Automatic Device Driver Synthesis

Adam Walker

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

May 2015

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Date .....

Abstract

Automatic device driver synthesis is a radical approach to creating drivers faster and with fewer defects by generating them automatically based on hardware device specifications. This thesis presents the design and implementation of a device driver synthesis toolkit called Termite. Termite is the first tool to combine the power of automation with the flexibility of conventional development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications.

At the core of Termite is a scalable synthesis engine, which enables it to handle real-world device specifications that are beyond the reach of existing synthesis techniques. This thesis presents the predicate-based abstraction refinement algorithm that this synthesis engine is based on. We develop solutions to the key problems involved in the implementation of efficient predicate abstraction which have not been addressed previously in a synthesis settings: (1) keeping abstractions concise by identifying relevant predicates only, (2) solving abstract games efficiently, and (3) computing and solving abstractions symbolically.

Lastly, we demonstrate the practicality of Termite by synthesizing drivers for a number of I/O devices representative of a typical embedded platform.

For Lord Xenu, ruler of the Galactic Confederacy.

Contents

List of Figures

List of Tables

Publications

- Leonid Ryzhyk, Adam Christopher Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. pages 661–676, Broomfield, CO, USA, October 2014
- Adam Christopher Walker and Leonid Ryzhyk. Predicate abstraction for reactive synthesis. Lausanne, Switzerland, October 2014

1 Introduction

Device driver synthesis has been proposed as a radical alternative to traditional driver development that offers the promise of creating drivers faster and with far fewer defects [Ryzhyk et al., 2009b]. The idea is to automatically generate the driver code responsible for controlling device operations from a behavioral model of the device and a specification of the driver-OS interface.

The primary motivations for device driver synthesis are the fact that device drivers are hard and tedious to write, and they are notorious for being unreliable [**??**]. Drivers generally take a long time to bring to production—given the speed at which new devices can be brought to market today, it is not uncommon for a device release to be delayed by driver rather than by silicon issues [Yavatkar, 2012].

Automatic driver synthesis was proposed in earlier work on the Termite-1 project [Ryzhyk et al., 2009b], which formulated the key principles behind the approach and demonstrated its feasibility by synthesizing drivers for several real-world devices. The next logical step was to develop driver synthesis into a practical methodology, capable of replacing the conventional driver development process. To this end we have addressed the key problems left open by Termite-1:

- The scalability of the Termite-1 synthesis algorithm was severely limited, which made synthesis of drivers for real-world devices intractable. Termite-1 got around the problem by using carefully crafted simplified device specifications, which is acceptable in a proof-of-concept prototype, but not in a practical tool. Addressing this issue is the major contribution of this dissertation.

- Termite-1 produced poor quality synthesized code. While functionally correct, Termite-1 drivers were bloated and poorly structured. This made it impossible for a programmer to maintain and improve the generated code and prevented synthesized drivers from being adopted by Linux and other major operating systems. Furthermore, it was impossible to enforce non-functional properties such as CPU and power efficiency.

1.1 Scalable Synthesis Algorithm

To address the scalability problem, we created a new scalable synthesis algorithm, which mitigates the computational bottleneck in driver synthesis. Following the approach proposed in Termite-1, we treat the driver synthesis problem as a two-player game between the driver and its environment, comprised of the device and the OS. In this work, we develop this approach into the first precise mathematical formulation of the driver synthesis problem based on game theory. This enables us to apply theoretical results and algorithmic techniques from game theory to driver synthesis.

Our game-based synthesis algorithm relies on abstraction and symbolic reasoning to achieve orders of magnitude speed up compared to the current state-of-the-art synthesis techniques. The algorithm is described in Chapter 5.

Game solving involves exploring potentially large state spaces. *Abstraction* offers an effective approach to mitigating the state space explosion problem. For example, in the model checking domain abstraction proved instrumental in enabling automatic verification of complex hardware and software systems [Clarke et al., 2000; **?**; Henzinger et al., 2002]. The reactive synthesis community has also identified the key role of abstraction in tackling real-world synthesis problems; however most research in this area has so far been of theoretical nature [de Alfaro and Roy, 2007; **?**].

In this dissertation I present the first practical abstraction-refinement algorithm for solving games. Our algorithm is based on *predicate abstraction*, which proved to be particularly successful in model checking [Graf and Saïdi, 1997]. Predicate abstraction partitions the state space of the game based on a set of predicates, which capture essential properties of the system. States inside a partition are indistinguishable to the abstraction, which limits the maximal precision of solving the game achievable within the given abstraction. The abstraction is iteratively refined by introducing new predicates.

The key difficulty in applying predicate abstraction to games is to efficiently solve the abstract game arising at every iteration of the abstraction refinement loop. This requires computing the abstract *controllable predecessor* operator, which maps a set of abstract states, winning for one of the players, into the set of states from which the player can force the game into the winning set in one round of the game. This involves enumerating concrete moves available to both players in each abstract state, which can be prohibitively expensive.

We address the problem by further approximating the expensive controllable predecessor computation and refining the approximation when necessary. To this end, we introduce additional predicates that partition the set of actions available to the players into *abstract actions*. The controllable predecessor computation then consists of two steps: (1) computing abstract actions available in each abstract state, and (2) evaluating controllable predecessor over abstract states and actions.

The first step involves potentially expensive analysis of concrete transitions of the system and is therefore computed approximately. More specifically, solving the abstract game requires overapproximating moves available to one of the players, while underapproximating moves available to the other [**?**]. The former is achieved by allowing an abstract action in an abstract state if it is available in at least one corresponding concrete state, the latter allows an action only if it is available in all corresponding concrete states. We compute the overapproximation by initially allowing all actions in all states and gradually refining the abstraction by eliminating spurious actions. Conversely, we start with an empty underapproximation and add available actions as necessary.

This dissertation makes three contributions in the field of reactive synthesis:

1. We propose the first practical predicate-based abstraction refinement algorithm for two-player games.

2. We introduce a new type of refinement, which increases the precision of controllable predecessor computation without refining the abstract state space of the game. This approach avoids costly operations involved in solving the abstract game, approximating them with a sequence of light-weight operations performed on demand, leading to dramatically improved scalability.

3. We evaluate the algorithm by implementing it as part of the Termite driver synthesis toolkit [Ryzhyk et al., 2014] and using it to synthesise drivers for complex real-world devices. Our algorithm efficiently solves games with very large state spaces, which is impossible without using abstraction or using simpler forms of abstraction.

1.2 User Guided Synthesis

Despite substantially improving the scalability of the synthesis algorithm during this work, we came to the conclusion that the approach taken by Termite-1 was *critically flawed*. The fundamental problem, in our view, was that the synthesis was viewed as a "push-button" technology that generated a specification-compliant implementation without any user involvement. As a result, the user had to rely on the synthesis tool to produce a good implementation. Unfortunately, even the most intelligent algorithm cannot fully capture the user-perceived notion of high-quality code. While in theory one might be able to enforce some of the desired properties by adding appropriate constraints to the input specification, in our experience creating such specifications is extremely hard and seldom yields satisfactory results.

A radically different approach was needed—one that combines the power of automation with the flexibility of conventional development, and that involves the developer from the start, guiding the generation of the driver. In many ways, synthesis and conventional development are conflicting. Hence, a key challenge was to conceive of a way that allowed the two to be combined so that the developer could do their job more efficiently and with fewer errors without having the synthesis tool get in the way.

This dissertation presents a novel *user-guided* approach to driver synthesis implemented in our new tool called Termite-2 (further referred to as Termite). In Termite, the user has full control over the synthesis process, while the tool acts as an assistant that suggests, but does not enforce, implementation options and ensures correctness of the resulting code. At any point during synthesis the user can modify or extend previously synthesized code. The tool automatically analyses user-provided code and, on user's request, suggests possible ways to extend it to a complete implementation. If such an extension is not possible due to an error in the user code, the tool generates an explanation of the failure that helps the user to identify and correct the error.

