UC Projeto

3º ano Licenciatura em Ciências da Computação

Construção de um ferramenta genérica de verificação SAT para propriedades de segurança de sistemas de transição de 1ª ordem (FOTS)

Alef Keuffer (A91683)

Alexandre Baldé (A70373)

Bruno Machado (A91680)

Pedro Pereira (A88062)

Supervisor: Professor José Manuel Esgalhado Valença

30 de junho de 2022

Resumo

Neste relatório apresenta-se o trabalho realizado para a UC de Projeto, que consistiu no desenvolvimento, em Python[1], de uma ferramenta genérica de verificação de propriedades de "*First-Order Transition Systems*" através de uma biblioteca-interface para SMT-LIB.

Para a verificação de correção de propriedades de segurança de FOTS, foram analisados os seguintes métodos: "*Bounded Model-Checking*", "*k-induction*", "*Property Directed Reachability*", "*Interpolant-Based Model-Checking*".

[1] `https://www.python.org/`

Conteúdo

Lista de Figuras

Lista de Excertos de Código

Capítulo 1

Introdução

Este relatório contém a descrição do projeto realizado pelos autores para a UC de Projeto da Licenciatura em Ciências da Computação, para o ano letivo de 2021/2022.

1.1 Estrutura do Relatório

A estrutura do relatório é a seguinte:

- No capítulo **??** faz-se uma análise do trabalho já existente na área, e das referência usadas para o projeto.
  - Na secção **??** consideram-se as técnicas relacionadas com indução
  - Na secção **??**, apresenta-se o estado de arte para a técnica de BMC
  - Na secção **??**, considera-se a origem e estado das técnicas de interpolação
  - Na secção **??**, está uma análise a PDR
- No capítulo **??** explicam-se alguns aspetos mais técnicos e concretos da implementação, assim como decisões tomadas e alternativas consideradas.
  - Na secção **??** olha-se para a parte da ferramenta que implementa "*k-induction*" e BMC
  - Em **??** está a implementação de IMC, e discutem-se o trabalho alternativo que se considerou
  - Em **??** apresenta-se a implementação do PDR, e descreve-se igualmente as alternativas consideradas à implementação escolhida
- No capítulo **??** apresenta-se um case de estudo com um FOTS que servirá para apresentação das funcionalidades desenvolvidas.
- No capítulo **??** termina-se o relatório com as conclusões e o trabalho futuro.
- Nos anexos **??** e **??** , encontra-se informação relativa ao código Python desenvolvido, assim como o repositório GitHub que o contém, e como utilizar o código.

1.2 Problema em análise

Parte da verificação formal de "*software*" prende-se com, se possível, extrair garantias de segurança de programas, e se impossível, obter contra-exemplos que demonstrem a insegurança do sistema.

Neste projeto, o objeto de estudo serão "*First-Order Transition Systems*", que por terem uma representáveis através de lógica de primeira ordem, são então manipuláveis através de "*solvers*" — este tema já foi abordado na UC de Lógica Computacional em [**?**], pelo que para evitar repetição, utilizar-se-ão referências ao material da UC.

Existe um vasto corpo de conhecimento relativo à verificação de propriedades de "*SMT solvers*", que inclui vários métodos diferentes para efetuar este tipo de provas — ver **??** para uma breve exploração das referências atuais no campo.

O objetivo deste projeto foi implementar uma ferramenta que permitisse a verificação de propriedades de "*First-Order Transition Systems*" através de alguns desses métodos.
Para implementar a ferramenta de verificação, utilizou-se a linguagem de programação Python, e a uma biblioteca de "*SMT solvers*" disponível **PySMT**[1].

1.2.1 Trabalho realizado

O resultado deste trabalho é uma ferramenta escrita em Python capaz de, dado um FOTS definido na DSL do PySMT, provar/refutar propriedades desse FOTS recorrendo às técnicas:

- "*k-induction*"
- "*Bounded Model-Checking*"
- "*Interpolant-Based Model-Checking*"
- "*Property Directed Reachability*"

1.3 Resolução e Estratégias adotadas

Para resolver o problema proposto, e implementar a ferramenta, consideraram-se as seguintes estratégias:

1.3.1 "*k*-*induction*" e "*Bounded Model-Checking*"

- As versões básicas destas duas técnicas foram estudadas e implementadas pelas autores após frequência da UC de Lógica Computacional no ano letivo de 2021/2022.
- Logo, na ferramenta deste projeto utiliza-se a implementação desenvolvida pelos autores nessa UC, com auxílio dos docentes.

[1] Documentação disponível em `https://pysmt.readthedocs.io/en/latest/`

1.3.2 "*Interpolant-Based Model-Checking*"

Para implementar esta técnica, consideraram-se duas formas principais:

- Uma permite provar propriedades sobre "*First-Order Transition Systems*" arbitrários, e requer o uso do teorema do interpolante de Craig[2]
- Outra possibilidade que se considerou foi converter "*First-Order Transition Systems*" para um programa na linguagem de fluxos abordada na UC de Lógica Computacional— apenas quando fosse possivel, porque nem sempre o será —, e depois utilizar as noções de WPC/SPC como interpolante de fórmulas

O item acima não foi completado, mas considerou-se também, caso tivesse sido, implementar uma versão do "*Interpolant-Based Model-Checking*" que utilizasse ambas técnicas em simultâneo de forma dinâmica, consoante as características do "*First-Order Transition System*", com o propósito de melhorar o desempenho do método

Linguagem de representação para "*First-Order Transition Systems*"

Uma das referências que se considerou para implementar "*Interpolant-Based Model-Checking*" foi [**?**] — ver **??**. Aí, utiliza-se a ferramenta CPAChecker, que possui uma linguagem própria para a definição de FOTS [3].

Para este projeto, considerou-se a implementação de uma linguagem própria semelhante à usada pelo CPAChecker para representar FOTS, recorrendo à análise semântica do FOTS para verificar se é possível convertê-lo para linguagem de fluxos.

Esta ideia foi discutida e fragmentos de um protótipo estão identificados no projeto; em última instância não se completou a implementação devido a restrições temporais.

1.3.3 "*Property Directed Reachability*"

A técnica do PDR foi abordada brevemente na UC de LC.

Em suma, consideraram-se duas abordagens.

- A primeira consistiu em seguir [**?**], e reimplementar a noção de "*induction-guided abstraction-refinement*", com a implementação em Java deste método servindo de referência. Optou-se por não terminar esta a favor da seguinte, dada a complexidade da implementação-guia, e o tempo disponível.
- A segunda foi baseada em [**?**], que é mais simples por ser uma extensão da k-indução.

1.4 Agradecimentos

Os autores gostariam de agradecer ao Professor Valença, supervisor deste projeto, pela sua paciência infindável ao ajudá-los durante o semestre letivo no desenvolvimento desta ferramenta, e pela sua disponibilidade e gentileza.

[2] `https://en.wikipedia.org/wiki/Craig_interpolation`

[3] veja-se um exemplo em `https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/config/specification/TerminatingStatements.spc`

Capítulo 2

Estado de arte

Os autores propuseram-se completar a UC Projeto com um tema relacionado com métodos de verificação de propriedades de FOTS após frequência da UC de Lógica Computacional da Licenciatura em Ciências da Computação da Universidade do Minho.

O material desta UC [**?**] que serviu de ponto de partida encontra-se no capítulo 3 (parte 2), em particular:

- Introdução aos "*First-Order Transition Systems*"
- Introdução a "*Bounded Model-Checking*"
- introdução à k-indução
- Introdução a PDR

Depois de revisto este material, prosseguiu-se a uma análise de outras referências na área de verificação de propriedades através de SMT, que se enumeram de seguida.

