent executions that can be obtained by swapping adjacent independent events. However, this is no longer true with conditional dependency. In Example 4, using conditional independence, the partial order of the explored derivation $p.q.r$ would be empty, which means there would be 6 possible linearizations. However $r.q.p$ is not equivalent to $p.q.r$ since q and p are dependent in $s_{[r]}$, i.e., when $z = 0$. An extra condition, called *uniformity*, is proposed in [15] to allow using conditional independence within the POR theory. Intuitively, *uniform independence* adds a condition to Definition 1 to ensure that independence holds at all successor states for those events that are enabled and are *uniformly independent* with the two events whose independence is being proved. While this notion can be checked *a posteriori* in a given exploration, it is unclear how it could be applied in a

dynamic setting where decisions are made *a priori*. Here we propose a weaker notion of uniformity, called *transitive uniformity*, for which we have been able to prove that the *dynamic*-POR framework is sound. The difference with [15] is that our extra condition ensures that independence holds at all successor states for *all* events that are enabled, which is thus a superset of the events considered in [15]. We notice that the general happens-before definition of [1,2] does not capture our transitive uniform conditional independence below (namely property seven of [1,2] does not hold), hence CDPOR cannot be seen as an instance of SDPOR but rather as an extension.

Definition 3. *The* transitive uniform *conditional independence relation, written unif*(α, β, S)*, fulfills (i1) and (i2) and, (i3) unif*$(\alpha, \beta, S_\gamma)$ *holds for all* $\gamma \notin \{\alpha, \beta\}$ *enabled in* S*, where* S_γ *is defined by* $S \xrightarrow{\gamma} S_\gamma$*.*

During the exploration of the sequence $p.q.r$ in Example 4, the algorithm will now find a reversible race between p and q, since the independence is not transitively uniform in $z = -1, x = -2$. Namely, *(i3)* does not hold since r is enabled and we have $x = -1$ and $z = 0$ in $s_{[r]}$, which implies $\neg$*unif*$(p, q, s_{[r]})$ (*(i2)* does not hold).

We now introduce sufficient conditions for transitive uniformity that can be precomputed statically, and efficiently checked, in our dynamic algorithm. Condition *(i1)* is computed dynamically as usual during the exploration simply storing enabling dependencies. Condition *(i2)* is provided by the ICs. Our sufficient conditions to ensure *(i3)* are as follows. For each atomic block b, we precompute *statically* (before executing DPOR) the set $W(b)$ of the global variables that can be modified by the full execution of b, i.e., by an instruction in b or by any other block called from, or enabled by, b (transitively). To this end, we do a simple analysis which consists in: (1) First we build the call graph for the program to establish the calling relationships between the blocks in the program. Note that when we find a process creation instruction $spawn(P[ini])$ we have a calling relationship between the block in which the spawn instruction appears and P. (2) We obtain (by a fixed point computation) the largest relation fulfilling that g belongs to $W(b)$ if either g is *modified* by an instruction in b or g belongs to $W(c)$ for some block c called from b. This computation can be done with different levels of precision, and it is well-studied in the static analysis field [18]. We let $G(C)$ be the set of global variables evaluated on constraint C in I.

Definition 4 (sufficient condition for transitive uniformity, $U_\Rightarrow$). *Let* E *be a sequence,* I *a set of constraints,* α *and* β *be two events enabled in* $s_{[E]}$*, and* $T = next_{[E]}(enabled(s_{[E]})) \setminus \{\alpha, \beta\}$*, we define* $U_\Rightarrow(I, \alpha, \beta, s_{[E]}) \equiv \exists C \in I : C(s_{[E]}) \wedge ((G(C) \cap \bigcup_{t \in T} W(\bar{t})) = \emptyset)$*.*

Intuitively, our sufficient condition ensures transitive uniformity by checking that the global variables involved in the constraint C of the IC used to ensure the uniformity condition are not modified by other enabled events in the state.

Theorem 1. *Given a sequence* E *and two events* α *and* β *enabled in* $s_{[E]}$*, we have that* $U_\Rightarrow(I_{\bar{\alpha},\bar{\beta}}, \alpha, \beta, s_{[E]}) \Rightarrow$ *unif*$(\alpha, \beta, s_{[E]})$*.*

3.3 The Constrained DPOR Algorithm

The code highlighted in blue in Algorithm 1 provides the extension to apply conditional independence within DPOR. In addition to the pruning explained in Sect. 3.1, it achieves two further types of pruning:

1. *Back-set reduction.* The race detection is strengthened with an extra condition (line 9) so that e and n (the next event of p) are in race only if they are not conditionally independent in state $s_{[E'.u]}$ (using our sufficient condition above). Here u is the sub-sequence of events of E that occur after e and "happen-before" n. This way the conditional independence is evaluated in the state after the shortest subsequence so that the events are adjacent in an equivalent execution sequence.
2. *Sleep-set extension.* An extra condition to propagate down elements in the sleep set is added (line 17) s.t. a sequence x, with just one process, is propagated if its corresponding event is conditionally independent of n in $s_{[E]}$.

It is important to note also that the inferred conditional independencies are recorded in the happens-before relation to be later re-used for subsequent computations of the $\precsim$ and *dep* definitions.

Example 5. Let us describe the exploration for the example in Fig. 1 using our CDPOR. At state 1, the algorithm checks whether p and q are in race. $U_{\Rightarrow}(I_{\bar{p},\bar{q}}, p, q, S)$ does not hold in $z = -2$ since, although $(z \leq -1) \in I_{\bar{p},\bar{q}}$ holds, we have that $G(z \leq -1) \cap W(r) = \{z\} \neq \emptyset$. Process q is hence added to $back(\epsilon)$. On the other hand, since $(z \leq -1) \in I_{\bar{p},\bar{q}}$ holds in $z = -2$ (line 11), $q.p$ is added to $sleep(\epsilon)$ (line 12). At state 2 the algorithm checks the possible race between q and r after executing p. This time the transitive uniformity of the independence of q and r holds since $(z \leq -2) \in I_{\bar{q},\bar{r}}$ holds, and there are no enabled events out of $\{q, r\}$. Our algorithm therefore avoids the addition of r to $back(p)$ (pruning 1 above). The algorithm also checks the possible race between p and r in $z = -2$. Again, $true \in I_{\bar{p},\bar{r}}$ holds and is uniform since $G(true) = \emptyset$ (pruning 1). The algorithm finishes the exploration of sequence $p.q.r$ and then backtracks with q at state 0. At state 5 the algorithm selects process r (p is in the sleep set of 5 since it is propagated down from the $q.p$ in $sleep(\epsilon)$). It then checks the possible race between q and r, which is again discarded (pruning 1), since transitive uniformity of the independence of q and r can be proved: we have that $(z \leq -2) \in I_{\bar{q},\bar{r}}$ holds in $z = -2$ and $W(p) \cap G(z \leq -2) = \emptyset$, where p is the only enabled event out of $\{q, r\}$ and $W(p) = \{x\}$. This avoids adding r to $back(\epsilon)$. Finally, at state 5, p is propagated down in the new sleep set (pruning 2), since as before $true \in I_{\bar{p},\bar{r}}$ ensures transitive uniformity. The exploration therefore finishes at state 6.

Overall, on our working example, CDPOR has been able to explore only one complete sequence $p.q.r$ and the partial sequence $q.r$ (a total of 6 states). The latter one could be avoided if a more precise sufficient condition for uniformity is provided which, in particular, is able to detect that the independence of p and q in ϵ is transitive uniform, i.e., it still holds after r (even if r writes variable z).

Theorem 2 (soundness). *For each Mazurkiewicz trace T defined by the happens before relation, Explore$(\epsilon, \emptyset)$ in Algorithm 1 explores a complete execution sequence T' that reaches the same final state as T.*

4 Automatic Generation of ICs Using SMT

Generating ICs amounts to proving (conditional) program equivalence w.r.t. the global memory. While the problem is very hard in general, proving equivalence of smaller blocks of code becomes more tractable. This section introduces a novel SMT-based approach to synthesize ICs between pairs of atomic blocks of code. Our ICs can be used within any transformation or analysis tool –beyond DPOR– which can gain accuracy or efficiency by knowing that fragments of code (conditionally) commute. Section 4.1 first describes the inference for basic blocks; Sect. 4.2 extends it to handle process creation and Sect. 4.3 outlines other extensions, like loops, method invocations and data structures.

4.1 The Basic Inference

In this section we consider blocks of code containing conditional statements and assignments using linear integer arithmetic (LIA) expressions. The first step to carry out the inference is to transform q and r into two respective *deterministic* Transition Systems (TSs), T_q and T_r (note that q and r are assumed to be deterministic), and compose them in both reverse orders $T_{q \cdot r}$ and $T_{r \cdot q}$. Consider r and q in Fig. 1 whose associated TSs are (primed variables represent the final value of the variables):

$$T_q : \begin{array}{l} z \geq 0 \rightarrow z' = x; \\ z < 0 \rightarrow z' = z; \end{array} \qquad T_r : true \rightarrow x' = x + 1, z' = z + 1;$$

The code to be analyzed is the composition of T_q and T_r in both orders:

$$T_{q \cdot r}: \begin{array}{l} z \geq 0 \rightarrow x' = x + 1, z' = x + 1; \\ z < 0 \rightarrow x' = x + 1, z' = z + 1; \end{array} \qquad T_{r \cdot q}: \begin{array}{l} z \geq -1 \rightarrow x' = x + 1, z' = x + 1; \\ z < -1 \rightarrow x' = x + 1, z' = z + 1; \end{array}$$

In what follows we denote by $T_{a \cdot b}$ the deterministic TS obtained from the concatenation of the blocks a and b, such that all variables are assigned in one instruction using parallel assignment. We let $A \mid_G$ be the restriction to the global memory of the assignments in A (i.e., ignoring the effect on local variables). The following definition provides an SMT formula over LIA (a boolean formula where the atoms are equalities and inequalities over linear integer arithmetic expressions) which encodes the independence between the two blocks.

Definition 5 (IC generation). *Let us consider two atomic blocks q and r and a global memory G and let $C_i \rightarrow A_i$ (resp. $C'_j \rightarrow A'_j$) be the transitions in $T_{q \cdot r}$ (resp. $T_{r \cdot q}$). We obtain $F_{q,r}$ as the SMT formula: $\bigvee_{i,j}(C_i \wedge C'_j \wedge A_i \mid_G = A'_j \mid_G)$.*

Intuitively, the SMT encoding in the above definition has as solutions all those states where both a condition C_i of a transition in $T_{q \cdot r}$ and C'_j of a transition in $T_{r \cdot q}$ hold (and hence are compatible) and the final global state after executing all instructions in the two transitions (denoted A_i and A'_j) remains the same.

Next, we generate the constraints of the independence condition $I_{q,r}$ by obtaining a compact representation of all models over linear arithmetic atoms (computed by an *allSAT* SMT solver) satisfying $F_{q,r}$. In particular, we add a constraint in $I_{q,r}$ for every obtained model.

Example 6. In the example, we have the TS with conditions and assignments:

$$\begin{array}{rl|rl} T_{q\cdot r}\colon & C_1{:}z \geq 0 \quad A_1{:}x' = x+1, z' = x+1 & T_{r\cdot q}\colon & C'_1{:}z \geq -1 \quad A'_1{:}x' = x+1, z' = x+1 \\ & C_2{:}z < 0 \quad A_2{:}x' = x+1, z' = z+1 & & C'_2{:}z < -1 \quad A'_2{:}x' = x+1, z' = z+1 \end{array}$$

and we obtain a set with three constraints $I_{q,r} = \{(z \geq 0), (z = x), (z < -1)\}$ by computing all models satisfying the following resulting formula:

$$\begin{array}{l} (z \geq 0 \wedge z \geq -1 \wedge x+1 = x+1 \wedge x+1 = x+1) \vee \\ (z \geq 0 \wedge z < -1 \wedge x+1 = x+1 \wedge x+1 = z+1) \vee \\ (z < 0 \wedge z \geq -1 \wedge x+1 = x+1 \wedge z+1 = x+1) \vee \\ (z < 0 \wedge z < -1 \wedge x+1 = x+1 \wedge z+1 = z+1) \end{array}$$

The second conjunction is unsatisfiable since there is no model with both C_1 and C'_2. On the other hand, the equalities of the first and the last conjunctions always hold, which give us the constraints $z \geq 0$ and $z \leq -2$. Finally, all equalities hold when $x = z$, which give us the third constraint as a result for our SMT encoding.

Note that, as in this case $F_{q,r}$ describes not only a sufficient but also a necessary condition for independence, the obtained constraints IC are also a sufficient and necessary conditions for independence. This allows removing line 14 in the algorithm, since the context-sensitive check will fail if line 11 does. However, the next extensions do not ensure that the generated ICs are necessary conditions.

4.2 IC for Blocks with Process Creation

Consider the following two methods whose body constitutes an atomic block (e.g., the lock is taken at the method start and released at the return). They are inspired by a highly concurrent computation for the Fibonacci used in the experiments. Variables `nr` and `r` are global to all processes:

```
fib(int v) {                          res(int v) {
   if (v≤1) {spawn(res(v));}             if (nr>0) {nr=0; r=v; }
   else {spawn(fib(v-1));                else {spawn(res(r+v));
         spawn(fib(v-2));}                     r=0;nr=1;}
}                                     }
```

We now want to infer $I_{\mathsf{fib(v)},\mathsf{fib(v_1)}}$, $I_{\mathsf{fib(v)},\mathsf{res(v_1)}}$, $I_{\mathsf{res(v)},\mathsf{res(v_1)}}$. The first step is to obtain, for each block r, a *TS with uninterpreted functions*, denoted TS^u_r, in which transitions are of the form $C \rightarrow (A, S)$ where A are the parallel assignments as in Sect. 4.1, and S is a multiset containing calls to fresh *uninterpreted* functions associated to the processes spawned within the transition (i.e., a process creation $spawn(P)$ is associated to an uninterpreted function $spawn\_P$).

$$T^u_{\mathsf{fib}}\colon\ v \le 1 \to (skip, \{spawn\_res(v)\})$$
$$\quad\ v > 1 \to (skip, \{spawn\_fib(v-1), spawn\_fib(v-2)\}$$
$$T^u_{\mathsf{res}}\colon\ nr \ge 0 \to (nr' = 0, r' = v, \{\})$$
$$\quad\ nr < 0 \to (nr' = 1, r' = 0, \{spawn\_res(r+v)\}$$

The following definition extends Definition 5 to handle process creation. Intuitively, it associates a fresh variable to each different element in the multisets (mapping P' below) and enforces equality among the multisets.

Definition 6 (IC generation with process creation). *Let us consider $TS^u_{r\cdot q}$ and $TS^u_{q\cdot r}$. We define $P = \{\cup s \mid s \in S,\ with\ C \to (A, S) \in TS^u_{r\cdot q} \cup TS^u_{q\cdot r}\}$. Let P' be a mapping from the elements in P to fresh variables, and $P'(S)$ be the replacement of the elements in the multiset S applying the mapping P'. Let $C_i \to (A_i, S_i)$ (resp. $C'_j \to (A'_j, S'_j)$) be the transitions in $TS^u_{q\cdot r}$ (resp. $TS^u_{r\cdot q}$). We obtain $F_{q,r}$ as the SMT formula: $\bigvee_{i,j}(C_i \wedge C'_j \wedge A_i \mid_G = A'_j \mid_G \wedge P'(S_i) \equiv P'(S'_j))$.*

For simplicity and efficiency, we consider that $\equiv$ corresponds to the syntactic equality of the multisets. However, in order to improve the precision of the encoding we apply P' to S_i and S_j replacing two process creations by the same variable if they are equal modulo associativity and commutativity (AC) of arithmetic operators and after substituting the equalities already imposed by $A_i \mid_G = A'_j$ (see example below). A more precise treatment can be achieved by using equality with uninterpreted functions (EUF) to compare the multisets of processes.

Example 7. Let us show how we apply the above definition to infer $I_{\mathsf{res(v)},\mathsf{res(v_1)}}$. We first build $T_{\mathsf{res(v)}\cdot\mathsf{res(v_1)}}$ from $T_{\mathsf{res(v)}}$ by composing it with itself:

$$nr \le 0 \to (nr' = 0, r' = v_1, \{spawn\_res(r{+}v)\})$$
$$nr > 0 \to (nr' = 1, r' = 0, \{spawn\_res(v{+}v_1)\})$$

and $T_{\mathsf{res(v_1)}\cdot\mathsf{res(v)}}$ which is like the one above but exchanging v and $\mathsf{v_1}$. Next, we define $P' = \{spawn\_res(r+v) \mapsto x_1, spawn\_res(v+v_1) \mapsto x_2, spawn\_res(r+v_1) \mapsto x_3, spawn\_res(v_1{+}v) \mapsto x_4\}$ and apply it with the improvement described above

$$(nr \le 0 \wedge nr \le 0 \wedge 0 = 0 \wedge v = v_1 \wedge \{x_1\} = \{x_1\}) \vee$$
$$(nr \le 0 \wedge nr > 0 \wedge 0 = 1 \wedge v_1 = 0 \wedge \{x_1\} = \{x_4\}) \vee$$
$$(nr > 0 \wedge nr \le 0 \wedge 1 = 0 \wedge 0 = v \wedge \{x_2\} = \{x_3\}) \vee$$
$$(nr > 0 \wedge nr > 0 \wedge 1 = 1 \wedge 0 = 0 \wedge \{x_2\} = \{x_2\})$$

Note that the second and the third conjunction are unfeasible and hence can be removed from the formula. In the first one $spawn\_res(r+v_1)$ is replaced by x_1 (instead of x_3) since we can substitute v_1 by v as $v = v_1$ is imposed in the conjunction and in the fourth one $spawn\_res(v_1+v)$ is replaced by x_2 (instead of x_4) since it is equal modulo AC to $spawn\_res(v+v_1)$. Then we finally have

$$(nr \le 0 \wedge nr \le 0 \wedge 0 = 0 \wedge v = v_1) \ \vee \ (nr > 0 \wedge nr > 0 \wedge 1 = 1 \wedge 0 = 0)$$

As before, $I_{\mathsf{res(v)},\mathsf{res}(v_1)} = \{(nr > 0), (v = v_1)\}$ is then obtained by computing all satisfying models. In the same way we obtain $I_{\mathsf{fib(v)},\mathsf{res(v_1)}} = I_{\mathsf{fib(v)},\mathsf{fib(v_1)}} = \{true\}$.

The following theorem states the soundness of the inference of ICs, that holds by construction of the SMT formula.

Theorem 3 (soundness of independence conditions). *Given the assumptions in Definition 6, if* $\exists C \in I_{r,q}$ *s.t.* $C(S)$ *holds, then* $S \xrightarrow{r \cdot q} S'$ *and* $S \xrightarrow{q \cdot r} S'$.

We will also get a necessary condition in those instances where the use of syntactic equality modulo AC on the multisets of created processes (as described above) is not loosing precision. This can be checked when building the encoding.

4.3 Other Extensions

We abstract loops from the code of the blocks so that we can handle them as uninterpreted functions similarly to Definition 6. Basically, for each loop, we generate as many uninterpreted functions as variables it modifies (excluding local variables of the loop) plus one to express all processes created inside the loop. The functions have as arguments the variables accessed by the loop (again excluding local variables). This transformation allows us to represent that each variable might be affected by the execution of the loop over some parameters, and then check in the reverse trace whether we get to the loop over the same parameters.

Definition 7 (loop extraction for IC generation). *Let us consider a loop* L *that accesses* $x_1, \ldots, x_n$ *variables and modifies* $y_1, \ldots, y_m$ *variables (excluding local loop variables) and let* $l_1, \ldots, l_{m+1}$ *be fresh function symbol names. We replace* L *by the following code:*

$$x'_1 = x_1; \ldots;\ x'_n = x_n;\ y_1 = l_1(x'_1, ..., x'_n); \ldots;\ y_m = l_m(x'_1, ..., x'_n);$$
$$spawn(f_{m+1}(x'_1, ..., x'_n)); \quad (only\ if\ there\ are\ spawn\ operations\ inside\ the\ loop)$$

Existing dependency analysis can be used to infer the subset of $x_1, \ldots, x_n$ that affects each y_i, achieving more precision with a small pre-computation overhead.

The treatment of method invocations (or function calls) to be executed atomically within the considered blocks can be done analogously to loops by introducing one fresh function for every (non-local) variable that is modified within the method call and one more for the result. The parameters of these new functions are the original ones plus one for each accessed (non-local) variable. After the transformations for both loops and calls described above, we have TSs with function calls that are treated as uninterpreted functions in a similar way to Definition 6. However these functions can now occur in the conditions and the assignments of the TS. To handle them, we use again a mapping P'' to remove all function calls from the TS and replace them by fresh integer variables. After that the encoding is like in Definition 6, and we obtain an SMT formula over LIA, which is again sent to the allSAT SMT solver. Once we have obtained the models we replace back the introduced fresh variables by the function calls using the mapping P''. Several simplifications on equalities involving function calls can be done before and after invoking the solver to improve the result. As a

final remark, data structures like lists or maps have been handled by expressing their uses as function calls, hence obtaining constraints that include conditions on them.

5 Experiments

In this section we report on experimental results that compare the performance of three DPOR algorithms: SDPOR [1,2], CSDPOR [3] and our proposal CDPOR. We have implemented and experimentally evaluated our method within the SYCO tool [3], a systematic testing tool for message-passing concurrent programs. SYCO can be used online through its web interface available at http://costa.fdi.ucm.es/syco. To generate the ICs, SYCO calls a new feature of the VeryMax program analyzer [6] which uses Barcelogic [5] as SMT solver. As benchmarks, we have borrowed the examples from [3] (available online from the previous url) that were used to compare SDPOR with CSDPOR. They are classical concurrent applications: several concurrent sorting algorithms (QS, MS, PS), concurrent Fibonacci Fib, distributed workers Pi, a concurrent registration system Reg and database DBP, and a consumer producer interaction BB. These benchmarks feature the typical concurrent programming methodology in which computations are split into smaller atomic subcomputations which concurrently interleave their executions, and which work on the same shared data. Therefore, the concurrent processes are highly interfering, and both inferring ICs and applying DPOR algorithms on them becomes challenging.

We have executed each benchmark with size increasing input parameters. A timeout of 60 s is used and, when reached, we write $>X$ to indicate that for the corresponding measure we encountered X units up to that point (i.e., it is at least X). Table 1 shows the results of the executions for 6 different inputs. Column Tr shows the number of traces, S the number of states that the algorithms explore, and T the time in *sec* it takes to compute them. For CDPOR, we also show the time T^{smt} of inferring the ICs (since the inference is performed once for all executions, it is only shown in the first row). Times are obtained on an Intel(R) Core(TM) i7 CPU at 2.5 GHz with 8 GB of RAM (Linux Kernel 5.4.0). Columns $G^{\mathbf{s}}$ and $G^{\mathbf{cs}}$ show the time speedup of CDPOR over SDPOR and CSDPOR, respectively, computed by dividing each respective T by the time T of CDPOR. Column $G^{\mathbf{smt}}$ shows the time speedup over CSDPOR including T^{smt} in the time of CDPOR. We can see from the speedups that the gains of CDPOR increase exponentially in all examples with the size of the input. When compared with CSDPOR, we achieve reductions up to 4 orders of magnitude for the largest inputs on which CSDPOR terminates (e.g., Pi, QS). It is important to highlight that the number of non-unitary sequences stored in sleep sets is 0 in every benchmark except in BB for which it remains quite low (namely for BB(11) the peak is 22).

W.r.t. SDPOR, we achieve reductions of 4 orders of magnitude even for smaller inputs for which SDPOR terminates (e.g., PS). Note that since most examples reach the timeout, the gains are at least the ones we show, thus the

Table 1. Experimental evaluation

	SDPOR			CSDPOR			CDPOR				Speed-up		
Bench.	**Tr**	**S**	**T**	**Tr**	**S**	**T**	**Tr**	**S**	**T**	$\mathbf{T^{smt}}$	$\mathbf{G^{s}}$	$\mathbf{G^{cs}}$	$\mathbf{G^{smt}}$
Fib(6)	3k	26k	7.7	1	244	0.1	1	50	0.03	0.12	366	4	0.6
Fib(7)	>13k	>160k	60.0	1	551	0.3	1	82	0.05		>1364	6	1.4
Fib(8)	>8k	>101k	60.0	1	2k	0.7	1	134	0.12		>527	6	3.0
Fib(9)	>4k	>51k	60.0	1	3k	2.8	1	218	0.25		>242	12	7.5
Fib(10)	>2k	>27k	60.0	1	8k	11.5	1	354	0.69		>88	17	14.3
Fib(14)	>10	>3k	60.0	>1	>4k	60.0	1	3k	42.67		>2	>2	>1.5
QS(10)	512	9k	2.6	1	4k	1.0	1	38	0.02	11.99	199	71	0.1
QS(13)	5k	91k	29.5	1	29k	7.9	1	50	0.03		1474	395	0.7
QS(15)	>7k	>157k	60.0	1	115k	42.6	1	58	0.05		>1500	1064	3.6
QS(20)	>4k	>98k	60.0	>1	>148k	60.0	1	78	0.04		>1539	>1539	>5.0
QS(25)	>3k	>96k	60.0	>1	>133k	60.0	1	98	0.06		>1017	>1017	>5.0
QS(200)	>5	>2k	60.0	>1	>87k	60.0	1	798	4.45		>14	>14	>3.7
MS(10)	628	7k	2.9	1	187	0.1	1	42	0.02	0.12	175	6	0.7
MS(30)	>6k	>55k	60.0	1	974	1.0	1	118	0.13		>484	8	4.0
MS(65)	>2k	>16k	60.0	1	3k	3.5	1	258	0.47		>131	8	6.1
MS(100)	>2k	>15k	60.0	>1	>19k	60.0	1	398	0.97		>63	>63	>55.6
MS(150)	>2k	>21k	60.0	>1	>18k	60.0	1	598	2.21		>28	>28	>26.0
MS(220)	>341	>6k	60.0	>1	>5k	60.0	1	878	4.49		>14	>14	>13.1
Pi(7)	6k	49k	16.2	74	2k	0.4	1	23	0.02	0.05	1243	27	5.6
Pi(8)	>10k	>105k	60.0	264	5k	1.7	1	26	0.02		>4616	128	26.9
Pi(9)	>11k	>120k	60.0	2k	19k	7.0	1	29	0.02		>4000	465	108.9
Pi(10)	>10k	>128k	60.0	6k	91k	45.2	1	32	0.02		>3530	2655	683.7
Pi(12)	>9k	>122k	60.0	>7k	>128k	60.0	1	38	0.03		>2400	>2400	>810.9
Pi(20)	>5k	>101k	60.0	>5k	>115k	60.0	1	62	0.09		>723	>723	>454.6
PS(4)	288	2k	0.4	2	41	0.1	1	16	0.01	0.59	72	2	0.1
PS(5)	35k	156k	43.2	8	142	0.1	1	22	0.01		5391	5	0.1
PS(6)	>32k	>141k	60.0	72	2k	0.4	1	29	0.02		>4286	28	0.7
PS(7)	>29k	>130k	60.0	2k	28k	7.5	1	37	0.03		>2858	357	12.3
PS(9)	>25k	>109k	60.0	>11k	>165k	60.0	1	56	0.06		>1053	>1053	>92.9
PS(11)	>23k	>103k	60.0	>9k	>132k	60.0	1	79	0.09		>690	>690	>88.8
DBP(5)	243	8k	2.0	133	4k	1.0	32	193	0.08	0.09	27	14	6.2
DBP(6)	729	33k	8.2	308	11k	3.2	64	386	0.16		53	21	13.3
DBP(7)	3k	134k	36.9	699	32k	10.8	128	771	0.33		113	33	26.2
DBP(8)	>4k	>157k	60.0	2k	91k	36.1	256	2k	0.79		>77	47	41.6
DBP(10)	>6k	>116k	60.0	>4k	>125k	60.0	2k	7k	3.23		>19	>19	>18.2
DBP(12)	>9k	>79k	60.0	>8k	>111k	60.0	5k	25k	15.79		>4	>4	>3.8
BB(6)	924	4k	1.3	215	2k	0.5	64	382	0.91	0.18	2	1	0.4
BB(7)	4k	13k	4.3	580	4k	1.2	128	830	1.49		3	1	0.8
BB(8)	13k	49k	17.2	2k	11k	3.3	256	2k	2.79		7	2	1.1
BB(9)	>41k	>156k	60.0	5k	30k	9.0	512	4k	6.15		>10	2	1.5
BB(10)	>46k	>176k	60.0	12k	81k	23.6	2k	9k	12.50		>5	2	1.9
BB(11)	>44k	>169k	60.0	>44k	>169k	60.0	3k	18k	25.74		>3	>3	>2.4

concrete numbers shown should not be taken into account. In some examples (e.g., BB, MS), though the gains are linear for the small inputs, when the size of the problem increases both SDPOR and CSDPOR time out, while CDPOR can still handle them efficiently.

Similar reductions are obtained for number of states explored. In this case, the system times out when it has memory problems, and the computation stops progressing (hence the number of explored states does not increase with the input any more). As regards the time to infer the annotations T^{smt}, we observe that in most cases it is negligible compared to the exploration time of the other methods. QS is the only example that needs some seconds to be solved and this is due to the presence of several nested conditional statements combined with the use of

built-in functions for lists, which makes the generated SMT encoding harder for the solver and the subsequent simplification step. Note that the inference is a pre-process which does not add complexity to the actual DPOR algorithm.

6 Related Work and Conclusions

The notion of conditional independence in the context of POR was first introduced in [11,15]. Also [12] provides a similar strengthened dependency definition. CSDPOR was the first approach to exploit this notion within the state-of-the-art DPOR algorithm. We advance this line of research by fully integrating conditional independence within the DPOR framework by using *independence constraints* (ICs) together with the notion of *transitive uniform* conditional independence –which ensures the ICs hold along the whole execution sequence. Both ICs and transitive uniformity can be approximated statically and checked dynamically, making them effectively applicable within the dynamic framework. The work in [14,21] generated for the first time ICs for processes with a single instruction following some predefined patterns. This is a problem strictly simpler than our inference of ICs both in the type of IC generated (restricted to the patterns) and on the single-instruction blocks they consider. Furthermore, our approach using an AllSAT SMT solver is different from the CEGAR approach in [4]. The ICs are used in [14,21] for SMT-based bounded model checking, an approach to model checking fundamentally different from our stateless model checking setting. As a consequence ICs are used in a different way, in our case with no bounds on number of processes, nor derivation lengths, but requiring a uniformity condition on independence in order to ensure soundness. Maximal causality reduction [13] is technically quite different from CDPOR as it integrates SMT solving within the dynamic algorithm.