In an extreme scenario, Termite can be used to synthesize the complete implementation fully automatically. At the other extreme, the user can build the complete implementation by hand, in which case Termite acts as a static verifier for the driver. In practice, we found the intermediate approach, where most of the code is auto-generated, but manual involvement is used when needed to improve the implementation, to be the most practical.

From the developer's perspective, user-guided synthesis appears as an enhancement of the conventional development process with very powerful autocomplete functionality, rather than a completely new development methodology. This vision is implemented in all aspects of the design of Termite. In particular, input specifications for driver synthesis are written as imperative programs that model the behavior of the device and the OS. The driver itself is modelled as a source code template where parts to be synthesized are omitted. This approach enables the use of familiar programming techniques in building input specifications. In contrast, previous synthesis tools, including Termite-1, require specifications to be written in formal languages based on state machines and temporal logic, which proved difficult and error-prone to use even for formal methods experts, not to mention software development practitioners.

Most previous research on automatic synthesis, including Termite-1, considered input specifications to be "correct by definition". In contrast, we recognise that input specifications produced by human developers are likely to contain defects, which can prevent the synthesis algorithm from finding a correct driver implementation. Therefore Termite incorporates powerful debugging tools that help the developer identify and fix specification defects through well-defined steps, similar to how conventional debuggers help troubleshoot implementation errors.

We evaluate Termite by synthesizing drivers for several I/O devices. Our experience demonstrates that our methodology meets our design goals, and indeed makes automatic driver synthesis practical.

This dissertation makes two contributions in the field of practical device driver synthesis:

1. A practical synthesis tool that allows the user to work anywhere on the spectrum from full automation to verified manual development.

2. A counterexample guided debugging environment that the developer

may use to interactively eliminate defects in the input specifications.

1.3 Chapter outline

The rest of this dissertation is structured as follows. Chapter 2 provides the necessary background information on game solving. Chapter 4 presents our user guided device driver synthesis tool and methodology and, finally, Chapter 5 presents our scalable synthesis algorithm.

2 Background

2.1 Device Drivers

A device driver provides an interface for the operating system (OS) to use a hardware device that is attached to the computer. It translates requests between the OS and the hardware. It allows the operating system to use the device without knowing the exact details of the hardware. As an example, a network interface driver may provide functions to send and receive packets from the network. It performs the low level device register and memory writes necessary to perform these operations while presenting a high level interface to the OS.

Writing a device driver requires in depth knowledge of both the device to be controlled and the interface to the OS. This makes writing a device driver a tedious and error prone process. Additionally, device drivers usually operate in a privileged part of the OS so that it can access the hardware. As a result, errors often lead to failures of the system.

Device drivers are usually developed by the hardware vendor as they have the best access to information about the design of the hardware. This includes hardware documentation as well as the actual description of the hardware in a hardware description language. If is common for the documentation to never be released to the public.

Device driver functions

The key functions of a device driver are:

- *Abstraction.* As discussed, the driver abstracts away the low level details of the device and presents the functionality through a high level set of

operations.

- *Unification.* The driver provides a unified interface to similar devices. For example, another network interface may have present a completely different hardware interface to the first but its driver presents the same send and receive functionality to the OS as before.

- *Multiplexing.* The driver and OS cooperate to make sure that the services that the hardware provides are available to all users of the system with sufficient permissions to access the hardware.

Device driver organisation

I/O buses

Device drivers communicate with devices over an I/O bus such as the *Peripheral Component Interconnect* (PCI) or *Universal Serial Bus* (USB). Modern operating systems contain drivers that operate these buses and provide low level access to devices over them. In this thesis we assume that these bus level drivers already exist.

Memory Mapped I/O

Device registers and data buffers are mapped into the system's memory space by the bus controller. When the CPU initiates a load or store transaction which falls in the range of memory mapped to the device the PCI controller translates this to a transaction with the device. This transaction reads and writes the device's internal registers.

Direct Memory Access

It is also possible for the device to initiate a transaction to transfer data to or from main memory. This is known as *Direct Memory Access* or DMA. The CPU is not involved at all. DMA is necessary for efficient I/O with devices that transfer large amounts of data as otherwise the CPU would be required to copy all of the data to and from the device word by word.

Interrupts

Interrupts are asynchronous transactions initiated by the device to signal to the CPU that an event has happened that requires the driver's attention. For example, a network interface may raise an interrupt when a packet has arrived or when a packet has been successfully been sent. Interrupts are also often raised on error conditions within the device.

On a Linux system, when an interrupt is raised, the currently executing task is temporarily suspended and control is transferred to the kernel's interrupt handler routine. This determines which driver is responsible for handling the interrupt and invokes the driver's interrupt handler.

Concurrency

Most operating systems are multithreaded, including the device drivers. Driver entry points and interrupt handlers may be invoked concurrently and in a nested fashion. Careful use of synchronization primitives are required to avoid race conditions and deadlock. In this work, we assume that only one driver entry point is invoked at a time, removing all concurrency. This can be achieved by using an event based framework such as Dingo [Ryzhyk et al., 2009a], writing a wrapper to translate the concurrent interface required by the operating system to the single threaded interface of the generated drivers, or by post processing the generated driver as described by [Cerny et al., 2013]

Operating system services

In addition to bus level drivers, the OS provides various services to device drivers. These include memory management, timing and synchronization. Memory management includes allocating memory and support for memory mapped IO and DMA. Timer functionality includes both synchronous (thread delays) and asynchronous (callback) timers. Various synchronization primitives are also provided, but, given that Termite does not make any use of them, we do not discuss them further.

2.2 Fixed points

Syntax

We define a logic with fixed points capable of expressing greatest and least solutions of fixed point equations $X = f(X)$ where f is a monotone function. The set of formulas in this logic is defined as follows:

Let P be a set of propositions and V be a set of variables. Then

- each proposition $p \in P$ is a formula
- if ϕ and ψ are formulas then $\phi \wedge \psi$ is a formula
- if ϕ is a formula then $\neg\phi$ is a formula
- if ϕ is a formula and Z a variable then $\nu Z.\phi$ is a formula provided that every free occurrence of Z in ϕ occurs under an even number of negations
- if ϕ is a formula and Z is a variable then $\forall Z.\phi$ is a formula

Given these definitions, we also have:

- $\phi \vee \psi$ meaning $\neg(\phi \wedge \psi)$
- $\mu Z.\phi$ meaning $\neg\nu Z.\neg\phi[Z := \neg Z]$ where $\phi[Z := \neg Z]$ means substituting $\neg Z$ for all free occurrences of Z in ϕ
- $\exists Z.\phi$ meaning $\neg\forall Z.\neg\phi$

Semantics

Given the tuple $\langle S, F\rangle$ where

- S is the set of states
- $F : P \to 2^S$ maps each proposition to the set of states where the proposition is true

A fixed point formula is interpreted as follows:

- p holds in the set of states $F(p)$

- $\phi \wedge \psi$ holds in the set of states where both ϕ and ψ hold
- $\neg\phi$ holds in every state where ϕ does not hold
- $\nu Z.\phi$ holds in any set of states T such that when the variable Z is set to T in ϕ then ϕ holds for all of T. It is the greatest fixpoint of ϕ.
- $\forall X.\phi$ holds when ϕ holds for all possible values of X.

From fixed point formulas to algorithms

It is straightforward to convert a fixed point formula into an algorithm that returns the set of states for which the formula holds.