2.1 "k-*induction*"

Na obra [**?**] apresentaram-se o que foram, à data da publicação, novidades no uso de técnicas baseadas em indução para provar propriedades de segurança de "*Finite State Machines*".
No entanto, nessa obra não se introduz a automatização parcial do processo de prova no que diz respeito à derivação e inserção manual de invariantes no sistema de "*SAT-solving*" que depois se utilizá para fazer as provas —recorde-se que este process nunca poderá ser completamente automatizado devido à indecidibilidade da lógica de primeira ordem.

Relativamente a métodos baseados inteira ou principalmente na indução, trabalho recente inclui [**?**], com análise a formas de mecanizar, total ou parcialmente, o processo de procura de invariantes em "*k-induction*", e uma implementação na já referida ferramenta CPAChecker.

2.2 "*Bounded Model-Checking*"

Em [**?**] e [**?**], discute-se BMC, e apresentam-se duas implementações genéricas da técnica, acompanhadas de medições de desempenho que mostravam ser competitiva com outras técnicas populares à data da pu-

blicação desses artigos para verificação de propriedades de FOTS, que eram baseadas em "*Binary Decision Diagrams*", ou BDDs, que foram introduzidos em [**?**].

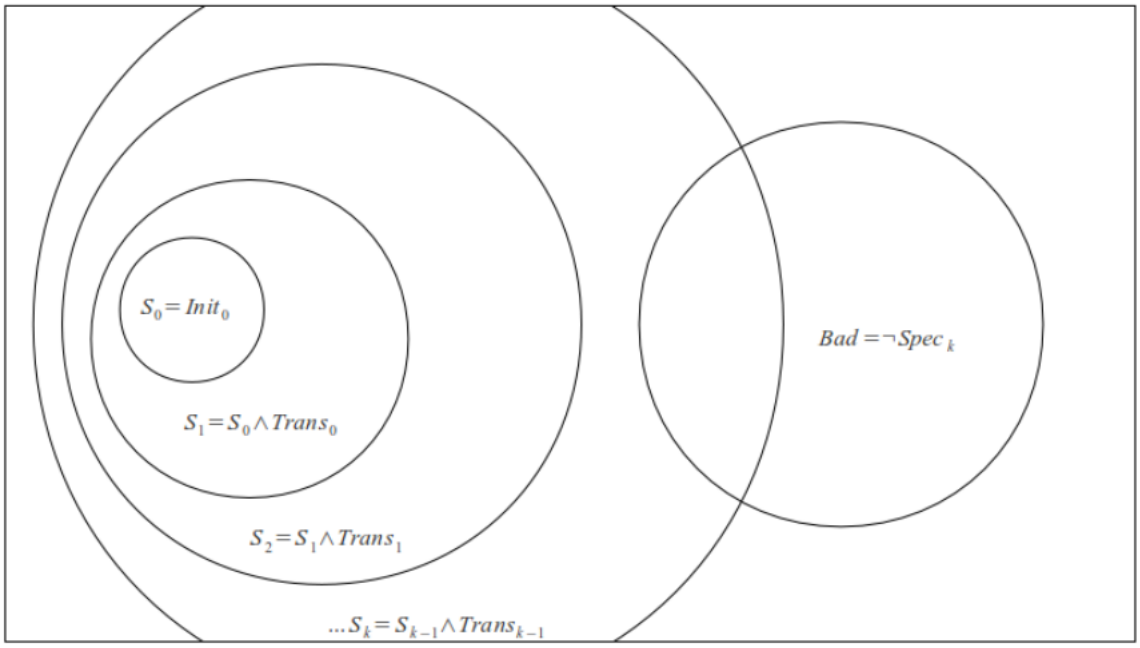


Figura 2.1: Diagrama para funcionamento de BMC retirado de [**?**, p. 16].

Na figura **??**, está um diagrama que elucida a metodologia BMC. Para um FOTS = $(Init, Trans)$, e uma propriedade *Spec*, a técnica de BMC tenta verificar a atingibilidade de $\neg Spec$ em até k passos, podendo apenas garantiar a segurança do sistema em traços finitos.

2.3 "*Interpolant-Based Model-Checking*"

O corpo de técnicas de IMC começou em [**?**], com uma técnica para prova (ou refutação) de propriedades de FOTS através da utilização do interpolante de Craig, que se mostrou ser favorável em comparação com as técnicas comuns à data da sua publicação, que se baseavam também nas BDDs referidas acima.

No entanto, a restrição aqui é de apenas se considerarem FOTS finitos, que são os aplicáveis à verificação de circuitos e "*hardware*" industrial —para modelar programas de software para o ramo da verificação formal, estes geralmente tomam a forma de FOTS infinitos.

Em [**?**], que foi uma das referências inicias dadas pelo Professor Valença, construiu-se sobre [**?**] para permitir FOTS infinitos, com o intuito de aplicar a técnica à verificação de "*software*". Fez-se novamente uma análise do desempenho que comprovou a viabilidade da técnica numa suite de instâncias não-triviais de problemas-teste [1].

Tem-se também o trabalho presente em [**?**], que construindo sobre [**?**] faz várias contribuições adicionais ao estabelecer resultados teóricos sobre IMC, entre eles:

- Todos os interpolantes gerados durante o IMC são aproximações, pelo que a sua precisão vai influenciar

[1] `https://github.com/sosy-lab/benchexec`

a velocidade da convergência do algoritmo até se chegar à indutividade da fórmula ou contra-exemplo. Este trabalho sistematiza formas de gerar interpolantes com propriedades e "força" (no sentido lógico, WPC/SPC) desejadas.

- À interpolação de fórmulas podem ser impostas propriedades específicas para garantir o funcionamento de uma dada técnica baseada em IMC. Esta obra estabelece relações entres as diversas propriedades, e uma hierarquia entre elas, que, relacionada com o ponto anterior, permitirá a construção de algoritmos de IMC "genéricos" com uma componente modular no algoritmo de interpolação utilizado, consoante o resultado que se pretende obter.

As inovações descritas nesta obra encontram-se em [**?**], capítulo 1.3.

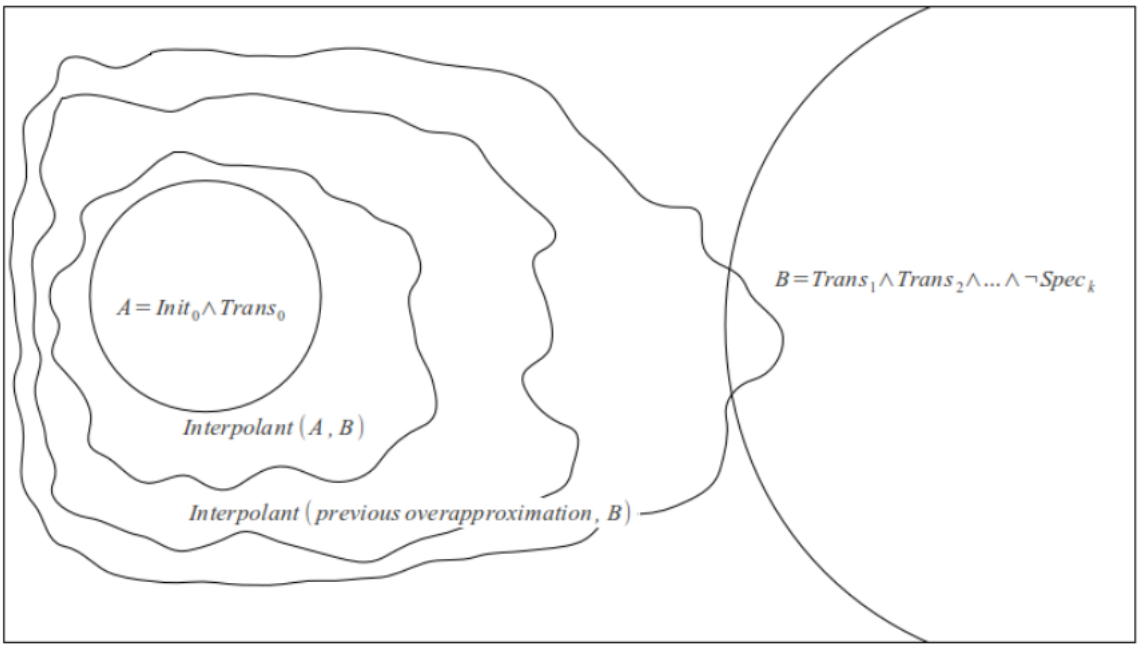


Figura 2.2: Diagrama para funcionamento de IMC retirado de [**?**, p. 17].