Finally, data-centric DPOR (DCDPOR) [7] presents a new DPOR algorithm based on a different notion of dependency according to which the equivalence classes of derivations are based on the pairs read-write of variables. Consider the following three simple processes $\{p, q, r\}$ and the initial state $x = 0$:

p: `write(x=5)`, q: `write(x=5)`, r: `read(x)`. In DCDPOR, we have only three different observation functions: (r, x) (reading the initial value), (r, p) (reading the value that p writes), (r, q) (reading the value that q writes). Therefore, this notion of relational independence is finer grained than the traditional one in DPOR. However, DCDPOR does not consider conditional dependency, i.e., it does not realize that (r, p) and (r, q) are equivalent, and hence only two explorations are required (and explored by CDPOR). The example in conclusion, our approach and DCDPOR can complement each other: our approach would benefit from using a dependency based on the read-write pairs as proposed in DCDPOR, and DCDPOR would benefit from using conditional independence as proposed in our work. It remains as future work to study this integration. Related to DCDPOR, [16] extends optimal DPOR with observers. For the previous example, [16] needs to explore five executions: $r.p.q$ and $r.q.p$, are equivalent because p and q do not have any observer. Another improvement orthogonal to ours is to inspect dependencies over chains of events, as in [17,19].

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017)
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384 (2014)
3. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_26
4. Bansal, K., Koskinen, E., Tripp, O.: Commutativity condition refinement (2015)
5. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_27
6. Borralleras, C., Larraz, D., Oliveras, A., Rivero, J.M., Rodríguez-Carbonell, E., Rubio, A.: VeryMax: tool description for termCOMP 2016. In: WST (2016)
7. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Vaidya, K., Sinha, N.: Data-centric dynamic partial order reduction. In: POPL 2018 (2018)
8. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. STTT **2**(3), 279–287 (1999)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121 (2005)
10. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7
11. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_36
12. Günther, H., Laarman, A., Sokolova, A., Weissenbacher, G.: Dynamic reductions for model checking concurrent software. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 246–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_14
13. Huang, S., Huang, J.: Speeding up maximal causality reduction with static dependency analysis. In: ECOOP, pp. 16:1–16:22 (2017)
14. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_31
15. Katz, S., Peled, D.A.: Defining conditional independence using collapses. TCS **101**(2), 337–359 (1992)
16. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 229–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_14
17. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L.: Quasi-optimal partial order reduction. CoRR, abs/1802.03950 (2018)
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6

19. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: CONCUR, pp. 456–469 (2015)
20. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
21. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_29

Actor-Based Model Checking for SDN Networks

Elvira Albert[a], Miguel Gómez-Zamalloa[a], Miguel Isabel[a,*], Albert Rubio[a], Matteo Sammartino[b], Alexandra Silva[b]

[a] *Complutense University of Madrid, Spain*
[b] *University College London, UK*

Abstract

Software-Defined Networking (SDN) is a networking paradigm that has become increasingly popular in the last decade. The unprecedented control over the global behavior of the network it provides opens a range of new opportunities for formal methods and much work has appeared in the last few years on providing bridges between SDN and verification. This article advances this research line and provides a link between SDN and traditional work on formal methods for verification of concurrent and distributed software—actor-based modelling. We show how SDN programs can be seamlessly modelled using *actors*, and thus existing advanced model checking techniques developed for actors can be directly applied to verify a range of properties of SDN networks, including consistency of flow tables, violation of safety policies, and forwarding loops. Our model checker for SDN networks is available through an online web interface, that also provides the SDN actor-models for a number of well-known SDN benchmarks.

Keywords: Software-Defined Networks, Verification, Concurrency, Actor-based modelling, Model checking

1. Introduction

SDN is a relatively recent networking paradigm which is now widely used in industry, with many companies—such as Google and Facebook—using SDN to control their backbone networks and data-centers. The core principle in SDN is the separation of control and data planes—there is a centralized *controller* which operates a collection of distributed interconnected switches. The controller can dynamically update switches' policies depending on the observed flow of packets, which is a simple but powerful way to react to unexpected events in the network.

*Corresponding author: Miguel Isabel, Department of Sistemas Informáticos y Computación , C/ Profesor José García Santesmases, s/n Complutense University of Madrid, E-28040 - Madrid (Spain). Phone/Fax +34 91 3947641 / +34 91 3947529.

Email addresses: elvira@sip.ucm.es (Elvira Albert), mzamalloa@ucm.es (Miguel Gómez-Zamalloa), miguelis@ucm.es (Miguel Isabel), albert@cs.upc.edu (Albert Rubio), m.sammartino@ucl.ac.uk (Matteo Sammartino), alexandra.silva@ucl.ac.uk (Alexandra Silva)

Network verification has gained an extra boost since SDN was introduced, as in this new paradigm the amount of detailed information available about network events is rich enough and can be centrally gathered to check for properties, both statically and dynamically, of the network behavior. Moreover, the controller itself is a program which can be analyzed and verified before deployment.

The distributed and concurrent nature of network behavior makes programming and verification tasks challenging. Some of the bugs that can be found in existing (programmable) networks are reminiscent of faults that have appeared in distributed and concurrent systems, and which have inspired much research in the verification and formal methods communities. With this observation as a starting point, this article provides a new bridge between SDN and a strand of formal methods—actor-based modelling [1]— which was originally developed to analyze concurrent systems. Actors, entities equipped with a private memory, form the basic unit of computation in such framework and can interact with each other through *asynchronous* messages. This setup enables reasoning about local properties of the system without knowledge of the whole program, which gives rise to more compositional and thus scalable methods. Actors provide the foundations for the concurrency model of languages used in industry, e.g., *Erlang* and *Scala*, and libraries used in mainstream languages, e.g., *Akka*.

1.1. Summary of contributions

This article makes five main contributions:

1. SDN-semantics: A formalization of the semantics of SDN networks which allows us to define the transitions that occur in the network and formalize the concept of execution trace needed to prove soundness of our modelling.

2. SDN-Actors: An encoding of all basic components of an SDN network (switches, hosts, controller) into the actor-based language ABS [2] and a soundness proof of our encoding using the semantics of SDN networks in point 1.

3. Barriers: One of the most challenging aspects to encode are the OpenFlow *barrier* messages, special instructions that the controller can use to force switches to execute all their queued tasks. We provide an implementation of barriers using conditional synchronization and a soundness result.

4. Model checker: A model checker for our SDN models built on top of the SYCO tool [3] that incorporates several dynamic partial-order reduction (DPOR) algorithms.

5. Case studies: Several case studies of SDN and properties to illustrate the versatility and potential of the approach. We were able to find bugs related to programming errors in the controller, forwarding loops, and violation of safety policies, and scale to larger networks than related techniques.

This article extends and improves the conference paper that appeared in the FM'18 proceedings [4] as follows. On the theoretical side, we have formalized

the semantics of SDN networks and used it to prove soundness of the basic encoding of SDN-Actors, ensuring thus the correctness of our models. On the practical side, we have carried out a new experimental evaluation using the Constrained DPOR algorithm [5]. This DPOR algorithm can take advantage of independence conditions that we have defined specifically for the SDN domain and that allow us to treat larger networks than by using related techniques and than in our FM'18 paper. We have also extended the SYCO tool with a mechanism to detect the violation of the property under check that stops the exploration, while before SYCO was restricted to full exploration.

1.2. Organization of the article

Section 2 gives an intuition of the main ideas in the article by means of a simple example. In Section 3 we present the semantics of SDN programs and of actor systems, in two parts. First, Section 3.1 introduces a semantics for SDN networks that describes the communication patterns in this kind of networks and that allows us to formalize the notion of execution trace in the SDN network. Next, we recall the semantics of actor systems from [2] which will constitute the semantics of our models. Section 4 introduces the concept of *SDN-Actor* by providing the encoding of all components in an SDN network as actors. We formally prove the soundness of the encoding by relying on the semantics introduced in Section 3.1. Section 5 extends both the SDN semantics and our models to handle barriers and formalizes the soundness of this extension. Section 6 describes our DPOR-based model checker which instantiates an off-the-shelf model checker for actor systems with tailored independence conditions to efficiently verify SDN-Actor models. Section 7 describes the experimental evaluation of the tool. Related work and conclusions appear in Section 8.

2. Overview

This section contains an overview of the technical contributions via an extended example, which we also use to introduces basic concepts and notations.

2.1. Concurrency errors in SDN networks

SDN is a networking architecture where a central software *controller* can dynamically change how network switches forward packets by monitoring the traffic. Switches can be connected to hosts and to other switches via bidirectional channels that may reorder packets. Each switch has a *flow table*, that is a collection of guarded forwarding rules to determine the route of incoming packets. Whenever a switch receives a packet, it checks if one of the flow table rules applies. If no rule applies, the switch sends a message to the controller via a dedicated link, and the packet is buffered until instructions arrive. Depending on its policy, the controller instructs the switch, and possibly other switches in the network, on how to update their flow tables. Such control messages between the controller and the switches can be processed in arbitrary order.

We now show how a simple load-balancer can be implemented in SDN (example taken from [6]) and how potential bugs can easily arise due to the concurrent

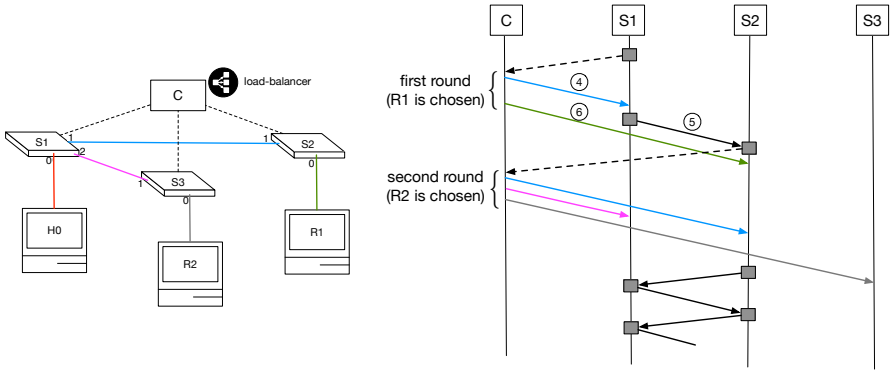


Figure 1: Example SDN load-balancer. On the left: structure of the SDN. On the right: messages exchanged in a possible execution of a naive controller program. Coloured arrows stand for control messages to switches, indicating which flow rule to install (colours specify the link to be used for the forwarding). Grey boxes and arrows among them represent packet forwardings. Dashed arrows indicate messages to the controller.

behavior and asynchrony of message passing. Suppose we want to balance the traffic to a server by using two replicas R1 and R2 to which the controller alternates the traffic in a round-robin fashion. The structure of the SDN is shown in Figure 1, on the left: H0 is any host that wants to communicate with the server and S1, S2 and S3 are switches (numbers on endpoints stand for port numbers).

Even in this simple network, an incorrect implementation of the controller can lead to serious problems. In Figure 1, on the right, we show an execution of a naive controller, which simply instructs switches to forward packets along the shortest path to the chosen replica. This implementation ignores the potential concurrency in actions taken by switches and controller, leading to a forwarding loop between S1 and S2. In the first round, when S1 queries the controller, R1 is chosen. The figure shows S1 forwarding the packet to S2 before the end of the first round, i.e., before a rule is installed on S2 (green arrow). This causes S2 to query the controller, which triggers the second round in which the controller chooses R2. Thus, it sends instructions to install rules on S2, S1 and S3 to forward the packet to S1, S3 and R2, respectively. When the controller rules arrive at S1, it will have two contradictory instructions, telling to forward the packet either to S2 or to S3. In the former case, the loop at the bottom of the figure occurs. This issue can be avoided if the implementation uses barriers—the controller will then guarantee that S2 receives and processes control messages before taking any other action.

2.2. Actor-based modelling of SDN networks

We now explain how we can automatically detect the above problem using actors and model checking. We use the object-oriented actor language ABS [2, 7], where each actor type is specified as a class, consisting of a set of fields and methods. Actors are instances of actor classes. For instance, the instructions: Controller ctrl = new Controller(); Switch s1 = new Switch("S1",ctrl); Host h0 =

new Host("H0",s1,0); create 3 actors: a controller ctrl; a switch s1 with name "S1" and a reference to ctrl; a host h0, with name "H0", connected to the switch s1 via the port 0. The SDN in Figure 1 can be modeled using one actor per component (additional data structures for network links will be shown later).

The execution model of actors is *asynchronous*. Each actor can be thought of as a processor, with a queue of pending tasks and a local memory. Actors are executed in parallel and, at each actor, one task is *non-deterministically* selected among all the pending ones and executed. The syntax Fut<type> f=a!m($\overline{x}$) spawns an asynchronous task m($\overline{x}$), that is added to the queue of pending tasks of a, type is the type of the data returned by m or Unit if no data is returned. This task consists in executing the method m of a with arguments $\overline{x}$. The variable f is a *future variable* [8] that will allow us to check if such task has been completed. Left-hand side of the assignment can be omitted in case the future variable is not needed.

A partial trace of execution of our SDN actor model computed by the model checker is (the code that the tasks below execute will be given in Section 4):

$$\boxed{1\colon \text{h0!sendIn}} \xrightarrow{1} \boxed{2\colon \text{s1!switchHandlePacket}} \xrightarrow{2} \boxed{3\colon \text{ctrl!controlHandleMessage}}$$
$$\xrightarrow{3} \boxed{4\colon \text{s1!switchHandleMessage(s2)}, 5\colon \text{s1!sendOut}, 6\colon \text{s2!switchHandleMessage(r1)}}$$

Intuitively, a packet sending (sendIn) is executed on h0 (label 1), which causes the packet to be forwarded to the switch s1 (2), then s1 sends a control message to the controller (3). Finally, the controller spawns the three tasks in the last state (parameters tell where to forward the packet). When executed, these tasks will produce the messages in Figure 1 with the same numbers. Their execution order is arbitrary: if it is the one shown in Figure 1, the execution trace may lead to a state exhibiting a forwarding cycle between s1 and s2. As we will show later, this situation can be easily detected by our model checker SYCO via an exploration of a *reduced* execution tree, which avoids equivalent executions (Section 6).

The ABS language provides a convenient await primitive that will be used to model barriers and to rule out the behavior described above. The instruction await f? synchronizes with the termination of the task associated to the future variable f, by releasing the processor (so that another task can be scheduled) if the task is not finished. Once the awaited task is finished, the suspended task can resume. The await can be used also with boolean conditions await b? to suspend the execution of the current active task until condition b holds. The formal semantics of the language can be found in Section 3.2.

3. Semantics for SDN Networks and for Actors

This section presents two semantics that provide the formal basis on which we build our models: we first introduce the semantics of SDN Networks in Section 3.1, and then the semantics of actors in Section 3.2. The semantics

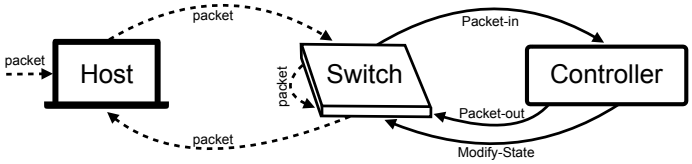


Figure 2: Information flow in SDN networks

of actors has been already defined in several works (ours is a simplification of [2]). Our formalization of the SDN semantics is similar to that of [9]. We have considered a simplification of the Openflow specification that captures the essence of the communications of SDN networks (e.g., we have not included the operation *flood* as it behaves similarly to the considered switch operations).

3.1. SDN Networks

Let us first describe the information flow of packets and messages among the different elements in an SDN network that we have depicted in Figure 2. As in standard networks, packets can be sent from hosts to switches and viceversa, and also from switches to switches (see dashed arrows). The leftmost dashed arrow represents the reception by a host of a new packet which is fed into the network. The specific communications of SDN networks are performed by means of *Openflow messages* (see regular arrows), which in our simplification can be of three types:

- Packet-in: This message is sent from a switch to the controller when the switch processes a packet for which it has no action rule to apply. The message includes the switch identifier and the identifier and header of the packet. The packet is buffered in the switch until a message of type *Packet-out* is received.
- Modify-State: This message is sent from the controller to a switch with new action rules to be inserted into the switch's flow-table. The message includes a flow-table entry with an action rule.
- Packet-out: This message is sent from the controller to a switch to notify that it must re-try applying an action rule to a buffered packet. The message includes the packet header.

Figure 3 shows the semantics of the flow of communications performed in our simplified SDN networks. The three types of messages below are respectively abbreviated as pktIn, modState and pktOut.

- A host is a term of the form $h(id, sid, o, in)$, where id is the host identifier, sid and o are, respectively, the switch identifier and port to which the host is connected, and in its input channel.

- A switch is of the form $s(id, ft, b, in)$, where id is the switch identifier, ft its flow-table, b its internal buffer of packets and in its input channel.
- The controller is of the form $c(top, in)$ where top is the topology of the network and in is its input channel.
- A state of the SDN network is a tuple of the form $\langle H, S, C\rangle$ where $H = \{h \mid h \text{ is a host}\}$, $S = \{s \mid s \text{ is a switch}\}$, that is, H is a set of hosts, S is a set of switches, and, C is the controller.

Letter p denotes a packet. Function $header(p)$ returns its header. Flow-tables are represented as mappings from pairs packet-header/port to actions and are treated as a black-box through the following functions: $lookup(ft, \langle ph, o\rangle)$ that returns the action associated to the packet with header ph received through port o in the flow-table ft, or $\perp$ if there is no entry for it; and, $put(ft, \langle ph, o\rangle, a)$ that returns the new flow-table after inserting in ft the entry $\langle ph, o\rangle \mapsto a$. For simplicity, we only consider actions of the form $send(id)$ or $\langle send(id), o\rangle$, which indicate that the corresponding packet should be sent, respectively, to the host id, or to the switch with id as identifier using port o. Function $applyPol(top, sid, o, ph)$ represents the application of the controller's policy using the current network topology top in result to a packet received via port o with header ph that the switch with identifier sid has not been able to handle. It returns a set of pairs $\langle id, m\rangle$ where m is a modifyState message with an associated new flow-table entry that has to be forwarded to the switch with identifier id.

A *transition* or *step* in the network corresponds to the processing of a packet or message by a host, switch or the controller. There are six (sets of) transition rules corresponding to the different types of incoming arrows in Figure 2:

- sendIn (abbreviated as SI): It corresponds to the processing by a host of a new packet which is fed into the network (denoted as $new(p)$), in which case the packet is forwarded to the switch to which the host is connected via the corresponding port. Note that the port is attached to the packet (denoted $o{:}p$) since there is only one input channel in switches.
- hostHandlePacket (HHP): This corresponds to the processing by a host h of a packet received from its switch, in which case the packet is consumed without any further action.
- switchHandlePacket (SHP): When a switch processes a received packet, either from a host or from another switch, it looks up if there is any rule matching with the header of the packet and port in its flow table ft. There are three cases: (cases 1 and 2) there is a send action rule in the switch's flow-table, hence the packet is forwarded to the host (case 1) or switch (case 2) indicated in the action (in the latter case also the switch's port is included in the action); or (case 3) there is no rule for this packet, in which case the packet is buffered and a Packet-in message is sent to the controller.

- sendOut (SO): This corresponds to the processing of a Packet-out message by a switch. After looking up the header of the packet p in its own flow-table ft, there are three cases which are analogous to those of switch-HandlePacket except that the packet is in the switch's buffer (instead of in its input channel), and that if no action rule is found in the switch's flow-table the packet is dropped.

- switchHandleMessage (SHM): It corresponds to the processing of a Modify-State message by a switch, in which case the received action rule is inserted into the switch's flow-table.

- controlHandleMessage (CHM): The controller receives a Packet-in message from a switch s in result to a packet that the switch s has not been able to handle. As a result, the controller sends a set ms of Modify-State messages with new action rules to a selected set of switches (as specified by the controller's policy with the current network topology), and a Packet-out message to switch s.

A derivation $E \equiv S_0 \rightarrow \cdots \rightarrow S_n$ is *complete* if S_0 is the initial state and $S_n = \langle H, S, C \rangle$ is the final state such that every message and packet in their channels has been processed (their input channels are empty). We use $exec(S)$ to denote the set of all possible executions starting at state S.

3.2. Syntax and Semantics for Actor Programs

The grammar below describes the syntax of the language ABS in which SDN models will be defined:

$$
\begin{array}{lcl}
P & ::= & M\ \bar{C} \\
C & ::= & \textbf{class}\ c(\bar{T}\ \bar{x})\{\bar{M}\} \\
M & ::= & T\ m(\bar{T}\ \bar{x})\{s;\} \\
s & ::= & s\ ;\ s \mid x = e \mid \texttt{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ b\ \textbf{do}\ s \mid m(\bar{z}) \\
 & & \mid x = \textbf{new}\ C(\bar{y}) \mid f = x!m(\bar{z}) \mid \textbf{await}\ f? \mid \textbf{await}\ b?
\end{array}
$$

Here, x, y, z denote variables names, f a future variable name, and s a sequence of instructions. For any entity A, the notation $\bar{A}$ is used as a shorthand for $A_1, ..., A_n$. We use the special identifier `this` to denote the current actor. For generality, the syntax of expressions e, Boolean conditions b and types T is left unspecified. As in the object-oriented paradigm, a class denotes a type of actors including their behavior, and it is defined as a set of fields and methods. Lastly, $m(\bar{z})$ denotes standard (synchronous) method calls, which are only allowed on the actor itself, whereas "!" is used for asynchronous method calls (see Section 2.2).

Figure 4 presents the semantics of the actor model. An *actor* is a term of the form $a(o, tk, h, \mathcal{Q})$, where o is the actor identifier, tk is the identifier of the *active task* that holds the actor's lock or $\perp$ if the actor's lock is free, h is its local heap and $\mathcal{Q}$ is the queue of tasks in the actor. A *heap* h is a mapping $h : \mathit{fields}(C) \mapsto \mathbb{V}$, where $\mathbb{V}$ stands for the set of references and values. A *task* tk is a term $\textsc{tk}(tk, m, l, s)$ where tk is a unique task identifier,

$$(\text{SI})\ \frac{h = h(id, sid, o, in \cup \{new(p)\}) \quad s = s(sid, ft, b, in')}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o, in)\} \cup H, \{s(sid, ft, b, in' \cup \{o{:}p\})\} \cup S, C\rangle}$$

$$(\text{HHP})\ \frac{h = h(id, sid, o, in \cup \{p\})}{\langle \{h\} \cup H, S, C\rangle \to \langle \{h(id, sid, o, in)\} \cup H, S, C\rangle}$$

$$(\text{SHP}_1)\ \frac{\begin{array}{c} s = s(sid, ft, b, in \cup \{o{:}p\}) \quad h = h(id, sid, o', in') \\ send(id) = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o', in' \cup \{p\})\} \cup H, \{s(sid, ft, b, in)\} \cup S, C\rangle}$$

$$(\text{SHP}_2)\ \frac{\begin{array}{c} s = s(sid, ft, b, in \cup \{o{:}p\}) \quad s' = s(sid', ft', b', in') \\ send(sid', o') = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s, s'\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, in), s(sid', ft', b', in' \cup \{o'{:}p\})\} \cup S, C\rangle}$$

$$(\text{SHP}_3)\ \frac{s{=}s(sid, ft, b, in \cup \{o{:}p\}) \quad s' = s(sid, ft, b \cup \{o{:}p\}, in) \quad \bot = lookup(ft, \langle header(p), o\rangle)}{\langle H, \{s\} \cup S, c(top, in')\rangle \to \langle H, \{s'\} \cup S, c(top, in' \cup \{\mathsf{pktIn(sid, o, id(p), header(p))}\})\rangle}$$

$$(\text{SO}_1)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, in \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad h = h(id, sid, o', in') \quad send(id) = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o, in' \cup \{p\})\} \cup H, \{s(sid, ft, b, in)\} \cup S, C\rangle}$$

$$(\text{SO}_2)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, in \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad s' = s(sid', ft', b', in') \quad send(sid', o') = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s, s'\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, in), s(sid', ft', b', in' \cup \{o'{:}p\})\} \cup S, C\rangle}$$

$$(\text{SO}_3)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, in \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad \bot{=}lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, in)\} \cup S, C\rangle}$$

$$(\text{SHM})\ \frac{s = s(sid, ft, b, in \cup \{\mathsf{modState}(\langle \mathsf{ph}, \mathsf{o}\rangle \mapsto \mathsf{a})\})}{\langle H, \{s\} \cup S, C\rangle \to \langle H, \{s(sid, put(ft, \langle ph, o\rangle, a), b, in)\} \cup S, C\rangle}$$

$$(\text{CHM})\ \frac{\begin{array}{c} c = c(top, cin \cup \{\mathsf{pktIn(sid, o, pid, ph)}\}) \quad s = s(sid, ft, b, sin) \\ ms = applyPol(top, sid, o, ph) \quad ms_{id} = \{m \mid \langle id, m\rangle \in ms\} \\ S' = \{s(sid', ft', b', in') \mid s(sid', ft', b', in) \in S, in' = in \cup ms_{sid'}\} \end{array}}{\langle H, S \cup \{s\}, c\rangle \to \langle H, S' \cup s(sid, ft, b, sin \cup ms_{sid} \cup \{\mathsf{pktOut(ph)}\}), c(top, cin)\rangle}$$

Figure 3: Semantics of SDN networks

$$(\text{MSTEP})\ \frac{a(o,\bot,h,\mathcal{Q}) = selectAct(S) \qquad \text{TK}(tk,m,l,s) = selectTask(a(o,\bot,h,\mathcal{Q})) \quad s \neq \epsilon \quad S \overset{o\cdot tk}{\rightsquigarrow^*} S'}{S \longmapsto S'}$$

$$(\text{ASY})\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\mathtt{x_f} = y\ !\ m_1(\overline{z});s) \quad o_1 = l(y) \quad tk_1 = \text{fresh}() \quad l_1{=}newlocals(\bar{z},m_1,l)}{\begin{array}{c} a(o,tk,h,\mathcal{Q}\cup\{\mathsf{tk}\})\cdot a(o_1,tk',h',\mathcal{Q}') \overset{o\cdot tk}{\rightsquigarrow} \\ a(o,tk,h,\mathcal{Q}\cup\{\text{TK}(tk,m,l[\mathtt{x_f}\mapsto tk_1],s)\})\cdot a(o_1,tk',h',\mathcal{Q}'\cup\{\text{TK}(tk_1,m_1,l_1,body(m_1))\}) \end{array}}$$

$$(\text{SYN})\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,m_1(\overline{z});s) \quad l_1{=}newlocals(\bar{z},m_1,l)}{(\text{SYN})\ a(o,tk \quad h,\mathcal{Q}\cup\{\mathsf{tk}\})\cdot \overset{o\cdot tk}{\rightsquigarrow}\ a(o,tk,h,\mathcal{Q}\cup\{\text{TK}(tk,m,l_1,body(m_1);s)\})}$$

$$(\text{NEW})\ \frac{\begin{array}{c}\mathsf{tk} = \text{TK}(tk,m,l,x = \mathbf{new}\ D(\bar{y});s) \quad o_1{=}\text{fresh}() \\ h' = newheap(D) \quad l' = l[x\to o_1] \quad \mathbf{class}\ D(\bar{f})\{\ldots\}\end{array}}{a(o,tk,h,\mathcal{Q}\cup\{\mathsf{tk}\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,tk,h,\mathcal{Q}\cup\{\text{TK}(tk,m,l',s)\})\cdot a(o_1,\bot,h'[\bar{f}\mapsto l(\bar{y})],\emptyset)}$$

$$(\text{AWAIT})_1\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s) \quad l(\mathtt{x_f}) = tk_1 \quad \text{TK}(tk_1,m_1,l_1,\epsilon)\in S}{a(o,tk,h,\mathcal{Q}\cup\{\mathsf{tk}\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,tk,h,\mathcal{Q}\cup\{\text{TK}(tk,m,l,s)\})}$$

$$(\text{AWAIT})_2\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s) \quad l(\mathtt{x_f}) = tk_1 \quad \text{TK}(tk_1,m_1,l_1,\epsilon)\notin S}{a(o,tk,h,\mathcal{Q}\cup\{\mathsf{tk}\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,\bot,h,\mathcal{Q}\cup\{\text{TK}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s)\})}$$

$$(\text{RETURN})\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\epsilon)}{a(o,tk,h,\mathcal{Q}\cup\{\mathsf{tk}\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,\bot,h,\mathcal{Q}\cup\{\mathsf{tk}\})}$$

Figure 4: Semantics of concurrent primitives of actor programs

m is the method name executing in the task, l is a mapping from local variables to $\mathbb{V}$, and s is the sequence of instructions to be executed. Finally, a global state S is a set of actors. As actors do not share their states, the semantics can be presented as a macro-step semantics [10] (defined by means of the transition "$\longmapsto$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to a *release point*, i.e., a point in which the actor's processor becomes idle due to the return or an await instruction. In this case, rule (MSTEP) is applied to select an available task from an actor, namely relation $selectAct(S)$ is applied to select *non-deterministically* an actor $a(o,\bot,h,\mathcal{Q})$ in the state with a non-empty queue $\mathcal{Q}$, and, $selectTask(a(o,\bot,h,\mathcal{Q}))$ to select *non-deterministically* a task of $\mathcal{Q}$. Micro-step transitions are written $\overset{o\cdot tk}{\rightsquigarrow}$ and define evaluations in task tk by actor o within a given macro-step. As before, the sequential instructions are standard and thus omitted. In (NEW), an active task tk in actor o creates a new actor of class D with a fresh identifier $o_1 = \text{fresh}()$, which is introduced to the state with a free lock. Here $h' = newheap(D)$ stands for a default initialization

on the fields of class D. Rule (SYN) simply replaces in task tk the statement with the method call to m_1 by its body. Rule (ASY) spawns a new task (the initial state is created by *newlocals*) with a fresh task identifier tk_1 which is stored in the future variable $\mathtt{x_f}$. We assume $o \neq o_1$, but the case $o = o_1$ is analogous, the new task tk_1 is simply added to the queue $\mathcal{Q}'$ of actor o_1. In rule $(\text{AWAIT})_1$, the future variable $\mathtt{x_f}$ we are awaiting for points to a finished task and thus the **await** can be completed. The finished task identified with tk_1 is looked up in all actors in the current state (written as $\text{TK}(tk_1, m_1, l_1, \epsilon) \in S$). Otherwise, $(\text{AWAIT})_2$ yields the lock so that any other task of the same actor can take it. The behaviour of AWAIT on Boolean conditions is analogous. When rule (RETURN) is executed, the task is *finished*, but it remains in the queue so that rules $(\text{AWAIT})_1$ and $(\text{AWAIT})_2$ can be applied. A derivation $E \equiv S_0 \longmapsto \cdots \longmapsto S_n$ is *complete* if S_0 is the initial state and all actors in S_n are of the form $a(o, \bot, h, \mathcal{Q})$, where for all $\mathsf{tk} \in \mathcal{Q}$ it holds that $\mathsf{tk} \equiv \text{TK}(tk, m, l, \epsilon)$. We use $exec(S)$ to denote the set of all possible executions starting at state S.