Greatest and least fixed points are computed by iteration. To compute the greatest fixed point of a function $f(x)$ i.e. $\nu X.f(X)$ we iterate f starting with the universal set. As f is monotonic, the result is guaranteed to shrink or stay the same on each iteration. Furthermore, if the set of states, S is finite, this iteration must eventually converge to some set as S being finite prevents the result from shrinking forever.

We give the algorithm for computing the set of states for which a fixed point formula holds in Algorithm 1. It assumes the existence of an abstract set datatype that supports set union, complementation and equality checking.

2.3 ω-regular languages

The ω-regular languages generalise the regular languages. An ω-regular language is a set of sequences of symbols of infinite length. An ω-regular language has the form:

- A^ω where A is a nonempty regular language not containing the empty string.
- AB, the concatenation of a regular language A and an ω-regular language B. Note that B must not be a regular language as this would imply that the sequence is finite.
- $A \cup B$ where A and B are ω-regular languages.

Elements of A^ω are obtained by concatenating words from A infinitely many times. Intuitively, ω-regular languages are similar to regular languages

Algorithm 1 MuSemantics, given a fixed point formula, returns the set of states that satisfy the formula.

```
function MuSemantics(φ)
    if φ = p then
        return F(p)
    else if φ = ψ ∨ ρ then
        return MuSemantics(ψ) ∪ MuSemantics(ρ)
    else if φ = ¬ψ then
        return S − MuSemantics(ψ)
    else if φ = ∀X.ψ then
        return {s ∈ S | ∀x. s ∈ MuSemantics(ψ[X := x])}
    else if φ = νX.ψ then
        Z ← S
        loop
            Z' ← MuSemantics(ψ[X := Z])
            if Z' = Z then
                return Z
            end if
            Z ← Z'
        end loop
    end if
end function
```

but with an additional ω operator that concatenates words from the regular language infinitely many times.

2.4 Two player games

Two-player games are a useful formalism for reactive synthesis. Many problems in electronic design automation, industrial automation and robotics can be formalised as a game. In particular, the driver synthesis problem can be formalised as a game, and, this is the formalism around which Termite is built. Here, we present the fundamentals of two player games.

Formalism

A two player game is played by player 1 against its opponent, player 2. It consists of a possibly infinite state space S on which the game is played. The game is always in some state $s \in S$ called the *current state*. The game progresses from state to state according to a transition relation, $\delta \subseteq S \times L \times S$

where S is the set of states and L is a set of label variables. A transition $t \in S \times L \times S$ is allowed in the game iff $t \in \delta$.

The meaning of the label $l \in L$ depends on the type of game, but for now we will consider turn based games. In a turn based game, S is partitioned into two sets: the player 1 set τ_1 and the player 2 set τ_2, where $\tau_1 \cap \tau_2 = \emptyset$ and $\tau_1 \cup \tau_2 = S$. When $s \in \tau_1$ player 1 gets to pick l and when $s \in \tau_2$ player 2 gets to pick l. We refer to the opponent of player i as $\bar{i}$ ($\overline{1} = 2$, $\overline{2} = 1$).

Lastly, each game has an associated set of initial states $I \in 2^S$ where execution of the game begins.

Putting this all together, we can identify a *turn based game structure* $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$ with a turn based game.

A game proceeds in an infinite sequence of rounds, starting from some initial state. The infinite sequence of states visited $(s_0, s_1, \ldots) \in S^\omega$ is called a path. An *objective* $\Phi \subseteq S^\omega$ is a subset of state sequences of G. We are concerned with ω-regular objectives, i.e., objectives characterised by ω-regular languages [**?**].

A *strategy* for player i is a function $\pi_i : S^* \times \tau_i \to L$ that, in any player i state, associates the history of the game with a label to play. The set of initial states I and a player i strategy π_i determines a set $Outcomes_i(I, \pi_i)$ of paths $s_0, s_1, s_2, \ldots$ such that $s_0 \in I$ and $s_{k+1} = \delta(s_k, \pi_i(s_0, \ldots, s_k))$ when $s_k \in \tau_i$ and $s_{k+1} = \delta(s_k, l)$ for some l when $s_k \in \tau_{\bar{i}}$. Given an objective $\Phi \subseteq S^\omega$ we say that state $s \in S$ is winning for player i if there is a strategy π_i such that $Outcomes_i(s, \pi_i) \subseteq \Phi$. That is, if, by picking suitable labels, they can force the path to be within the set of winning sequences. An arbitrary set of infinite sequences is an extremely general, but not practically useful, way of defining an objective. In the following sections we will consider some more restricted objectives that have practical uses.

Safety and reachability

The two simplest objectives are safety and reachability. A safety objective is defined by a set $SAFE \subseteq S$ that player 1 must force the game to stay within, regardless of the labels that player 2 picks. Formally, a run is safe if

$$\forall i.\ s_i \in SAFE \tag{2.1}$$

The dual of a safety objective is a reachability objective. A reachability objective is defined by a set $REACH \subseteq S$ that player 1 must force the game to visit at least once, regardless of the labels that player 2 picks. Formally, a reachability run is winning if

$$\exists i.\ s_i \in REACH \tag{2.2}$$

Büchi A Büchi objective is defined by a set $BUCHI \subseteq S$ that player 1 must always be able to force execution of the game into. This differs from a reachability game in that the region must always be reachable, not just once. When it has been reached once, it must be reachable again, and so on. So, it must be reachable infinitely many times. Formally, a run is Büchi winning if

$$\forall i.\ \exists j > i.\ s_j \in BUCHI \tag{2.3}$$

Generalized Büchi A Generalized Büchi objective is defined by a finite set of sets $BUCHIS \subseteq 2^S$. Player 1 must always be able to force execution into each set $BUCHI \in BUCHIS$. Formally, a run satisfies a generalized Büchi objective if

$$\forall i.\ \forall BUCHI \in BUCHIS.\ \exists j > i.\ s_j \in BUCHI \tag{2.4}$$

Fairness Sometimes it is necessary to rule out invalid plays that are not easily ruled out by changing the state machine. As an example, consider a progress assumption that guarantees that the game will eventually leave a set of 'pending' states. For this we use fairness conditions. A fairness condition is a set of states, the *fair* set, which we may assume that all valid runs of the game eventually enter. Or, equivalently, a set of states, called the *unfair* set which we may assume that all valid runs of the game eventually leave. If a spoiling strategy exists that results in an unfair run, i.e. it does not always, at some point in the future, enter the fair set, then it does not count. Formally, a run satisfies a fair reachability objective if

$$\forall i.\ \exists j > i.\ s_j \in FAIR \rightarrow \exists i.\ s_i \in REACH \tag{2.5}$$

GR(1) A Generalized Reactive 1, or GR(1) [Piterman et al., 2006] objective, is a generalized Büchi objective with multiple fairness conditions. In practice, this turns out to be a very useful type of objective. Formally, given a set $FAIRS$ of fair sets of states, a run satisfies a GR(1) objective if

$$\forall i.\, \forall FAIR \in FAIRS.\, \exists j > i.\, s_j \in FAIR \rightarrow \\ \forall i.\, \forall BUCHI \in BUCHIS.\, \exists j > i.\, s_j \in BUCHI \quad (2.6)$$

Informally, a run is winning in a GR(1) game if for all fair runs the generalised Büchi condition holds.

Perfect information games

Perfect information games are two player games where both players know the current state of the transition system at all times. We will assume that all games are perfect information games.

Strategies and counterexamples

If a game is winnable for player 1 then there exists (by definition) a strategy that tells player 1, in any state, which label it must play and ensures that if player 1 adheres to the strategy then the objective will be satisfied.

It is known that for perfect information safety, reachability and Büchi games, if there exists a strategy, then there also exists one that only depends on the current state [**?**]. Thus, we can simplify the definition of a strategy in this case for player 1 to be a function $\pi_1 : \tau_1 \rightarrow L$ that only depends on the current state.