Na figura **??**, exemplifica-se a versão mais simples desta técnica. Para um FOTS = $(Init, Trans)$, e uma propriedade *Spec*, a técnica de IMC prova a validade de *Spec* se for consistente com alguma "overapproximation" de $Init \wedge Trans$ indutiva (obtidas através de interpolação). Ao contrário do BMC, não se limita a traços infinitos, embora em [**?**] McMillan prova um limite superior às iterações necessárias para se atingir um estado indutivo, ou a refutação.

2.4 "*Property Directed Reachability*"

A técnica "*Property Directed Reachability*" é introduzido em [**?**], com o "*caveat*" de não suportar, nesta descrição inicial, sistemas de transição de estados infinitos. Baseia-se principalmente na noção de "generalização indutiva", descrita em [**?**].

Consecutivo a [**?**] vem [**?**], que apresenta melhorias à implementação e desempenho do algoritmo PDR original, e discute a possibilidade de melhoramentos adicionais futuros.

Para poder utilizar PDR em sistemas de estados infinitos —e daí poder trabalhar com FOTS que representem programas, e fazer verificação formal —, é necessário generalizar para FOTS de estados infinitos, o que foi feito em [**?**]. Isto requer uma combinação de interpolação, "*k-induction*" e o PDR original. Foi esta a referência dada aos autores para começar o estudo deste tópico.

CTIGAR e IC3

A título complementar, mas sem ligação direta a este trabalho, nota-se que em [**?**] está uma introdução à primeira implementação de um "*model checker*" baseado no PDR de [**?**] pelo próprio autor.

Em [**?**] descreveu-se uma extensão do IC3 apta para sistemas de transição com estados infinitos nomeada CTIGAR.
O IC3 foi ainda alvo de trabalho adicional em [**?**] que, baseado em [**?**], desenvolveu outra extensão do IC3 original que também pode lidar com sistemas de transição de estados infinitos.

Capítulo 3

Análise do trabalho

Neste capítulo estará uma breve análise do trabalho feito, assim como algumas das decisões tomadas na implementação de cada uma das técnicas de verificação, assim como exemplos dessa implementação em PySMT quando não forem extensos demais.

Nota-se que todo o código aqui referenciado estará disponível no anexo em **??**, assim como no GitHub em **??**.

3.1 "*k-induction*" e "*Bounded Model-Checking*"

Para a "*k-induction*" e "*Bounded Model-Checking*", para além do trabalho feito nas aulas da UC de Lógica Computacional **??**, recorreu-se ao exemplo contido na documentação do PySMT[1], que combina "*k-induction*" e "*Bounded Model-Checking*" numa só classe Python, que é construída a partir da representação em PySMT de um FOTS. Este exemplo usa como referências [**?**, **?**].

Apresenta-se de seguida um fragmento da implementação. A implementação completa está em **??**.

```
def BMC_IND(I, T, P):
    """Interleaves BMC and K-Ind to verify the property."""
    print(f"Checking property {P[0]}...")
    from ply.cpp import xrange
    for b in xrange(100):
        f = get_bmc(I, T, P, b)
        print(f"   [BMC]    Checking bound {b + 1}...")
        if is_sat(f):
            print(f"--> Bug found at step {b + 1}")
            return

        f = get_k_induction(I, T, P, b)
        print(f"   [K-IND]  Checking bound {b + 1}...")
        if is_unsat(f):
            print("--> The system is safe!")
            return
```

[1] https://pysmt.readthedocs.io/en/latest/tutorials.html#model-checking-an-infinite-state-system-bmc-k-induction-in-

Observe-se que esta implementação faz "*interleaving*", ou "entrelaçamento", das duas técnicas BMC e "*k-induction*", gerando traços sucessivamente maiores para o BMC, e só no caso de sucesso com esta técnica se considera a utilização de k-indução, para k igual ao tamanho que se considera para os traços em cada momento, ou seja, também crescente.

3.2 "*Interpolant-Based Model-Checking*"

Como foi descrito acima, considerou-se a implementação da interpolação presente em [**?**], pp. 38.
Em seguida está o pseudocódigo presente na citação acima:

Input: Transition system (S, T), property P
Output: SAFE or UNSAFE and a counterexample
Data: k: bound, $R(i)$: overapproximation of states at distance at most i from S, $I(i)$: interpolant

1 **begin**
2 **if** $S \wedge \neg P$ *is satisfiable* **then** return UNSAFE, counterexample
3 $k \leftarrow 1, i \leftarrow 0$
4 $R(i) \leftarrow S$
5 **while** true **do**
6 $A \leftarrow R(i) \wedge T^0$
7 $B \leftarrow \bigwedge_{j=1}^{k-1} T^j \wedge \bigvee_{l=0}^{k} \neg P^l$
8 **if** $A \wedge B$ *is satisfiable* **then**
9 **if** $R(i) = S$ **then** return UNSAFE, counterexample
10 **else**
11 $k \leftarrow k+1, i \leftarrow 0$
12 $R(i) \leftarrow S$
13 **else**
14 $I(i) \leftarrow Itp(A, B)$
15 **if** $I(i) \models R(i)$ **then** return SAFE
16 **else**
17 $R(i+1) \leftarrow R(i) \vee I(i)$
18 $i \leftarrow i+1$
19 **end**
20 **end**

Figura 3.1: Pseudocódigo para IMC retirado de [**?**]

Considere-se agora a implementação quase direta em Python, através do PySMT:

```
def IMC(S: Predicate,
        T: Predicate,
        P: Predicate,
        interpolator: Callable[[FNode, FNode], FNode] = binary_interpolant,
        print_info: bool = True):

    print("Checking if initial states violates safety property")
    if m := get_model(S[0] & ~P[0]):
        # halt return a counterexample
        if print_info:
```

```
            print(f"[step 0] Initial state violates property:")
            print(f"{INDENT}Counterexample:")
            print(textwrap.indent(f"{m}", INDENT))
        return Status.UNSAFE1

    k = 2

    i = 0
    R = S[0]

    while True:
        A = R & T[0]
        B = T[1:k - 1] & Or(~P[l] for l in range(k + 1))
        print(f"[{i=},{k=}] Checking BMC from R(i)")
        if m := get_model(A & B):
            if is_valid(EqualsOrIff(R, S[0])):
                print(f"[{i=},{k=}] Checking if R=S")
                print(m)
                return Status.UNSAFE2
            else:
                print(f"[{i=},{k=}] R != S")
                k += 1
                i = 0
                R = S[0]
        else:
            print(f"[{i=},{k=}] Calculating interpolant")
            I = interpolator(A, B)

            if is_valid(I.Implies(R)):
                if print_info:
                    print(f"[{i=},{k=}] Proved safety: all states have been covered, "
                          f"and the system is safe")
                return Status.SAFE
            else:
                print(f"[{i=},{k=}] I !=> R")
                R |= I
                i += 1
```

Excerto de Código 3.1: Implementação em PySMT do algoritmo de IMC retirado de [**?**]

A versão com comentário encontra-se no anexo em **??**, ou no GitHub com o resto do código fonte, também no anexo **??**.