4. SDN-Actors: an actor based encoding of SDN programs

We present the concept of *SDN-Actor* in 3 steps: Section 4.1 describes the creation and initialization of the actors according to the topology. Section 4.2 provides the encoding of the operations and communication for Switch and Host actors. Section 4.3 proposes the encoding of the controller. Altogether, our encoding provides an actor-based semantics foundation of SDN networks that follow the OpenFlow specification [11] captured by the semantics in Section 3.1.

4.1. Network topology

The topology can be given as a relation with two types of links:

1. *SHlink(s,h,o)*: switch s is connected to host h through the port o
2. *SSlink(s_1,i_1,s_2,i_2)*: switch s_1 is connected via port i_1 to port i_2 ofs s_2

from which we automatically generate the initial configuration as follows.

Definition 4.1 (initial configuration). *Let S and H be, respectively, the set of different switch and host identifiers available in the* link *relations that define the network topology. The initial configuration (method* init_conf*) is defined as:*

- *We create a controller actor* Controller ctrl=**new** Controller()
- *For each* sid$\in S$*, we create an actor* Switch s=**new** Switch(sid,ctrl)
- *For each* hid$\in H$*, we create an actor* Host h=**new** Host(hid,s,o) *where* s *is the reference to the switch actor,* o *the port identifier, that* hid *is connected to.*
- *The data structures* srefs *and* hrefs *store, respectively, the relations between identifier in the topology and reference in the program, for all switches in S and hosts in H.*

- *The data structure* ntw *contains the link relations in the network topology.*
- *The synchronous call* ctrl.addConfig(srefs,hrefs,ntw) *initializes in the controller the topology relations and the references to switches and hosts such that the controller can send control messages to redirect the traffic to the involved links.*

Example 4.2. *By applying Definition 4.1 to the topology in Figure 1, given as the relation:* $SHlink(S1, H0, 0)$, $SHlink(S2, R1, 0)$, $SHlink(S3, R2, 0)$, $SSlink(S1, 1, S2, 1)$, *and* $SSlink(S1, 2, S3, 1)$, *we obtain the following initial configuration which constitutes the* init_conf *method from which the execution starts:*

```
1 init_conf() { Controller ctrl = new Controller(); Switch s1 = new Switch("S1",ctrl);
2       Switch s2 = new Switch("S2",ctrl); Switch s3 = new Switch("S3",ctrl);
3       Host h0 = new Host("H0",s1,0); Host r1 = new Host("R1",s2,0);
4       Host r2 = new Host("R2",s3,0);
5       Map<SwitchId,Switch> srefs = {"S1":s1, "S2":s2, "S3":s3};
6       Map<HostId,Host> hrefs = {"H0":h0, "R1":r1, "R2":r2};
7       List<Link> ntw = [SHLink("S1","H0",0), SSLink("S1",1,"S2",1),..];
8       ctrl.addConfig(srefs,hrefs,ntw); }
```

The data structures srefs *and* hrefs *are implemented using maps, and the network* ntw *as a heterogeneous list. The use of data structures is nevertheless orthogonal to the encoding as actors. We just assume standard functions to create, initialize, access them (like getters, put, take, lookup, etc.) that will appear in italics in the code.*

4.2. The switch and host classes

Figure 5 presents the actor-based Switch and Host classes. We include at the top some **type** declarations that are assumed and must be implemented (such as identifiers, packets and their headers, etc.). There are two main data structures implemented in more detail to make explicit the information they contain:

- the buffer at Line 22 (L22 for short) is a *map* that must contain pairs of packet and input port indexed by their PacketId.
- the flow table flowT (L21) is implemented as a *map* indexed by the so-called *match field* [11] represented by type MatchF in Figure 5. The match field is composed by information stored in the header of a Packet (retrieved by function $getHeader$) and the input port. For a given matching, the flow table contains the Action the switch has to perform upon the reception of the Packet. An action l can be of three types: i) send the packet to a host h, ii) send the packet to the port o of a switch s, iii) drop the packet. Given an action l, function $isSwitch$ respectively $isHost$ succeeds if the action is of type ii) respectively i), and functions $getSwitch$, $getHost$ and $getPort$ return the s, h and o respectively. The full implementation must allow duplicate entries (non-deterministically selected), and the use of wildcards in the match fields, but these aspects are unrelated to the encoding of SDN actors, and skipped for simplicity.

```
type SwitchId=... type HostId=... type PortId=... type PacketId=...
type PacketH=... type Packet=... type Action=... type Link=...
type MatchF=(PacketH,PortId);
```

```
class Host(HostId hid, Switch s, PortId o) {
 Unit sendIn(Packet p){
    s!switchHandlePacket(p,o);
 }
 Unit hostHandlePacket(Packet p){
    /* output packet */
 }
}
class Switch(SwitchId sid, Controller ctrl) {
   Map<MatchF,Action> flowT={};
   Map<PacketId,(Packet,PortId)> buffer={};
 Unit switchHandlePacket(Packet p, PortId o){
    Action l=lookup(flowT,(getHeader(p),o));
    if (isSwitch(l))
        getSwitch(l)!switchHandlePacket(p,getPort(l));
    else if (isHost(l))
        getHost(l)!hostHandlePacket(p);
        else {
        buffer=put(buffer,getId(p),(p,o));
        ctrl!controlHandleMessage(sid,o,getId(p),getHeader(p));
    }
 }
 Unit sendOut(PacketId pi){
    Packet p; PortId o;
   (p,o)=take(buffer,pi);
    Action l=lookup(flowT,(getHeader(p),o));
    if (isSwitch(l))
        getSwitch(l)!switchHandlePacket(p,getPort(l));
    else if (isHost(l))
        getHost(l)!hostHandlePacket(p);
    /* else packet is dropped */
 }
 Unit switchHandleMessage(MatchF m, Action a){
   flowT=put(flowT,m,a);
 }
}
```

Figure 5: Type declarations (top) and actor-based host and switch classes (bottom)

Upon creation, hosts receive their identifier and a reference to the switch and the port identifier they are connected to (defined as class parameters that are initialized at the actor creation). Their method sendIn is used to send a packet to the switch, and method hostHandlePacket to receive a packet from the switch. Switches receive upon creation their identifier and a reference to the controller. They have as additional fields: (a) the flow table flowT (as described above) in which they store the actions to take upon receiving each kind of package, and (b) a buffer in which they store packets that are waiting for a response from the controller. Switches can perform three operations: (1) switchHandlePacket receives a packet, looks up in the flow table the action to be made L24, and, if there is an entry for the packet in the table, it asynchronously makes the corresponding action (either send it to a host L27 or to a switch L25). Otherwise, it sends a controlHandleMessage request and puts the packet and input port in the buffer (L30 and L31) until it can be handled later upon receipt of a sendOut; (2) sendOut receives a packet identifier that corresponds to a waiting packet, retrieves it from the buffer (L35), looks up the action l to be performed in the flow table, and makes the corresponding asynchronous call (as in switchHandlePacket); (3) switchHandleMessage corresponds to a message received from the controller with an instruction to update the flow table. Other switch operations like *forward packet*, that is similar to sendOut but directly tells the switch the action to be performed, or *flood*, that sends a packet through all ports except the input port, can be encoded similarly and are used in the experiments in Section 7.

Example 4.3. *In init_conf, after L8, we add h0!sendIn(p), where p is a packet to be sent to the IP address of the replica servers (the information on the destination is part of the packet header). This is the only asynchronous task that init_conf spawns. Its execution in turn spawns a new task s1!switchHandlePacket(p,0) at L13, that does not find an entry in flowT at L24 and spawns a controlHandleMessage task on the controller at L31, whose code is presented in the next section.*

4.3. The controller

After creating the controller actor, the method addConfig is invoked synchronously to initialize the references to switches and hosts and set up the initial network topology (see L8). A simple controller is presented in Figure 6. When a switch asynchronously invokes controlHandleMessage, the controller applies the current policy—function *applyPolicy* must be implemented for each different type of controller. The implementation of the policy typically requires the definition of new data structures in the controller to store additional information (see Section 7). When applying the policy for a given SwitchId, PortId and PacketH , we obtain a list of switch identifiers and corresponding actions to be applied to them (as a data-structure of type *List*<(SwitchId,MatchF,Action)>). The while loop at L57 in controlHandleMessage asynchronously invokes switchHandleMessage at L82 on each of the switches in the list, and passes as parameter the corresponding action to be applied for the given match entry. Finally, it notifies at L63 the switch from which the packet came that this can be sent out. More sophisticated controllers that build upon this encoding are described in Section 7.

```
48 class Controller() {
49   Map<SwitchId,Switch> srefs={};
50   Map<HostId,Host> href={};
51   List<Link> ntw=[];
52   Unit addConfig(Map<SwitchId,Switch> sr, Map<HostId,Host> hr, List<Link> n){
53         /* references to switches and hosts and network topology initialized */
54   }
55   Unit controlHandleMessage(SwitchId sid, PortId o, PacketId p, PacketH h){
56       List<(SwitchId,MatchF,Action)> l=applyPolicy(sid,o,h);
57       while (not(isEmpty(l))) {
58           SwitchId s1; Action a1; MatchF m1;
59           (s1,m1,a1)=head(l);
60           lookup(srefs,s1)!switchHandleMessage(m1,a1);
61           l=tail(l);
62       }
63        lookup(srefs,sid)!sendOut(p);
64    }
65  }
```

Figure 6: Controller class (without barriers)

Example 4.4. *In the example,* applyPolicy *corresponds to the load-balancer described in Section 2, which directs external requests to a chosen replica in a round-robin fashion. For the call* applyPolicy(s1,0,h)*, it chooses* r1 *and thus, it returns in L56 two actions:* (s1→s2)*,* (s2→r1)*, i.e., one action to install in* s1 *the rule to send the packet to* s2*, and the second to install in* s2 *the rule to send it to* r1*. For simplicity, we assume that the* Action *just contains the location to which the packet has to be sent (without including the port). The while loop thus spawns two asynchronous calls,* s1!switchHandleMessage(m1,s2) *and* s2 !switchHandleMessage(m1,r1)*. Besides, it sends a* s1!sendOut(p) *in L63. Several problems may arise in this implementation. One problem, as explained in Section 2, is that the packet is sent from* s1 *to* s2 *before the control message is processed by* s2*. Then,* s2 *gets the packet and it does not find any matching rule, thus it sends a* controlHandleMessage *to the controller. Applying the above policy, the controller chooses now as replica* r2 *and returns the actions:* (s2→s1)*,* (s1→ s3)*,* (s3→r2)*, i.e., the packet should be sent to* r2 *by first sending from* s2 *to* s1 *(first action), and so on. This might create the circularity depicted in Figure 1.*

4.4. Soundness of the Encoding

An execution in the network is characterized by the messages in the queues of the switches, hosts, and controller and the state of their data structures. First of all, let us define the equivalence between an input channel with its buffer (in, b) and a queue of pending tasks with its buffer $(\mathcal{Q}, \text{buffer})$. Let us notice here that even though we have used different notation for b and buffer, we use $b = \text{buffer}$ to denote the equality of information in both structures, that is, they have exactly the same packets and with the same ports.

Definition 4.5. *An input channel in with a buffer of pending packets b and a queue of pending tasks $\mathcal{Q}$ with a buffer of pending packets buffer are equivalent, written $(in, b) \equiv (\mathcal{Q}, buffer)$ if and only if:*

1. *$in = \emptyset = \mathcal{Q}$ and $b = buffer$ or*
2. *otherwise on the following holds:*

   pktOut: *$in = \{pktOut(ph)\} \cup in'$, $\exists \text{TK}, p$ such that $\mathcal{Q} = \{\text{TK}(\_, sendOut, l, \_)\} \cup \mathcal{Q}', b = b' \cup \{o{:}p\}$, $buffer = buffer' \cup \{(p, o)\}$, $getId(p) = l[pi]$ and $getHeader(p) = ph$, and $(in', b') \equiv (\mathcal{Q}', buffer')$.*

   modState: *$in = \{modState(\langle ph,o \rangle \mapsto a)\} \cup in', \exists \text{TK}$ such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, switchHandleMessage, l, \_)\}$, $(\langle ph,o \rangle \mapsto a) = (l[m] \mapsto l[a])$ and $(in', b) \equiv (\mathcal{Q}', buffer)$,*

   pktIn: *$in = \{pktIn(sid,o,pid,ph)\} \cup in'$, $\exists \text{TK}$ such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, controlHandleMessage, l, \_)\}$, $sid = l[sid]$, $pid = l[p]$, $ph = l[h]$, $o = l[o]$ and $(in', b) \equiv (\mathcal{Q}', buffer)$.*

   packet: *$in = \{o{:}p\} \cup in'$, $\exists \text{TK}$ such that $\mathcal{Q} = \{\text{TK}(\_, switchHandlePacket, l, \_)\} \cup \mathcal{Q}'$, $o = l[o]$, $p = l[p]$ and $(in', b) \equiv (\mathcal{Q}', buffer)$.*

   packet-out: *$in = \{p\} \cup in'$, $\exists \text{TK}$ such that $\mathcal{Q} = \{\text{TK}(\_, hostHandlePacket, l, \_)\} \cup \mathcal{Q}'$, $p = l[p]$, and $(in', b) \equiv (\mathcal{Q}', buffer)$.*

   packet-in: *$in = \{new(pkt)\} \cup in'$, $\exists \text{TK}$ such that $\mathcal{Q} = \{\text{TK}(\_, sendIn, l, \_)\} \cup \mathcal{Q}'$, $pkt = l[p]$, and $(in', b) \equiv (\mathcal{Q}', buffer)$.*

Now, we can define the equivalence between an SDN state and an SDN-Actor state.

Definition 4.6 (equivalence). *An SDN state $S = \langle H, Sw, C \rangle$ and an SDN-actor state S_a are equivalent, written $S \equiv S_a$, if and only if:*

Host: *$\forall h(id, sid, o, in) \in H, \exists! a(\_, \_, h, \mathcal{Q}) \in S_a$ such that $(in, \emptyset) \equiv (\mathcal{Q}, \emptyset)$, $id = h[hid], sid = h[s]$, and $o = h[o]$.*

Switch: *$\forall s(id, ft, b, in) \in Sw, \exists! a(\_, \_, h, \mathcal{Q}) \in S_a$ such that $(in, b) \equiv (\mathcal{Q}, h[buffer])$, $id = h[sid]$, and $ft = h[flowT]$.*

Controller: *$C = c(top, cin)$ and $\exists! a(id, \_, h, \mathcal{Q}) \in S_a$ such that $(cin, \emptyset) \equiv (\mathcal{Q}, \emptyset)$, $related(top, \{h[srefs], h[href], h[ntw]\})$, and $\forall a(\_, \_, h', \_) \in S_a$, $id = h'[ctrl]$.*

Let us notice here that we use $related(top, \{h[srefs], h[href], h[ntw]\})$ to clarify that information about the topology is coherent in both the controller and the controller actor.

The following theorem ensures the soundness of our modelling. Essentially we guarantee that, for a given SDN network that follows the OpenFlow specification, any execution in the network has an equivalent execution in the SDN-Actor model. The proof can be found in the appendix. We denote as S_a^{ini} the SDN-Actor state defined in Definition 4.1, i.e., after executing method init_conf() and

all asynchronous calls to method sendIn containing the packets to be delivered. Furthermore, $S^{ini} \equiv S_a^{ini}$.

Theorem 4.7. *Let S^{ini} and S_a^{ini} be an SDN state and an SDN-Actor state, respectively.*

1. *For every execution $S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini}), \exists S_a^{ini} \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini})$ such that $S^n \equiv S_a^n$.*
2. *For every execution $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini}), \exists S^{ini} \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ such that $S^n \equiv S_a^n$.*

5. Implementing barriers using conditional synchronization

Barriers [11] have been designed to force a switch to handle previous control messages, and thus avoid problems such as the one described above.

Definition 5.1 (OF barrier). *Following OpenFlow [11], upon receipt of a* barrier message*, the switch must finish processing all previously-received controller messages, before executing any messages received after the* barrier message*.*

Figure 7 shows our modelling that intuitively consists in the controller not sending further messages to any switch on which a barrier has been activated, until this switch acknowledges that all previous control messages have been already processed. The main points in the implementation are:

1. The controller creates a future variable at L82 for every asynchronous task that it posts on all switches.
2. it keeps in barrierMap the list of future variables (not yet acknowledged) for each of the switches (putAdd in L82 adds the future variable to the list indexed by s1 in the map).
3. The controller keeps in barrierOn the set of switches with an active barrier.
4. A barrier on a switch consists in the controller awaiting on the list of future variables that the switch needs to acknowledge to ensure that its control messages have already been processed (method barrierRequest).
5. All control messages must be now preceded by a call to barrierWait that checks if the corresponding switch has an active barrier, L97. This is because while suspended in a barrier, the controller can start to process another controlHandleMessage unrelated to the previous one, but which affects (some of) the same switches for which a barrier was set. So, we cannot send messages to them until their barriers are set to off. Similarly, the call to barrierRequest must also be preceded by a call to barrierWait since barrierRequest is indeed modelling the send to the switch of a control message (the *barrier message*).

```
66  class Controller() {
67    Map<SwitchId,Switch> srefs={};
68    Map<HostId,Host> href={};
69    List<Link> ntw=[];
70    Map<SwitchId,List<Fut<Unit>> barrierMap={};
71    Set<SwitchId> barrierOn = ∅;
72    Unit addConfig(Map<SwitchId,Switch> sr, Map<HostId,Host> hr, List<Link> n){
73          /* references to switches and hosts and network topology initialized */
74    }
75    Unit controlHandleMessage(SwitchId sid, PortId o, PacketId p, PacketH h){
76         List<(SwitchId,MatchF,Action)> l=applyPolicy(sid,o,h);
77         List<SwitchId> ls = [];
78         while (not(isEmpty(l))) {
79              SwitchId s1; Action a1; MatchF m1;
80              (s1,m1,a1)=head(l);
81              barrierWait(s1);
82              Fut<Unit>f=lookup(srefs,s1)!switchHandleMessage(m1,a1);
83              barrierMap=putAdd(barrierMap,s1,f);
84              ls = add(ls,s1);
85              l=tail(l);
86         }
87         while(not(isEmpty(ls))) {
88              barrierWait(head(ls));
89              barrierRequest(head(ls));
90              ls=tail(ls);
91         }
92         barrierWait(sid);
93         Fut<Unit>f=lookup(srefs,sid)!sendOut(p);
94         barrierMap=putAdd(barrierMap,sid,f);
95    }
96    Unit barrierWait (SwitchId sid){
97           await not(contains(barrierOn,sid))?;
98    }
99    Unit barrierRequest (SwitchId sid){
100        barrierOn=add(barrierOn,sid);
101        List<Fut<Unit>> futSid=take(barrierMap,sid);
102        while (not(isEmpty(futSid)) {
103              Fut<Unit> fi=head(futSid);
104              await fi?;
105              futSid=tail(futSid);
106        }
107       barrierOn=delete(barrierOn,sid);
108   }
109 }
```

Figure 7: Extension of Controller class with barriers

Note that this is not a restriction on the type of controllers we model, but rather an effective way to encode barriers using actors and conditional synchronization (by means of the **await** instructions) that ensures the behaviour of OpenFlow barriers.

The next theorem states that our implementation of barriers via methods barrierRequest and barrierWait provide a sound encoding of the OF *barrier* messages in Definition 5.1.

Theorem 5.2 (soundness of barriers). *Given any state S in any execution of the SDN-Actor model right before executing L89 with switch sid as parameter (i.e., the state before activating a barrier over sid), and the state S' right before executing L90 (i.e., the state after receiving the acknowledgement of the barrier), the following holds:*

- *All* switchHandleMessage *and* sendOut *tasks in the queue of switch sid in state S have been completely executed in state S'.*
- *No* switchHandleMessage *nor* sendOut *task have been spawned over switch sid in any middle state between S and S'.*
- *No other* barrierRequest *call for switch sid is performed between S and S'.*

Proof.
Let us firstly define an invariant which holds for every possible state S'' of any execution of the SDN-Actor model:

$$\forall a(sid, \_, \_, \mathcal{Q}) \in S'' \text{ and } \forall \text{TK}(tk, m, \_, \_) \in \mathcal{Q},\ m \in \{\mathsf{switchHandleMessage}, \mathsf{sendOut}\}$$

$$\exists! a(cid, \_, h, \_) \in S'' \text{ such that } tk \in h[\mathsf{barrierMap}][\mathsf{sid}]$$

The invariant states that every spawned switchHandleMessage or sendOut task tk on a switch *sid* is recorded by means of a future variable in the list associated to *sid* in the barrierMap field of the controller (i.e. $tk \in h[\mathsf{barrierMap}][\mathsf{sid}]$). Note the abuse of notation $\in$ to check existence of an element in a *List* data-structure, and [] to access the value of a key in a *Map* data-structure. It can be seen that after making any asynchronous call to method switchHandleMessage (L82) or sendOut (L93), the corresponding future variable is always recorded in the barrierMap field (L83 and L94).

Now, given the controller of the state S, $a(cid, \_, h_c, \mathcal{Q}_c) \in S$, for every task $\text{TK}(tid, \mathsf{controlHandleMessage}, l_c, \mathsf{barrierRequest(l)}; s) \in \mathcal{Q}_c$, we have a derivation $S = S_0 \overset{cid.tid}{\rightsquigarrow^*} S_1 \longmapsto ... \longmapsto S_n \overset{cid.tid}{\longmapsto} S_{n+1} = S'$ such that S_1 is the global state after executing the micro-step transitions of such task until it stops at L104, and S_{n+1} is the first state where $h_c[\mathsf{barrierOn}]$ does not contain the switch $sw = l[\mathsf{sid}]$. Let us notice that if such stop is not performed, then every task in sw has already finished and barrierRequest is performed in a single macro-step ($S_0 = S_n$). Then, we know that $\forall i \in \{0, ..., n+1\}, \exists a(sw, \_, \_, \mathcal{Q}_i) \in S_i$ with $\mathcal{Q}_i = SHP_i \cup SO_i \cup SHM_i$ such that:

- SHP_i contains all switchHandlePacket tasks,
- SO_i contains all sendOut tasks, and,
- SHM_i contains all switchHandleMessage tasks.

By the definitions of the states S_1 and S_{n+1}, we know that $\forall i \in \{1, ..., n\}$, $sw \in h_c[\mathsf{barrierOn}]$. Hence, $\forall i \in \{1, ..., n\}$, the condition of the **await** instruction at L97 does not hold, thus, the task is suspended in state S_i, and, consequently, no switchHandleMessage nor sendOut task can be spawned in any state S_i. Therefore, $\forall i \in \{1, ..., n\}, \forall j \in \{i, ..., n\}, SHM_j$ (resp. SO_j) never contains more tasks than SHM_i (resp. SO_i). Similarly, no other call to barrierRequest can be performed due to the call to barrierWait in L88, which implies that there cannot be two active barriers over the same switch.

Finally, since sw no longer belongs to barrierOn in S_{n+1}, we know that $\forall fut \in h_c[\mathsf{barrierMap}][\mathsf{sid}]$, the task $l_c[fut]$ has finished, since for each variable fut, the **await** statement in L104 has succeeded. Moreover, using the invariant, we know that all the tasks in SHM_n and SO_n have their corresponding future variable in $h_c[\mathsf{barrierMap}][\mathsf{sid}]$, and therefore all of them have finished.

□

6. DPOR-based model checking of SDN-Actors

Model checking tools deal with a combinatorial blow-up of the state space (a.k.a. the state space explosion problem) that must be faced to solve real-world problems. This problem is exacerbated in the context of SDN programs, because of the concurrent and distributed nature of networks: all network components (switches, hosts, controllers) are distributed nodes that run in parallel and whose concurrent tasks can interact. As we have seen, a controller message sent from a switch can change the state of another switch, and affect the route of an incoming packet. Thus, a model checker needs to explore all possible reorderings of *dependent* tasks (i.e., those whose execution might interfere with each other) leading to a huge number of possible executions even for networks with a low number of nodes and packets. Additionally, the state space is unbounded because hosts may generate unboundedly many packets that could be simultaneously traversing the network.

There are two *incomplete* approaches to handle unbounded inputs: one is to impose a bound k on the number of packets of each type (as e.g. in[12]) and the other one is to use abstraction (as e.g. in [13]). In the former, the search space is exhausted for the considered input, but there could be bugs that only show up when more packets are considered. In the latter, abstraction requires to lose information and bugs may only show up when the omitted information is considered. Therefore, the sources of incompleteness are different, and the approaches can complement each other. Our tool SYCO uses the former, e.g., in Example 4.3 we have considered one packet (limit $k = 1$). The rest of the section presents the key features of our approach assuming such a k bound.

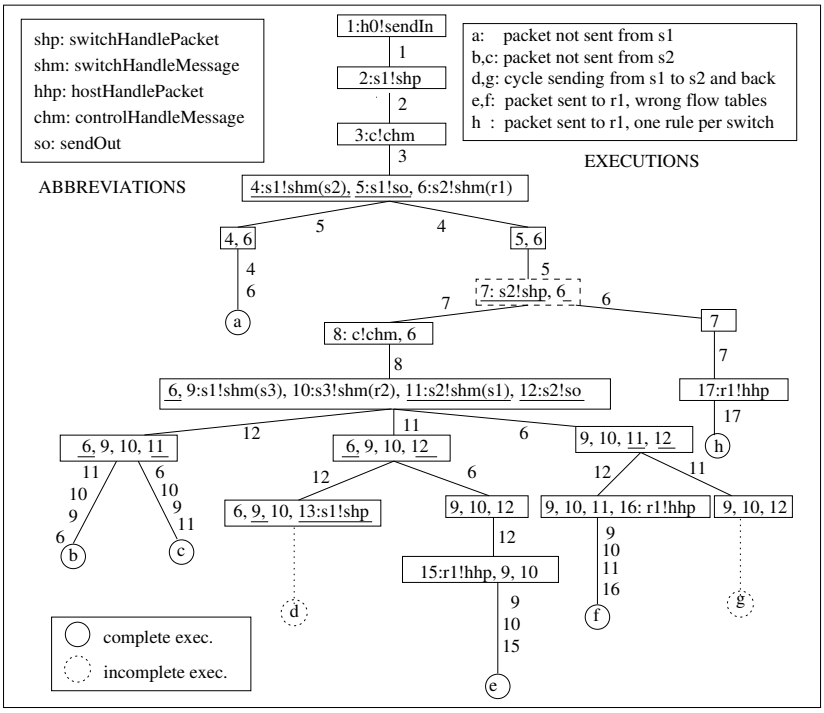


Figure 8: Search tree for running example w/o barriers (rightmost branch w/ barriers)

6.1. DPOR-based model checking in actors

DPOR [14] is able to dynamically identify and avoid the exploration of redundant executions and prune the search space exponentially. It is based on the idea of initially exploring an arbitrary interleaving of the various concurrent tasks, and *dynamically* tracking dependent interactions between them to identify backtracking points where alternative paths in the state space need to be explored. Two tasks are *independent* when changing their order of execution will not affect their combined effect. When DPOR is applied to actor systems, there are inherent reductions [15] because: (i) we can atomically execute each task (without re-orderings) until a return or an **await** instruction are found, as concurrency is non-preemptive and the active task cannot be interrupted. This avoids having to consider the reorderings at the level of instructions (as one must do in thread-based concurrency), and allows us to work at the level of tasks. (ii) Also, two tasks can have a dependency only if they belong to the same actor. This is because only the actor itself can modify its private memory.

Example 6.1. *Figure 8 shows the search tree computed by DPOR for our SDN-Actor program without barriers. It has no redundancy, i.e., each execution corresponds to a different behavior on the packet arrival and/or the actions installed in the flow tables (see top right descriptions). At each node (i.e., state),*

we show the available tasks. A task is given an identifier the first time it appears, and afterwards only its identifier is shown. Method names are abbreviated as shown in the top left, and parameters are omitted except in tasks executing switchHandleMessage*, for which we only include the switch identifier that is part of the* Action *to be installed. For instance,* 4:s1!shm(s2) *is a task with identifier* 4*, that will execute method* switchHandleMessage *on* s1 *and will add to its flow table the information that the packet must be sent to* s2*. Labels on the edges show the task(s) that have been executed. At each state, we underline the tasks which have an interacting dependency. The execution starts by executing the* init_conf *method in Example 4.2 with the instruction* sendIn *added in Example 4.3 which appears in the root. The next two steps have one task available, but in the fourth state we have tasks* 4 *and* 5*, belonging to the same actor, whose reordering needs to be considered (leading to branching), while* 6 *is independent of them. Out of the 8 branches of the tree, only the rightmost execution* (h) *corresponds to the correct behavior in which the packet is actually sent to* r1 *and the actions are installed in the flow tables in the expected order. In execution* (a) *the packet does not arrive at the destination because the* sendOut *is executed before the action has been installed. Executions* (d) *and* (g) *correspond to the cycle described in Section 2, each of them with different installations of actions.*

Importantly, we do not need specific optimizations to use the DPOR algorithm in [16] to model check SDN-Actors. The use of **await** (is already covered by DPOR and) does not require any change either and, as expected, the search tree for the implementation with barriers only contains branch (h). The difference arises from task 3 in the tree: in the presence of barriers, this leads to a state in which we have asynchronous calls 4 and 6 and task 3 suspended at the **await** in L104 (awaiting for the termination of 4 and then of 6). Therefore, the dependent tasks 4 and 5 will not coexist because 5 is not spawned until 4 and 6 terminate.

6.2. Entry-level and context-sensitive independence

When two tasks that belong to the same actor are found, in the context of DPOR techniques, independence is commonly over-approximated by requiring that actor fields accessed by one task are not modified by the other. In our model, all tasks posted on a given switch access its flow table, namely sendOut and switchHandlePacket read it and switchHandleMessage writes it. Thus, in principle, any task executing switchHandleMessage is considered dependent on the other two. This explains the tasks underlinings in the figure and the branching in the tree. When there are multiple packets traversing the network usually different packets access distinct entries in the flow table. This results in the inaccurate detection of many dependencies hence producing redundant executions. Using Constrained DPOR [5], we alleviate this state space explosion:

1. *Entry-level independence.* We adopt a finer-grained notion of *entry-level independence* for which an access to entry i is independent from an access to j if $i \neq j$. This aspect is not visible when considering a single packet as in the example, as all accesses to the flow table refer to the same entry.

However, by simply adding another packet to the erroneous program, the state explosion is huge and the system times out if entry-level independence is not implemented, while it computes 92 executions (exploring 761 states) with entry-level independence.