Furthermore, it is known that for generalised Büchi and GR(1) games, if there exists a strategy, then there also exists one that only depends on the current state and some finite additional state [**?**]. We can simplify the definition of a strategy in this case for player 1 to be a function $\pi_1 : \tau_1 \times \sigma \rightarrow L \times \sigma$ that only depends on the current state and the finite additional state (σ), and, in addition to returning the label to play, returns the updated additional state.

Perfect information ω-regular games are *determined* ***TODO: this is not true in our case because they are non-deterministic, do we use the simpler deterministic case here and fix this later?***. That is, if the game is losing for

player 1 then the game is winning for player 2, i.e. there exists a strategy for player 2 to violate the objective [**?**].

This is known as a *counterexample strategy*. It is a strategy for player 2 which ensure that player 1 cannot satisfy their objective. Again, in the case of perfect information safety, reachability and Büchi games, if there exists a counterexample strategy (or, equivalently if the game is not winnable for player 1) then there exists a counterexample strategy that only depends on the current state. This is a function $\pi_2 : \tau_2 \to L$ that associates any player 2 state with a label for player 2 to play.

Again, in the case of generalised Büchi and GR(1) games, the counterexample strategy may depend on an additional finite state: $\pi_1 : \tau_2 \times \sigma \to L \times \sigma$.

2.5 Solving games

Given a game and an objective there are two questions we can ask:

- can player 1 win?
- if so, what is the strategy for player 1 and, if not, what is the counterexample strategy for player 2

We present algorithms to answer these questions in this section. We begin with the simplest game, the reachability game, and progress to GR(1) games as are used in Termite.

Controllable predecessor

All of the algorithms for games I will be describing use a function called the *controllable predecessor* (abbreviated CPre). CPre is a function from a target set of game states (2^S) called the *target set* to another set of game states. Given a set T, $CPre(T)$ returns the set of states from which player 1 can force execution into T in one step.

The exact details of CPre depend on the type of game being solved. Here we consider it to be a parameter to the game solving algorithms. The only property of $CPre$ that the game solving algorithms require is that it is monotonic, i.e. if $X \subseteq Y$ then $CPre(X) \subseteq CPre(Y)$. Clearly, any reasonable CPre function will have this property.

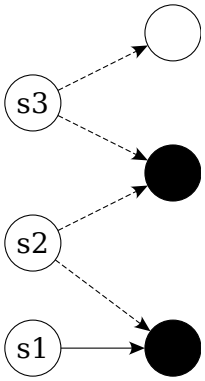


Figure 2.1: Turn based controllable predecessor

In the following sections we describe the controllable predecessor for turn based games as well as Termite's games.

Turn based controllable predecessor

$$CPre(X) = \tau_1 \cap \{x | \exists l. \langle x, l, x' \rangle \in \delta \rightarrow x' \in X\} \\ \cup \tau_2 \cap \{x | \forall l. \langle x, l, x' \rangle \in \delta \rightarrow x' \in X\} \quad (2.7)$$

The controllable predecessor for turn based games, given by Equation 2.7, returns the subset of τ_1 where there exists a label which player 1 can choose to take execution into X together with the subset of τ_2 where all labels which player 2 can choose take execution into X.

This is illustrated in Figure 2.1. The two dark states are winning and they form the set X passed to $CPre$. $CPre$ returns state $s1$ in the winning set because it is a controllable state and there exists a transition into a winning state. State $s2$ is also winning because it is an uncontrollable state and all outgoing transitions go to winning states. Lastly, $s3$ is *not* winning because it is uncontrollable and one of its outgoing transitions does not lead to a winning state.

Termite's controllable predecessor

Concurrent Games Games where there are states in which both player 1 and player 2 have moves are called *concurrent games*. Concurrent games differ from simpler turn based games in that in each state both players get to pick

a label and the next state that the game transitions to is some (possibly non-deterministic) function, λ, of both of those labels. Turn based games are a special case of concurrent games where in player i states (a concept that does not generally exist in concurrent games) λ only depends on the label played by player i and the other player's label is ignored.

Termite's controllable predecessor In Termite, however, the concurrent game is almost as simple as a turn based game. Labels, not states, are classified as controllable or uncontrollable. λ chooses the effective label non-deterministically. This means that while player 1 may choose a label in any state, there is no guarantee that it will be played. There is, however, a fairness guarantee that each player eventually gets a turn that will be dealt with later.

We simulate this non-determinism with the input variable `controllable`. This variable is always chosen non-deterministically. Our (now deterministic) λ function picks the player 1 chosen label if `controllable` is $True$ and it picks the player 2 chosen label otherwise.

Disregarding fairness for now, given a target state X, we define a state s to be winning if both:

1. there exists a controllable label originating from S such that all transitions with this label lead to X, and
2. all uncontrollable transitions with any label originating from S lead to X

are simultaneously true.

Condition 1 is captured by the function $CpreC$ given in Equation 2.8. It returns the set of states from which there exists a controllable label such that all transitions with this label lead to a state in the target set X.

$$CpreC(X) = \exists L.controllable \wedge \forall N.Trans(S, L, N) \rightarrow X(N) \tag{2.8}$$

Condition 2 is captured by the function $CpreU$ given in Equation 2.9. It returns the set of states from which all transitions with uncontrollable labels lead to a state in X.

$$CpreU(X) = \forall L.\neg controllable \rightarrow \forall N.Trans(S, L, N) \rightarrow X(N) \tag{2.9}$$

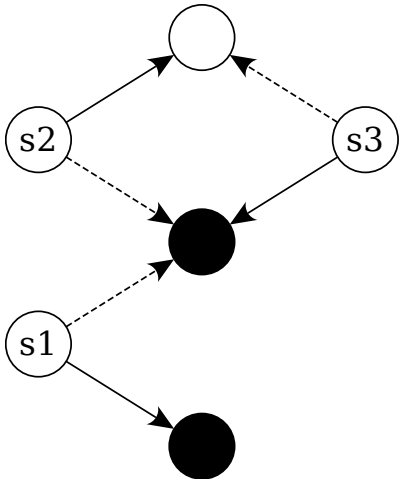


Figure 2.2: Concurrent controllable predecessor

The final controllable predecessor, $Cpre$, given in Equation 2.10 returns the set of states for which both conditions are satisfied.

$$Cpre(X) = CpreC(X) \wedge CpreU(X) \tag{2.10}$$

This is illustrated in Figure 2.2. Again, the two dark states are winning and they form the set X passed to $CPre$. $CPre$ returns state $s1$ in the winning set because there exists a controllable transition into the winning set and all uncontrollable transitions also lead to the winning set. State $s2$ is *not* winning because it does not have a controllable transition into the winning set. Lastly, $s3$ is *not* winning because there exists an uncontrollable transition to a non-winning state.

Safety and reachability

We will start with reachability as it is the most straightforward. We solve games by finding the set of states from which player 1 can win, called the *winning region* and denoted WIN. If the winning region contains the set of initial states, then we know that player 1 can win from any state in the initial set and thus can win the game.

According to Equation 2.2, a state is winning in a reachability game if there is some finite i such that we can guarantee that after i rounds of the game, execution will have, at least once, entered a state in $REACH$.

We can define the winning region inductively. It is obviously possible to reach the set $REACH$ from $REACH$ in 0 steps as we are already there. This is the base case. It is also possible to reach $REACH$ from $x \in S$ in $N+1$ steps or fewer iff there exists $Y \subseteq 2^S$ such that it is possible to reach $REACH$ from all of $Y \subseteq 2^S$ in N steps or fewer and $x \in CPre(Y)$.