3.2.1 Problema da implementação em PySMT

Nota-se que surgiu um problema com esta implementação que os autores lamentavelmente não puderam resolver antes da data de submissão do projeto. A função acima, relativamente ao FOTS em **??**, dava como válidas propriedades obviamente falsas que o PDR em **??** corretamente refutava.

Após várias tentativas de "*debugging*", os autores suspeitam que se trate de uma, ou mais, das seguintes situações:

- O pseudocódigo de [**?**] apresenta alguma imprecisão —improvável.
- A função

3.2.2 Interpolante de Craig, escolha de lógica SMT

Para implementar este algoritmo, foi necessário utilizar o teorema de interpolação de Craig[2], que já estava implementado no PySMT.

É preciso notar que a implementação do PySMT não suporta todas as lógicas SMT-LIB. Por exemplo, no "caso de estudo" apresentado em **??**, a lógica QF_BV [3] é suficiente para considerar o problema.

No entanto, se não for **explicitamente** selecionada, o PySMT escolherá a lógica mais geral que conseguir, que no caso era QF_AUFBVLIRA[4], que não tem, à data da submissão do projeto, uma implementação de um algoritmo para cálculo do interpolante de Craig entre duas fórmulas arbitrárias.

Isto significa que o utilizador da ferramenta tem a responsabilidade de escolher manualmente a lógica apropriada para garantir o funcionamento desta implementação de IMC.

3.2.3 Interpolante utilizando noção de WPC e SPC, e linguagem de fluxos

Como se referiu em **??**, uma das propostas para a técnica do IMC foi fazer a conversão de FOTS para programas num fragmento da linguagem de fluxos estudada em Lógica Computacional, e utilizar as metodologias de "*Weakest pre-condition*" (WPC) e "*Strongest post-condition*" (spc) para obter interpolantes em vez do interpolante de Craig.

Elabora-se esta ideia de seguida.

Relação entre linguagem de fluxos e FOTS

Considere-se o seguinte programa simples retirado do material de LC[5]:

```
assert (z >= 0)
0: while z > 0:
1:     z = z - 1
2: stop
```

Excerto de Código 3.2: Exemplo de programa simples em Python

Aplicando a metodologia estudada na UC de LC (que se explica no link anterior), isto dá origem ao FOTS

[2] https://en.wikipedia.org/wiki/Craig_interpolation
[3] https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV
[4] https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_AUFLIA
[5] https://paper.dropbox.com/doc/Capitulo-3-Satisfiability-Modulo-Theories-2-Parte-zorZj2G3ceOIi92zrE0n1

3. O conjunto dos estados iniciais é determinado pelo predicado

$$\mathbf{init}(z, c) \equiv (c = 0) \wedge (z \geq 0)$$

4. A relação de transição é definida pelo predicado

$$\mathbf{trans}(z, c, z', c') \equiv \begin{cases} (c = 0) \wedge (z = 0) \wedge (c' = 2) \wedge (z' = z) & \vee \\ (c = 0) \wedge (z > 0) \wedge (c' = 1) \wedge (z' = z) & \vee \\ (c = 1) \wedge (z > 0) \wedge (c' = 0) \wedge (z' = z - 1) & \vee \\ (c = 2) \wedge (c' = 2) \wedge (z' = z) \end{cases}$$

Figura 3.2: Programa de **??** convertido para FOTS

Pode-se também considerar aumentar o FOTS com um estado de erro, como exemplificado em [**?**]. Para o exemplo acima, seria:

$$\mathbf{err}(z, c) \equiv (c = 2) \wedge (z > 0)$$

Considere-se agora a tradução do FOTS acima para a linguagem de programas anotados.

```
{
assume c = 0 and z = 0;
c = 2;
} ||
{
assume c = 0 and z > 0;
c = 1;
} ||
{
assume c = 1 and z > 0;
c' = 0;
z = z - 1;
} ||
{
assume c = 2;
skip;
}
```

Excerto de Código 3.3: Tradução de FOTS para LPA

Observe-se uma das razões para se considerar o IMC neste formato - no caso de uma transição não modificar uma variável do FOTS, ela não precisará ser incluída no programa LPA, levando a programas mais pequenos, que por si levarão depois a fluxos menores, e logo fórmulas mais simples que potenciam melhor desempenho.

A representação normal de um FOTS obriga à inclusão de todas as variáveis em todas as transições.

Note-se que esta tradução nem sempre é possível - no caso de o FOTS em causa representar um sistema ciber-físico, algumas das suas transições poderão não ser representáveis no fragmento da LPA que apenas contém atribuições e `assumes`. Isto acontece porque no caso das transições "*timed*", a discretização das equações diferenciais que regem a transição poderá não ser representável no fragmento da LPA que se escolheu.

Seja *Prog* o fluxo gerado do programa acima, e considere-se agora uma esquematização do IMC com LPA.

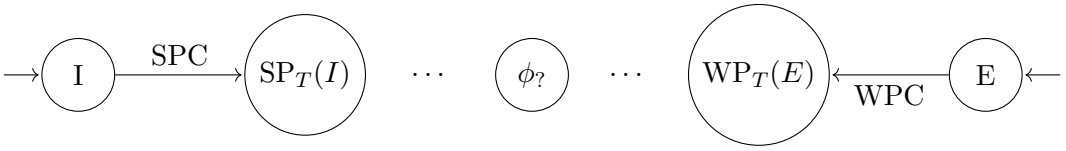


Figura 3.3: Interpolante em programas LPA com WPC/SPC

A ideia é partir dos dois estados, I e E, e ir sucessivamente aplicando as regras WPC/SPC descritas em [6], [7] para ir gerando novos predicados, e ver se eles alguma vez se intersectam para provar a validade da propriedade P que se quer provar:

- Se $\phi_?$ existir, ou seja, se houver algum estado alcançável simultaneamente a partir de I (através de SPC) e de E (através de WPC), então o sistema é inseguro
- Se $\phi_?$ não existir, ou seja, se se chegar a estados I_k, E_k que sejam indutivos e não sejam equivalentes, e tais que I_k seja consistente com P, então o sistema é seguro, e a propriedade é válida.

Para entender este conceito, recorreu-se a [**?**].

Esta ideia **não** foi implementada na ferramenta, mesmo após uma contribuição inicial do Professor Valença. No entanto, deixou-se o código não funcional no GitHub para referência futura; está indicado em **??**.

3.3 "*Property Directed Reachability*"

Na documentação da ferramenta PySMT, para além de implementações das técnicas "*k-induction*" e "*Bounded Model-Checking*"— ver **??** —, existe também uma implementação básica de PDR (vinda de [**?**]) que serviu aos autores para entender o funcionamento do método.

No entanto, essa implementação apenas suporta FOTS com um número de estados **finito**. Assim sendo, para suportar FOTS infinitos, é necessário outro algoritmo. Foi referido na introdução **??** que se consideraram duas formas de implementar a técnica de PDR para FOTS infinitos:

- Uma em [**?**] (capítulo 3)
- A outra em [**?**], que foi a opção escolhida no final.