2. *Context-sensitiveness.* Even when two tasks t and p access the same entry, Constrained DPOR [5] introduces some further checks that avoid redundant explorations. If the state before executing both tasks satisfies a certain *independence annotation*, then the executions of p and q are guaranteed to commute. Hence, one of the derivations can be pruned and further exploration from it is avoided. For instance, executing two consecutive switchHandleMessage on the same entry might lead to the same state if the flow table contains duplicate entries, as our implementation allows. An example of independence annotation for these two tasks is the check of duplicate entries in the state.

Although entry-level independence in theory could be proved automatically by using SMT solvers (see [5]), this is not yet possible in our system, and we have declared annotations which are valid for any SDN model. Let us explain the most representative annotations for method switchHandleMessage(m,a):

1. indep(switchHandlePacket(pi,pk),!matchHead&Port(getHeader(pk),pi,m)) denotes that tasks executing switchHandleMessage(m,a) are independent of those executing switchHandlePacket(pi,pk) if the matched field of the message does not match the header and the input port of the packet (the condition is checked by the auxiliary function matchHead&Port).

2. indep(switchHandleMessage(m2,a2),indepSwitchMsgeMsge(m,a,m2,a2)) denotes that tasks executing switchHandleMessage(m,a) are independent of those executing switchHandleMessage(m2,a2) if the matched fields m and m2 are independent (they do not match with the same entries in the flow table), but actions a and a2 are equals (the condition is checked by the auxiliary function indepSwitchMsgeMsge).

6.3. Comparison of DPOR reductions with related work

Other model checkers for SDN programs have used DPOR-based algorithms before [12, 13]. According to the experiments in the NICE tool, DPOR only achieves a 20% reduction of the search space because even the finest granularity does not distinguish independent flows. The reason for this modest reduction might be that it does not take advantage of the inherent independence of the code executed by the distributed elements of the network (switches, host, clients), nor to the fact that barriers allow removing dependencies, as our actor-based SDN model does. In Kuai [13], a number of optimizations are defined to take advantage of these aspects. Such optimizations must be (1) identified and formalized in the semantics, (2) proven correct and, (3) implemented in the model checker. Instead, due to our formalization using actors, the optimizations are already implicit in the model and handled by the model checker

without requiring any extension. Another main difference with Kuai is that they make two important simplifications to the kind of SDNs they can handle: (i) they assume a simplified model of switches in which a switch gets suspended (i.e., does not process further packets nor controller messages) while awaiting a controller request. The error showed in Example 1 would thus not be captured. We do not make any simplification and thus a switch can start to process a new packet while awaiting the controller and can also receive other controller actions (triggered by other switches). (ii) It works on a class of SDNs in which the size of the controller queue is one. Therefore, it will not capture potential errors that arise due to the reordering of messages by the controller. In contrast, our model checker works on the general model of SDN networks.

7. Implementation and experimental evaluation

This section describes how to use our model checking tool and its visualization capabilities in Section 7.1, and then the experimental evaluation carried out on a series of standard SDN benchmarks in Section 7.2.

7.1. The model checking tool and its visualization capabilities

We have built an extension for property model checking on top of the SYCO tool [3]. It can be used through an online web interface available at: `http://costa.fdi.ucm.es/syco` by selecting the POR algorithm `CDPOR` and disabling the automatic generation of independence constraints. All benchmarks we are describing in this section can be found in the folder JSS19. In order to run the model checker, the user first opens one of these benchmarks and clicks over the button Apply. By default SYCO makes a full exploration of the execution. However, by using the Settings, it is possible to change the default options. In particular, by selecting Property checking, the exploration finishes after finding an execution trace that violates the property being checked. In order to define the property P under test, we add to the controller a new method called `error_message` and encode P as a Boolean function F_p using the programming language itself. Then, in all places where the property has to hold, we add an `if` statement checking the negation of F_p and if it holds we call asynchronously to `error_message` on the controller. Then property holds for the given input if and only if there is no trace in the execution tree including a call to `error_message`.

The result of executing the model checker is shown in the console at the bottom, where SYCO first prints the number of executions explored and the output state for each explored execution. The output state contains the actors modelling the controller, the switches and the hosts created during the execution. Each actor is represented as a term with three arguments: the actor identifier, the actor type or class, and the final values of their fields.

7.2. Checking SDN properties in case studies

To evaluate our approach, we have implemented a series of standard SDN benchmarks used in previous work [13, 17, 6]. Our goal is on the one hand to

show the versatility of our approach to check properties that are handled using different approaches in the literature (e.g., programming errors in the controller as in [17], safety policy violations as in [17, 13], or loop detection as in [6]). And, on the other hand, to show that we are able to handle networks larger than in related systems [13], but without requiring simplifications to the SDN models, nor extensions for DPOR reduction, and in spite of using a non-distributed model checker. We should note though that a precise comparison of figures is not possible due to the differences described in Section 6.3 and the use of different implementations of controllers.

Times are obtained on an Intel Core i7 at 3.4Ghz with 8GB of RAM (Linux Kernel 3.2). For each benchmark, we show in the second column the number of switches, hosts and packets, **Execs** corresponds to the number of different executions (i.e., branches in the search tree), **States** to the number of nodes in the search tree, and **Time** is the time taken by the analysis in ms. Results are shown in Figure 9.

Controller with load balancer [6] (LB/LBB). This corresponds to the controller of [6], similar to our running example. It performs stateless load balancing among a set of replica identified by a virtual IP (VIP) address. When receiving packets destined to a VIP, the controller selects a particular host and installs flow rules along the entire path. For a buggy controller without barriers (LB) and a network with 3 switches and 3 hosts, we detect that there is a forwarding loop (i.e., that a packet reaches a switch more than once) in 9ms after exploring 21 states. For this, we have added to the switches a field to store the packet identifiers that they have already received, and when the same packet reaches it, it sends an error message, which is observable from the final state. We are able to scale this version up to 302 hosts and 300 packets. Once we check the correct version with barriers (LBB), we are able to scale up to 127 hosts and 125 packets. As it can be observed, for the largest network, 1499 states are explored and in all cases we verify that the traffic is balanced. The experiments in [6] do not specify the time to detect the bug for this controller (they only mentioned that their analysis finishes in less than 32s in the vast majority of cases). Nevertheless, the underlying techniques to find the bugs are unrelated (see Section 8), and thus time comparison is not meaningful.

SSH controller [13] (SSHE/SSHB). This case study is based on a controller that dynamically modifies the behaviors of the switches as follows: it can update the switches with a rule that states that no SSH packets are forwarded, and another that states that all non-SSH packets are forwarded. We have two versions of the SSH controller. The first three evaluations correspond to an erroneous SSH controller that installs the rule to forward packets and the rule to drop SSH packets with the same priority, and thus the safety policy can be violated. As in [13], we evaluate a network with 2 switches and 2 hosts. As for packets, we write `100ssh`, `120other`, and `50each` to indicate that we send 100 SSH packets, 120 non-SSH packets and 50 of each type. We detect the error by checking in the switch if two contradictory drop and forward packet actions are received

Name	SxHxP	Execs	States	Time
LB	3x52x50	4	313	1305
LB	3x102x100	4	613	7301
LB	3x202x200	4	1213	38203
LB	3x302x300	4	1813	110220
LBB	3x52x50	1	599	11117
LBB	3x77x75	1	899	31644
LBB	3x102x100	1	1199	68059
LBB	3x127x125	1	1499	127740

(a) Controller with load balancer.

Name	SxHxP	Execs	States	Time
SSH	2x2x100ssh	1	407	83824
SSH	2x2x120oth	1	490	146151
SSH	2x2x50each	1	410	117245
SSH	2x2x2cor	179	1691	1340
SSHB	2x2x2	6	120	104
SSHB	2x2x3	65	1419	2506
SSHB	3x3x4	421	10951	33470

(b) SSH controller.

Name	SxHxP	Execs	States	Time
LE	3x3x2	3	71	42
LE	3x3x5	10	383	355
LE	6x3x2	5	217	272
LE	6x3x5	16	1040	2045
LE	9x2x2	10	787	4570
LE	15x2x2	16	2074	49274

(c) Network authentication with learning.

Name	SxHxP	Execs	States	Time
MIb	1x5x12	32	599	1029
MIb	1x5x14	64	1107	2730
MIb	1x5x16	748	9418	24870
MIb	1x8x20	2242	45539	153419
MI	1x5x8	32	1004	865
MI	1x5x10	256	9436	9176
MI	1x5x12	960	17941	29675
MI	1x8x14	1727	55200	119908

(d) Firewall with migration.

Figure 9: Experimental results.

for the same entry. The results that we obtain for 1 packet suggest higher performance of our approach: in [13] they find the bug in 0.1s and we do it in 0.004s or 0.007s, depending on the type of packet. The last evaluation 2 cor corresponds to the correct SSH controller for which we achieve a notable improvement as we have now less tasks that match the same entry (as priority is different). The row SSHB is a correct implementation with barriers that reduces the number of executions for 2 packets notably because it guarantees that forward rules are installed and thus switches will not send further requests. They prove the correctness for SSHB-2-2 in 6.4 seconds by exploring 13 states, we explore 15 states (in 6ms) or 18 states (in 8 ms), depending on the type of packet. Furthermore we are able to scale up to 3 hosts and 3 switches.

Network authentication with learning [17, 13] (LE). This implements a composition of a learning switch with authentication in [17]. Also, [13] evaluates a MAC learning controller but using a different implementation. LE implements a controller with barriers for which we can verify flow-table consistency and that the packet flows satisfy the intended policy. We have considered configurations of 3x3, 6x3, 9x2 and 15x2. When compared to [13], we handle larger sizes of networks and for similar sizes, we explore less **States** in less **Time**. We note that this might be due to the differences pointed out in Section 6.3 and different implementations of the controller.

Firewall with migration[17] (MIb/MI). MI is the implementation of a firewall that supports migration of trusted hosts. A host is trusted if it either sent/received (on some switch) a message through/from port 1. Thus, when a trusted host migrates to a new switch, the controller will remember it was trusted before and will allow communication from either port. For the same network 1x5 as [17], we can scale the number of packets up to 12 packets that actually modify the data base for trusted hosts. We can keep on adding more packets if those do not affect the shared data base. In MIb, we introduce the same bug in the controller as [17], which forgets to check if trusted on events from port 2. We detect the error by checking in the final state of the derivations that a packet arrives to a host that is not in the trusted data base. The scalability of MI and MIb are rather similar. However, we can handle larger sizes of networks (1x8). Both [17] and us find the bug in a negligible time.

8. Conclusions

We have proposed an actor-based framework to model and verify SDN programs. A unique feature of our approach is that we can use existing advanced verification algorithms without requiring any specific extension to handle SDN features. This has allowed us to model and analyse several SDN scenarios: a controller with load balancer, an SSH controller, a learning switch with authentication, and a firewall with migration. Experiments have given evidence of the versatility and scalability of our approach.

We conclude with a review of related work in verification of software-defined networks and some directions for future work.

8.1. Related work

Static and Dynamic verification.. The last years have witnessed the development of many static and dynamic techniques for verification that are closely related to our approach. Static approaches have the main advantage that, when the property can be proved, it is ensured for any possible execution, while using dynamic analysis only guarantees the property for the considered inputs. As a counterpart, in order to cover all possible behaviors, static analysis needs to perform abstraction, which can give a don't-know answer, and, possibly, false positives. In [17], the work on Horn-based verification is lifted to the SDN programming paradigm, but excluding barriers. Using this kind of verification, one can prove safety invariants on the program. Our framework can additionally check liveness invariants (e.g., loop detection) by inspecting the traces computed by the model checker. Static algebraic techniques are used in NetKAT [18, 19, 20], to prove properties of SDN programs. NetKAT does not include primitives for concurrency, and has a significantly higher level of abstraction. Therefore capturing features and scenarios we are interested in would be difficult. In [21], a particular type of attacks in the context of SDN networks has been modeled in Maude using the so-called hierarchically structured composite actor systems described in [22]. This work does not provide a general model for SDN networks and, besides, barriers are not considered. On the other hand, it applies a statistical model checker, which requires to have a given scheduler for the messages. Such scheduler determines the exact order in which messages are handled while our framework captures all possible behaviours. Hence, both their aim and their SDN model are radically different from ours.

Concerning dynamic techniques, our work is mostly related to the model checkers NICE and Kuai for SDN programs, which have been compared in detail in Section 6.3. Our approach could be adapted to apply abstractions that bound the size of buffers [13] and to consider environment messages [23]. The approach of [6, 24] is based on analyzing dynamically given snapshots of the network from real executions. Instead, we try to find programming errors by inspecting only the SDN program and considering all possible execution traces, thus enabling verification at system design time.

Data and Control-plane verification. There is a substantial body of work on verification techniques for SDN focussing specifically on the data or the control plane. Data-plane approaches include: Anteater [25], which uses static analysis via SAT solving; FlowChecker [26], which applies symbolic model-checking to OpenFlow configurations; VeriFlow [27], which provides an infrastructure to check data-plane properties in real-time. Control-plane approaches include: Flowlog [28], a declarative language to program SDN controllers, which uses the Alloy model-checker to perform verification; [29], which uses differential analysis to discover bugs in different versions of the same controller program.

We stress that our approach targets both control and data-plane, and in particular it is capable of detecting bugs that arise from their interaction. Moreover, concurrency and barriers are not considered in the mentioned works.

Quantitative verification. In [30], SDN components are modelled via a quantitative process algebra. Their focus is on quantitative properties, e.g., latency and congestion. In particular, concurrency and barriers are not considered.

Network verification via actors. Another actor-based verification framework is *Rebeca* (see [31] for a survey). Rebeca supports a variety of state-reduction techniques, and has been used to model and verify wireless networks [32, 33]. Our approach uses the ABS language and the SYCO tool. SYCO includes recent DPOR techniques [5, 16] which, by exploiting specific features of SDNs, enabled us to better scale and analyse larger networks.

8.2. Future Work

Although we did not explore it in this article, the encoding we provide opens the door to apply a range of techniques other than model checking. For instance, static analysis, runtime monitoring or simulation of network behavior can be done now using the ABS toolsuite [7]. Other tools and methods for verification of message-passing and concurrent-object systems could be also easily adapted [34, 35, 36, 37]. In addition, because the encoding is not very far from the original flow tables, both model extraction from existing network code and code generation from an actor model should be achievable with a small extension of the tool. This is left for future work.

Acknowledgments This work was partially funded by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish FPU Grant FPU15/04313, the Spanish MINECO projects TIN2015-69175-C4-2-R, TIN2015-69175-C4-3-R, the Spanish MCIU, AEI and FEDER (EU) through projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33 and the CM project S2018/TCS-4314, the ERC starting grant Profoundnet (679127) and a Leverhulme Prize (PLP-2016-129).

References

[1] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, 1986.

[2] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: FMCO, 2010, pp. 142–164.

[3] E. Albert, M. Gómez-Zamalloa, M. Isabel, SYCO: a systematic testing tool for concurrent objects, in: CC, 2016, pp. 269–270. doi:10.1145/2892208.2892236.

[4] E. Albert, M. Gómez-Zamalloa, A. Rubio, M. Sammartino, A. Silva, Sdn-actors: Modeling and verification of SDN programs, in: FM, 2018, pp. 550–567. doi:10.1007/978-3-319-95582-7_33.

[5] E. Albert, M. Gómez-Zamalloa, M. Isabel, A. Rubio, Constrained dynamic partial order reduction, in: CAV, 2018, pp. 392–410. doi:10.1007/978-3-319-96142-2_24.

[6] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, M. T. Vechev, Sdnracer: concurrency analysis for software-defined networks, in: POPL, 2016, pp. 402–415. doi:10.1145/2908080.2908124.

[7] The ABS tool suite, `http://abs-models.org`.

[8] F. S. de Boer, D. Clarke, E. B. Johnsen, A Complete Guide to the Future, in: ESOP, Vol. 4421, 2007, pp. 316–330.

[9] A. Guha, M. Reitblatt, N. Foster, Machine-verified network controllers, in: PLDI, 2013, pp. 483–494. doi:10.1145/2491956.2462178.

[10] K. Sen, G. Agha, Automated Systematic Testing of Open Distributed Programs, in: FASE, 2006, pp. 339–356.

[11] Openflow switch specification, version 1.4.0 (October 2013).

[12] M. Canini, D. Venzano, P. Peresíni, D. Kostic, J. Rexford, A NICE way to test openflow applications, in: NSDI, 2012, pp. 127–140.

[13] R. Majumdar, S. D. Tetali, Z. Wang, Kuai: A model checker for software-defined networks, in: FMCAD, 2014, pp. 163–170. doi:10.1109/FMCAD.2014.6987609.

[14] C. Flanagan, P. Godefroid, Dynamic partial-order reduction for model checking software, in: POPL, 2005, pp. 110–121.

[15] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha, Transdpor: A novel dynamic partial-order reduction technique for testing actor programs, in: FMOODS/FORTE, 2012, pp. 219–234.

[16] E. Albert, P. Arenas, M. G. de la Banda, M. Gómez-Zamalloa, P. J. Stuckey, Context-sensitive dynamic partial order reduction, in: CAV, Vol. 10426, 2017, pp. 526–543.

[17] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, A. Valadarsky, Vericon: towards verifying controller programs in software-defined networks, in: PLDI, 2014, pp. 282–293. doi:10.1145/2594291.2594317.

[18] N. Foster, D. Kozen, M. Milano, A. Silva, L. Thompson, A coalgebraic decision procedure for netkat, in: POPL, 2015, pp. 343–355. doi:10.1145/2676726.2677011.

[19] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, D. Walker, Netkat: semantic foundations for networks, in: POPL, 2014, pp. 113–126. doi:10.1145/2535838.2535862.

[20] R. Beckett, M. Greenberg, D. Walker, Temporal netkat, in: PLDI, 2016, pp. 386–401. doi:10.1145/2908080.2908108.

[21] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, V. Nigam, Slow TCAM exhaustion ddos attack, in: SEC, 2017, pp. 17–31.

[22] J. Eckhardt, T. Mühlbauer, J. Meseguer, M. Wirsing, Statistical model checking for composite actor systems, in: WADT, 2012, pp. 143–160.

[23] D. Sethi, S. Narayana, S. Malik, Abstractions for model checking SDN controllers, in: FMCAD, 2013, pp. 145–148.

[24] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: Static checking for networks, in: NSDI, 2012, pp. 113–126.

[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, S. T. King, Debugging the data plane with anteater, in: ACM SIGCOMM, 2011, pp. 290–301. doi:10.1145/2018436.2018470.

[26] E. Al-Shaer, S. Al-Haj, Flowchecker: configuration analysis and verification of federated openflow infrastructures, in: SafeConfig, 2010, pp. 37–44. doi:10.1145/1866898.1866905.

[27] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P. B. Godfrey, Veriflow: Verifying network-wide invariants in real time, in: NSDI, 2013, pp. 15–27.

[28] T. Nelson, A. D. Ferguson, M. J. G. Scheer, S. Krishnamurthi, Tierless programming and reasoning for software-defined networks, in: NSDI, 2014, pp. 519–531.

[29] T. Nelson, A. D. Ferguson, S. Krishnamurthi, Static differential program analysis for software-defined networks, in: FM, 2015, pp. 395–413. doi:10.1007/978-3-319-19249-9_25.

[30] V. Galpin, Formal modelling of software defined networking, in: IFM, 2018, pp. 172–193. doi:10.1007/978-3-319-98938-9_11.

[31] M. Sirjani, M. M. Jaghoori, Ten years of analyzing actors: Rebeca experience, in: Formal Modeling: Actors, Open Systems, Biological Systems, 2011, pp. 20–56. doi:10.1007/978-3-642-24933-4_3.

[32] B. Yousefi, F. Ghassemi, R. Khosravi, Modeling and efficient verification of wireless ad hoc networks, Formal Asp. Comput. 29 (6) (2017) 1051–1086. doi:10.1007/s00165-017-0429-z.

[33] E. Khamespanah, M. Sirjani, K. Mechitov, G. Agha, Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking, STTT 20 (5) (2018) 547–561. doi:10.1007/s10009-017-0480-3.

[34] M. Christakis, A. Gotovos, K. F. Sagonas, Systematic Testing for Detecting Concurrency Errors in Erlang Programs, in: ICST, 2013, pp. 154–163.

[35] S. Lauterburg, R. K. Karmani, D. Marinov, G. Agha, Basset: a tool for systematic testing of actor programs, in: SIGSOFT FSE, 2010, pp. 363–364. doi:10.1145/1882291.1882349.

[36] A. Bouajjani, M. Emmi, C. Enea, J. Hamza, Tractable refinement checking for concurrent objects, in: POPL, 2015, pp. 651–662. doi:10.1145/2676726.2677002.

[37] H. Liang, X. Feng, A program logic for concurrent objects under fair scheduling, in: POPL, 2016, pp. 385–399. doi:10.1145/2837614.2837635.

Appendix A. Soundness Proofs

Appendix A.1. Proof of Theorem 4.7

Let α be the one-to-one function that pairs each element of S with its unique actor in S_a such that $S \equiv S_a$.

Assumption 1. *Given an SDN state $S = \langle H, Sw, C\rangle$ and an SDN-Actor state S_a such that $S \equiv S_a$, $C = c(top, cin)$ and $a(c, tk, h, \mathcal{Q}) \in S_a$ with $\alpha(C) = a(c, tk, h, \mathcal{Q})$, then*

$$applyPol(top, sid, o, ph) = applyPolicy(sid, o, ph)$$

Assumption 2. *$S^{ini} \equiv S_a^{ini}$, where S_a^{ini} is the SDN-Actor state defined in Def.4.1, that is, the state after executing method* init_conf() *and all asynchronous calls to method* sendIn *containing the packets to be delivered.*

Definition Appendix A.1 (succ(S) (succ(S_a))). *Given an SDN(resp. SDN-Actor) state S (resp. S_a), succ(S) (resp. succ(S_a)) denotes the set of final states of the executions in exec(S) (resp. exec(S_a)).*

Lemma Appendix A.2. *Given an SDN state and an SDN-Actor state S_a such that $S \equiv S_a$ and $S \in succ(S^{ini})$ and $S_a \in succ(S_a^{ini})$,*

1. *If $S \to S'$, then $\exists S_a'$ such that $S_a \longmapsto S_a'$ and $S' \equiv S_a'$.*
2. *If $S_a \longmapsto S_a'$, then $\exists S'$ such that $S \to S'$ and $S' \equiv S_a'$.*

Proof.
Let us see reason about both points at the same time. We distinguish several cases depending on the semantics rule applied during the step $S \to S'$. Let S and S' be $\langle H, Sw, C\rangle$ and $\langle H', Sw', C'\rangle$, respectively.

1. If rule (SI) is applied,
   - $H = H'' \cup \{h(id, sid, o, in \cup \{new(p)\})\}$ and $H' = H'' \cup \{h(id, sid, o, in)\}$
   - $Sw = Sw'' \cup \{s(sid, ft, b, in')\}$ and $Sw' = Sw'' \cup \{s(sid, ft, b, in' \cup \{o{:}p\})\}$
   - $C = C'$

   Moreover, we know that $S \equiv S_a$, hence $\alpha(h(id, sid, o, in \cup \{new(p)\})) = a(hoid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendIn}, l, \_)\} \cup \mathcal{Q}'$. Thus, task tid can be executed in state S_a by actor $hoid$ and it will send a task shp (where shp stands for switchHandlePacket) to $h[\mathsf{s}]$ such that $h[\mathsf{s}] = \alpha(s(sid, ft, b, in')) = a(soid, \_, h2, \mathcal{Q}'') \in S_a$ (by the equivalence of S and S_a).

   a) $S_a = S_a'' \cup \{a(hoid, \_, h, \mathcal{Q}), a(soid, \_, h2, \mathcal{Q}''), \}$
   b) $S_a' = S_a'' \cup \{a(hoid, \_, h, \mathcal{Q}'), a(soid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\})\}$, such that $l2[\mathsf{p}] = l[\mathsf{p}]$ and $l2[\mathsf{o}] = h[\mathsf{o}]$.

By such equivalence we know that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(in, \emptyset) \equiv (\mathcal{Q}', \emptyset)$ and by a) and b) $(in' \cup \{o : p\}, \emptyset) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, \emptyset)$ Consequently, $S' \equiv S'_a$.

2. If rule (HHP) is applied,

   - $H = H'' \cup \{h(id, sid, o, in \cup \{p\})\}$ and $H' = H'' \cup \{h(id, sid, o, in)\}$
   - $Sw = Sw'$ and $C = C'$

   We also know $S \equiv S_a$, hence $\alpha(h(id, sid, o, in \cup \{p\})) = a(hoid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{hhp}, l, \_)\} \cup \mathcal{Q}'$ (where hhp stands for $\mathsf{hostHandlePacket}$). Therefore task tid can be executed in state S_a by $hoid$ and the packet is removed from the buffer.

   - $S_a = S''_a \cup \{a(hoid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(hoid, \_, h, \mathcal{Q}')\}$.

   Furthermore, $p = l[p]$. By the equivalence between S_a and S, we know that $(in \cup \{p\}, \emptyset) \equiv (\mathcal{Q}, \emptyset)$ and then, (1) $(in, \emptyset) \equiv (\mathcal{Q}', \emptyset)$ and also, (2) $\langle H'', Sw, C\rangle \equiv S''_a$. Consequently, $S' \equiv S'_a$.

3. If rule (SHP_1) is applied,

   - $H = H'' \cup \{h(id, sid, o, in'\}$ and $H' = H'' \cup \{h(id, sid, o, in' \cup \{p\})\}$
   - $Sw{=}Sw'' \cup \{s(sid, ft, b, in \cup \{o2 : p\})\}$ and $Sw'{=}Sw'' \cup \{s(sid, ft, b, in)\}$
   - $C = C'$ and $\langle send(id)\rangle = lookup(ft, \langle header(p), o\rangle)$.

   Moreover, we know $S \equiv S_a$, hence $\alpha(s(sid, ft, b, in)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Therefore, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(id)$. Therefore, the check in line 21 does not succeed, but the check in line 22 does, and, consequently, it spawns a task hhp to $\alpha(h(id, sid, o, in')) = a(hoid, \_, h2, \mathcal{Q}'') \in S_a$. As a result

   - $S_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

   where $l[\mathsf{p}] = l2[\mathsf{p}]$. We also know by $S \equiv S_a$, that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(in \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Since $p = l2[p]$, then $(in' \cup \{p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}, h[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

4. If rule (SHP_2) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, in \cup \{o2 : p\}), s(sid', ft', b', in')\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, in), s(sid', ft', b', in' \cup \{o : p\})\}$
   - $H = H',\ C = C'$ and $send(sid', o) = lookup(ft, \langle header(p), o2\rangle)$.

Moreover, we know that $S \equiv S_a$, so $\alpha(s(sid, ft, b, in)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(soid', o)$. Therefore, the check in line 21 succeeds and, consequently, it spawns a task shp to $\alpha(s(sid', ft', b', in')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$. As a result

- $S_a = S''_a \cup \{a(soid', \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{s(soid', \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

where $l[p] = l2[p]$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(in \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(in' \cup \{o : p\}, b') \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, h2[\mathsf{buffer}])$ because $o = l2[\mathsf{o}], p = l2[\mathsf{p}]$. Consequently, we have that $S' \equiv S'_a$.

5. If rule SHP_3 is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, in \cup \{o : p\})\}$ and $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o : p\}, in)\}$
   - $C = c(top, in')$ and $C' = c(top, in' \cup \{\mathsf{pktIn(sid,o,id(p),header(p))}\})$
   - $H = H'$ and $\perp = lookup(ft, \langle header(p), o\rangle)$.

   Moreover, we know that $S \equiv S_a$, so $\alpha(s(sid, ft, b, in)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, so the action returned by flowT is the same that the one returned by ft, that is $\perp$. Therefore, the checks in line 21 and 22 do not succeed and, consequently, it (1) spawns a task chm (where chm stands for$\mathsf{controlHandleMessage}$) to $h[ctrl] = \alpha(c(top, in')) = a(coid, \_, h2, \mathcal{Q}'') \in S_a$, and, (2) it stores the packet and the port $o : p$ in $h[\mathsf{buffer}]$. As a result

   - $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q}), a(coid, \_, h2, \mathcal{Q}'')\}$.
   - $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}'), a(coid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{chm},\ l2, \_)\})\}$ and $h' := h$ but $h'[\mathsf{buffer}] := h[\mathsf{buffer}] \cup \{(p, o)\}$.

   Furthermore, we know that (1) $\langle H, Sw'', C\rangle \equiv (S''_a \cup \{a(coid, \_, h2, \mathcal{Q}'')\})$ and (2) $(in, b \cup \{o : p\}) \equiv (\mathcal{Q}', h[\mathsf{buffer}] \cup \{(p, o)\})$. Moreover, we have $(in', \emptyset) \equiv (\mathcal{Q}'' \cup \{task\ (\_, \mathsf{chm}, l2, \_\}, \emptyset)$ since $soid = l2[\mathsf{sid}],\ p = l[\mathsf{p}]$ and $o = l[\mathsf{o}]$. Consequently, we have $S' \equiv S'_a$.

6. If rule (SO_1) is applied,

   - $H = H'' \cup \{h(id, sid, o, in'\}$ and $H' = H'' \cup \{h(id, sid, o, in' \cup \{o : p\})\}$
   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o2 : p\}, in \cup \{\mathsf{pktOut(ph)}\})\}$ and $Sw' = Sw'' \cup \{s(sid, ft, b, in)\}$
   - $C = C'$, $ph = header(p)$ and $send(id) = lookup(ft, \langle header(p), o\rangle)$.

We also know $S \equiv S_a$, so $\alpha(s(sid, ft, b \cup \{o2 : p\}, in \cup \{\mathsf{pktOut(ph)}\})) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, so the action returned by flowT is the same that the one returned by ft, that is $send(soid, o)$. Therefore, the check in line 28 does not succeed, but the check in line 29 does, and, consequently, it spawns a task hhp to $\alpha(h(id, sid, o, in')) = a(hoid, \_, h2, \mathcal{Q}'') \in S_a$. As a result

- $S_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}), a(soid, \_, h', \mathcal{Q}')\}$

where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(in \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(in' \cup \{o : p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}, h[\mathsf{buffer}])$ since $(p, o) = take(h[\mathsf{buffer}], l[\mathsf{pi}]) = (l2[\mathsf{p}], o)$. Consequently, we have that $S' \equiv S'_a$.

7. If rule (SO_2) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o2 : p\}, in \cup \{\mathsf{pktOut(ph)}\}), s(sid', ft', b', in')\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, in), s(sid', ft', b', in' \cup \{o : p\})\}$
   - $H = H',\ C = C'$ and $send(sid', o) = lookup(ft, \langle header(p), o2\rangle)$.