This suggests an algorithm to find the winning region. We start with $REACH$, apply CPre to get the states winning in 1 step, combine with $REACH$ again to get the states reachable in 1 step or less, and then repeat. After N iterations, we find the states winning in N or less steps. The algorithm is given in Algorithm 2 and the equivalent fixed point formula is given in Equation 2.11. Two questions remain:

- An algorithm must terminate, so, when should we stop iterating?
- After termination, has the algorithm found all the winning states?

We denote the set of states from which it is possible to reach REACH in N or less steps W_N. Observe that $W_{N+1} \supseteq W_N$. So, on each iteration, we either grow the winning set or it remains the same. Also observe that our set S is finite, which means that 2^S is also finite. As set inclusion is a transitive relation, we cannot grow our winning set forever, so eventually we must reach a fixed point of the CPre function.

A more formal proof of termination is based in the Knaster-Tarski theorem [Tarski, 1955]. The power set of S can be ordered by set inclusion to obtain a complete lattice with supremum S and infimum $\emptyset$. $f(X) = CPre\ (X \cup REACH)$ is an order preserving function, so by the Knaster-Tarski theorem the set of fixed points of f is also a complete lattice. Thus there exists a greatest and least fixed point (as the lattice is complete). The least fixed point of f is clearly obtained by iteration starting from the least element of the lattice, ie. $\emptyset$.

Thus, we will eventually reach a fixed point, and, this is when we should stop iterating as further iterations will not change the winning set. Furthermore, we know that after any finite number N of iterations where N is greater than the number of iterations required to reach the fixed point, the winning region will remain WIN. Thus $WIN_N = WIN$ and we have found all winning states.

Finally, to answer the question of whether the game is winning for player 1, we check if the winning region contains the initial state set. If it does, we return *Yes*; otherwise, we return *No*.

Algorithm 2 Solving a reachability game

Input: A set of target states $REACH \subseteq S$, an initial set of states I and a monotonic controllable predecessor $CPre$.
Output: *Yes* if $I \subseteq ReachWin(T, Cpre)$ and *No* otherwise.

```
function ReachWin(REACH, I, CPre)
    Y ← ∅
    loop
        Y' ← CPre(Y ∪ REACH)
        if Y' = Y then
            if Y ⊆ I then
                return Yes
            else
                return No
            end if
        end if
        Y ← Y'
    end loop
end function
```

Safety games are the dual of reachability games and are also solved by iterating the controllable predecessor. The algorithm for solving safety games is also the dual of the algorithm for solving reachability games and will not be given here. However, we do give the fixed point formula in Equation 2.12 and the algorithm can be deduced from this using Algorithm 1.

$$ReachWin(T) = \mu Y.CPre(Y \vee T) \tag{2.11}$$

$$SafeWin(T) = \nu Y.CPre(Y \wedge T) \tag{2.12}$$

Büchi

To solve a Büchi game, we first find the set from which we can reach the goal once as we do for a reachability game. Then, we use this set to find the set from which we can reach the goal twice, three times, and so on until we get to another fixed point.

Imagine we have solved the reachability game $R = ReachWin(T)$ where T is the Büchi target set and R is the winning region. Then $V = R \wedge T$ is a subset of the goal from which player 1 can reach the goal one more time. Computing $W = ReachWin(V)$ gets us the set of states from which we can

Algorithm 3 Solving a generalized Büchi game

```
function GENERALIZED_BUCHI(T, CPre)
    X ← S
    loop
        X' ← S
        for each G in Goals do
            X' ← X' ∩ REACHWIN(X ∩ G)
        end for
        if X' = X then
            return X
        end if
        X ← X'
    end loop
end function
```

reach the goal twice. Iterating this procedure will eventually lead to a fixed point as $ReachWin$ is a monotonic function.

This fixed point is the set of states from which we can reach the goal any number of times. Thus, it is the winning set of the Büchi game. The fixed point formula is given in Equation 2.13 and the algorithm to solve Büchi games can be deduced from this using Algorithm 1.

$$BuchiWin(T) = \nu X.\mu Y.CPre(Y \vee (X \wedge T)) \tag{2.13}$$

Generalized Büchi

Solving a generalized Büchi game is similar to a Büchi game except that we find the set from which we can reach any goal once, then any goal twice, etc. The algorithm is a small modification to the Büchi algorithm and is given in Algorithm 3. The equivalent fixed point formula is given in Equation 2.14.

$$GenBuchiWin(Goals) = \nu X. \bigwedge_{G \in Goals} \mu Y.CPre(Y \vee (X \wedge G)) \tag{2.14}$$

GR(1)

Next, we need to add fairness. Consider a fair reachability game. An unfair region is a region that we can assume execution will leave, regardless of the loops it contains. We modify the controllable predecessor to create a fair

Algorithm 4 The fair controllable predecessor

```
function FAIRCPRE(CPre, φ, T)
    Z ← S
    loop
        Z' ← CPRE(Z ∩ φ ∪ T)
        if Z' = Z then
            return Z
        end if
        Z ← Z'
    end loop
end function
```

Algorithm 5 GR(1) game

```
function GR1WIN(CPre, T)
    return BUCHIWIN(T, fairCpre(CPre))
end function
```

controllable predecessor that takes this into account. Intuitively, the algorithm considers a set of unfair states to be winning if the only way out leads to an already winning state. To achieve this, we play a variation of a safety game where we can win if we stay indefinitely within the unfair set or upon exiting the unfair set we are immediately in the target set T. The procedure for the fair controllable predecessor is given in Algorithm 4 and the equivalent fixed point formula is given in Equation 2.15.

$$fairCpre(F, T) = \nu Z.CPre((\neg F \wedge Z) \vee T) \tag{2.15}$$

Using $fairCpre$ as the controllable predecessor operator in the reachability algorithm (Algorithm 2) yields the fair reachability algorithm. Finally, if we combine the Büchi game with the fair controllable predecessor we get a GR(1) game. The procedure is given in Algorithm 5 and the equivalent fixed point formula is given in Equation 2.16.

$$GR1Win(Goals, Fairs) = \nu X. \bigwedge_{G \in Goals} \mu Y. \bigvee_{F \in Fairs} \nu Z.CPre((\neg F \wedge Z) \vee (G \wedge X) \vee Y) \tag{2.16}$$

2.6 Strategies and Counterexamples

Extracting strategies

Reachability

Once we have solved a game and determined that it is winning, we can extract a strategy. In driver synthesis, the strategy is used to generate the driver. A strategy for a reachability game is a relation between states and labels for player 1 to play such that if player 1 sticks to this strategy, the game will eventually end up in the goal. Strategy extraction requires a straightforward modification to the game solving algorithm.

When solving a reachability game, we discover the sets of states for which we can force execution into the goal in 1 step, 2 steps, etc. When extracting a strategy for a reachability game we need to record, for each iteration, how we got one step closer to the goal. The algorithm is given in Algorithm 6.

The strategy relation is initialized to the empty set on line 2. Then, we perform an iteration of the controllable predecessor. This time we also use CPRE_STRAT on line 9, which computes a relation between the winning states discovered so far and the labels that take execution from these newly discovered states one step closer to the goal.

On each iteration we combine this strategy for the newly discovered states with the strategy for the previously discovered states. We are careful on line 10 not to add new labels for states that have already been discovered to ensure that the labels take us towards the goal. We iterate until our winning region reaches a fixed point as before. In the end, the strategy relates each state in the final winning region to a set of labels that take the game one step closer to the goal.