[6] `https://paper.dropbox.com/doc/Capitulo-5-Verificacao-Formal-de-Software-e95D7fVpc0dArh4pnVl1l#:uid=847823822173927851567448&h2=Denota%C3%A7%C3%A3o-WPC-e-sua-linguagem-`

[7] `https://paper.dropbox.com/doc/Capitulo-5-Verificacao-Formal-de-Software-e95D7fVpc0dArh4pnVl1l#:uid=999909737185703645851102&h2=Denota%C3%A7%C3%A3o-SPC-(%E2%80%9Cstrongest-post`

Em CTIGAR existem as noções de "*counterexamples to induction (CTIs)*", de "*abstraction*" e "*consecution*" — ver [**?**] (capítulos 2, 3).

Devido a dificuldades na implementação da noção de "*consecution*", a partir da implementação em Java do CTIGAR, optou-se pela segunda via do PDR. Como no caso da metodologia SPC+WPC para o IMC **??**, manteve-se o código resultante desta tentativa no GitHub do projeto **??**.

Em seguida está o pseudocódigo presente em [**?**], que foi o ponto de partida para a implementação em PySMT:

Algorithm 1 Iterative-Deepening k-Induction with Property Direction

Input: the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$,
the initial states defined by the predicate I,
the transfer relation defined by the predicate T,
a safety property P,
a function get_currently_known_invariant to obtain auxiliary invariants,
a Boolean pd that enables or disables property direction,
a function $\mathsf{lift} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \times S \to (S \to \mathbb{B})$, and
a function $\mathsf{strengthen} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \to (S \to \mathbb{B})$,
where S is the set of program states.
Output: **true** if P holds, **false** otherwise
Variables: the current bound $k := k_{init}$,
the invariant $InternalInv := true$ computed by this algorithm internally, and
the set $O := \{\}$ of current proof obligations.

1: **while** $k \leq k_{max}$ **do**
2: $\quad O_{prev} := O$
3: $\quad O := \{\}$
4: $\quad base\_case := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
5: $\quad$ **if** $\mathsf{sat}(base\_case)$ **then**
6: $\quad\quad$ **return false**
7: $\quad forward\_condition := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
8: $\quad$ **if** $\neg\mathsf{sat}(forward\_condition)$ **then**
9: $\quad\quad$ **return true**
10: $\quad$ **if** pd **then**
11: $\quad\quad$ **for each** $o \in O_{prev}$ **do**
12: $\quad\quad\quad base\_case_o := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg o(s_n) \right)$
13: $\quad\quad\quad$ **if** $\mathsf{sat}(base\_case_o)$ **then**
14: $\quad\quad\quad\quad$ **return false**
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad step\_case_{o_n} := \bigwedge_{i=n}^{n+k-1} (o(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg o(s_{n+k})$
17: $\quad\quad\quad\quad ExternalInv := \mathsf{get\_currently\_known\_invariant}()$
18: $\quad\quad\quad\quad Inv := InternalInv \wedge ExternalInv$
19: $\quad\quad\quad\quad$ **if** $\mathsf{sat}(Inv(s_n) \wedge step\_case_{o_n})$ **then**
20: $\quad\quad\quad\quad\quad s_o :=$ satisfying predecessor state
21: $\quad\quad\quad\quad\quad O := O \cup \{\neg\mathsf{lift}(k, Inv, o, s_o)\}$
22: $\quad\quad\quad\quad$ **else**
23: $\quad\quad\quad\quad\quad InternalInv := InternalInv \wedge \mathsf{strengthen}(k, Inv, o)$
24: $\quad step\_case_n := \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
25: $\quad ExternalInv := \mathsf{get\_currently\_known\_invariant}()$
26: $\quad Inv := InternalInv \wedge ExternalInv$
27: $\quad$ **if** $\mathsf{sat}(Inv(s_n) \wedge step\_case_n)$ **then**
28: $\quad\quad$ **if** pd **then**
29: $\quad\quad\quad s :=$ satisfying predecessor state
30: $\quad\quad\quad O := O \cup \{\neg\mathsf{lift}(k, Inv, P, s)\}$
31: $\quad$ **else**
32: $\quad\quad$ **return true**
33: $\quad k := \mathsf{inc}(k)$
34: **return unknown**

Figura 3.4: Pseudocódigo para PDR retirado de [**?**]

Considere-se agora a implementação quase direta do pseucódigo acima em PySMT:

```
def PDR(I: Predicate,
        T: Predicate,
        P: Predicate,
        get_currently_known_invariant=lambda: TRUE(),
        strengthen=lambda k, Inv, o: Inv,
        lift=_lift,
```

```
    k_init: int = 1,
    k_max: int = float('inf'),
    pd: bool = True,
    inc: Callable[[int], int] = lambda n: n + 1,
    print_info=True
    ) -> bool | Status:
k: int = k_init

InternalInv: FNode = TRUE()

O: Set[Predicate] = set()

while k <= k_max:
    O_prev: Set[Predicate] = O
    O = set()

    # begin: base-case check (BMC)
    base_case = get_base_case(k, I, T, P)
    if m := get_model(base_case):
        if print_info:
            print(f"[{k=}] base-case check failed")
            print(f"{INDENT}Counterexample:")
            print(textwrap.indent(f"{str_model(m)}", INDENT))
        return False
    # end ############################################################################

    # begin: forward-condition check (as described in Sec. 2)
    forward_condition = I[0] & T[:k - 1]
    if is_unsat(forward_condition):
        print(f"[{k=}] Proved correctness: successful forward condition check")
        pprint(forward_condition.serialize())
        return True
    # end ############################################################################

    # begin: attempt to prove each proof obligation using k-induction
    if pd:
        for o in O_prev:
            # begin: check the base case for a proof obligation o
            base_case_o = get_base_case(k, I, T, o)
            if is_sat(base_case_o):
                print(f"[{k=}] Found violation for proof obligation {o}")
                return False
            # end ############################################################################

            else:
                # no violation was found

                # begin: check the inductive-step case to prove o
```

```
                    step_case_o_n = get_step_case(k, T, o)
                    ExternalInv = get_currently_known_invariant()
                    Inv = Predicate(InternalInv & ExternalInv)
                    if m := get_model(Inv[0] & step_case_o_n):
                        s_o = Predicate(get_assignment_as_formula_from_model(m))
                        predicate_describing_set_of_CTI_states = lift(k, Inv, P, s_o, T)
                        if predicate_describing_set_of_CTI_states:
                            O = O.union(Not(predicate_describing_set_of_CTI_states))
                    else:
                        InternalInv &= strengthen(k, Inv, o)
                    # end ########################################################
        # end: attempt to prove each proof obligation using k-induction

        # begin: check the inductive-step case for the safety property P
        step_case_n = get_step_case(k, T, P)
        ExternalInv = get_currently_known_invariant()
        Inv = Predicate(InternalInv & ExternalInv)
        if m := get_model(Inv[0] & step_case_n):
            if pd:
                s = get_assignment_as_formula_from_model(m)
                # Try to lift this state to a more abstract state that still satisfies
                # the property that all of its successors violate the safety property.
                if abstract_state := lift(k, Inv, P, Predicate(s), T):
                    O = O.union(Not(abstract_state))
        else:
            print(f"[{k=}] Proved correctness: safety property is inductive")
            return True
        # end ############################################################################

        k = inc(k)
    print("Property's status is unknown: exceeded maximum number of iterations")
```

Excerto de Código 3.4: Implementação em PySMT do algoritmo de PDR retirado de [**?**]

A versão original com comentários encontra-se no anexo em **??**. Como no caso do IMC, o PySMT facilita a prototipagem de funções a partir de pseudocódigo de forma quase imediata.