   Moreover, we know $S \equiv S_a$, so $\alpha(s(sid, ft, b \cup \{o2 : h\}, in \cup \{\mathsf{pktOut(ph)}\})) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and, by the equivalence, we know that $ft = h[\mathsf{flowT}]$, so the action returned by flowT is the same that the one returned by ft, that is $send(soid', o)$. Therefore, the check in line 28 succeeds, and consequently, it spawns a task shp to $\alpha(s(sid', ft', b', in')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$. As a result

   - $S_a = S''_a \cup \{a(soid', \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{s(soid', \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

   where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(in \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(in' \cup \{o : p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, h[\mathsf{buffer}])$ since $(p, o) = take(h[\mathsf{buffer}], l[\mathsf{pi}]) = (l2[\mathsf{p}], l2[\mathsf{o}])$. Consequently, we have that $S' \equiv S'_a$.

8. If rule (SO_3) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o : p\}, in \cup \{\mathsf{pktOut(ph)}\})\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, in)\}$
   - $H{=}H',\ C{=}C'$, and $ph{=}header(p)$ and $\perp{=}lookup(ft, \langle header(p), o\rangle)$.

Moreover, we know that $S \equiv S_a$, so $\alpha(s(sid, ft, b \cup \{o{:}p\}, in \cup \{\mathsf{pktOut(ph)}\}))$ $= a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $\perp$. Therefore, the checks in line 28 and 29 do not succeed, and consequently, it drops the packet without spawning any other task. As a result

- $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}')\}$

where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(in, b) \equiv (\mathcal{Q}', h'[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

9. If rule (SHM) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, in \cup \{\mathsf{modState}(\langle \mathsf{ph},\ \mathsf{o}\rangle \mapsto \mathsf{a})\})\}$
   - $Sw' = Sw'' \cup \{s(sid, put(ft, \langle ph, o\rangle, a), b, in)\}$
   - $H = H',\ C = C'$.

   Moreover, we know that $S \equiv S_a$, so $\alpha(s(sid, ft, b, in \cup \{\mathsf{modState}(\langle \mathsf{ph},\mathsf{o}\rangle \mapsto \mathsf{a})\}))$ $= a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{switchHandleMessage}, l, \_)\} \cup \mathcal{Q}'$. Therefore, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, hence $put(ft, m, a) = put(h[\mathsf{flowT}], \langle ph, o\rangle, a)$, and $m = \langle ph, o\rangle$ because of Assumption 1, that is, *applyPol* and *applyPolicy* behaves similarly for $\alpha(s)$ and s, $\forall s \in Sw$ and $\alpha(s) \in S$.

   - $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}')\}$

   where $h' := h$ but $h'[\mathsf{flowT}] := put(h[\mathsf{flowT}], m, a)$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(in, b) \equiv (\mathcal{Q}', h[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

10. If rule (CHM) is applied,

    - $Sw = Sw'' \cup \{s(sid, ft, b, in)\}$
    - $Sw' = Sw''_{ms} \cup \{s(sid, ft, b, in \cup ms_{sid} \cup \{\mathsf{pktOut(ph)}\}\}$,
    - $H = H'$, $C = c(top, cin \cup \{\mathsf{pktIn(sid,o,pid,ph)}\})$ and $C' = c(top, cin)$

    where $ms = applyPol(top, sid, o, ph)$ and $ms_{id} = \{m | \langle id, m\rangle \in ms\}$. Moreover, we know that $S \equiv S_a$, so $\alpha(c(top, cin \cup \{\mathsf{pktIn(sid,o,pid,ph)}\}))$ $= a(coid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{chm}, l1, \_)\} \cup \mathcal{Q}'$ (where chm stands for chm) such that $soid = l1[\mathsf{sid}]$, $o = l1[\mathsf{o}]$, $pid = l1[\mathsf{p}]$ and $ph = l1[\mathsf{h}]$. Therefore, task tid can be executed, and by the equivalence of S and S_a

and Assumption 1, we know that the list l is equivalent to ms, in the sense that it contains exactly the same switches and the same actions. Hence, in line 41, actor *coid* spawns tasks shm to every switch in the list l (where shm stands for switchHandleMessage) and finally, it spawns a a task sendOut to actor *soid* in line 43. Let us see the equivalence between S'_a and S'.

- $\forall s(sid', ft', b', in') \in Sw''$ such that $ms_{sid'} = \emptyset$, then $s(sid, ft', b', in') \in Sw''_{ms}$. Furthermore, if $ms_{sid'} = \emptyset$, then $sid' \notin l$, hence sid' will not receive any message. Therefore, $\alpha(s(sid', ft', b', in')) \in S_a \cap S'_a$.
- $\forall s(sid', ft', b', in') \in Sw''$ such that $ms_{sid'} \neq \emptyset$,then $s(sid, ft', b', in' \cup ms_{sid'}) \in Sw''_{ms}$. Then, *coid* will spawn as many shm tasks as messages in $\{(m, a) | (soid', m, a) \in l\}$ (and in $ms_{soid'}$). By $S_a \equiv S$, we know $\alpha(s(sid, ft', b', in')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$ and $(in', b') \equiv (\mathcal{Q}'', h2[\mathsf{buffer}])$. Furthermore, $a(soid', \_, h2, \mathcal{Q}'' \cup tks_{l,soid'}) \in S'_a$, where $tks_{l,soid'} := \{\text{TK}(\_, \mathsf{shm}, l', \_) \mid (soid', m, a) \in l, l'[\mathsf{m}] := m,\ l'[\mathsf{a}] := a\}$ which is equivalent to the information contained in $ms_{sid'}$. Then, $(in' \cup ms_{sid'}, b) \equiv (\mathcal{Q}'' \cup tks_{l,soid'}, h2[\mathsf{buffer}])$.
- Regarding the switch $s(sid, ft, b, in)$, by the equivalence of S and S_a, we know that $\alpha(s(sid, ft, b, in)) = a(soid, \_, h1, \mathcal{Q}_{soid}) \in S_a$ and since $soid = l[\mathsf{sid}]$, actor *coid* spawns a task sendOut, and as many shm tasks as messages in $\{(m, a) | (soid, m, a) \in l\}$, and then $a(soid, \_, h1, \mathcal{Q}_{soid} \cup tks_{l,soid} \cup \{\text{TK}(\_, \mathsf{sendOut}, l', \_)\}) \in S'_a$. Again, by the equivalence of S and S_a, we know that $(in, b) \equiv (\mathcal{Q}_{soid}, h1[\mathsf{buffer}])$ and, since $tks_{l,soid}$ is the equivalent information contained in ms_{sid}, then we get that $(in \cup ms_{sid}, b) \equiv (\mathcal{Q}_{soid} \cup tks_{l,soid}, h1[\mathsf{buffer}])$. Finally, pktOut(ph) contains the equivalent information to $t_{out} := \text{TK}(\_, \mathsf{sendOut}, l', \_)$, thus we get $(in' \cup (ms_{sid'} \cup \{\mathsf{pktOut(ph)}\}, b) \equiv (\mathcal{Q}'' \cup tks_{l,soid'} \cup \{t_{out}\}, h[\mathsf{buffer}])$.
- Regarding the controller C, we know that $(cin \cup \{\mathsf{pktIn(sid,o,pid,ph)}\}, \emptyset) \equiv (\mathcal{Q} \cup \{\text{TK}(tkid, \mathsf{chm}, l1, \_)\}, \emptyset)$. Furthermore, by Assumption 1, we know that $related(top, \{h[\mathsf{srefs}, \mathsf{href}, \mathsf{ntw}]\})$. As a consequence $(cin, \emptyset) \equiv (\mathcal{Q}, \emptyset)$.

All in all, we conclude that $S' \equiv S'_a$.

Let us notice here that even though we have distinguished the different cases depending on the semantics rule for SDN networks, the previous reasoning also includes each possible execution of a task in the SDN-Actor model. Hence, each possible execution of a task corresponds exactly with one of the semantics rule for SDN networks. □

Theorem 4.7. *Let S^{ini} and S^{ini}_a be an SDN state and an SDN-Actor state, respectively.*

1. *For every execution $S^{ini} \to S^1 \to ... \to S^n \in exec(S^{ini}), \exists S^{ini}_a \longmapsto S^1_a \longmapsto ... \longmapsto S^n_a \in exec(S^{ini}_a)$ such that $S^n \equiv S^n_a$.*

2. *For every execution* $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini}), \exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ *such that* $S^n \equiv S_a^n$.

Proof.
Let us prove both cases by induction on the length n of the execution.

- If $n = 0$, it is straightforward to see that $S^0 = S^{ini} \equiv S_a^{ini} = S_a^0$.
- Let us suppose that both cases are true for n and let us prove them for $n + 1$

    1. We need to prove that for every execution $S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^{n+1} \in exec(S^{ini})$, $\exists S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \longmapsto S_a^{n+1} \in exec(S_a^{ini})$ such that $S^{n+1} \equiv S_a^{n+1}$. Applying the induction hypothesis we know that $\exists S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini})$ such that $S^n \equiv S_a^n$. Therefore, now we have $S^n \rightarrow S^{n+1}$, and $S^n \equiv S_a^n$, hence, applying Lemma Appendix A.2.1 we get that $\exists S_a^{n+1}$ such that $S_a^n \longmapsto S_a^{n+1}$ and $S^{n+1} \equiv S_a^{n+1}$.
    2. We need to prove that for every execution $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^{n+1} \in exec(S_a^{ini})$, $\exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \rightarrow S^{n+1} \in exec(S^{ini})$ such that $S^{n+1} \equiv S_a^{n+1}$. Applying the induction hypothesis we know that $\exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ such that $S^n \equiv S_a^n$. Therefore, now we have $S_a^n \longmapsto S_a^{n+1}$, and $S^n \equiv S_a^n$, hence, applying Lemma Appendix A.2.2 we get $\exists S^{n+1}$ such that $S^n \rightarrow S^{n+1}$ and $S^{n+1} \equiv S_a^{n+1}$.

□

Conditional Dynamic Partial Order Reduction and Optimality Results

Miguel Isabel
Complutense University of Madrid
Spain
miguelis@ucm.es

ABSTRACT

Testing concurrent systems requires exploring all possible non-deterministic interleavings that the concurrent execution may have, as any of the interleavings may reveal an erroneous behaviour of the system. This introduces a combinatorial explosion on the number of states that must be considered, which leads often to a computationally intractable problem. In the present PhD thesis, this challenge will be addressed through the development of new Partial Order Reduction techniques (POR). The cornerstone of POR theory is the notion of *independence*, that is used to decided whether each pair of concurrent events p and t are in a race and thus both executions $p \cdot t$ and $t \cdot p$ must be explored. A fundamental goal of this thesis is to introduce notions of *conditional independence* –which ensure the commutativity of the considered events p and t under certain conditions that can be evaluated in the explored state– with a DPOR algorithm in order to alleviate the combinatorial explosion problem. The new techniques that we propose in the thesis have been implemented within the SYCO tool. We have carried out accompanying experimental evaluations to prove the effectiveness and applicability of the proposed techniques. Finally, we have successfully verified a range of properties for several case studies of Software-Defined Networks to illustrate the potential of the approach, scaling to larger networks than related techniques.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Testing, Software Verification, Partial-Order Reduction

ACM Reference Format:
Miguel Isabel. 2019. Conditional Dynamic Partial Order Reduction and Optimality Results. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338987

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338987

1 INTRODUCTION

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is present everywhere in today's software applications. It is widely recognized that concurrent programs are difficult to develop, debug, test and analyze. This is even more so in the context of concurrent *imperative* languages that use a global memory (so called heap) to which the different tasks can have access. These accesses introduce additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. Therefore, software validation techniques urge especially in the context of concurrent programming.

Testing is the most widely-used methodology for software validation. However, due to the non-deterministic interleaving of tasks, traditional testing for concurrent programs is not as effective as for sequential programs. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must systematically explore all possible ways in which the tasks can interleave. This is known as *systematic testing* [24] in the context of concurrent programs. Such full systematic exploration of all task interleavings produces the well known state explosion problem and is often computationally intractable (see, e.g., [25] and its references).

Partial-order reduction (POR) [14] is a general theory that helps mitigate this combinatorial explosion by formally identifying *equivalence* classes of redundant explorations. POR is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps. Such equivalence class is called a Mazurkiewicz trace [21], and POR guarantees that it is sufficient to explore one interleaving per equivalence class. For this purpose, a *happens-before* partial order among the events of the execution sequences is defined. This order relation induces a set of equivalent execution sequences, i.e., those sequences with the same happens-before order belong to the same equivalence class. As a result, only one of them needs to be explored.

2 STATE OF THE ART

Early POR algorithms [12, 14, 26] relied on static over-approximations to detect possible *future* conflicts. The Dynamic-POR (DPOR) algorithm, introduced by Flanagan and Godefroid [13] in 2005, was a breakthrough in the area because it does not need to look at the future. It keeps track of the independence races witnessed along its execution and uses them to decide the required exploration dynamically, without the need of static approximation. DPOR is nowadays considered one of the most scalable techniques for concurrent software testing. The key of DPOR algorithms is in the dynamic construction of two types of sets at each scheduling point: the *sleep set* that contains processes whose exploration has been

proven to be redundant (and hence should not be selected), and the *backtrack set* that contains the processes that have not been proven independent with previously explored steps (and hence may need to be explored). Source-DPOR (SDPOR) [1, 2] improves the precision computing backtrack sets (named there *source* sets).

An extension of the DPOR algorithm, named Optimal DPOR (ODPOR) [2], guarantees the optimality of the exploration, i.e., the proposed DPOR algorithm never explores redundant states (they are equivalent to others already explored). In addition to using source sets, a major extension is needed to achieve optimality: the use of *wakeup trees* to guide the initial steps in the exploration. By means of these extensions, ODPOR guarantees that redundant explorations are never even initiated, proving optimality for *any* number of processes w.r.t. an *unconditional independence* relation.

Both the original DPOR and its extension ODPOR are based on an *unconditional* dependency relation (also called unconditional happens-before relation) which determines the partial order of transitions, i.e., for two transitions be considered independent they must commute in all possible contexts. Let us consider the following three simple processes $\{p, q, r\}$ and the initial state $x = 0$:

p: `write(x=5)`, q: `write(x=5)`, r: `assert(x>0)`.

The transitions `write(x=5)` and `write(x=5)` are unconditionally independent but `write(x=5)` and `assert(x>0)` are not. The latter are independent only if $x > 0$.

Conditional independence was earlier introduced in the context of POR [19], where it was proven that only *uniform* conditional independence can be used, i.e., independence must hold along the whole trace. A notion of uniformity is needed because, unlike unconditional independence, pruning the DPOR search space by relying on whether conditions like $x > 0$ above, called *independence constraints (ICs)*, are satisfiable in the explored state is unsound in general. The ICs provide the conditions under which each pair of transitions commutes, i.e., both execution orders leads to the same state. The first algorithm that took advantage of independence constraints is [27], but this algorithm can only ensure optimality between *two* threads, (while DPOR ensures it among any number of threads) and infers ICs for *single* instructions. The first algorithm that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR (DPOR_{cs}) [4]. However, DPOR_{cs} does not use ICs (it rather checks state equivalence dynamically during the exploration) and exploits conditional (context-sensitive) independence only *partially* to extend the sleep sets. ODPOR with Observers (ODPOR^{ob}) [3] introduces the notion of *observability*, where dependencies between execution steps p and t are conditional to the existence of future steps, called observers, which read the values modified by p and t. For the previous example, when ODPOR^{ob} explores the execution $r.p.q$, we have that p and q are considered as independent, since there is not a observer executed after them. This is because r is executed before, so it does not read the value written by p and q.

3 GOALS OF THE RESEARCH

The overall goal of this PhD thesis is to be able to explote notions of *conditional independence* –which ensure the commutativity of the considered events p and t under certain conditions that can be evaluated in the explored state– within DPOR algorithms in order to alleviate the combinatorial explosion problem described in Section 1. This goal is materialized in the following research challenges:

i) combine and exploit the notions of Context-Sensitive DPOR, Optimal DPOR with Observers and study their synergies to gain further pruning,
ii) propose (sufficient) conditions that ensure uniformity and enable new forms of pruning using ICs,
iii) integrate the notion of uniform conditional independence (which requires to look ahead) to prune the search space in a *dynamic* algorithm using ICs,
iv) statically synthesize ICs for *atomic blocks* of instructions in a pre-analysis using a SMT approach,
v) extend the DPOR framework, that ensures optimality for *any* number of processes, to exploit ICs during the exploration in order to both reduce the backtrack sets and expand the sleep sets as much as possible,
vi) carry out a thorough experimental evaluation to compare the different extensions, and
vii) apply the techniques to a realistic setting.

4 CURRENT STATE OF THE RESEARCH

At the Conference ISSTA 2019 [6], we have presented Optimal Context-Sensitive DPOR with Observers which addresses the first challenge. We have formulated Context-Sensitive DPOR over Optimal DPOR, which is named Optimal Context-Sensitive DPOR (ODPOR_{cs}), and it includes the extension of wakeup trees used to ensure optimality. Furthermore, we have also integrated the notion of observability into ODPOR_{cs} and the resulting algorithm is called Optimal Context-Sensitive DPOR with Observers (ODPOR^{ob}_{cs}). To illustrate this, let us consider again the previous example. ODPOR^{ob}_{cs} detects $p.q.r$ and $q.p.r$ as redundant, because in both sequences r observes the same result (the assert holds). Consequently, ODPOR^{ob}_{cs} only needs to explore one of them, whereas ODPOR^{ob} explores both executions. Finally, we have implemented this new algorithm and perform an experimental evaluation that shows it can explore exponentially less sequences than DPOR_{cs} and ODPOR^{ob}_{cs}.

At the Conference CAV 2018 [5], we have presented *Constrained Dynamic Partial Order Reduction* (CDPOR) which addresses the second, third and fourth challenges. As a result, we have introduced sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework. Moreover, we have extended the state-of-the-art DPOR algorithm with new forms of pruning (by means of expanding sleep sets and reducing backtrack sets). We have also presented an SMT-based approach to automatically synthesize ICs for *atomic blocks*, whose applicability goes beyond the DPOR context. For `write(x)` and `write(x)`, it infers *true* as IC (that is, they are unconditional independent) and, for `write(x)` and `assert(x>0)` it infers $x > 0$. During the exploration, CDPOR detects p and r as dependent in the execution $p.r.q$ (because the condition does not hold in the initial state) and as independent in the execution $q.p.r$ (since after q, the IC holds), avoiding the exploration of $q.r.p$. Moreover, we have experimentally shown the exponential gains achieved by CDPOR.

However, CDPOR extends Source-DPOR only with sound ways of exploiting ICs and, although it has experimentally shown to

achieve exponential reductions, they have not been proven optimal w.r.t. the equivalence classes induced by a conditional happens-before relation. Currently, we are working on a new direction to address the fifth challenge. We aim at characterizing the properties of a *conditional happens-before* relation which allows defining the equivalence classes to be explored by an optimal DPOR algorithm as a disjunction of partial orders. Furthermore, we aim at defining a provably Optimal Constrained DPOR algorithm that varies from unconditional ODPOR in the construction of the sequences to be explored when races are detected. We plan to use a conditional happens-before relation that can be checked efficiently during the execution of the DPOR algorithm and is sufficiently accurate to detect redundancies that can only be captured using a conditional relation. Finally, we will perform an experimental evaluation against ODPOR and CDPOR to show the gains of this new approach.

5 EXPERIMENTS

The thesis will be backed up by a thorough experimental evaluation that addresses goal vi). Namely, in [5], we have reported on experimental results that compare the performance of three DPOR algorithms: SDPOR [1, 2], DPOR$_{cs}$ [4] and our proposal CDPOR [5]. We have implemented and experimentally evaluated CDPOR within the SYCO tool [4], a systematic testing tool for message-passing concurrent programs. SYCO can be used online through its web interface available at http://costa.fdi.ucm.es/syco. To generate the ICs, SYCO calls a new feature of the VeryMax program analyzer [10] which uses Barcelogic [9] as SMT solver.

As benchmarks, we have borrowed the examples from [4] (available online from the previous url) that were used to compare SDPOR with DPOR$_{cs}$ (see Table 1). They are classical concurrent applications: a concurrent sorting algorithm (*QS*), concurrent Fibonacci (*Fib*) and several distributed workers (*Pi, PS*). These benchmarks feature the typical concurrent programming methodology in which computations are split into smaller atomic subcomputations which concurrently interleave their executions, and which work on the same shared data. Therefore, the concurrent processes are highly interfering, and both inferring ICs and applying DPOR algorithms on them becomes challenging. We have executed each benchmark with size increasing input parameters. As it can be observed in the table, he results show that the gains of CDPOR increase exponentially in all examples respect to the size of the input. When compared with DPOR$_{cs}$, we achieve reductions up to 4 orders of magnitude for the largest inputs on which DPOR$_{cs}$ terminates. W.r.t. SDPOR, we achieve reductions of 4 orders of magnitude even for smaller inputs for which SDPOR terminates. In *QS* and *PS*, though the gains are linear for the small inputs, when the size of the problem increases both SDPOR and DPOR$_{cs}$ time out, while CDPOR can still handle them efficiently. As regards the time to infer ICs, we observe that in most cases it is negligible compared to the exploration time of the other methods. Let us also notice that the inference is a pre-process which does not add complexity to the actual DPOR algorithm.

In [6], we have reported on an experimental comparison of the performance of DPOR$_{cs}$, ODPOR^{ob} and our proposal ODPOR$^{ob}_{cs}$. We have used two sets of benchmarks: the same set described above, and a subset of the synthetic examples used in [3] to compare Optimal DPOR and Optimal DPOR with Observers. In general, we obtain speedups with respect to both methods, although when the reduction in explored sequences is small, the overhead of the complex context-sensitive checks does not pay off. Furthermore, our proposal obtains gains, scaling by several orders of magnitude.

In summary, we conclude that our experimental results in [5] and [6] show exponential reductions of explored executions and our gains increase exponentially.

A potential hazard of using conditional independence within DPOR algorithms is that it needs to check independence with respect to states explored in the current execution sequence but not in the current state. It does not need to revisit states that have been completely explored and backtracked, but only those in the current execution sequence. There are several strategies to confront this challenge: *on-demand recomputing*, all states are recomputed following the same events order that led to them (and then no memory usage is needed); *full storage*, all states are stored to be used until the state is backtracked; and *state caching* [23], where states are stored until the memory is approaching full utilization. Our current implementation follows the second strategy. To the light of our experimental results, full storage performs efficiently, since the number of stored states is limited by the number of events in each execution sequence and it remains quite low for the experiments.

6 FOR THE VERIFICATION OF SOFTWARE-DEFINED NETWORKS

To address the vii) challenge, we want to apply our techniques for the verification of Software-Defined Network (SDN). SDN is a relatively recent networking paradigm which is now widely used in industry, with many companies—such as Google and Facebook—using SDN to control their backbone networks and data-centers. The core principle in SDN is the separation of control and data planes—there is a centralized *controller* which operates a collection of distributed interconnected switches. *Hosts* communicate with each other by sending packets to their *switches*. Each switch checks if its own *flow table* contains the destination of the packet. If it does, the packet is sent to the next switch or to the final host. Otherwise, it sends a *message* to the controller in order to receive information about the destination of the packet. The controller answers by means of messages and dynamically updates switches' policies depending on the observed flow of packets, which is a simple but powerful way to react to unexpected events in the network.

Network verification has become increasingly popular since SDN was introduced, because in this new paradigm the amount of detailed information available about network events is rich enough and can be centrally gathered to check for properties of the network behavior. Moreover, the controller itself is a program which can be analyzed and verified before deployment.

We have encoded all basic components of an SDN network (switches, hosts, controller) into the message-passing concurrent language ABS [17]. Furthermore, we have formalized the semantics of SDN networks that allows us to prove soundness of the equivalence between such semantics and the semantics of our encoding. Furthermore, we have overcome one of the most challenging aspects to encode SDN networks, which is the *barrier* messages, special instructions used by the controller to force switches to execute all their messages previously received. We have built a model checker for our SDN models on top of SYCO (choosing the appropriate configuration). This tool uses the DPOR algorithms proposed in this

Table 1: Experimental evaluation comparing SDPOR, DPOR$_{cs}$ and CDPOR

	SDPOR			DPOR$_{cs}$			CDPOR				Speed-up		
Bench.	**Tr**	**S**	**T**	**Tr**	**S**	**T**	**Tr**	**S**	**T**	**T^{smt}**	**G^{s}**	**G^{cs}**	**G^{smt}**
Fib(7)	>13k	>160k	60.0	1	551	0.3	1	82	0.05	0.12	>1364	6	1.4
Fib(8)	>8k	>101k	60.0	1	2k	0.7	1	134	0.12		>527	6	3.0
Fib(9)	>4k	>51k	60.0	1	3k	2.8	1	218	0.25		>242	12	7.5
Fib(10)	>2k	>27k	60.0	1	8k	11.5	1	354	0.69		>88	17	14.3
Fib(14)	>10	>3k	60.0	>1	>4k	60.0	1	3k	42.67		>2	>2	>1.5
QS(13)	5k	91k	29.5	1	29k	7.9	1	50	0.03	11.99	1474	395	0.7
QS(15)	>7k	>157k	60.0	1	115k	42.6	1	58	0.05		>1500	1064	3.6
QS(20)	>4k	>98k	60.0	>1	>148k	60.0	1	78	0.04		>1539	>1539	>5.0
QS(25)	>3k	>96k	60.0	>1	>133k	60.0	1	98	0.06		>1017	>1017	>5.0
QS(200)	>5	>2k	60.0	>1	>87k	60.0	1	798	4.45		>14	>14	>3.7
Pi(8)	>10k	>105k	60.0	264	5k	1.7	1	26	0.02	0.05	>4616	128	26.9
Pi(9)	>11k	>120k	60.0	2k	19k	7.0	1	29	0.02		>4000	465	108.9
Pi(10)	>10k	>128k	60.0	6k	91k	45.2	1	32	0.02		>3530	2655	683.7
Pi(12)	>9k	>122k	60.0	>7k	>128k	60.0	1	38	0.03		>2400	>2400	>810.9
Pi(20)	>5k	>101k	60.0	>5k	>115k	60.0	1	62	0.09		>723	>723	>454.6
PS(5)	35k	156k	43.2	8	142	0.1	1	22	0.01	0.59	5391	5	0.1
PS(6)	>32k	>141k	60.0	72	2k	0.4	1	29	0.02		>4286	28	0.7
PS(7)	>29k	>130k	60.0	2k	28k	7.5	1	37	0.03		>2858	357	12.3
PS(9)	>25k	>109k	60.0	>11k	>165k	60.0	1	56	0.06		>1053	>1053	>92.9
PS(11)	>23k	>103k	60.0	>9k	>132k	60.0	1	79	0.09		>690	>690	>88.8

thesis to avoid exploring redundant executions and incorporates visualization tools to view the exploration and execution diagram.

To evaluate this approach, we have modelled and analysed several SDN scenarios: a controller with load balancer, a SSH controller, a learning switch with authentication, and a firewall with migration. We have found bugs related to programming errors in the controller [7], forwarding loops [15] and violation of safety policies [7, 20]. SYCO needs to explore all possible reorderings of dependent messages and packets, leading to a combinatorial explosion. Thanks to the use of conditional independence within DPOR algorithms, SYCO can handle networks larger than in related systems [20], but without requiring simplifications to the SDN models. This is achieved by means of ICs that are satisfied if two messages or packets are independent (for instance, they do not access to the same entries of the flow table) and they can be proven automatically by using SMT solvers, as in [5]. In our current experiments, we have declared by-hand ICs which are valid for any SDN model. It is part of our future work to infer them automatically.

7 RELATED AND FUTURE WORK

The work in [18, 27] generated for the first time ICs for processes with a single instruction following some predefined patterns. This is a problem strictly simpler than our inference of ICs both in the type of IC generated (restricted to the patterns) and on the single-instruction blocks they consider. Furthermore, our approach using an AllSAT SMT solver is different from the CEGAR approach in [8]. The ICs are used in [18, 27] for SMT-based bounded model checking, an approach to model checking fundamentally different from our stateless model checking setting. Consequently ICs are used in a different way, in our case with no bounds on number of processes, nor derivation lengths, but requiring a uniformity condition on independence in order to ensure soundness.

It remains as future work the combination of Constrained DPOR and Context-Sensitive ODPOR with Observers. To the best of our knowledge, it has not been studied yet. We believe the integration of both techniques can achieve exponential gains compared with these approaches. Data-centric DPOR (DCDPOR) [11] presents a new DPOR algorithm based on a different notion of dependency according to which the equivalence classes of derivations are based on the pairs read-write of variables. Let us consider again the running example with the three processes $\{p, q, r\}$ and the initial state $x = 0$. In DCDPOR, we have only three different observation functions: (r, x) (reading the initial value), (r, p) (reading the value that p writes), (r, q) (reading the value that q writes). Therefore, this notion of relational independence is finer grained than the traditional one in DPOR. However, DCDPOR does not consider conditional dependency, i.e., it does not realize that (r, p) and (r, q) are equivalent, and hence only two explorations are required. In conclusion, our approaches and DCDPOR can complement each other and it is in our agenda to study this integration.

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the MINECO/FEDER, UE project TIN2015-69175-C4-3-R, the Spanish MINECO project TIN2015-69175-C4-2-R, the Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, and by the CM project P2018/TCS-4314.

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In *POPL*, pages 373–384, 2014.

[3] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 229–248, 2018.

[4] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In *CAV*, pages 526–543, 2017.

[5] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained Dynamic Partial Order Reduction. In CAV, volume 10982 of - *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.

[6] Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal Context-sensitive Dynamic Partial Order reduction with Observers. In *ISSTA*, 2019.
[7] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *PLDI*, pages 282–293, 2014.
[8] Kshitij Bansal, Eric Koskinen, and Omer Tripp. Commutativity condition refinement, 2015.
[9] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *CAV*, pages 294–298, 2008.
[10] Cristina Borralleras, Daniel Larraz, Albert Oliveras, José Miguel Rivero, Enric Rodríguez-Carbonell, and Albert Rubio. VeryMax: Tool description for termCOMP 2016. In *WST*, 2016.
[11] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Kapil Vaidya, and Nishant Sinha. Data-centric dynamic partial order reduction. In *POPL 2018*, 2018.
[12] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.
[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
[14] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
[15] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. Sdnracer: concurrency analysis for software-defined networks. In *POPL*, pages 402–415, 2016.
[16] Shiyou Huang and Jeff Huang. Speeding up maximal causality reduction with static dependency analysis. In *ECOOP*, pages 16:1–16:22, 2017.
[17] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10*, LNCS 6957, pp. 142-164. Springer, 2012.
[18] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, pages 398–413, 2009.
[19] Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *TCS*, 101(2):337–359, 1992.
[20] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, pages 163–170, 2014.
[21] Antoni W. Mazurkiewicz. Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986. pages 279–324, 1986.
[22] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. *CoRR*, abs/1802.03950, 2018.
[23] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *CONCUR*, pages 456–469, 2015.
[24] Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, pages 339–356, 2006.
[25] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In FMOODS/FORTE, volume 7273 of Lecture Notes in Computer Science, pages 219-234. Springer, 2012.
[26] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets*, pages 491–515, 1990.
[27] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.