CPRE_STRAT is a parameter to the strategy extraction algorithm in the same way as CPRE is to the game solving algorithm. Its definition depends on the exact details of the game. For completeness, we give the definition of CPRE_STRAT for turn based games.

$$CPre\_Strat(X) = \{\langle x, l\rangle | x \in \tau_1 \wedge (\langle x, l, x'\rangle \in \delta \rightarrow x' \in X)\} \tag{2.17}$$

It returns a set of $\langle state, label\rangle$ pairs where the state belongs to the player 1 controllable set such that if the label is played, execution ends up in the set X,

Algorithm 6 Extracting a strategy for a reachability game

```
1:  function REACH_STRAT(REACH, CPre, CPre_Strat)
2:      Y ← ∅
3:      STRAT ← ∅
4:      loop
5:          Y' ← CPre(Y ∪ REACH)
6:          if Y' = Y then
7:              return (Y, STRAT)
8:          end if
9:          STRAT' ← CPre_Strat(Y ∪ REACH)
10:         STRAT ← STRAT ∪ {⟨s, l⟩ | ⟨s, l⟩ ∈ STRAT' ∧ s ∉ Y}
11:         Y ← Y'
12:     end loop
13: end function
```

Algorithm 7 Extracting a strategy for a Büchi game

```
1: function BUCHI_STRAT(BUCHI, CPre, CPre_Strat)
2:     win ← BUCHI(BUCHI)
3:     return REACH_STRAT(win ∩ BUCHI, CPre, CPre_Strat)
4: end function
```

the argument to the function.

Büchi

Like a reachability strategy, a strategy for a Büchi game must ensure that we eventually get to the goal. However, it must also ensure that once we get to the goal it is still possible to reach the goal again.

If we find a reachability strategy for the intersection of the goal set and the winning region, then when we follow this strategy, we are guaranteed to reach the goal once and then be able to reach it again by following the same strategy because we remain in the winning region. Thus, the algorithm for finding the strategy for a Büchi game is to find the winning region of the game and then compute the reachability strategy for the intersection of the goal and the winning region.

The algorithm for computing the strategy in a Büchi game is given in Algorithm 7. It simply computes the Büchi winning region on line 2 and then computes the strategy to reach the intersection of the winning region and the goal set on line 3.

Generalized Büchi

A generalized Büchi strategy must ensure that we can always get to any of the goals. If we compute, for each goal, the reachability strategy for the intersection of that goal and the generalised Büchi winning region, we get a strategy that takes us to that goal and, by keeping execution in the winning region, ensures that we can reach any of the other goals using their strategies.

Thus, a strategy for a generalised Büchi game is a set of strategies, one for each goal, that we must play in a round robin manner to ensure that each goal is eventually reached. The finite state required to ensure that the strategies are played in a round robin order is the finite state that the generalised Büchi strategy requires as discussed in Section 2.4. This, for example, could be a counter that that loops through the goals and is incremented each time the current goal is reached.

The algorithm for computing the strategy in a generalized Büchi game is given in Algorithm 8. It first computes the winning region of the generalised Büchi on line 2 and then it computes a strategy to reach the intersection of this and each goal on line 5.

Algorithm 8 Extracting a strategy for a generalized Büchi game

```
1: function GEN_BUCHI_STRAT(BUCHIS)
2:     win ← GENERALIZED_BUCHI(BUCHIS)
3:     strats ← [ ]
4:     for each G ∈ BUCHIS do
5:         strats ← strats ⊕ REACH_STRAT(win ∩ G)
6:     end for
7:     return strats
8: end function
```

Fairness

To compute strategies for fair variants of reachability, Büchi and GR(1) games we use the same approach as solving them and introduce a modified controllable predecessor that takes fairness into account, called CPre_Strat_Fair. This modified controllable predecessor, in turn, is parameterised by the CPre and CPre_Strat of the underlying game.

The fair controllable predecessor must ensure that we can reach the target set assuming that an unfair condition is not forever true. Conceptually, it tries

to keep execution within an unfair region (as a safety game would) for which the only way out takes us a step closer to the goal.

The algorithm for computing the strategy in a fair reachability game is given in Algorithm 9. It essentially solves a safety game for $\neg\phi$ with the exception that it is possible to leave safe regions as long as you enter T. When a fixed point is reached on line 6 it uses CPre_Strat to compute a strategy to either stay within the computed unfair region or enter the target set. This strategy is guaranteed to reach the target set as long as fairness if not violated.

Algorithm 9 Fair controllable predecessor strategy extraction

```
1:  function FAIR_CPRE_STRAT(CPre, CPre_Strat, φ, T)
2:      Z ← U
3:      loop
4:          Z' ← CPRE(Z ∧ ¬φ ∨ T)
5:          if Z' = Z then
6:              return CPRE_STRAT(Z ∧ ¬φ ∨ T)
7:          end if
8:          Z ← Z'
9:      end loop
10: end function
```

GR(1)

Strategies for GR(1) games, as used in Termite, can be computed using Algorithm 8 instantiated with the fair controllable predecessor, Algorithm 9.

Extracting counterexamples

Counterexamples are computed by solving the complement game and extracting the strategy.

Reachability

The complement of a reachability game is a safety game with negated objective. This can be seen by complementing the mu-calculus formula for the winning region, repeated below for convenience.

$$REACH(T) = \mu Y.CPre_1(Y \vee T) \tag{2.18}$$

$$\neg REACH = \neg \nu Y.CPre_2(Y \wedge \neg T) = SAFE(\neg T) \tag{2.19}$$

Thus, the counterexample strategy for a reachability game is the strategy for a safety game with complemented objective and the other player's controllable predecessor. ***TODO: We dont actually give an algorithm for safety game strategy, so this is a bit useless.***

GR(1)

The formula for a GR(1) game and its complement are given below:

$$GR1(Goals, Fairs) = \nu X. \bigwedge_{G \in Goals} \mu Y. \bigvee_{F \in Fairs} \nu Z.CPre_1((\neg F \wedge Z) \vee (G \wedge X) \vee Y) \tag{2.20}$$

$$\neg GR1(Goals, Fairs) = \mu X. \bigvee_{G \in Goals} \nu Y. \bigwedge_{F \in Fairs} \mu Z.CPre_2((F \vee Z) \wedge (\neg G \vee X) \wedge Y) \tag{2.21}$$

The negation of a GR(1) objective is considerably more complex than a reachability objective. To ensure that player 1 cannot win a GR(1) game, player 2 must ensure that there is at least one goal that is only reached finitely many times, and each fairness condition is visited infinitely many times.

2.7 Symbolic games

The algorithm described so far appears very inefficient. Consider a reachability game. We are performing a backwards breadth-first search starting from $REACH$. If we were to implement it directly as described, we would need a set abstract datatype to represent the winning set. Some of the games we have solved with Termite have upwards of 2^{80} states, even after abstraction. Clearly, explicitly representing the winning set will never succeed.

Identical problems are encountered in model checking. The breakthrough that revolutionised model checking was to represent state sets implicitly as a characteristic equation over state variables.

State variable encoding

Symbolic games are defined over a finite set of state variables, Σ, and a finite set of label variables Λ. A *valuation* for a set of variables V is a function from each of those variables to an element of its domain. We redefine S, the set of states, to be the set of possible valuations of each state variable in Σ. That is, each state $s \in S$ is given by a valuation of all of the state variables in Σ. Similarly, we redefine L, the set of labels, to be the set of possible valuations of each label variable in Λ.

For a set Z of variables, we denote by $\mathcal{F}(Z)$ the set of propositional formulas constructed from the variables in Z. The characteristic formula of a set of states T is a function $f \in \mathcal{F}(\Sigma)$ that evaluates to true for the valuation corresponding to a state $s \in T$ and false otherwise. We use characteristic formulas to represent sets of states without explicitly listing each member of the set. This is called a symbolic representation.

Likewise, δ is specified by a formula in $\mathcal{F}(\Sigma \bigcup \Lambda \bigcup \Sigma')$, where $\Sigma' = \{\sigma' \mid \sigma \in \Sigma\}$ is the set of next state variables.

Sometimes, we make the variables that a characteristic formula depends on explicit. For example, we may write the transition relation as $\delta(\Sigma, \Lambda, \Sigma')$.