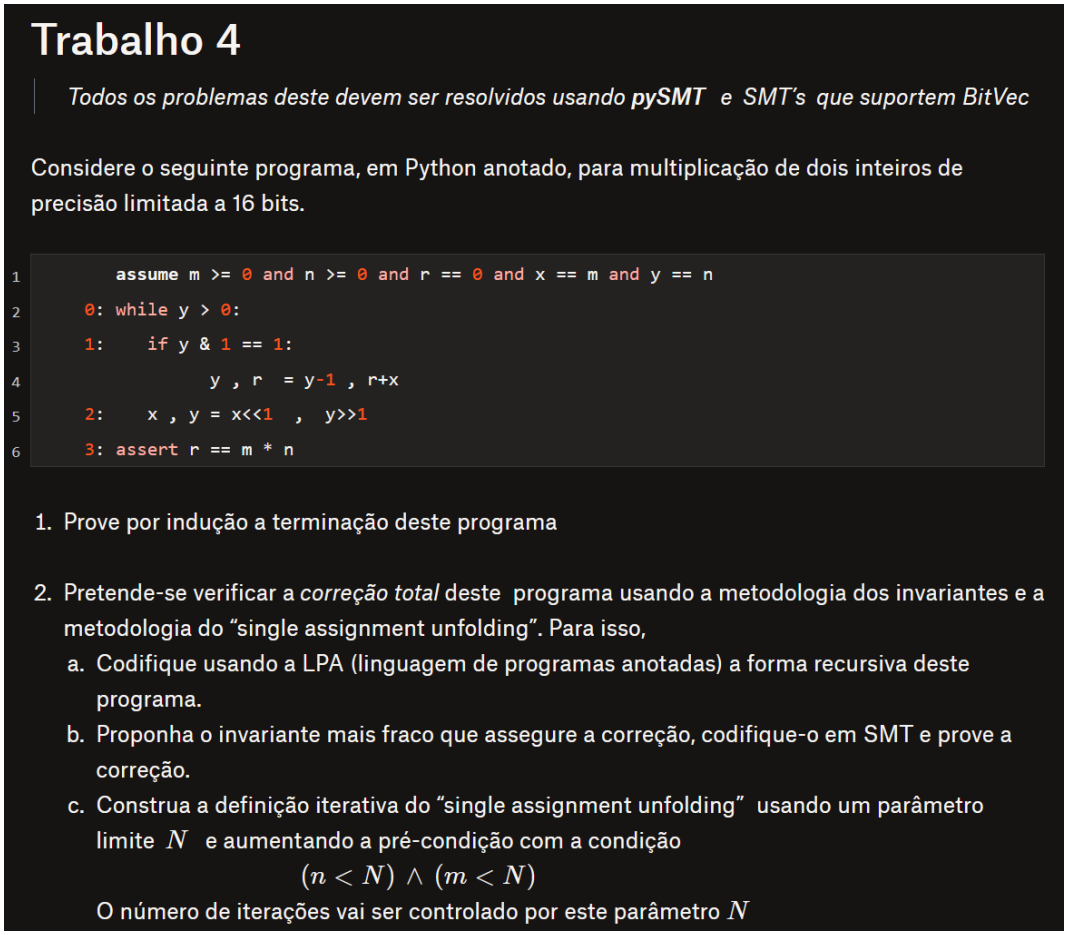
Notes sobre a implementação do PDR

Veja-se que no pseudocódigo em **??**, a função para o PDR recebe transformadores de predicados como argumentos: $lift, strengthen$. Devido a limitações de tempo, não se pôde considerar os tranformadores descritos em [**?**], pelo que usa-se a identidade para *strengthen* por omissão.

Capítulo 4

Caso de Estudo

Para testar a ferramenta desenvolvida, escolheu-se um exemplo de um programa sem grande complexidade, mas não trivial, de um "*First-Order Transition System*" visto num dos trabalhos práticos da UC de Lógica Computacional.



Trabalho 4

> *Todos os problemas deste devem ser resolvidos usando **pySMT** e SMT's que suportem BitVec*

Considere o seguinte programa, em Python anotado, para multiplicação de dois inteiros de precisão limitada a 16 bits.

```
1       assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
2   0: while y > 0:
3   1:    if y & 1 == 1:
4                 y , r  = y-1 , r+x
5   2:    x , y = x<<1  ,  y>>1
6   3: assert r == m * n
```

1. Prove por indução a terminação deste programa

2. Pretende-se verificar a *correção total* deste programa usando a metodologia dos invariantes e a metodologia do "single assignment unfolding". Para isso,
   a. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.
   b. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.
   c. Construa a definição iterativa do "single assignment unfolding" usando um parâmetro limite N e aumentando a pré-condição com a condição
   $$(n < N) \wedge (m < N)$$
   O número de iterações vai ser controlado por este parâmetro N

Figura 4.1: FOTS de trabalho 4 de Lógica Computacional, ano letivo 2021/2022

Na documentação referenciada no anexo **??** apresenta-se um FOTS para o programa em **??**, e mostra-se um exemplo de utilização da ferramenta para provar algumas propriedades sobre ele.

Capítulo 5

Conclusão

Conclui-se desta forma a apresentação do trabalho desenvolvido pelos autores para a UC *Projeto* no ano letivo 2021/2022.

5.1 Comentários

Os autores concluem, após terminar o projeto, que as capacidades oferecidas pelo PySMT são boas, e que é uma ferramenta capaz de implementar conceitos complexos sem grande dificuldade.

No entanto, a documentação oferecida pelo projeto é esparsa, e quando existe é muitas vezes insuficiente. Fruto de ser um projeto "*open-source*", é alvo de constantes mudanças, o que tornam a biblioteca inapropriada para projetos sérios que vão além de prototipagem. Veja-se por exemplo o erro "silencioso" que se obtinha em **??** fruto de permitirà biblioteca deduzir a escolha da lógica SMT

Referenciada em várias das obras consultadas para este projeto ([**?**, **?**, **?**], a ferramenta CPACheckerseria uma alternativa a considerar para trabalhos futuros deste género.

5.2 Trabalho Futuro

Por ser um projeto académico a nível de licenciatura, é improvável que veja qualquer uso futuro adicional.

No entanto, há várias possibilidades para o melhorar:

- Mais importante que todas outras, referenciada em **??** —corrigir o problema com a implementação do IMC em PySMT, que leva o algoritmo a dar, por vezes, resultados incorretos.
- Implementar a ideia de IMC através de WPC+SPC descrita em **??**
  - Esta própria ideia poderia ser melhorada ao implementar um algoritmo dinâmico de escolha de metodologia de prova IMC para usar a técnica adequada para um dado FOTS
  - Testes comparativos ao desempenho das duas técnicas também seriam esclarecedores
- Como mencionado em **??**, teria sido positivo terminar a implementação do PDR semelhante àquela que o CTIGAR usa, descrita em [**?**], assim como comparar o desempenho das duas técnicas
- Teria também sido relevante estudar o efeito de transformadores descritos em **??** de predicados adicionais no comportamento do PDR.