Combining Static Analysis and Testing for Deadlock Detection

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel( )

Complutense University of Madrid (UCM), Madrid, Spain
miguelis@ucm.es

Abstract. Static deadlock analyzers might be able to verify the absence of deadlock. However, they are usually not able to detect its presence. Also, when they detect a potential deadlock cycle, they provide little (or even no) information on their output. Due to the complex flow of concurrent programs, the user might not be able to find the source of the anomalous behaviour from the abstract information computed by static analysis. This paper proposes the combined use of static analysis and testing for effective deadlock detection in asynchronous programs. When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. While if the program does not have deadlock, but the analyzer inaccurately spotted it, we might prove deadlock freedom.

1 Introduction

In concurrent programs, *deadlocks* are one of the most common programming errors and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, with no shared memory among them, and which has two operations for blocking and non-blocking synchronization with the termination of asynchronous tasks. In this setting, in order to detect deadlocks, all possible *interleavings* among tasks executing at the distributed locations must be considered. Basically, each time that the processor can be released, any of the available tasks can start its execution, and all combinations among the tasks must be tried, as any of them might lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. This aspect of static analysis is especially important in security assurance – security attacks try to exercise an application in unpredictable and

This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 409–424, 2016.
DOI: 10.1007/978-3-319-33693-0_26

untested ways. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a deadlock is found, state-of-the-art analysis tools [6,7,12] provide little (and often no) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem.

In contrast, testing consists of executing the application for concrete input values. Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings. The primary advantage of *systematic testing* [4,14] for deadlock detection is that it can provide the detailed deadlock trace. There are two shortcomings though: (1) Although recent research tries to avoid redundant exploration as much as possible [1,3–5], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming. (2) There is only guarantee of deadlock freedom for finite-state terminating programs (terminating executions with concrete inputs).

This paper proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [6] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide a testing tool in order to find associated deadlock traces (or discard them). In summary, the main contributions of this paper are:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made and the status of tasks.
2. We provide a formal characterization of *deadlock state* which can be checked along the execution and allows us to early detect deadlocks.
3. We present a new methodology to detect deadlocks which combines testing and static analysis as follows: the deadlock cycles inferred by static analysis are used to guide the testing process towards paths that might lead to a deadlock cycle while discarding deadlock-free paths.
4. We have implemented our methodology in the SYCO system (see Sect. 6) and performed a thorough experimental evaluation on some classical examples.

2 Asynchronous Programs: Syntax and Semantics

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffers of any processor, including its own, and synchronize with the termination of tasks. The language uses *future variables* to check if the execution of an asynchronous

$$(\textsc{mstep})\ \frac{selectLoc(S) = loc(\ell, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, selectTask(\ell) = tsk(tk, m, l, s), \quad S \diamond \boldsymbol{\rho_\emptyset} \overset{\ell \cdot tk}{\leadsto}{}^{*}\ S' \diamond \boldsymbol{\rho}}{S \xrightarrow{\ell \cdot tk} S'}$$

$$(\textsc{newloc})\ \frac{tk = tsk(tk, m, l, pp{:}x = \mathsf{new}\ D; s), \mathrm{fresh}(\ell'), h' = newheap(D), l' = l[x \to \ell']}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l', s)\}) \cdot loc(\ell', \bot, h', \{\}) \diamond \boldsymbol{\rho_0}}$$

$$(\textsc{async})\ \frac{tk = tsk(tk, m, l, pp{:}y{=}x!m_1(\bar{z}); s), l(x){=}\ell_1,\ \mathrm{fresh}(tk_1),\ l_1{=}buildLocals(\bar{z}, m_1, l)}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot loc(\ell_1, \_, \_, \mathcal{Q}') \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot \\ loc(\ell_1, \_, \_, \mathcal{Q}' \cup \{tsk(tk_1, m_1, l_1, body(m_1))\}) \cdot fut(y, o_1, tk_1, ini(m_1)) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$(\textsc{return})\ \frac{tk = tsk(tk, m, l, pp{:}\mathsf{return}; s), \boldsymbol{\rho_1} = \mathsf{return}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, \bot, h, \mathcal{Q} \cup \{tsk(tk, m, l, \epsilon)\}) \diamond \boldsymbol{\rho_1}}$$

$$(\textsc{await1})\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$(\textsc{await2})\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{await}}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, \bot, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1} \end{array}}$$

$$(\textsc{block1})\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$(\textsc{block2})\ \frac{tk{=}tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{block}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1}}$$

Fig. 1. Macro-step semantics of asynchronous programs

task has finished. An asynchronous call $\mathsf{m}(\bar{\mathsf{z}})$ spawned at location x is associated with a future variable f as follows $\mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}})$. Instructions $f.\texttt{block}$ and $f.\texttt{await}$ allow, respectively, blocking and non-blocking synchronization with the termination of m. When a task completes, or when it is awaiting with a non-blocking await for a task that has not finished yet, its processor becomes idle again, chooses the next pending task, and so on. The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to a *concurrent object* and can be dynamically created using the instruction **new**. The program consists of a set of methods of the form $M ::= T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form $s ::= s; s \mid x{=}e \mid \textbf{if}\ e\ \texttt{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \mid \textbf{return} \mid b = \textbf{new} \mid f = x\ !\ m(\bar{z}) \mid f.\texttt{await} \mid f.\texttt{block}$. For the sake of generality, the syntax of expressions e and types T is left open.

Figure 1 presents the semantics of the language. The information about $\boldsymbol{\rho}$ in bold font is part of the extensions for testing in Sect. 4 and should be ignored for now. A *state* or *configuration* is a set of locations and future variables $loc_0 \cdots loc_n \cdot fut_0 \cdots fut_m$. A *location* is a term $loc(\ell, tk, h, \mathcal{Q})$ where ℓ is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *future variable* is a term $fut(id, \ell, tk, m)$ where id is a unique

future variable identifier, ℓ is the location identifier that executes the task tk awaiting for the future, and m is the initial program point of tk. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated. We assume that the execution starts from a `main` method without parameters. The initial state is St=$\{loc(0, 0, \bot, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0. Here, l maps local variables to their initial values (**null** in case of reference variables) and $\bot$ is the empty heap. $body(m)$ is the sequence of instructions in method m, and we can know the program point pp where an instruction s is in the program as follows pp:s.

As locations do not share their states, the semantics can be presented as a macro-step semantics [14] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to an await or return instruction. In this case, we apply rule MSTEP to select an available task from a location, namely we apply the function $selectLoc(S)$ to select non-deterministically one *active* location in the state (i.e., a location with a non-empty queue) and $selectTask(\ell)$ to select non-deterministically one task of ℓ's queue. The transition $\rightsquigarrow$ defines the evaluation within a given location. NEWLOC creates a new location without tasks, with a fresh identifier and heap. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tk_1, and it adds a new future to the state. $ini(m)$ refers to the first program point of method m. We assume $\ell \neq \ell_1$, but the case $\ell = \ell_1$ is analogous, the new task tk_1 is added to $\mathcal{Q}$ of ℓ. The rules for sequential execution are standard and are thus omitted. AWAIT1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task t_1 is only looked up but it does not disappear from the state as its status may be needed later on. AWAIT2: Otherwise, the task yields the lock so that any other task of the same location can take it. RETURN: When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction ϵ). BLOCK2: A y.`block` instruction waits for the future variable but without yielding the lock. Then, when the future is ready, BLOCK1 allows continuing the execution.

In what follows, a *derivation* or *execution* $E \equiv St_0 \longrightarrow \cdots \longrightarrow St_n$ is a sequence of macro-steps (applications of rule MSTEP). The derivation is *complete* if St_0 is the initial state and $\nexists\ St_{n+1} \neq St_n$ such that $St_n \longrightarrow St_{n+1}$. Since the execution is non-deterministic, multiple derivations are possible from a state. Given a state St, $exec(St)$ denotes the set of all possible derivations starting at St. We sometimes label transitions with $\ell \cdot tk$, the name of the location ℓ and task tk selected (in rule MSTEP) or evaluated in the step (in the transition $\rightsquigarrow$). The systematic exploration of $exec(St)$ thus corresponds to the standard systematic testing setting with no reduction of any kind.

```
1  main() {
2   Ba barber = new Ba();
3   Cl client = new Cl();
4   Ch chair = new Ch();
5   client!wakeup(barber,chair);
6   barber!sleeps(client,chair);
7  }
8  class Ba{
9    Unit sleeps(Cl cl, Ch ch){
10     Fut f=ch!taken(cl);
11     f.block;}
12   Unit cuts(){}
13 }
14 class Ch{
15   Unit taken(Cl cl){
16     Fut f=cl!sits();
17     f.await;}
18   Unit isClean(){}
19 }
20 class Cl{
21   Unit wakeup(Ba b, Ch ch){
22     Fut f=b!cuts();
23     ch!isClean();
24     f.block;}
25   Unit sits(){}
26 }
```

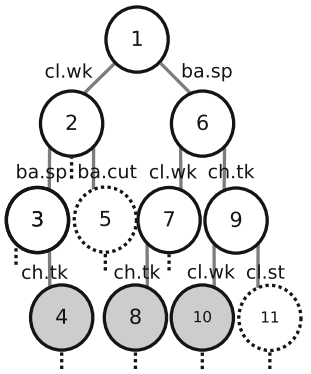


Fig. 2. Classical sleeping barber problem (left) and execution tree (right)

3 Motivating Example

Our running example is a simple version of the classical sleeping barber problem where a barber sleeps until a client arrives and takes a chair, and the client wakes up the barber to get a haircut. Our implementation in Fig. 2 has a `main` method shown on the left and three classes `Ba`, `Ch` and `Cl` implementing the barber, chair and client, respectively. The `main` creates three locations `barber`, `client` and `chair` and spawns two asynchronous tasks to start the `wakeup` task in the `client` and `sleeps` in the `barber`, both tasks can run in parallel. The execution of `sleeps` spawns an asynchronous task on the `chair` to represent the fact that the client takes the chair, and then blocks at line 11 (L11 for short) until the chair is taken. The task `taken` first adds the task `sits` on the client, and then `awaits` on its termination at L17 without blocking, so that another task on the location `chair` can execute. On the other hand, the execution of `wakeup` in the client spawns an asynchronous task `cuts` on the barber and one on the chair, `isClean`, to check if the chair is clean. The execution of the client blocks until `cuts` has finished. We assume that all methods have an implicit return at the end.

Figure 2 summarizes the systematic testing tree of the `main` method by showing some of the macro-steps taken. Derivations that contain a dotted node are not deadlock, while those with a gray node are deadlock. A main motivation of our work is to detect as early as possible that the dotted derivations will not lead us to deadlock and prune them. Let us see two selected derivations in detail. In the derivation ending at node 5, the first macro-step executes `cl.wakeup` and then `ba.cuts`. Now, it is clear that the location `cl` will not deadlock, since the `block` at L24 will succeed and the other two locations will be also able to complete their tasks, namely the `await` at L17 of location `ch` can finish because the client is certainly not blocked, and also the `block` at L11 will succeed because the task in `taken` will eventually finish as its location is not blocked. However, in the branch of node 4, we first select `wakeup` (and block client), then we select

`sleeps` (and block barber), and then select `taken` that will remain in the await at L17 and will never succeed since it is awaiting for the termination of a task of a blocked location. Thus, we have a deadlock. Let us outline five states of this derivation:

$$St_1 \equiv loc(ini, ..) \cdot loc(cl, .., \{tsk(1, wk, ..)\}) \cdot loc(ba, .., \{tsk(2, sp, ..)\}) \cdot loc(ch, ..) \xrightarrow{cl,1}$$
$$St_2 \equiv loc(cl, .., \{tsk(1, wk, f_0.block)\}) \cdot loc(ba, .., \{tsk(3, cut, ..), ..\}) \cdot fut(f_0, ba, 3, 12) \cdot .. \xrightarrow{ba,2}$$
$$St_3 \equiv loc(ba, .., \{tsk(2, sp, f_1.block)\}) \cdot loc(ch, .., \{tsk(5, tk, ..), ..\}) \cdot fut(f_1, ch, 5, 15) \cdot .. \xrightarrow{ch,5}$$
$$St_4 \equiv loc(ch, .., \{tsk(5, tk, f_2.await), ..\}) \cdot loc(cl, .., \{tsk(6, st, ..), ..\}) \cdot fut(f_2, cl, 6, 25) \cdot ..$$
$$\xrightarrow{ch,4} St'_4 \equiv loc(ch, ..\{tsk(4, isClean, \epsilon), ..\}) \cdot ..$$

$$\text{(MSTEP2)} \quad \frac{\begin{array}{c} selectLoc(S) = loc(\ell, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, selectTask(\ell) = tsk(tk, m, l, pp : s), \\ \mathbf{check_{\mathfrak{C}}(S, table)}, S \diamond \rho_0 \overset{\ell \cdot tk}{\rightsquigarrow^*} S' \diamond \rho, S \neq S', \boldsymbol{not}(\mathsf{deadlock}(\boldsymbol{S'})) \\ clock(n), table' = table \cup t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle \end{array}}{(S, table) \xrightarrow{\ell \cdot tk} (S', table')}$$

Fig. 3. MSTEP2 rule for combined testing and analysis

The first state is obtained after executing the `main` where we have the initial location *ini*, three locations created at L2, L3 and L4, and two tasks at L5 and L6 added to the queues. Note that each location and task is assigned a unique identifier (we use numbers as identifiers for tasks and short names as identifiers for locations). In the next state, the task `wakeup` has been selected and fully executed (we have shortened the name of the methods, e.g., `wk` for `wakeup`). Observe at St_2 the addition of the future variable created at L22. In St_3 we have executed task `sleeps` in the barber and added a new future term. In St_4 we execute task `taken` in the chair (this state is already deadlock as we will see in Sect. 4.2), however location chair can keep on executing an available task `isClean` generating St'_4. From now on, we use the location and task names instead of numeric identifiers for clarity.

4 Testing for Deadlock Detection

The goal of this section is to present a framework for early detection of deadlocks during systematic testing. This is done by enhancing our standard semantics with information which allows us to easily detect *dependencies* among tasks, i.e., when a task is awaiting for the termination of another one. These dependencies are necessary to detect in a second step *deadlock states*.

4.1 An Enhanced Semantics for Deadlock Detection

In the following we define the *interleavings table* whose role is twofold: (1) It stores all decisions about task interleavings made during the execution. This way, at the end of a concrete execution, the exact ordering of the performed

macro-steps can be observed. (2) It will be used to detect deadlocks as early as possible, and, also to detect states from which a deadlock cannot occur, therefore allowing to prune the execution tree when we are looking for deadlocks. The interleavings table is a mapping with entries of the form $t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{\ell,tk,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location ℓ and task tk that have been selected in the macro-step, and the program point pp of the first instruction that will be executed;
- n is an integer representing the time when the macro-step starts executing;
- ρ is the status of the task after the macro-step and it can take three values as it can be seen in Fig. 1: `block` or `await` when executing these instructions on a future variable that is not ready (we also annotate in ρ the information on the associated future); **return** that allows us to know that the task finished.

We use a function ***clock*(*n*)** to represent a clock that starts at 0, is increased by one in every execution of ***clock***, and returns the current value **n**. The initial entry is $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$, 0 being the identifier for the initial location and task, and 1 the first program point of *main*. The clock also assigns the value 0 as the first element in the tuple and a fresh variable in the second element ρ_0. The next macro-step will be assigned clock value 1, next 2, and so on. As notation, we define the relation $t \in table$ if there exists an entry $t \mapsto \langle n, \rho \rangle \in table$, and the function $status(t, table)$ which returns the status ρ_t such that $t \mapsto \langle n, \rho_t \rangle \in table$. The semantics is extended by changing rule MSTEP as in Fig. 3. The function deadlock will be defined in Theorem 1 to stop derivations as soon as deadlock is detected. Function $\mathsf{check}_{\mathfrak{C}}$ should be ignored for now, it will be defined in Sect. 5.2. Essentially, there are two new aspects: (1) The state is extended with the status ρ, namely all rules include a status ρ attached to the state using the symbol $\diamond$. The status is showed in bold font in Fig. 1 and can get a value in rules block2, await2 and return. The initial value ρ_0 is a fresh variable. (2) The state for the macrostep is extended with the interleavings table *table*, and a new entry $t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle$ is added to *table* in every macrostep if there has been progress in the execution, i.e., $S' \neq S$, n being the current clock time.

Example 1. The interleavings table below (left) is computed for the derivation in Sect. 3. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The right column shows the future variables in the state that store the location and task they are bound to.

St_1	$t_{ini,main,1} \mapsto \langle 0, return \rangle$	$\emptyset$
St_2	$t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block \rangle$	$fut(f_0, ba, cuts, 12)$
St_3	$t_{ba,sleeps,9} \mapsto \langle 2, 11{:}f_1.block \rangle$	$fut(f_1, ch, taken, 15)$
St_4	$t_{ch,taken,15} \mapsto \langle 3, 17{:}f_2.await \rangle$	$fut(f_2, cl, sits, 25)$

4.2 Formal Characterization of Deadlock State

Our semantics can easily be extended to detect deadlock just by redefining function *selectLoc* so that only locations that can proceed are selected. If, at a given state, no location is selected but there is at least a location with a non-empty queue then there is a deadlock. However, deadlocks can be detected earlier. We present the notion of *deadlock state* which characterizes states that contain a *deadlock chain* in which one or more tasks are waiting for each other's termination and none of them can make any progress. Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit. Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using this naive extension. The early detection of deadlocks is crucial to reduce state exploration as our experiments show in Sect. 6.

We first introduce the auxiliary notion of *waiting interval* which captures the period in which a task is waiting for another one to terminate. In particular, it is defined as a tuple $(t_{stop}, t_{async}, t_{resume})$ where t_{stop} is the macro-step at which the location stops executing a task due to some block/await instruction, t_{async} is the macro-step at which the task that is being awaited is selected for execution, and, t_{resume} is the macro-step at which the task will resume its execution. t_{stop}, t_{async} and t_{resume} are time identifiers as defined in Sect. 4.1. t_{resume} will also be written as $next(t_{stop})$. When the task stops at t_{stop} due to a block instruction, we call it *blocking interval*, as the location remains blocked between t_{stop} and $next(t_{stop})$ until the awaited task, selected in t_{async}, has already finished. The execution of a task can have several points at which macro-steps are performed (e.g., if it contains several await or block the processor may be lost several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step: $suc(t_{\ell,tk,pp_0}, table) = \{t_{\ell,tk,pp_i} : t_{\ell,tk,pp_i} \in table, t_{\ell,tk,pp_i} \geq t_{\ell,tk,pp_0}\}$.

Definition 1 (Waiting/Blocking Intervals). *Let $St = (S, table)$ be a state, $I = (t_{stop}, t_{async}, t_{resume})$ is a* waiting interval *of St, written as $I \in St$, iff:*

1. $\exists\ t_{stop} = t_{\ell,tk_0,pp_0} \in table$, $\rho_{stop} = status(t_{stop}) \in \{pp_1 : x.\texttt{await}, pp_1 : x.\texttt{block}\}$,
2. $t_{resume} \equiv t_{\ell,tk_0,pp_1}$, $fut(x, \ell_x, tk_x, pp(M)) \in S$,
3. $t_{async} \equiv t_{\ell_x,tk_x,pp(M)}, \nexists\ t \in suc(t_{async}, table)$ *with* $status(t) = return$.

If $\rho_{stop} = x.\texttt{block}$, then I is blocking.

In condition 3, we can see that if the task starting at t_{async} has finished, then it is not a waiting interval. This is known by checking that this task has not reached return, i.e., $\nexists\ t \in suc(t_{async}, table)$ such that $status(t) = return$. In condition 1, we see that in ρ_{stop} we have the name of the future we are awaiting (whose corresponding information is stored in fut, condition 2). In order to define t_{resume} in condition 2, we search for the same task tk_0 and same location ℓ that executes the task starting at program point pp_1 of the `await`/`block`, since this is the point that the macro-step rule uses to define the macro-step identifier t_{ℓ,tk_0,pp_1} associated to the resumption of the waiting task.

Example 2. Let us consider again the derivation in Sect. 3. We have the following blocking interval $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_2$ with $St_2 \equiv (S_2, table_2)$, since $t_{cl,wakeup,21} \in table_2$, $status(t_{cl,wakeup,21}, table_2) =$ [24:f.`block`], $(f, ba, cuts, 12) \in St_2$ and $t_{ba,cuts,12} \notin table_2$. This blocking interval captures the fact that the task at $t_{cl,wakeup,21}$ is blocked waiting for task *cuts* to terminate. Similarly, we have the following two intervals in St_4: $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ and $(t_{ch,taken,15}, t_{cl,sits,25}, t_{ch,taken,17})$.

The following notion of *deadlock chain* relies on the waiting/blocking intervals of Definition 1 in order to characterize chains of calls in which intuitively each task is waiting for the next one to terminate until the last one which is waiting on the termination of a task executing on the initial location (that is blocked). Given a time identifier t, we use $loc(t)$ to obtain its associated location identifier.

Definition 2 (Deadlock Chain). *Let $St = (S, table)$ be a state. A chain of time identifiers $t_0, ..., t_n$ is a* deadlock chain *in St, written as $dc(t_0, ..., t_n)$ iff $\forall t_i \in \{t_0, ..., t_{n-1}\}$ s.t. $(t_i, t'_{i+1}, next(t_i)) \in St$ one of the following conditions holds:*

1. $t_{i+1} \in suc(t'_{i+1}, table)$, or
2. $loc(t'_{i+1}) = loc(t_{i+1})$ and $(t_{i+1}, \_, next(t_{i+1}))$ is blocking.
and for t_n, we have that $t_{n+1} \equiv t_0$, and condition 2 holds.

Let us explain the two conditions in the above definition: In condition (1), we check that when a task t_i is waiting for another task to terminate, the waiting interval contains the initial time t'_{i+1} in which the task will be selected. However, we look for any waiting interval for this task t_{i+1} (thus we check that t_{i+1} is a successor of time t'_{i+1}). As in Definition 2, this is because such task may have started its execution and then suspended due to a subsequent await/block instruction. Abusing terminology, we use the time identifier to refer to the task executing. In condition (2), we capture deadlock chains which occur when a task t_i is waiting on the termination of another task t'_{i+1} which executes on a location $loc(t'_{i+1})$ which is blocked. The fact that is blocked is captured by checking that there is a blocking interval from a task t_{i+1} executing on this location. Finally, note the circularity of the chain, since we require that $t_{n+1} \equiv t_0$.

Theorem 1 (Deadlock state). *A state St is deadlock, written $\mathsf{deadlock}(S)$, if and only if there is a deadlock chain in St.*

Derivations ending in a deadlock state are considered complete derivations. We prove that our definition of deadlock is equivalent to the standard definition of deadlock in [6] (proof can be found in [16]).

Example 3. Following Example 1, St_4 is a deadlock state since there exists a *deadlock chain* $dc(t_{cl,wakeup,21}, t_{ba,sleeps,9}, t_{ch,taken,15})$. For the second element in the chain $t_{ba,sleeps,9}$, condition 1 holds as $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11}) \in St_4$ and $t_{ch,taken,15} \in suc(t_{ch,taken,15}, table_4)$. For the first element

$t_{cl,wakeup,21}$, condition 2 holds since $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_4$ and $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ is blocking. Condition 2 holds analogously for $t_{ch,taken,15}$.

5 Combining Static Deadlock Analysis and Testing

This section proposes a deadlock detection methodology that combines static analysis and systematic testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [6], which provides a set of abstractions of potential *deadlock cycles*. If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we systematically test the program using a novel technique to guide the exploration towards paths that might lead to deadlock cycles. The goals of this process are: (1) finding concrete deadlock traces associated to the feasible cycles, and, (2) discarding unfeasible deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input or, in our case, for the `main` method under test. As our experiments show in Sect. 6, our technique allows reducing significantly the search space compared to the full systematic exploration.

5.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [6] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. Three kinds of arrows can be distinguished, namely, *task-task* (a task is awaiting for the termination of another one), *task-location* (a task is awaiting for a location to be idle) and *location-task* (the location is blocked due the task). *Location-location* arrows cannot happen. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location ℓ by the program point at which it is created ℓ_{pp}, and each task by the method name executing. They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Both the analysis and the semantics can be made *object-sensitive* by keeping the k ancestor abstract locations (where k is a parameter of the analysis). For the sake of simplicity of the presentation, we assume $k = 0$ in the formalization (our implementation uses $k = 1$).

Example 4. In our working example there are three abstract locations, ℓ_2, ℓ_3 and ℓ_4, corresponding to locations barber, client and chair, created at lines 2, 3 and 4; and six abstract tasks, *sleeps*, *cuts*, *wakeup*, *sits*, *taken* and *isClean*. The following cycle is inferred by the deadlock analysis: $\ell_2 \xrightarrow{11:sleeps} taken \xrightarrow{17:taken} sits \xrightarrow{25:sits} \ell_3 \xrightarrow{24:wakeup} cuts \xrightarrow{12:cuts} \ell_2$. The first arrow captures that the location created at L2 is blocked waiting for the termination of task `taken` because of the synchronization at L11 of task `sleeps`. Observe that cycles contain dependencies

also between tasks, like the second arrow, where we capture that `taken` is waiting for `sits`. Also, a dependency between a task (e.g., `sits`) and a location (e.g., ℓ_3) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can observed, it is complex to figure out from them why these dependencies arise, and in particular the interleavings scheduled to lead to this situation.

5.2 Guiding Testing Towards Deadlock Cycles

Given an abstract deadlock cycle, we now present a novel technique to guide the systematic execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied. Uppercase letters in constraints denote variables to allow representing incomplete information.

Definition 3 (Deadlock-cycle constraints). *Given a state $St = (S, table)$, a deadlock-cycle constraint takes one of the following three forms:*

1. $\exists t_{L,T,PP} \mapsto \langle N, \rho \rangle$*, which means that there exists or will exist an entry of this form in table (*time constraint*)*
2. $\exists fut(F, L, Tk, p)$*, which means that there exists or will exist a future variable of this form in S (*fut constraint*)*
3. $\mathsf{pending}(Tk)$*, which means that task Tk has not finished (*pending constraint*)*

The following function ϕ computes the set of deadlock-cycle constraints associated to a given abstract deadlock cycle.

Definition 4 (Generation of deadlock-cycle constraints). *Given an abstract deadlock cycle $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, and two fresh variables L_i, Tk_i, ϕ is defined as $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_i) =$*

$$\begin{cases} \{\exists t_{L_i,Tk_i,\_} \mapsto \langle \_, \mathsf{sync}(p_i,F_i)\rangle, \exists fut(F_i,L_j,Tk_j,p_j)\} \cup \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_j, Tk_j) & \text{if } e_j = tk_j \\ \{\mathsf{pending}(Tk_i)\} \cup \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_j) & \text{if } e_j = \ell \end{cases}$$

Notation $\mathsf{sync}(p_i, F_i)$ is a shortcut for p_i:F_i.`block` or p_i:F_i.`await`. Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows (since e_j is a task abstraction), whereas the second case handles task-location arrows (e_j is an abstract location). Let us observe the following: (1) The abstract location and task identifiers of the abstract cycle are not used to produce the constraints. This is because constraints refer to concrete identifiers. Even if the cycle contains the same identifier on two different nodes or arrows, the corresponding variables in the constraints

cannot be bound (i.e., we cannot use the same variables) since they could refer to different concrete identifiers. (2) The program points of the cycle (p_i and p_j) are used in time and fut constraints. (3) Location and task identifier variables of fut constraints and subsequent time or pending constraints are bound (i.e., the same variables are used). This is done using the 2nd and 3rd parameters of function ϕ. (4) In the second case, Tk_j is a fresh variable since the location executing Tk_i can be blocked due to a (possibly) different task. Intuitively, deadlock-cycle constraints characterize all possible deadlock chains representing the given cycle.

Example 5. The following deadlock-cycle constraints are computed for the cycle in Example 4: $\{\exists t_{L_1,Tk_1,\_} \mapsto \langle \_, 11{:}F_1.block\rangle, \exists fut(F_1, L_2, Tk_2, 15), \exists t_{L_2,Tk_2,\_} \mapsto \langle \_, 17{:}F_2.await\rangle, \exists fut(F_2, L_3, Tk_3, 25), \mathsf{pending}(Tk_3), \exists t_{L_3,Tk_4,\_} \mapsto \langle \_, 24{:}F_3.block\rangle, \exists fut(F_3, L_4, Tk_5, 12), \mathsf{pending}(Tk_5)\}$. They are shown in the order in which they are computed by ϕ. The first four constraints require *table* to contain a concrete time in which *some* barber sleeps waiting at L11 for a *certain* chair to be taken at L15 and, during another concrete time, this one waits at L17 for a *certain* client to sit at L25. The client is not allowed to sit by the 5th constraint. Furthermore, the last three constraints require a concrete time in which *this* client waits at L24 to get a haircut by *some* barber at L12 and that haircut is never performed. Note that, in order to preserve completeness, we are not binding the first and the second barber. If the example is generalized with several clients and barbers, there could be a deadlock in which a barber waits for a client which waits for another barber and client, so that the last one waits to get a haircut by the first one. This deadlock would not be found if the two barbers are bound in the constraints (i.e., if we use the same variable name). In other words, we have to account for deadlocks which traverse the abstract cycle more than once.

The idea now is to monitor the execution using the inferred deadlock-cycle constraints for the given cycle, with the aim of stopping derivations at states that do not satisfy the constraints. The following boolean function $\mathsf{check}_{\mathfrak{C}}$ checks the satisfiability of the constraints at a given state.

Definition 5. *Given a set of deadlock-cycle constraints* $\mathfrak{C}$*, and a state* $St = (S, table)$*,* check *holds, written* $\mathsf{check}_{\mathfrak{C}}(St)$*, if* $\forall t_{L_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle \in \mathfrak{C}$*,* $fut(F_i, L_j, Tk_j, p_j) \in \mathfrak{C}$*, one of the following conditions holds:*

1. $\mathsf{reachable}(t_{L_i,Tk_i,p_i}, S)$
2. $\exists t_{\ell_i,tk_i,pp} \mapsto \langle n, \mathsf{sync}(p_i, f_i)\rangle \in table \wedge fut(f_i, \ell_j, tk_j, p_j) \in S \wedge$
   $(\mathsf{pending}(Tk_j) \in \mathfrak{C} \Rightarrow \mathsf{getTskSeq}(tk_j, S) \neq \epsilon)$

Function reachable checks whether a given task might arise in subsequent states. We over-approximate it syntactically by computing the transitive call relations from all tasks in the queues of all locations in S. Precision could be improved using more advanced analyses. Function getTskSeq gets from the state the sequence of instructions to be executed by a task (which is ϵ if the task has terminated). Intuitively, check does not hold if there is at least a time constraint so that: (i) its time identifier is not reachable, and, (ii) in the case that the

interleavings table contains entries matching it, for each one, there is an associated future variable in the state and a pending constraint for its associated task which is violated, i.e., the associated task has finished. The first condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table. The second condition (ii) implies that, for each potential cycle in the state, there is no deadlock chain since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given cycle, hence the derivation can be stopped.