Symbolic algorithms

We can redefine our game solving algorithms to use characteristic functions instead of explicit sets. The algorithms are superficially similar except they use conjunction and disjunction to modify sets instead of explicit set intersection and union. As an example, we convert the turn based controllable predecessor (Equation 2.7) to symbolic form in Equation 2.22 and, we convert the simplest algorithm, determining the outcome of a reachability game to symbolic form, in Algorithm 10.

$$CPre(X) = \tau_1 \wedge \exists\Lambda.\forall\Sigma'.\delta \rightarrow X' \vee \tau_2 \wedge \forall\Lambda.\forall\Sigma'.\delta \rightarrow X' \tag{2.22}$$

Strategy generation

TODO: Symbolic strategy generation: should I bother?

Algorithm 10 Solving a reachability game symbolically

Input: The characteristic function of the set of target states $REACH \subseteq S$, the characteristic function of the initial set of states I and a monotonic controllable predecessor CPre that operates on characteristic functions.

Output: *Yes* if $I \subseteq REACH(T, CPre)$ and *No* otherwise.

```
function REACH(REACH, I, CPre)
    Y ← False
    loop
        Y' ← CPre(Y ∨ REACH)
        if Y' = Y then
            if Y → I then
                return Yes
            else
                return No
            end if
        end if
        Y ← Y'
    end loop
end function
```

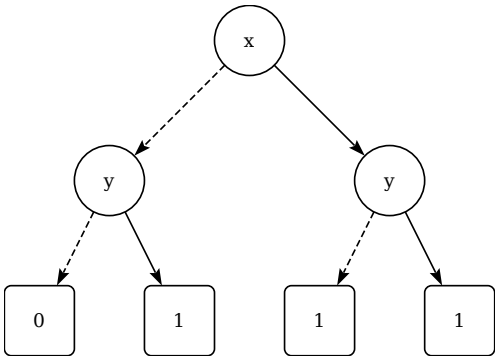


Figure 2.3: A binary decision tree for $x \vee y$

Binary decision diagrams

A binary decision diagram is an efficient data structure for manipulating large propositional formulas. They are the symbolic data structure that we use in Termite.

Binary decision trees

Figure 2.3 shows a decision tree for the disjunction of two variables. The root node represents the disjunction function. The child nodes, or internal

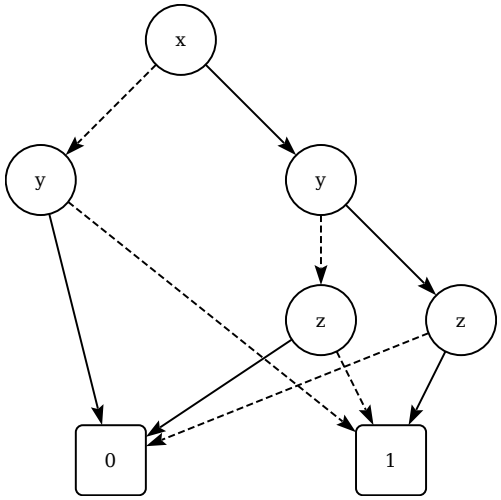


Figure 2.4: A binary decision diagram

nodes, represent variables and the leaf nodes, or terminal nodes represent the outcome of the function. Terminal nodes are labelled either True or False. Given a valuation of X and Y, we can evaluate the function by starting at the root and taking the solid edge if the variable represented by the node is assigned to True in the valuation and taking the dashed edge if the value is assigned to False.

For example, in Figure 2.3, for the valuation $x = True$ and $y = False$, we start at the root node which is labelled x as x is this node's decision variable. Variable X is assigned to True so we follow the solid edge to the next decision node which is labelled y. Y is assigned to false so we take the dashed edge and arrive at a terminal node whose value is 1, meaning that the function evaluates to 1 for this variable valuation.

Ordered binary decision trees

If the order in which the variables appear along all paths starting from the root node and ending at a terminal node are the same, then the decision tree is called an ordered decision tree. The decision tree in Figure 2.3 is ordered.

Reduced ordered binary decision diagrams

A reduced ordered binary decision diagram (from now on, just BDD) is created by sharing subtrees as much as possible within an ordered binary decision tree.

In particular:

- Terminal nodes with the same label are merged. This means there are only two terminal nodes: True and False.
- Internal nodes with the same children are merged.
- Nodes with two identical children are removed and all incoming nodes are redirected to the child.

Figure 2.4 is an example of a BDD.

Canonicity

Given a variable ordering, reduced ordered binary decision diagrams are a canonical representation of a function. This means that given a function f of some set S of variables, another function g that evaluates to the same value for each valuation of the variables in S will be represented by exactly the same BDD.

In practice, one uses a BDD library such as CUDD [Somenzi] to build and manipulate BDDs. CUDD keeps track of all BDDs and subtrees within the BDDs that have been created and reuses these to ensure that all BDDs remain in reduced form. This, along with canonicity, means that BDD equivalence can be checked in constant time simply by checking pointer equality of the two BDDs.

Complement arcs

Arcs may have a complement attribute. The function represented by an arc is the function of the node that it points to, unless it is a complement arc, in which case it is the complement of the function of the node that it points to.

Use of complement arcs has two important advantages:

- Decreased memory requirements
- Complementation can be done in constant time, allowing De Morgan's laws to be used freely

The second guarantees that complementation is free in practice.

Additionally, it can be shown that, if only the else arcs of internal nodes are ever complemented then canonicity can be preserved.

Conjunction and disjunction

BDDs are not usually built as decision trees and then reduced. Instead, they are built from the bottom up, starting with the terminal and variable nodes and combining these using conjunction, disjunction and negation.

Conjunction and disjunction are computed using a straightforward recursive algorithm that will not be given here, but if the reader is interested, more information can be found in [Bryant, 1986]. An important result is that, in the worst case, the procedure runs in time proportional to the product of the sizes of the two input BDDs. Furthermore, the size of the resulting BDD may be equal to the product of the sizes of the two input BDDs in the worst case. A strength of BDDs, however, is that this worst case rarely happens in practice.

Function composition and quantification

Function composition is where a BDD representing some function is substituted for a variable in another BDD.

Given a function $f(x_1, \cdots, x_n)$, we define existential quantification of f with respect to the Boolean variable x_i as $\exists x_i.\ f = f_{x_i} \vee f_{\overline{x_i}}$ and universal quantification of f with respect to x_i as $\forall x_i.\ f = f_{x_i} \wedge f_{\overline{x_i}}$.

Quantifications of the same type commute, so quantification with respect to a set of variables is well defined.

Variable ordering

The number of nodes in a BDD depends drastically on the ordering chosen for the variables. Therefore, the space occupied by the BDDs and the time spent performing operations on them also depends on the variable ordering. This directly affects the performance of game solving algorithms that use BDDs as the symbolic data structure.

Optimal variable orderings may be found using exact algorithms, but these are prohibitively expensive for BDDs with more than a few nodes. In practice heuristics are used which produce good, but not optimal orderings. One such heuristic is Ruddell's sifting algorithm [Rudell, 1993].

The CUDD BDD package performs *dynamic variable ordering*, which means that once the number of BDD nodes the package knows about grows past a certain threshold, the package automatically performs the requested reordering algorithm on all BDDs that exist in the manager. Dynamic variable ordering is

critical to the performance of game solving algorithms that utilise BDDs and therefore we always enable it.

3 Driver Synthesis as a Game

In this chapter we formalise the device driver synthesis problem using games. We show that GR(1) games are sufficient to capture the properties of the drivers that we require. We also develop the driver synthesis controllable predecessor.