Apêndice A

Excertos de Código Utilizado no Projeto

A.1 "k-*induction*" combinada com "*Bounded Model-Checking*"

```
def get_simple_path(I, T, P, k):
    """Simple path constraint for k-induction:
    each time encodes a different state
    """
    variables = get_variables(I, T, P)
    res = []
    for i in range(k + 1):
        for j in range(i + 1, k + 1):
            state = []
            for v in variables:
                state.append(Not(EqualsOrIff(v[i], v[j])))
            res.append(Or(state))
    return And(res)


def get_k_induction(I, T, P, k):
    """Returns the K-Induction encoding at step K"""
    return And(T[:k],
               P[:k],
               get_simple_path(I, T, P, k),
               Not(P[k]))


def get_bmc(I, T, P, k):
    """Returns the BMC encoding at step k"""
    return And(I[0], T[:k], Not(P[k]))


def BMC_IND(I, T, P):
    """Interleaves BMC and K-Ind to verify the property."""
```

```
    print(f"Checking property {P[0]}...")
    from ply.cpp import xrange
    for b in xrange(100):
        f = get_bmc(I, T, P, b)
        print(f"   [BMC]    Checking bound {b + 1}...")
        if is_sat(f):
            print(f"--> Bug found at step {b + 1}")
            return

        f = get_k_induction(I, T, P, b)
        print(f"   [K-IND]  Checking bound {b + 1}...")
        if is_unsat(f):
            print("--> The system is safe!")
            return
```

Excerto de Código A.1: Implementação do algoritmo de BMC + "*k-induction*" retirada de documentação de PySMT

A.2 "*Interpolant-Based Model-Checking*"

```
def IMC(S: Predicate,
        T: Predicate,
        P: Predicate,
        interpolator: Callable[[FNode, FNode], FNode] = binary_interpolant,
        print_info: bool = True):
    """
    Interpolating Model Checking

    As specified at S. Fulvio Rollini, \Craig Interpolation and Proof Manipulation:
    Theory and Applications to Model Checking," Università della Svizzera Italiana. p.
    38. available at https://verify.inf.usi.ch/sites/default/files/RolliniPhDThesis.pdf

    A property to be verified is encoded as a formula :math:`P` , so that the system is
    safe if the error states where :math:`¬P` holds are not reachable from :math:`S`.

    Verifying that the system satisfies :math:`P` reduces to prove
    that :math:`P` is an inductive invariant property:

    .. math:: S  P\\qquad P  T  P'

    If (i) the initial states satisfy :math:`P` and, (ii) assuming :math:`P` holds,
    it also holds after applying the transition relation, then :math:`P` holds in all
    reachable states. When the inductiveness of :math:`P` cannot be directly proved,
    it might be possible to show that another formula :math:`\\hat P`, stronger than
    :math:`P ( \\hat P  P )`, is an inductive invariant, from which :math:`P` would
    follow as a consequence; this algorithm, which combines interpolation and bounded
    model checking (BMC), is based on iteratively building such a :math:`\\hat P`.
```

```
"""

# first makes sure P is not violated by S
print("Checking if initial states violates safety property")
if m := get_model(S[0] & ~P[0]):
    # halt return a counterexample
    if print_info:
        print(f"[step 0] Initial state violates property:")
        print(f"{INDENT}Counterexample:")
        print(textwrap.indent(f"{m}", INDENT))
    return Status.UNSAFE1

# bound
k = 2

# overapproximation of states at distance at most i from S
i = 0
R = S[0]

# for a bound k and a current overapproximation R(i) of the states at distance at
# most i from S, the algorithm checks if P is violated by the states reachable
# from R(i) in at most k steps.
while True:
    A = R & T[0]
    B = T[1:k - 1] & Or(~P[l] for l in range(k + 1))
    print(f"[{i=},{k=}] Checking BMC from R(i)")
    if m := get_model(A & B):
        # the error might be real or spurious, caused by an insufficient value of k
        if is_valid(EqualsOrIff(R, S[0])):
            print(f"[{i=},{k=}] Checking if R=S")
            # error is real so the system is unsafe
            print(m)
            return Status.UNSAFE2
        else:
            # error is spurious so k is increased to allow finer
            # overapproximations, and the algorithm restarts from S.
            print(f"[{i=},{k=}] R != S")
            k += 1
            i = 0
            R = S[0]
    # R(i) _{j=0}^{k1} T^j _{l=0}^k ¬P^l is unsat
    else:
        # an interpolant I(i) is computed, which represents an approximation of the
        # image of R(i) (i.e., of the states reachable from R(i) in one step).
        print(f"[{i=},{k=}] Calculating interpolant")
        I = interpolator(A, B)

        # a fixpoint check is carried out: if I(i) |= R(i), it means that all
```

```
            # states have been covered, and the system is safe; otherwise, R(i + 1) is
            # set to R(i)  I(i) and the procedure continues.
            if is_valid(I.Implies(R)):
                # the current R(i) corresponds to an inductive invariant P stronger
                # than P: on one side, S |= R(i), moreover R(i)  T |= I'(i) and I(i)
                # |= R(i) imply R(i)  T |= R'(i); on the other side, the fact that at
                # each iteration 0  h  i, R(h)  _{j=0}^{k1} T |= _{l=0}^k P^l,
                # together with R(i) being an inductive invariant, yield R(i) |= P.
                if print_info:
                    # print(f"R({i}) = ", R.simplify().serialize())
                    print(f"[{i=},{k=}] Proved safety: all states have been covered, "
                          f"and the system is safe")
                return Status.SAFE
            else:
                print(f"[{i=},{k=}] I !=> R")
                R |= I
                i += 1
```

Excerto de Código A.2: Implementação comentada do algoritmo de IMC -[?]