Definition 6 (Deadlock-cycle guided-testing (DCGT)). *Consider an abstract deadlock cycle c, and an initial state* St_0. *Let* $\mathfrak{C} = \phi(c, L_{init}, Tk_{init})$ *with* L_{init}, Tk_{init} *fresh variables. We define DCGT, written* $exec_c(St_0)$, *as the set* $\{d : d \in exec(St_0), \mathsf{deadlock}(St_n)\}$, *where* St_n *is the last state in d.*

Example 6. Let us consider the DCGT of our working example with the deadlock-cycle of Example 4, and hence with the constraints $\mathfrak{C}$ of Example 5. The interleavings table at St_5 contains the entries $t_{ini,main,1} \mapsto \langle 0, return \rangle$, $t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block \rangle$ and $t_{ba,cuts,12} \mapsto \langle 2, return \rangle\}$. $\mathsf{check}_{\mathfrak{C}}$ does not hold since $t_{L_1,Tk_1,24}$ is not reachable from St_5 and constraint $\mathsf{pending}(Tk_5)$ is violated (task *cuts* has already finished at this point). The derivation is hence pruned. Similarly, the rightmost derivation is stopped at St_{11}. Also, derivations at St_4, St_8 and St_{10} are stopped by function deadlock of Theorem 1. Since there are no more deadlock cycles, the search for deadlock detection finishes with this DCGT. Our methodology therefore explores 19 states instead of the 181 explored by the full systematic execution.

Theorem 2 (Soundness). *Given a program P, a set of abstract cycles C in P and an initial state* St_0, $\forall d \in exec(St_0)$ *if d is a derivation whose last state is deadlock, then* $\exists c \in C$ *s.t* $d \in exec_c(St_0)$. *(The proof can be found in App. A)*

6 Experimental Evaluation

We have implemented our approach within the SYCO tool, a testing tool for *concurrent objects* which is available at http://costa.ls.fi.upm.es/syco, where most of the benchmarks below can also be found. Concurrent objects communicate via *asynchronous* method calls and use `await` and `block`, resp., as instructions for non-blocking and blocking synchronization. This section summarizes our experimental results which aim at demonstrating the effectiveness and impact of the proposed techniques. The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *SB* is an extension of the sleeping barber, *UL* is a loop that creates asynchronous tasks and locations, *PA* is the pairing problem, *FA* is a distributed factorial, *WM* is the water molecule making problem, *HB* the hungry birds problem; and, (ii) deadlock free versions of some of the above, named *fX* for the *X* problem, for which deadlock analyzers give false positives. We also include here a peer-to-peer system *P2P*.

Table 1 shows, for each benchmark, the results of our deadlock guided testing (DGT) methodology for finding a representative trace for each deadlock compared to those of the standard systematic testing. Partial-order reduction techniques are not applied since they are orthogonal. This way we focus on the reductions obtained due to our technique per-se. For the systematic testing setting we measure: the number of solutions or complete derivations (column *Ans*), the total time taken (column T) and the number of states generated (column S). For the DGT setting, besides the time and number of states (columns T and S), we measure the "number of deadlock executions"/"number of unfeasible cycles"/"number of abstract cycles inferred by the deadlock analysis" (column $D/U/C$), and, since the DCGTs for each cycle are independent and can be performed in parallel, we show the maximum time and maximum number of states measured among the different DCGTs (columns T_{max} and S_{max}). For instance, in the DGT for *HB* the analysis has found five abstract cycles, we only found a deadlock execution for two of them (therefore 3 of them were unfeasible), 44 s being the total time of the process, and 15 s the time of the longest DCGT (including the time of the deadlock analysis) and hence the total time assuming an ideal parallel setting with 5 processors. Columns in the group **Speedup** show the gains of DGT over systematic testing both assuming a sequential setting, hence considering values T and S of DGT (column T_{gain} for time and S_{gain} for number of states), and an ideal parallel setting, therefore considering T_{max} and S_{max} (columns T^{max}_{gain} and S^{max}_{gain}). The gains are computed as X/Y, X being the measure of systematic testing and Y that of DGT. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3 GHz with 8 GB of RAM, running Mac OS X 10.8.5. A timeout of 150 s is used. When the timeout is reached, we write $>X$ to indicate that for the corresponding measure we have got X units in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X. Also, we write X^* when DGT times out.

Our experiments support our claim that testing complements deadlock analysis. In the case of programs with deadlock, we have been able to provide concrete traces for feasible deadlock cycles and to discard unfeasible cycles. For deadlock-free programs, we have been able to discard all potential cycles and therefore prove deadlock freedom. More importantly, the experiments demonstrate that our DGT methodology achieves a notable reduction of the search space over systematic testing in most cases. Except for benchmarks HB and WM which are explained below, the gains of DGT both in time and number of states are enormous (more than three orders of magnitude in many cases). It can be observed that the gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in SB, UL and PA). In general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively (e.g. in HB and WM). Even in these cases, DGT outperforms systematic testing in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, each DCGT

Table 1. Experimental results: deadlock-guided testing vs. systematic testing

	Systematic			DGT (deadlock-per-cycle)					Speedup			
Bm.	Ans	T	S	$D/U/C$	T	T_{max}	S	S_{max}	T_{gain}	S_{gain}	T_{gain}^{max}	S_{gain}^{max}
HB	35k	32k	114k	2/3/5	44k	15k	103k	34k	0.73	0.9	2.15	3.33
FA	11k	11k	41k	2/1/3	2k	759	3k	2k	5.5	13.7	15.1	22.2
UL	>90k	>150k	>489k	1/0/1	133	133	5	5	>1.1k	>2.5k	>2.5k	>98k
SB	>103k	>150k	>584k	1/0/1	59	59	23	23	>2.5k	>25k	>2.5k	>25k
PA	>121k	>150k	>329k	2/0/2	42	4	12	6	>3.6k	>27k	>38k	>55k
WM	>82k	>150k	>380k	1/0/2	>150k	>150k	>258k	>258k	1*	1.47*	1*	1.47*
fFA	5k	7k	25k	0/1/1	5k	5k	11k	11k	1.61	2.35	1.61	2.35
fP2P	25k	66k	118k	0/1/1	34k	34k	52k	52k	1.96	2.28	1.96	2.28
fPA	7k	7k	30k	0/2/2	4k	2k	9k	4k	1.75	3.33	3.73	6.98
fUL	>102k	>150k	>527k	0/1/1	410	410	236	236	>1k	>2k	>1k	>2k

cannot stop until all potential deadlock paths have been considered. As expected, when we consider a parallel setting, the gains are much larger.

All in all, we argue that our experiments show that our methodology complements deadlock analysis, finding deadlock traces for the potential deadlock cycles and discarding unfeasible ones, with a significant reduction.

7 Conclusions and Related Work

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [6,7] and thread-based programs [11,13], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for pointer aliasing) which might lead to a "don't know" answer. Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove deadlock freedom. Using our method, we try to find a deadlock by exploring the paths given by our deadlock detection algorithm that relies on the static information.

Deadlock detection has been also studied in the context of dynamic testing and model checking [4,9,10,15], where sometimes has been combined with static information [2,8]. As regards combined approaches, the approach in [8] first performs a transformation of the program into a trace program that only keeps the instructions that are relevant for deadlock and then dynamic testing is performed on such program. The approach is fundamentally different from ours: in their case, since model checking is performed on the trace program (that overapproximates the deadlock behaviour), the method can detect deadlocks that do not exist in the program, while in our case this is not possible since the testing is performed on the original program and the analysis information is only used to drive the execution. In [2], the information inferred from a type system is used to accelerate the detection of potential cycles. This work shares with our work that information inferred statically is used to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but works on a

general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: Proceedings of POPL 2014, pp. 373–384. ACM (2014)
2. Agarwal, R., Wang, L., Stoller, S.D.: Detecting potential deadlocks with static analysis and run-time monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006)
3. Albert, E., Arenas, P., Gómez-Zamalloa, M.: Actor- and task-selection strategies for pruning redundant state-exploration in testing. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 49–65. Springer, Heidelberg (2014)
4. Christakis, M., Gotovos, A., Sagonas, K.F.: Systematic testing for detecting concurrency errors in erlang programs. In: ICST 2013, pp. 154–163. IEEE (2013)
5. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings POPL 2005, pp. 110–121. ACM (2005)
6. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013)
7. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis of concurrent objects: theory and practice. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 394–411. Springer, Heidelberg (2013)
8. Joshi, P., Naik, M., Sen, K., Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In: Proceedings of FSE 2010, pp. 327–336. ACM (2010)
9. Joshi, P., Park, C., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: Proceedings of PLDI 2009. ACM (2009)
10. Kheradmand, A., Kasikci, B., Candea, G.: Lockout: efficient testing for deadlock bugs. Technical report (2013)
11. Masticola, S.P., Ryder, B.G.: A model of ada programs for static deadlock detection in polynomial time. In: Parallel and Distributed Debugging. ACM (1991)
12. Naik, M., Park, C., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of ICSE, pp. 386–396. IEEE (2009)
13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: a dynamic data race detector for multithreaded programs. ACM TCS **15**(4), 391–411 (1997)
14. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006)
15. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 245–264. Springer, Heidelberg (2000)
16. Albert, E., Gómez-Zamalloa, M., et al.: Combining Static Analysis and Testing for Deadlock Detection. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 409–424. Springer, Heidelberg (2016). http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf

Generation of Initial Contexts for Effective Deadlock Detection

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel( )

Complutense University of Madrid (UCM), Madrid, Spain
{elvira,mzamalloa}@fdi.ucm.es, miguelis@ucm.es

Abstract. It has been recently proposed that testing based on symbolic execution can be used in conjunction with static deadlock analysis to define a deadlock detection framework that: (i) can show deadlock presence, in that case a concrete test-case and trace are obtained, and (ii) can also prove deadlock freedom. Such symbolic execution starts from an *initial distributed context*, i.e., a set of locations and their initial tasks. Considering all possibilities results in a combinatorial explosion on the different distributed contexts that must be considered. This paper proposes a technique to effectively generate initial contexts that can lead to deadlock, using the possible conflicting task interactions identified by static analysis, discarding other distributed contexts that cannot lead to deadlock. The proposed technique has been integrated in the abovementioned deadlock detection framework hence enabling it to analyze systems without the need of any user supplied initial context.

1 Motivation

Deadlocks are one of the most common programming errors and they are therefore one of the main targets of verification and testing tools. We consider a distributed programming model with explicit *locations* (or distributed nodes) and *asynchronous* tasks that may be spawned and awaited among locations. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is non-empty, some task is selected for execution, this selection is non-deterministic. Let us see now our motivating example in Fig. 1 which simulates a simple communication protocol between a database location and a worker location. Our implementation has the main method, and two classes Worker and DB implementing the worker and the database, respectively. The main method creates two distributed locations: the database and the worker, and (asynchronously) invokes methods register and work on each of them, respectively. The work method of a worker simply accesses the database (invoking asynchronously method getData) and then *blocks* until it gets the result, which

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

F. Fioravanti and J. P. Gallagher (Eds.): LOPSTR 2017, LNCS 10855, pp. 3–19, 2018.
https://doi.org/10.1007/978-3-319-94460-9_1

```
1  main(){
2      DB db = new DB();
3      Worker w = new Worker();
4      db!register(w);
5      w!work(db);}
6
7  class Worker{
8    Data data;
9    int work(DB db){
10     Future⟨Data⟩ f;
11     f = db!getData(this);
12     data = f.get;
13     return 0;
14   }
15   int ping(int n){return n;}
16 }// end of class Worker
17
18 class DB{
19   Data data = ...;
20   Worker client = null;
21   int connected = 1;
22   int connect(){
23       connected = 3;
24       return connected;
25   }
26   int register(Worker w){
27       connected = 5;
28       Future⟨Data⟩ g;
29       g = this!getData(w);
30       await g?;
31       if (connected > 0){
32           connected = connected − 1;
33           Future⟨int⟩ f = w!ping(5);
34           if (f.get == 5) client = w;
35       }
36       return 0;
37   }
38   Data getData(Worker w){
39     if (client == w) return data;
40     else return null;
41   }
42 }// end of class DB
```

Fig. 1. Working example. Communication protocol between a DB and a worker

is assigned to its data field. The instruction get blocks the execution in the current location until the awaited task has terminated. We use future variables [7,8] to detect the termination of asynchronous tasks. The register method of the database makes a call to getData and waits for its execution. Once it has finished, it checks if the number of possible connections is bigger than 0. In that case connected is decreased by one, and the database makes sure that the worker is online. This is done by invoking asynchronously method ping with a concrete value and blocking until it gets the result with the same value. Then, the database registers the provided worker reference storing it in its client field. Method getData of the database returns its data field if the caller worker is registered, otherwise it returns **null**. Finally, method connect sets the field connected to 3. Depending on the sequence of interleavings, the execution of this program can finish: (1) as one would expect, i.e., with worker.data = db.data, (2) with w.data = **null** if getData is executed before the assignment at line 34, or, (3) in a deadlock.

We have recently proposed a deadlock detection framework [2,3] that combines static analysis and symbolic execution based testing [1,3,6,14]. The deadlock analysis (for example, [9]) is first used to obtain descriptions of potential deadlock cycles which are then used to guide the testing process. The resulting deadlock detection framework hence can: (i) show deadlock presence, in which case a concrete test-case and trace are obtained, and (ii) prove deadlock freedom (up to the symbolic execution exploration limit). However, the symbolic

execution phase needs to start from a concrete initial distributed context, i.e., a set of locations and their initial tasks. In our example, such an initial context is provided by the main method, which creates a Database and a Worker location, and schedules a work task on the worker with the database as parameter, and, a register task on the database with the worker as parameter. This is however only one out of the possible contexts, and, of course, it could be the case that it does not expose an error that occurs in other contexts (for example, it does not manifest any deadlock). This clearly limits the framework potential.

A fundamental challenge for a symbolic execution framework of distributed programs is to automatically and systematically generate *relevant* distributed contexts for the type of error that it aims at detecting. This would allow for instance applying symbolic execution for system and integration testing. The generation of relevant contexts involves two challenging aspects: (1) A first challenge is related to the elimination of redundant (useless) contexts. Observe that there is a combinatorial explosion on the different possible distributed contexts that can be generated when one considers all possible types and number of distributed locations and tasks within them. Therefore, it is crucial to provide the *minimal* set of initial contexts that contains only one representative of equivalent contexts. (2) For the particular type of error that one aims at detecting, an additional challenge is to be able to only generate initial contexts in which the error can occur. In the case of generating initial contexts for deadlock detection in our working example, this would mean generating for instance, a context with a database location and some worker location with a scheduled work task and a register task on the database for it, i.e., the context created by the main method. For instance, contexts that do not include both tasks would be useless for deadlock detection. Let us observe that if the assignment at Line 23 is changed to assign 0, then the initial contexts must also include a connect task, otherwise no deadlock will be produced. Interestingly, deadlock analyses provide [9,11,12] potential *deadlock cycles* which contain the possibly conflicting task interactions that can lead to deadlock. This information will be used to help our framework anticipate this information and discard initial distributed contexts that cannot lead to deadlock from the beginning. Briefly, the main contributions of this paper are the following:

- We introduce the concept of *minimal* set of initial contexts and extend a static testing framework to automatically and systematically generate them.
- We present a deadlock-guided approach to effectively generate initial contexts for deadlock detection and prove its soundness.
- We have implemented our proposal within the aPET/SYCO system [4] and performed an experimental evaluation to show its efficiency and effectiveness.

2 Asynchronous Programs

A program consists of a set of classes that define the types of locations, each of them defines a set of fields and methods of the form $M ::= T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form s::=s; s | x=e |if e then s else s | while e do s | **return** x; |

b=new $\mathsf{T}(\bar{z})$ | $\mathsf{f} = \mathsf{x}$! $\mathsf{m}(\bar{z})$ | **await** f? | $\mathsf{x} = \mathsf{f}$.**get**. Syntactically, a location will therefore be similar to a *concurrent object* that can be dynamically created using the instruction new $\mathsf{T}(\bar{z})$. The declaration of a future variable is as follows $\mathsf{Future}\langle\mathsf{T}\rangle$ f, where T is the type of the result r, it adds a new future variable to the state. Instruction $\mathsf{f} = \mathsf{x}$! $\mathsf{m}(\bar{z})$ spawns a new task (instance of method m) and it is set to the future f in the state. Instruction **await** f? allows non-blocking synchronization. If the future variable f we are awaiting for points to a finished task, then the **await** can be completed. Otherwise the task yields the lock so that any other task of the same location can take it. On the other hand, instruction f.**get** allows blocking synchronization. It waits for the future variable without yielding the lock, i.e., it blocks the execution of the location until the task that is awaiting is finished. Then, when the future is ready, it retrieves the result and allows continuing the execution. This instruction introduces possible deadlocks in the program, as two tasks can be awaiting for termination of tasks on each other's locations. Finally, instruction **return** x; releases the lock that will never be taken again by that task. Consequently, that task is *finished* and removed from the task queue. All statements of a task takes place serially (without interleaving with any other task) until it gets to a **return** or **await** f? instruction. Then, the processor becomes idle again, chooses non-deterministically the next pending task, and so on.

A *program state* or *configuration* is a set of locations $\{loc_0, ..., loc_n\}$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed. We assume that the execution starts from a *main* method without parameters. The initial state is S=$\{loc(0, 0, \bot, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0, maps local variables to their initial values, and $body(m)$ is the sequence of instructions in method m and $ini(main)$ is the initial program point in method m. From now on, we represent the state as a Prolog list, and we write $[x \mapsto v]$ to denote $h(x) = v$ (resp. $l(x) = v$), that is, field x in the heap h (resp. local variable x in the mapping l) takes the value v.

In what follows, a *derivation* or *execution* [20] is a sequence of states $S_0 \xrightarrow{o_1.t_1}$ ... $\xrightarrow{o_n.t_n} S_n$, where $S_i \xrightarrow{o_i.t_i} S_{i+1}$ denotes the execution of task t_i in location $o_i \in S_i$. The derivation is *complete* if S_0 is the initial state and $\nexists\, loc(o, \_, \_, \{tk\} \cup \mathcal{Q}) \in S_n$ such that $S_n \xrightarrow{o.tk} S_{n+1}$ and $S_n \neq S_{n+1}$. Given a state S, $exec(S)$ denotes the set of all possible complete executions starting at S.

3 Specifying and Generating Initial Contexts

In our asynchronous programs, the most *general* initial contexts consist of sets of locations with *free* variables in their fields, and initial tasks in each location

queue with *free* variables as parameters, i.e., neither the fields nor the parameters have concrete values. A first approach to systematically generate initial contexts could consist in generating, on backtracking, all possible multisets of initial tasks (method names), and for each one, generate all aliasing combinations with the locations of the tasks belonging to the same type of location. They are multisets because there can be multiple occurrences of the same task. To guarantee termination of this process we need to impose some limit in the generation of the multisets. For this, we could simply set a limit on the multiset global size. However it would be more reasonable and useful to set a limit on the maximum cardinality of each element in the multiset. To allow further flexibility, let us also set a limit on the minimum cardinality of each element. For instance, if we have a program with just one location type A with just one method m, and we set 1 and 2 as the minimum and maximum cardinalities respectively, then there are two possible multisets, namely, $\{m\}$ and $\{m, m\}$. The first one leads to one initial context with one location of type A with an instance of task m in its queue. The second one leads to two contexts, one with one location of type A with two instances of task m in its queue, and the other one with two different locations, each with an instance of task m in its queue.

On the other hand, it makes sense to allow specifying which tasks should be considered as initial tasks and which should not. A typical scenario is that the user knows which are the main tasks of the application and does not want to consider auxiliary or internal tasks as initial tasks. Another scenario is in the context of integration testing, where the tester might want to try out together different groups of tasks to observe how they interfere with each other. Also, the use of static analysis can help determine a subset of tasks of interest to detect some specific property. This is the case of our deadlock-guided approach of Sect. 4. With all this, the input to our automatic generation of initial contexts is: a set of tuples ($\mathsf{C.M}$,C^{min}, C^{max}), where $\mathsf{C.M}$ is an *abstract task*, i.e., a task name, being C and M the class and method name resp., and, C^{min} resp. C^{max} is the associated minimum resp. maximum cardinality. Note that this does not limit the approach in any way since one could just include in $\mathcal{T}_{ini}$ all methods in the program and set $C^{min} = 0$ and a sufficiently large C^{max}.

Example 1. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.connect}, 0, 1)\}$. The corresponding multisets are $\{\mathsf{register}\}$ and $\{\mathsf{register}, \mathsf{connect}\}$. All contexts must contain exactly one instance of task register and at most one instance of task connect. This leads to three possible contexts: (1) a DB location instance with a task register in its queue, (2) a DB location instance with tasks register and connect in its queue, and, (3) two different DB location instances, one of them with an instance of task register and the other one with an instance of task connect. For instance, the state corresponding to the latter context would be:

$$
\begin{aligned}
\mathtt{S} = [\mathtt{loc}(\mathtt{DB1}, \mathtt{bot}, \ & [\mathtt{data} \mapsto \mathtt{D1}, \mathtt{clients} \mapsto \mathtt{Cl1}, \mathtt{checkOn} \mapsto \mathtt{B1}], \\
& [\mathtt{tsk}(1, \mathtt{register}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB1}), \mathtt{m} \mapsto \mathtt{W1}], \mathtt{body}(\mathtt{register}))]) \\
\mathtt{loc}(\mathtt{DB2}, \mathtt{bot}, \ & [\mathtt{data} \mapsto \mathtt{D2}, \mathtt{clients} \mapsto \mathtt{Cl2}, \mathtt{checkOn} \mapsto \mathtt{B2}], \\
& [\mathtt{tsk}(2, \mathtt{connect}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB2})], \mathtt{body}(\mathtt{connect}))])],
\end{aligned}
$$

where D1, Cl1, and B1 (resp. D2, Cl2, and B2) are the fields data, clients, and checkOn of location DB1 (resp. DB2), and W1 resp. W2 the parameter of the task register resp. connect, and body(m) is the sequence of instructions in method m. Note that both fields and task parameters are fresh variables so that the context is the most general possible. Note that the first parameter of a task is always the location this and it is therefore fixed. □

In the following, we formally define the contexts that must be produced from a set of abstract tasks $\mathcal{T}_{ini}$ with associated cardinalities. We use the notation $\{[m_1, ..., m_n]_{o_i}\}$ for an initial context where there exists a location $loc(o_i, \bot, h, \{tk(tk_1, m_1, l_1, body(m_1))\} \cup ... \cup \{tk(tk_n, m_n, l_n, body(m_n))\})$. Note that we can have $m_i = m_j$ with $i \neq j$. For instance, the three contexts in Example 1 are written as $\{[\mathsf{register}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db}_1}\}$ and $\{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{connect}]_{\mathsf{db}_2}\}$, respectively. Let us first define the set of initial contexts from a given $\mathcal{T}_{ini}$ when all tasks belong to the same class.

Definition 1 (Superset of initial contexts (same class C_i)). *Let $\mathcal{T}_{ini} = \{(C_i.m_1, C_1^{min}, C_1^{max}), \ldots, (C_i.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities. Let us have $\sum_{i=1}^{n} C_i^{max}$ different identifiers: $o_{1,1}, \ldots, o_{1,C_1^{max}}, \ldots, o_{n,1}, \ldots, o_{n,C_n^{max}}$. We can find at most $\sum_{i=1}^{n} C_i^{max}$ instances of class C_i, that is, each abstract task m_i ($i \in [1, n]$) has at most C_i^{max} instances and each of them can be inside a different instance of class C_i. Let $u_{i,j}^{m_k}$ be an integer variable that denotes the number of instances of task m_k inside the location $o_{i,j}$ and let us consider the following integer system:*

$$\begin{cases} C_1^{min} \leq u_{1,1}^{m_1} + \ldots + u_{1,C_1^{max}}^{m_1} + \ldots + u_{n,1}^{m_1} + \ldots + u_{n,C_n^{max}}^{m_1} \leq C_1^{max} \\ \ldots \\ C_n^{min} \leq u_{1,1}^{m_n} + \ldots + u_{1,C_1^{max}}^{m_n} + \ldots + u_{n,1}^{m_n} + \ldots + u_{n,C_n^{max}}^{m_n} \leq C_n^{max} \end{cases}$$

Each formula requires at least C_k^{min} and at most C_k^{max} instances of task m_k. Each solution to this system corresponds to an initial context.
Let $(d_{1,1}^{m_1}, \ldots, d_{n,C_n^{max}}^{m_1}, \ldots, d_{1,1}^{m_n}, \ldots, d_{n,C_n^{max}}^{m_n})$ be a solution, then the corresponding initial context contains:

- *$loc(o_{i,j}, \bot, h, \mathcal{Q})$, that is, a location $o_{i,j}$ whose lock is free, the fields in h are mapped to fresh variables, and the queue $\mathcal{Q}$ contains: $d_{i,j}^{m_1}$ instances of abstract task m_1,..., and $d_{i,j}^{m_n}$ instances of m_n, if $i \in [1, n]$, $j \in [1, C_i^{max}]$ and $\exists d_{i,j}^{m_k} > 0, k \in [1, n]$, where each instance of m_i is $tsk(tk, m_i, l, body(m_i))$ and every argument in l is mapped to a fresh variable.*

Example 2. Let us consider the example $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 0, 1), (\mathsf{DB.connect}, 1, 1)\}$. The identifiers are $o_{1,1}$ and $o_{2,1}$, and the variables of the system are $u_{1,1}^{reg}$, $u_{2,1}^{reg}$, $u_{1,1}^{get}$ and $u_{2,1}^{get}$. Finally, we obtain the next system:

$$\begin{cases} 0 \leq u_{1,1}^{reg} + u_{2,1}^{reg} \leq 1 \\ 1 \leq u_{1,1}^{get} + u_{2,1}^{get} \leq 1 \end{cases}$$

We obtain 6 solutions: $(0,0,1,0), (0,0,0,1), (1,0,1,0), (1,0,0,1), (0,1,1,0)$ and $(0,1,0,1)$. Then, the superset of initial contexts is

$$\{\{[\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{connect}]_{\mathsf{o}_{2,1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{2,1}}\},$$

$$\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$$

□

Let us observe that the two last contexts are equivalent since they are both composed of two instances of DB with tasks register and connect respectively. Therefore, we only need to consider one of these two contexts for symbolic execution. Considering both would lead to *redundancy*. The notion of minimal set of initial contexts below eliminates redundant contexts, hence avoiding useless executions.

Definition 2 (Equivalence relation $\sim$). *Two contexts C_1 and C_2 are equivalent, written $C_1 \sim C_2$, if $C_1 = C_2 = \emptyset$ or $C_1 = \{loc(o_1, \bot, h_1, \mathcal{Q}_1)\} \cup C_1'$, and $\exists\ o_2 \in C_2$ such that:*

1. $C_2 = \{loc(o_2, \bot, h_2, \mathcal{Q}_2)\} \cup C_2'$,
2. $\mathcal{Q}_1$ *and* $\mathcal{Q}_2$ *contain the same number of instances of each task, and*
3. $C_1' \sim C_2'$.

Example 3. The superset in Example 2 contains 3 equivalence classes induced by the relation $\sim$: (1) the class $\{\{[\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts are composed of a location with a task connect, (2) the class $\{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[register, connect]_{o_{2,1}}\}\}$, whose locations have two tasks register and connect. and, finally, (3) the class $\{\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts have two locations with a task register and a task connect, respectively. □

Definition 3 (Minimal set of initial contexts $\mathcal{I}^{C_i}$ (same class Cl_i)). *Let $\mathcal{T}_{ini}$ be the set of abstract tasks, then the* minimal set of initial contexts $\mathcal{I}^{Cl_i}$ *is composed of a representative of each equivalence class induced by the relation $\sim$ over the superset of initial contexts for the input $\mathcal{T}_{ini}$.*

Example 4. As we have seen in the previous example, there are three different equivalence classes. So, the minimal set of initial contexts is composed of a representative of each class (we have renamed the identifiers for the sake of clarity):

$$\mathcal{I}^{DB} = \{\{[\mathsf{connect}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{connect}]_{\mathsf{db}_2}\}\}$$

□

Let us now define the set of initial contexts $\mathcal{I}$ when the input set $\mathcal{T}_{ini}$ contains tasks of different types of locations.

Definition 4 (Minimal set of initial contexts $\mathcal{I}$ (Different classes)). *Let $\mathcal{T}_{ini} = \{(C_1.m_1, C_1^{min}, C_1^{max}), \ldots, (C_n.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities, and let us consider a partition of this set where every equivalence class is composed of abstract tasks of the same class. Hence, we have: $\mathcal{T}_{ini}^{C_1} = \{C_1.m'_1, \ldots, C_1.m'_{j_1}\}, \ldots, \mathcal{T}_{ini}^{C_n} = \{C_n.m''_1, \ldots, C_n.m''_{j_n}\}$ where $C_i \neq C_j, \forall i, j \in [1, n], i \neq j$.*
Then, let $\mathcal{I}^{C_i}$ be the minimal set of initial contexts for the input $\mathcal{T}_{ini}^{C_i}$, $i \in [1, n]$ and $U : \mathcal{I}^{C_1} \times \ldots \times \mathcal{I}^{C_n} \to \mathcal{I}$, defined by $U(s_1, \ldots, s_n) = s_1 \cup \ldots \cup s_n$. The set $\mathcal{I}$ is defined by the image set of application U.

Example 5. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.connect}, 1, 1), (\mathsf{Worker.work}, 1, 1)\}$ from which we get the initial contexts $\mathcal{I}^{Worker} = \{\{[\mathsf{work}]_{\mathsf{w}_1}\}\}$ and $\mathcal{I}^{DB} = \{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db},1}\}, \{[register]_{db_1}, [connect]_{db_2}\}\}$. Then, by Definition 4,

$$\mathcal{I} = \{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db}_1}, [\mathsf{work}]_{\mathsf{w}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{connect}]_{\mathsf{db}_2}, [\mathsf{work}]_{\mathsf{w}_1}\}\}$$

□

It is straightforward to implement a function that generates the minimal set of initial contexts from a provided set of initial tasks (for instance [5]). Such a function is denoted as *generate_contexts*($\mathcal{T}_{ini}$). The main complication is to avoid the generation of equivalent contexts (Definition 2) as soon as possible during the process. For this aim one can rely on the definition of a normal form according to the number of tasks inside each location.