A.3 "*Property Directed Reachability*"

```
def PDR(I: Predicate,
        T: Predicate,
        P: Predicate,
        get_currently_known_invariant=lambda: TRUE(),
        strengthen=lambda k, Inv, o: Inv,
        lift=_lift,
        k_init: int = 1,
        k_max: int = float('inf'),
        pd: bool = True,
        inc: Callable[[int], int] = lambda n: n + 1,
        print_info=True
        ) -> bool | Status:
    """
    Iterative-Deepening k-Induction with Property Direction.

    As specified at D. Beyer and M. Dangl, \Software Verification with PDR:
    Implementation and Empirical Evaluation of the State of the Art" arXiv\:1908.06271
    [cs], Feb. 2020, Accessed: Mar. 05, 2022. [Online]. Available
    https://arxiv.org/abs/1908.06271.

    :param print_info: Whether info about the steps should be printed.
    :param k_init: the initial value :math:`1` for the bound `k`
    :param k_max: an upper limit for the bound `k`
    :param inc: a function :math:` → ` such that :math:`n  : \\inc(n) > n`
    :param TS: Contains predicates defining the initial states and the transfer relation
```

```
    :param P: The safety property
    :param get_currently_known_invariant: used to obtain the strongest invariant currently
        available via a concurrently running (external) auxiliary-invariant generator
    :param pd: boolean flag pd (reminding of \property-directed") is used to control
        whether failed induction checks are used to guide the algorithm towards a
        sufficient strengthening of the safety property P to prove correctness; if pd is
        set to false, the algorithm behaves exactly like standard k-induction.
    :param lift: Given a failed attempt to prove some candidate invariant :math:`Q` by
        induction, the function lift is used to obtain from a concrete
        counterexample-to-induction (CTI) state a set of CTI states described by a state
        predicate C. An implementation of the function :math:`k  , \\Inv   × (S →
        ) × (S → ) × S → (S → )` and :math:`C = \\lift(k, \\Inv , Q, s)`, lift needs to
        satisfy the condition that for a CTI :math:`s  S` where :math:`S` is the set of
        program states, the following holds:

        .. math:: C(s)  \\left( s_n  S: C(s_n) \
            \\Inv(s_n)
            \\displaystyle\\bigwedge_{i=n}^{n+k1} (Q(s_i)  T(s_i ,s_{i+1}))  ¬Q(s_{n+k}) \\r

        which means that the CTI s must be an element of the set of states described by
        the resulting predicate C and that all states in this set must be CTIs, i.e.,
        they need to be k-predecessors of :math:`¬Q`-states, or in other words,
        each state in the set of states described by the predicate :math:`C` must reach
        some :math:`¬Q`-state via :math:`k` unrollings of the transition relation
        :math:`T`.
    :param strengthen: The function strengthen: :math:` × (S → ) × (S → ) → (S →
        )` is used to obtain for a k-inductive invariant a stronger k-inductive
        invariant, i.e., its result needs to imply the input invariant, and, just like the
        input invariant, it must not be violated within k loop iterations and must be
        k-inductive.
    :return: `True` if `P` holds, `Status.UNKNOWN` if `k > k_max` , `False` otherwise.
    """

    # current bound
    k: int = k_init

    # the invariant computed by this algorithm internally
    InternalInv: FNode = TRUE()

    # the set of current proof obligations.
    O: Set[Predicate] = set()

    while k <= k_max:
        O_prev: Set[Predicate] = O
        O = set()

        # begin: base-case check (BMC)
        #
```

```
# Base Case. The base case of k-induction consists of running BMC with the
# current bound k. This means that starting from all initial program states, all
# states of the program reachable within at most k1 unwindings of the transition
# relation are explored. If a ¬P-state is found, the algorithm terminates.
base_case = get_base_case(k, I, T, P)
if m := get_model(base_case):
    if print_info:
        print(f"[{k=}] base-case check failed")
        print(f"{INDENT}Counterexample:")
        print(textwrap.indent(f"{str_model(m)}", INDENT))
        # print(textwrap.indent(f"{m}", INDENT))
    return False
# end ############################################################################

# begin: forward-condition check (as described in Sec. 2)
#
# Forward Condition. If no ¬P-state is found by the BMC in the base case, the
# algorithm continues by performing the forward-condition check, which attempts
# to prove that BMC fully explored the state space of the program by checking
# that no state with distance k > k1 to the initial state is reachable. If this
# check is successful, the algorithm terminates.
forward_condition = I[0] & T[:k - 1]
if is_unsat(forward_condition):
    print(f"[{k=}] Proved correctness: successful forward condition check")
    pprint(forward_condition.serialize())
    return True
# end ############################################################################

# begin: attempt to prove each proof obligation using k-induction
if pd:
    for o in O_prev:
        # begin: check the base case for a proof obligation o
        base_case_o = get_base_case(k, I, T, o)
        if is_sat(base_case_o):
            # If any violations of the proof obligation o are found, this means
            # that a predecessor state of a ¬P-state, and thus, transitively,
            # a ¬P -state, is reachable, so we return false.
            print(f"[{k=}] Found violation for proof obligation {o}")
            return False
        # end ########################################################################

        else:
            # no violation was found

            # begin: check the inductive-step case to prove o
            #
            # Inductive-Step Case. The forward-condition check, however,
            # can only prove safety for programs with finite (and, in practice
```

```
                # short) loops. To prove safety beyond the bound k, the algorithm
                # applies induction: The inductive-step case attempts to prove tha
                # after every sequence of k unrollings of the transition relation
                # that did not reach a ¬P-state, there can also be no subsequent
                # transition into a ¬P-state by unwinding the transition relation
                # once more. In the realm of model checking of software, however,
                # the safety property P is often not directly k-inductive for any
                # value of k, thus causing the inductive-step-case check to fail.
                # It is therefore state-of-the-art practice to add auxiliary
                # invariants to this check to further strengthen the induction
                # hypothesis and make it more likely to succeed. Thus,
                # the inductive-step case proves a program safe if the following
                # condition is unsatisfiable:
                #
                #   Inv(s_n) _{i=n}^{n+k-1}(P(s_i)  T(s_i,s_{i+1}))  ¬P(s_{n+k})
                #
                # where Inv is an auxiliary invariant, and s,...,s is any
                # sequence of states. If this check fails, the induction attempt is
                # inconclusive, and the program is neither proved safe nor unsafe
                # yet with the current value of k and the given auxiliary
                # invariant. In this case, the algorithm increases the value of k
                # and starts over.
                step_case_o_n = get_step_case(k, T, o)
                ExternalInv = get_currently_known_invariant()
                Inv = Predicate(InternalInv & ExternalInv)
                if m := get_model(Inv[0] & step_case_o_n):
                    s_o = Predicate(get_assignment_as_formula_from_model(m))
                    predicate_describing_set_of_CTI_states = lift(k, Inv, P, s_o, T)
                    if predicate_describing_set_of_CTI_states:
                        O = O.union(Not(predicate_describing_set_of_CTI_states))
                else:
                    # If the step-case check for o is successful,
                    # we no longer track o in the set O of unproven proof obligations.

                    # We could now directly use the proof obligation as an
                    # invariant, but instead, we first try to strengthen it into a
                    # stronger invariant that removes even more unreachable states
                    # from future consideration before conjoining it to our
                    # internally computed auxiliary invariant. In our
                    # implementation, we implement strengthen by attempting to drop
                    # components from a (disjunctive) invariant and checking if the
                    # remaining clause is still inductive.
                    InternalInv &= strengthen(k, Inv, o)
                # end ########################################################
    # end: attempt to prove each proof obligation using k-induction

    # begin: check the inductive-step case for the safety property P
    #
```

```
        # This check is mostly analogous to the inductive-step case check for the proof
        # obligations described above, except that if the check is successful,
        # we immediately return true.

        # Assume for any iteration n (k iterations from n to n + k  1 = n) that the
        # safety property holds, and from this assumption attempt to conclude that the
        # safety property will also hold in the next iteration n + 1 (n + k).
        step_case_n = get_step_case(k, T, P)
        ExternalInv = get_currently_known_invariant()
        Inv = Predicate(InternalInv & ExternalInv)
        if m := get_model(Inv[0] & step_case_n):
            if pd:
                s = get_assignment_as_formula_from_model(m)
                # Try to lift this state to a more abstract state that still satisfies
                # the property that all of its successors violate the safety property.
                if abstract_state := lift(k, Inv, P, Predicate(s), T):
                    # Negate this abstract state to obtain the proof obligation.
                    # This means that we have learned that we should prove the
                    # invariant ¬o, such that in future induction checks, we can remove
                    # all states that satisfy `o` from the set of predecessor states
                    # that need to be considered.
                    O = O.union(Not(abstract_state))
        else:
            print(f"[{k=}] Proved correctness: safety property is inductive")
            return True
        # end ##########################################################################

        k = inc(k)
    print("Property's status is unknown: exceeded maximum number of iterations")
    return Status.UNKNOWN
```

Excerto de Código A.3: Implementação comentada do algoritmo de PDR -[?]

Apêndice B

Repositório *GitHub* com código fonte e documentação

"*Source code*"

O código fonte da ferramenta desenvolvida é accessível através do link: `https://github.com/Alef-Keuffer/FOTS-Prover`.

Está sediado numa página de GitHub de um dos autores.

Implementações incompletas

Como referido em , as implementações incompletas do IMC através de WPC/SPC, e do PDR baseasdo em CTIGAR de [**?**] estão em: `https://github.com/Alef-Keuffer/FOTS-Prover/tree/main/src/unfinished`.

Documentação da ferramenta

A documentação da ferramenta encontra-se disponível aqui.

Documentação de exemplo

A documentação do exemplo em **??** está aqui.