4 On Automatically Inferring Deadlock-Interfering Tasks

The systematic generation of initial contexts produces a combinatorial explosion and therefore it should be used with small sets of abstract tasks (and low cardinalities). However, in the context of deadlock detection, in order not to miss any deadlock situation, one has to consider in principle all methods in the program, hence producing scalability problems. Interestingly, it can happen that many of the tasks in the generated initial contexts do not affect in any way deadlock executions. Our challenge is to only generate initial contexts from which a deadlock can show up. For this, the deadlock analysis provides the possibly conflicting task interactions that can lead to deadlock. We propose to use this information to help our framework discard initial contexts that cannot lead to deadlock from the beginning. Section 4.1 summarizes the concepts of the deadlock analysis used to obtain the deadlock cycles, and Sect. 4.2 presents the algorithm to generate the set of initial tasks $\mathcal{T}_{ini}$.

4.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [9] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points,

$tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location o by the program point at which it is created o_{pp}, and each task by the method name executing (as in Sect. 3). They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Points-to analysis [9,18] can be used to infer such abstractions with more precision, for instance, by distinguishing the actions performed by different location abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like "abstract location or task e is waiting for the termination of abstract location or task e' due to the synchronization instruction at program point p of abstract task tk". Three kinds of arrows can be distinguished, namely, *task-task* (an abstract task is awaiting for the termination of another one), *task-location* (an abstract task is awaiting for an abstract location to be idle) and *location-task* (the abstract location is blocked due the abstract task). *Location-location* arrows cannot happen.

Example 6. In our working example there are two abstract locations, o_2, corresponding to location database created at line 2 and o_3, corresponding to the n locations worker, created inside the loop at line 3; and four abstract tasks, *register*, *getD*, *work* and *ping*. The following cycle is inferred by the deadlock analysis: $o_2 \xrightarrow{34:register} ping \xrightarrow{15:ping} o_3 \xrightarrow{12:work} getD \xrightarrow{38:getD} o_2$. The first arrow captures that the location created at Line 2 is blocked waiting for the termination of task ping because of the synchronization at L34 of task register. Also, a dependency between a task and a location (for instance, ping and o_3) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can be observed, it is complex to figure out from them why these dependencies arise, and more importantly the interleavings scheduled to lead to this situation. □

4.2 Generation of Initial Tasks

The underlying idea is as follows: we select an abstract cycle detected by the deadlock analysis, and extract a set of potential abstract tasks which can be involved in a deadlock. In a naive approximation, we could take those abstract tasks that are inside the cycle and contain a blocking instruction. We also need to set the maximum cardinality for each task to ensure finiteness (by default 1) and require at least one instance for each task (minimum cardinality).

This approach is valid as long as we only have blocking synchronization primitives, i.e., when the location state stays unchanged until the resumption of a suspended execution. However, this kind of concurrent/distributed languages usually include some sort of non-blocking synchronization primitive. When a location stops its execution due to an **await** instruction, another task can interleave its execution with it, i.e., start to execute and, thus, modify the location state (i.e., the location *fields*). Then, if a call or a blocking instruction involved in a deadlock depends on the value of one of these fields, and we do not consider

all the possible values, a deadlock could be missed. As a consequence, we need to consider at release points, all possible interleavings with tasks that modify the fields in order to capture all deadlocks.

Let us consider now a simple modification of our working example. Line 27 is replaced by connected = 0. Now it is easy to see that if we only consider register and work as input, deadlocks are lost: once register is executed and the instruction at line 30 is reached, the location's queue only contains task getData but no connect and, therefore, when task register is resumed, field connected stays unchanged and the body of the condition is not executed, so we cannot have a deadlock situation.

In the following we define the *deadlock-interfering* tasks for a given abstract deadlock cycle, i.e., an *over-approximation* of the set of tasks that need to be considered in initial contexts so that we cannot miss a representative of the given deadlock cycle. In our extended example, those would be, register and work but also connect.

Definition 5 (initialTasks(C)). *Let C an abstract deadlock cycle. Then,*

$$initialTasks(C) := \bigcup_{i_{call} \in t \in C} initialTasks(t, i_{call}, C) \cup \bigcup_{i_{sync} \in t \in C} initialTasks(t, i_{sync}, C)$$

where:

- $initialTasks(t, i, C) = \emptyset \quad if\ o \xrightarrow{t} t_2 \notin C\ and\ i \neq i_{mod}\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t, i, C) = \{t\}\ if\ (o \xrightarrow{t} t_2 \in C\ or\ i = i_{mod}\ )\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t, i, C)$
  $= \{t\} \cup \bigcup_{f \in fields(i)} \left(\bigcup_{(i_{mod}, t_{mod}) \in mods(f)} initialTasks(t_{mod}, i_{mod}, C) \right)$
  $if\ \exists\ i_{await} \in [t_0, i]$

The definition relies on function fields(I) which, given an instruction I, returns the set of class fields that have been read or written until the execution of instruction I. Let mods(f) be the set of pairs (instruction,task) that modify field f. We can observe that *initialTasks(C)* is the union of the initial tasks for each relevant instruction inside the cycle C, i.e., asynchronous calls and synchronization primitives. We can also observe in the auxiliary function *initialTasks(t,i,C)* that: (1) if the instruction i is not producing a *location-task edge* and it is not an instruction modifying a field, then t does not need to be added as initial task, (2) if i produces a *location-task edge* or is modifying a field, and we do not have any **await** instruction between the beginning of the task and i, then i is going to be executed under the most general context, so we do not need to add more initial tasks but t, and (3) on the other hand, if there exists an **await** instruction between the beginning of task t, namely t_0, and instruction i, each field f inside the set fields(i) could be changed before the resumption of the **await** by any task modifying f. Thus, tasks containing any of the possible f-modifying instructions must be considered and, recursively, their initial tasks.

It is important to highlight that this definition could be non-terminating depending on the program we are working with. For instance, if we apply the

```
Data: An abstract cycle C and a maximum cardinality M
Result: A list with the interfering tasks for C
Q = ∅; L = ∅;
forall the t ∈ C do
    i_call = receiveCall(t,C); enqueue(Q,(i_call,t));
    i_await = receiveSync(t,C); enqueue(Q,(i_await,t));
    i_get = receiveSync(t,C); enqueue(Q,(i_get,t));
    if ∃ ∈ o --t--> t_2 ∈ C then
        insert(L,(i_get,t));
    end
end
while !empty(Q) do
    (i,t) = dequeue(Q);
    if ∃i_await ∈ t between the beginning of t and i then
        forall the f ∈ fields(i) do
            forall the (i_mod, t_mod) ∈ mods(f) do
                if !member(L,(i_mod, t_mod)) then
                    insert(L,(i_mod, t_mod));
                    enqueue(Q,(i_mod, t_mod));
                end
            end
        end
    end
end
return [(m,1,M) : m ∈ set(project_y(L))];
```

Algorithm 1. Algorithm to infer interfering tasks for a given deadlock cycle

definition to the abstract cycle C in Example 6, $initialTasks(\mathsf{db.register}, 32, C)$ will be evaluated. It fits well with the conditions on the third clause, as there exists an **await** instruction, $\mathsf{fields(32)} = \{\mathsf{connected}\}$ and then again 32 is a modifier instruction of field connected, so $initialTasks(\mathsf{db.register}, 32, C)$ will be evaluated again recursively.

Algorithm 1 shows how to finitely infer the interfering-tasks for a given deadlock cycle as defined by Definition 5. Function $receiveCall(t, C)$ ($receiveSync(t, C)$) receives the asynchronous call (synchronization instruction) of a task t inside the cycle C. Q is the queue of pending pairs {instruction, task}, and L is the list containing all such pairs whose tasks we have to consider. Finiteness is guaranteed because each instruction is added to Q and L at most once, and the number of instructions is finite. For each task in the cycle, we take the call and the corresponding synchronization instruction, and we add them to Q. Instructions **get** producing a *location-task edge*, are also added to L, as they have to be inside the initial context. The other tasks included in the initial context are the ones which could affect the conditions of the aforementioned instructions.

In the second loop, we take a pending instruction inside Q and we check if there exists an **await** instruction where the field values could be changed

(third clause in Definition 5). In case it does, we need to include all tasks which contain instructions modifying such field. However, this change could be inside an if-else body and we also need to consider the fields inside such condition. Therefore, we add the modifier instruction to the pending instructions queue Q. The algorithm finishes when Q is empty and L is the list of pairs with all interfering instructions and their container tasks. Finally, we only take the tasks, i.e., the second component of each pair (project_y), remove duplicates (set) and set their minimum and maximum cardinalities. From now on, we denote *initial_tasks(c,M)*, the set of initial tasks inferred for the abstract deadlock cycle c and the maximum cardinality M.

Example 7. Let us show how the algorithm works for our modified example and the maximum cardinality $M = 1$. For the sake of clarity, instructions are identified by their line numbers. After executing the first **forall** loop, the value of Q and L is $\{(33, \mathsf{DB.register}), (34, \mathsf{DB.register}), (11, \mathsf{Worker.work}), (12, \mathsf{Worker.work})\}$ and $[(34, \mathsf{DB.register}), (12, \mathsf{Worker.work})]$, respectively. Let us assume Q uses a LIFO policy, hence $(12, \mathsf{Worker.work})$ is taken first. Since $\mathsf{fields}(12) = \emptyset$, L stays unchanged. The same happens with $(11, \mathsf{Worker.work})$. At the beginning of the third loop, Q is $\{(33, \mathsf{DB.register}), (34, \mathsf{DB.register})\}$ and $(34, \mathsf{DB.register})$ is taken. Now, $\mathsf{fields}(34) = \{\mathsf{connected}\}$ and $\exists inst_{await}$ (line 30) between lines 26 and 34. We find three pairs modifying the field connected: (23,DB.connect), (27,DB.register) and (32,DB.register). None of them is a member of L and hence they are added to both queues. Now, Q is $\{(33, \mathsf{DB.register}), (27, \mathsf{DB.register}), (32, \mathsf{DB.register}), (23, \mathsf{DB.connect})\}$ but again $\mathsf{fields}(32) = \mathsf{fields}(23) = \emptyset$ and, thus, L stays unchanged. Finally, both $(33, \mathsf{DB.register})$ and $(27, \mathsf{DB.register})$ are taken and $\mathsf{fields}(33) = \mathsf{fields}(27) = \{\mathsf{connected}\}$, but the modifier instructions have been previously added to L, hence L remains unchanged. At the end of **while**, L is $\big\{(34, \mathsf{DB.register}), (12, \mathsf{Worker.work}), (27, \mathsf{DB.register}), (32, \mathsf{DB.register}), (23, \mathsf{DB.connect})\big\}$. Finally, the algorithm projects over the second component of each pair in the list, removes duplicates and returns the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{Worker.work}, 1, 1), (\mathsf{DB.connect}, 1, 1)\}$. Our generation of initial contexts for this set (see Example 5) produces

$$\mathcal{I} = \{\ \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db}_1}[\mathsf{work}]_{\mathsf{w}_1}\},\\ \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{connect}]_{\mathsf{db}_2}, [\mathsf{work}]_{\mathsf{w}_1}\}\},$$

where both initial contexts are composed of a worker location with a task work. However, the former context contains a database location with tasks register and connect, whereas the latter one contains two locations with a task register and a task connect, respectively. □

The next theorem establishes the soundness of our approach. Intuitively, soundness states that, for a given deadlock cycle c and maximum cardinality M, if there is an initial context, fulfilling M, from which a deadlock representative of c can be obtained, then our approach will generate a context (possibly different from the above) from which a deadlock representative of c is obtained.

Theorem 1 (Soundness). *Given a program P, an abstract deadlock cycle c and a maximum cardinality M, if there exists a derivation starting at a state S_{ini} and ending at S_{end} such that the cardinality of each task in S_{ini} is less than M and S_{end} is a representative of the cycle c, then there exists an initial context $St_0 \in generate\_contexts(initial\_tasks(c, M))$ such that $S_{end_2} \in exec(St_0)$ and S_{end_2} is also a representative of the cycle c.*

Proof. (Sketch) Let us define a task t as necessary in S_{ini} for the deadlock cycle c if and only if $\nexists S_{e'}$ such that $S_{ini}\backslash\{t\} \xrightarrow{*} S_{e'}$ and $S_{e'}$ is a representative of c, where $S\backslash\{t\}$ denotes the context S without the task t. Let us define now an initial context $nec(S)$ as the initial context that only contains the necessary tasks in S for c. In order to prove soundness, we need to prove that $nec(S_{ini}) \in generate\_contexts(initial\_tasks(M, c))$. We reason by contradiction. Assume that there exists a necessary task $t \in nec(S_{ini})$, instance of method m, which is not in any initial context generated. This is equivalent to assume that method m is not inferred by Algorithm 1. We can distinguish two different roles which task t plays in the deadlock situation:

- If task t gets blocked, then t contains an instruction pp:**get** where pp is the program point, and, by the soundness of the deadlock analysis (Theorem 1 of [9]), pp:**get** is the tag of an edge inside the deadlock cycle c. So, the pair (pp, m) is added to L in the first loop of Algorithm 1 and m is finally inferred. Thus, we have a contradiction.
- If task t modifies a field f at program point pp that appears in a condition of another task r, then we cannot get a deadlock if t is not executed before the evaluation of condition in task r (t is necessary). Here, we need to notice that if task r does not contain any **await**, symbolic execution explores all possible execution paths and t would be unnecessary. But we have supposed that t is necessary, then r contains an **await**. Then, (pp, m) will be added to L because of the third forall in Algorithm 1 and m is inferred, what contradicts our assumption. □

5 Experimental Evaluation

We have implemented the proposed techniques within the aPET/SYCO tool [4], a testing tool for the ABS [13] *concurrent objects* language. The tool is available for online use at http://costa.ls.fi.upm.es/syco, where the benchmarks below can also be found. This section summarizes our experimental evaluation whose objectives are the following:

1. Show the effectiveness of our approach in Sect. 4 to generate initial contexts for deadlock detection w.r.t the full systematic generation of Sect. 3.
2. Demonstrate the potential of the technique when being applied in practice within our deadlock detection framework.

The benchmarks we have used include classical concurrency patterns containing deadlocks, namely: *DBProt* is an extension of the database communication

protocol of our working example; *Barber* is an extension of the *sleeping barber* problem, *Fact* is a distributed and recursive implementation of a factorial function, *Loop* is a loop that creates asynchronous tasks and locations, and, *Pairing* is the pairing problem.

Effectiveness of generation of initial contexts for deadlock detection: Table 1 shows, for each benchmark: the number of generated initial contexts using the full systematic generation of contexts of Sect. 3 (column *Syst.*), the number of contexts generated using our deadlock-guided generation of Sect. 4 (column G), and, the number of contexts among those generated that lead to a deadlock (column D). This is done for three different values of maximum cardinality, namely, $M = 1$, $M = 2$ and $M = 3$. The rest of the columns are explained in the next paragraph. A timeout of 30 s is used and, when reached, we write $>X$ to indicate that we encountered X contexts up to that point. The reductions of our deadlock guided generation of contexts w.r.t the full systematic generation are huge. As expected the full systematic generation blows up fast for most examples. We can also observe that our deadlock guided generation of contexts is very precise, producing no false positives, i.e., contexts that do not lead to deadlock, except for *DBProt*. The reason of the loss of precision in the *DPProt* example is that task `register` only gets blocked if task `connect` changes the value of field `connected`. Therefore, contexts in which these two tasks do not belong to the same location will not lead to deadlock. This can be observed in Example 7. Improving our method to capture this situation is left for future work.

Table 1. Evaluating generation of initial contexts: Systematic vs. deadlock-guided

		M = 1				M = 2				M = 3			
Bench.	T_A/C	Syst.	G	D	T	Syst.	G	D	T	Syst.	G	D	T
DBProt	5/1	30	2	1	35	>12960	57	30	101s*	>6308	576	156	974s*
Barber	5/1	8	1	1	35	6859	9	9	57	>8310	36	36	309
Fact	6/2	15	2	2	11	2419	6	6	14	>4771	12	12	16
Loop	20/1	3375	1	1	30	>13433	27	27	495	>4771	216	216	77s*
Pairing	4/2	2	2	2	9	57	12	12	37	576	42	42	162

Application within our deadlock detection framework: Our deadlock-guided generation of initial contexts has been integrated within the deadlock detection feature of the testing system aPET/SYCO as follows: After running the static deadlock analysis, and only in case it outputs a non-empty set of potential abstract cycles (i.e. if the program is not already proven deadlock-free), we run our deadlock guided generation of initial contexts for each of the cycles inferred by the analysis. For each generated initial context, we start (possibly in parallel) a deadlock-guided symbolic execution [2,3] that stops as soon as it finds a deadlock. As a result, we obtain a concrete test-case with its associated

trace and sequence of interleavings. A local timeout for each symbolic execution is set so that it does not degrade the overall process in case a blowup is produced before finding a deadlock. This is relatively frequent with false-positive contexts (see paragraph above). Table 1 shows, for each benchmark, the time of the static deadlock analysis and the number of generated deadlock cycles (column T_A/C), and, the overall time of the rest of the process (column T), which includes both the time of the generation of contexts and the symbolic executions. Times are in milliseconds except where indicated and are obtained on an Intel(R) Core(TM) i7 CPU at 2.5 GHz with 8 GB of RAM, running Ubuntu 5.4.0. A timeout of 5s is set for each symbolic execution and an asterisk in the time indicates the timeout has been reached at least once.

Overall, our deadlock guided generation of initial contexts hence enables our deadlock detection framework to analyze systems without the need of any user supplied initial context. Also, it allows generating concrete test cases that lead to deadlock for integration and system testing.

6 Conclusions and Related Work

We have proposed a framework for the automatic generation of initial contexts for deadlock-guided symbolic execution. Such initial contexts are composed of the interfering tasks which, according to a static deadlock analyzer, might lead to deadlock. Given the initial contexts, we can drive symbolic execution towards paths that are more likely to manifest a deadlock, discarding safe contexts. There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [9,10] and thread-based programs [17,19], is based on static analysis techniques. Although we have used the static analysis of [9], the information provided by other deadlock analyzers could be used in an analogous way. Deadlock detection has been also studied in the context of dynamic testing and model checking [6,15,16], where sometimes has been combined with static information [1,14]. The initial contexts generated by our framework are of interest also in these approaches. As regards the application in a thread-based concurrency model, the fundamental difference is that our whole approach is defined at the level of atomic tasks that execute concurrently using non-preemptive scheduling, unlike thread-based preemption. However, our approach would be adaptable to thread-based applications that rely on synchronized blocks of code (such as in monitors or concurrent objects). As future work, we plan to investigate how our framework could be adapted to this model.

References

1. Agarwal, R., Wang, L., Stoller, S.D.: Detecting potential deadlocks with static analysis and run-time monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006). https://doi.org/10.1007/11678779_14
2. Albert, E., Gómez-Zamalloa, M., Isabel, M.: Deadlock Guided Testing in CLP. Technical report (2017). http://costa.ls.fi.upm.es/papers/costa/AlbertGI17tr.pdf
3. Albert, E., Gómez-Zamalloa, M., Isabel, M.: Combining static analysis and testing for deadlock detection. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 409–424. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_26
4. Albert, E., Gómez-Zamalloa, M., Isabel, M.: SYCO: a systematic testing tool for concurrent objects. In: Proceedings of CC 2016. ACM (2016)
5. Albert, E., Gómez-Zamalloa, M., Isabel, M.: On the generation of initial contexts for effective deadlock detection. Technical report, October 2017. https://arxiv.org/abs/1709.04255
6. Christakis, M., Gotovos, A., Sagonas, K.F.: Systematic testing for detecting concurrency errors in erlang programs. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, 18–22 March 2013. IEEE Computer Society (2013)
7. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
8. Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1995)
9. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE -2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_19
10. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis of concurrent objects: theory and practice. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 394–411. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_27
11. Giachino, E., Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 63–77. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_6
12. Laneve, C., Giachino, E., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. **15**(4), 1013–1048 (2016)
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. Joshi, P., Naik, M., Sen, K., Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In: Proceedings of FSE 2010. ACM (2010)
15. Joshi, P., Park, C., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: Proceedings of PLDI 2009. ACM (2009)
16. Kheradmand, A., Kasikci, B., Candea, G.: Lockout: Efficient Testing for Deadlock Bugs. Technical report (2013). http://dslab.epfl.ch/pubs/lockout.pdf

17. Masticola, S.P., Ryder, B.G.: A model of Ada programs for static deadlock detection in polynomial time. In: Parallel and Distributed Debugging, pp. 97–107. ACM (1991)
18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**, 1–41 (2005)
19. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
20. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006). https://doi.org/10.1007/11693017_25

SYCO: A Systematic Testing Tool for Concurrent Objects

Elvira Albert
Complutense University of Madrid
elvira@fdi.ucm.es

Miguel Gómez-Zamalloa
Complutense University of Madrid
mzamalloa@fdi.ucm.es

Miguel Isabel
Complutense University of Madrid
miguelis@ucm.es

Abstract

We present the concepts, usage and prototypical implementation of SYCO: a SYstematic testing tool for *Concurrent Objects*. The system receives as input a program, a selection of method to be tested, and a set of initial values for its parameters. SYCO offers a visual web interface to carry out the testing process and visualize the results of the different executions as well as the sequences of tasks scheduled as a sequence diagram. Its kernel includes state-of-the-art partial-order reduction techniques to avoid redundant computations during testing. Besides, SYCO incorporates an option to effectively catch *deadlock* errors. In particular, it uses advanced techniques which guide the execution towards potential deadlock paths and discard paths that are guaranteed to be deadlock free.

Categories and Subject Descriptors D1.3 [*Programming Techniques*]: Concurrent Programming; D2.5 [*Testing and Debugging*]: [testing tools, systematic execution]

Keywords systematic testing, concurrency, concurrent objects, software testing, partial-order reduction

1. Motivation

Testing is the most widely-used methodology for software validation in industry. Several studies point out that it requires at least half of the total cost of a software project. Software testing tools urge especially in the context of concurrent programming. This is because writing correct concurrent programs is more difficult than writing sequential ones as with concurrency come additional hazards not present in sequential programs such as race conditions, deadlocks, and livelocks. In order to catch such errors, the testing tool must consider the non-determinism caused by the fact that an execution can lead to different solutions depending on the way that the involved tasks interleave, and, ideally, all possible interleavings must be considered. A systematic exploration of the state space is usually not feasible. A lot of research has been done in the context of testing and model checking with the aim of avoiding redundant state exploration as much as possible [1, 2, 5, 10]. SYCO is a testing tool that targets the ABS concurrent objects language [8] and that incorporates state-of-the-art partial-order-reduction (POR) techniques to avoid redundant exploration.

Essentially, a concurrent object is a monitor that allows at most one *active* task to execute within the object. Task scheduling is non-preemptive, i.e., the active task has to release the object lock explicitly (using the **await** or **return** instructions). Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and start executing. Each object has a local heap or memory (set of fields) which can only be accessed from the owner object. The instruction f = ob!m() creates an asynchronous task to execute method m on object ob. Synchronization can be performed using the *future variable* f, namely the instruction **await** f? checks if the execution of the asynchronous task has finished. It not, the object lock is released and the task suspends until the value of f is ready. In contrast, the instruction v = f.**get** blocks the task until f is ready retaining the object lock. Once the execution of the task finishes, it assigns the obtained value to v.

Running Example. The following example simulates a simple communication protocol between a database and a worker.

```
1 {\\main block                    14   Int getD(Worker w){
2   DB db = new DB();              15     if (cl == w) return data;
3   Worker w = new Worker();       16     else return null;
4   db!register(w);                17   }
5   w!work(db);                    18 }// end class DB
6 }                                19 class Worker{
7 class DB{                        20   Data data;
8   Data data = ...;               21   void work(DB db){
9   Worker cl = null;              22     Fut<Data> f = db!getD(this);
10  void register(Worker w){       23     data = f.get;
11    Fut<Int> f = w!ping(5);      24   }
12    if (f.get == 5) cl = w;      25   Int ping(Int n){return n;}
13  }                              26 }// end of class Worker
```

The main method creates the two objects and invokes methods register and work resp. The work method of the worker simply accesses the database (invoking asynchronously method getD) and then blocks until it gets the result, which is assigned to its data field. The register method of the database, first checks that the worker is online (invoking asynchronously method ping), then blocks until it gets the result, and finally it registers the worker by storing its reference in its cl field. Method getD of the database returns its data field if the caller worker is registered, otherwise it returns **null**.

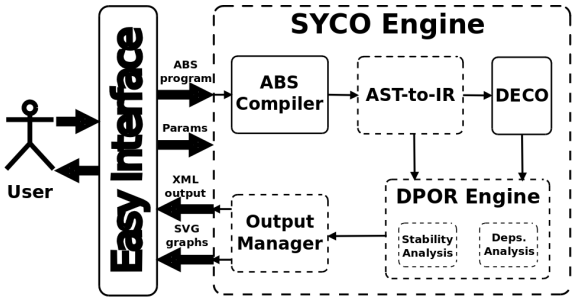
Depending on the sequence of interleavings, the execution of this program can finish: (i) as expected, i.e., with w.data = db.data , (ii) with w.data = **null**, or, (iii) in a deadlock. (i) happens when the worker is registered in the database (assignment in L12) before getD is executed. (ii) happens when getD is executed before the assignment at L12. A deadlock is produced if both register and work start executing before getD and ping.

2. The SYCO Tool

The figure above shows the main architecture of SYCO. Boxes with dash lines are internal components of SYCO whereas boxes with regular lines are external components. The user interacts with SYCO through its web interface which is provided by *EasyInter-*

CC'16, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892236

face [7]. Basically EasyInterface provides a generic IDE which can be instantiated to different languages and compilers and where external plugins can be easily added. The SYCO engine receives an ABS program and a selection of parameters. The *ABS compiler* compiles the program into an abstract-syntax-tree (AST) which is then transformed into the SYCO intermediate representation (IR). The *DPOR engine* carries out the actual systematic testing process. It comprises the ABS semantics, the DPOR algorithm of [2] and the *stability* and *dependencies* analyses of [2]. The *output manager* then generates the output in the format which is required by EasyInterface, including an XML file containing all the EasyInterface commands and actions and the SVG diagrams. In case a deadlock-guided testing is requested (see the corresponding parameter below), the *DECO* deadlock analyzer [6] is invoked, whose output is used by the DPOR engine to guide the testing process (discarding non-deadlock executions) [4]. Let us note that other actor-based languages with similar features could be handled by SYCO just by providing a compiler to the SYCO IR.



The web interface of SYCO is available at `costa.ls.fi.upm.es/syco`. Essentially, once the input program is ready, either selected from the available library of ABS programs or supplied by the user, a set of parameters are provided (or just left with by-default values), the SYCO engine is run and the output is obtained.

Parameters. The following parameters can be set:

- *Partial-order reduction*: It enables/disables POR.
- *Dependency over-approximation*: In case POR is applied, a central operation is the detection of independent tasks, which has to be over-approximated. SYCO includes the over-approximation of [10] which considers as dependent tasks those in the same actor, and, also, the enhancement of [2] for actors with local memory, which looks at field accesses within the involved tasks and considers as dependent only tasks belonging to the same actor and accessing at least a common field.
- *Deadlock-guided testing*: If this parameter is selected, the testing process is guided with the cycles inferred by DECO towards deadlocks, discarding non-deadlock executions, with the corresponding state space reduction.

Output. As a result, SYCO outputs a set of executions. For each one, SYCO shows the output state and the sequence of tasks/interleavings and concrete instructions of the execution (highlighting the source code). Also, it allows showing a sequence diagram from which it can be observed the task/object executing and the asynchronous calls made (with arrows from caller to callee) at each time of the simulation, the waiting and blocking dependencies, the deadlock cycles, etc. SYCO produces 6 executions for the running example with POR disabled. That covers all possible task interleavings that may occur. SYCO reports that 2 executions are deadlock executions corresponding to sequences main→register→work and main→work→register. Those correspond to scenario (iii) at the end of Sect. 1. Within the remaining 4 executions, two of them correspond to scenario (i) and the other two to scenario (ii). According to POR theory [2, 10], the remaining 4 executions can be grouped in two equivalence classes, therefore 2 executions are redundant and only two different results are obtained. When POR is enabled, SYCO produces these 4 executions, the two deadlock executions, and, the executions corresponding to scenarios (i) and (ii).

3. Discussion and Related Work

We have presented a systematic tester for an actor-based concurrency model which incorporates state-of-the-art POR methods. The tool can be used online through its web interface and provides information about all possible (non-redundant) behaviors that the input concurrent program may have, including trace highlighting and detailed sequence diagrams. It also has support for deadlock detection and debugging, incorporating novel techniques for deadlock-guided testing [4] in which an external deadlock analyzer [6] is embedded. We claim that the tool is very useful for testing and debugging models of concurrent systems.

Several related tools exist, being the most relevant Microsoft's *CHESS* [9] for .NET, *Concuerror* [5] for Erlang and *Basset* [10] for *ActorFoundry*. All of them incorporate state-of-the-art POR techniques. The most advanced in this sense is Concuerror which is equipped with the most recent *Optimal* DPOR algorithm [1]. Also, Concuerror is the only one providing graphical output similar to our sequence diagrams. None of them provides a web interface. Many other related tools exist in the context of *model-checking* that are left out of this comparison.

As regards future work, we are currently studying the most advanced POR techniques of [1] and the possibility of adapting them to our context. Also, we are in the process of incorporating the symbolic execution engine of [3] so that SYCO also allows performing static testing.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. POPL'14, pp. 373–384*. ACM, 2014.

[2] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *Proc. FORTE'14*, LNCS 8461, pp. 49-65. Springer, 2014.

[3] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *Proc. ESEC/FSE'13, pp. 595–598*. ACM, 2013.

[4] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015.

[5] S. Aronis and K. Sagonas. Concuerror: Systematic concurrency testing of Erlang programs.

[6] A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pages 273–288. Springer, 2013.

[7] S. Genaim and J. Doménech. The EasyInterface Framework, 2015. http://github.com/abstools/easyinterface.

[8] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10*, LNCS 6957, pp. 142-164. Springer, 2012.

[9] M. Musuvathi and S. Qadeer. Concurrency Unit Testing with CHESS. Tech. Report MSR-TR-2008-04, Microsoft Research, January 2008.

[10] D. Marinov G. Agha S. Lauterburg, R. K. Karmani. Basset: A Tool for Systematic Testing of Actor Programs. In *Proceedings of FSE 2010*, pages 363–364. ACM, 2010.