We formalize the driver synthesis problem as a *two-player game* [Thomas, 1995] between the driver and its environment. The game is played over a finite automaton that represents all possible states and behaviors of the system. Transitions of the automaton are classified into *controllable* transitions triggered by the driver and *uncontrollable* transitions triggered by the device or OS. A winning strategy for the driver in the game corresponds to a correct driver implementation. If, on the other hand, a winning strategy does not exist, this means that there exists no specification-conforming driver implementation.

Two-player games naturally capture the essence of the driver synthesis problem: the driver must enforce a certain subset of system behaviors while having only partial control over the system. The game-based approach leads to a precise mathematical formulation of the problem and enables us to apply theoretical results and algorithms from game theory to driver synthesis.

Figure 3.1 illustrates the concept using a trivial game automaton that models the core of our running example. Controllable and uncontrollable transitions of the automaton are shown with solid and dashed arrows respectively. The goal of the driver in the game is to infinitely often visit the initial state, labelled `G`, which represents the situation when the driver does not have any outstanding requests. After getting a `send` request from the OS, the driver must write data and command registers to start the data transfer. Writing the command register first may trigger a hardware send event before the driver has a chance to write the data register. As a result, wrong data value gets sent,

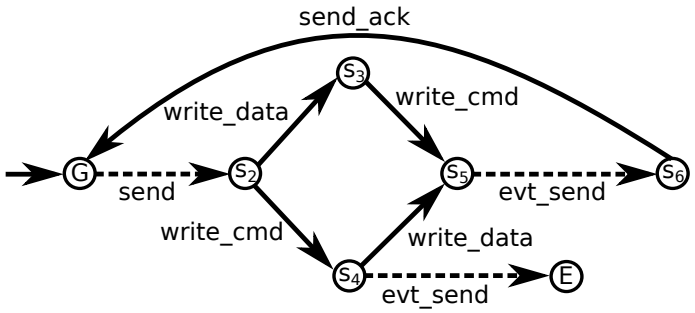


Figure 3.1: A simple two-player game.

taking the game into an error state `E`. Hence, state s_4 is losing for the driver. To avoid this state, the correct strategy for the driver is to play `write_data` in state s_2, followed by `write_cmd`. In s_5 the driver must remain idle until the environment executes the `evt_send` transition.

3.1 The Players

To formalise the driver synthesis problem using games, the first thing we must define is the players:

- Player 1 is the driver.
- Player 2 is the environment, which consists of the device to be controlled as well as the operating system.

3.2 Device and OS specifications and synchronization

A core concept of Termite is the separation of device and operating system specifications. Games so far have been defined to have a single state machine that contains all of the states of the system as well as all of the actions. In Termite, we decouple the description of the operating system from the device to be controlled and then combine these to form a state machine before synthesis.

Both the device and OS specifications are given as state machines. They are defined symbolically. Each specification has its own set of state variables that it must supply update functions for, however, one specification may refer

to another's state variables in its state update functions. There is also a global set of label variables which each specification may also refer to in its state update functions. An overview of a system consisting of a device and OS specification as well as their interconnections is given in figure **??**.

As the figure shows, device and OS specifications are identical to modules that describe Mealy machines in a hardware description language such as Verilog or VHDL. They consist of some private state, represented by D type flip-flops (though the variables are not restricted to single bits) that is updated solely by logic within that module. Other modules can read, but not write to, this private state through the Mealy machine's output variables. The label, which is chosen by one of the players, appears as input to both of the modules and they can use it in the update functions for their private state.

To ensure reusability of specifications, specifications do not directly refer to each other's state variables. Instead, they access them through well defined interfaces and there is a device class mechanism to standardise the interfaces of both the device and operating system specification interfaces. We delay description of this mechanism until Chapter **??**. For now we assume that specifications can access each other's state variables without restriction.

3.3 An Example System Specification

We give an example system specification to illustrate its decomposition into separate OS and device specifications. We give the system both as graphical state machines and in ASL.

State Machines

Figure 3.2a is our example device specification. The labelled circles are states. Arrows represent transitions between states. Solid arrows are controllable transitions whereas dashed arrows are uncontrollable, as described in Section **??**. Each transition is labelled by at least one event. Conceptually, these events trigger the transitions. All events for which there are no outgoing transitions do not change the state, i.e. they are self loops and are not shown in the interest of keeping the diagrams concise. The short arrow with no source state indicates the initial state. There may be more than one initial state, in which case the initial state is chosen non-deterministically.

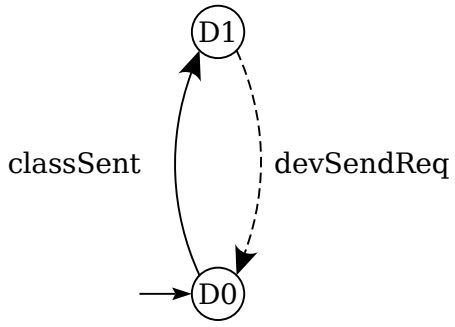
Device Specification

We start with the device specification given in Figure 3.2a. Our device is a hypothetical trivial UART-like device. Instead of sending bytes, it sends notifications, which carry no value. Its programming interface consists of an action, `devSendReq`, that requests that the device send a notification. In a real system this could be a write to a particular bit in a control register that triggers the device action. This action only has an effect in the `devIdle` state and it causes the device to transition to the `devSending` state. From there, when the `devSent` event happens, which corresponds to the device actually sending the notification, the device transitions back to the `devIdle` state. Additionally, it emits the `classSent` event. We will respond to the `classSent` event in the OS specification in the next section. From here, another notification may be requested by performing the `devSendReq` action again.

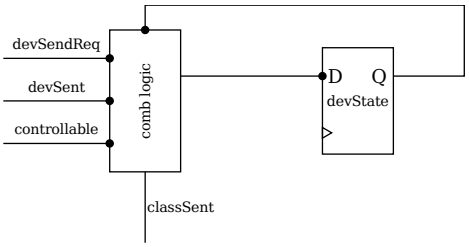
A fragment of the symbolic encoding of the device specification is given in Figure 3.2c. There is only one state variable, `devState`, that toggles between two values: `devIdle` and `devSending`. Additionally, there are three boolean label variabes: `controllable`, `devSendReq` and `devSent`. The `devSendReq` and `devSent` variables correspond to the actions in the provious paragraph. The `controllable` variable is an additional variable to distinguish between controllable and uncontrollable events.

The update function for the `devState` variable (the only variable in the device specification) consists of a single case statement. If the system is currently in the `devIdle` state and the `devSendReq` label variable is true, and, importantly, the `controllable` label variable is also true, then the system transitions to the `devSending` state. The `controllable` variable had to be true because this was a controllable transition. Additionally if the system is in the `devSending` state and the `devSent` variable is true and the transition is not controllable then the system transitions to the `devIdle` state. If neither of these conditions are met, the device remains in the same state.

On line 19 we defined the `classSent` event to occur when the device is in state `devSending`, the `devSent` label is true and the transition is not controllable. This matches the state machine. The `define` statement is actually an m4 [**?**] macro that will be used to substitute the definition of `classSent` into the operating system specification later. This behaves much like a signal or continuous assignment in a hardware description language, but, to keep



(a) Device specification



(b) Device specification

```
State
devState : {devIdle, devSending};

Label
devSendReq   : bool;
devSent      : bool;
controllable : bool;

Init
devState == devIdle

Transitions
devState := case {
    devState == devIdle && devSendReq && controllable  : devSending;
    devState == devSending && devSent && !controllable : devIdle;
    true                                               : devState;
}

define(classSent, (devState == devSending && devSent && !controllable))
```

(c) Symbolic device specification

our specification language minimal we have not implemented signals.

Lastly, this device model may be seen as a Mealy machine as shown in Figure 3.2b. There is a single register for storing the current state and this state is updated on each transition using combinational logic which additionally takes the label variables as input. This combinational logic also produces the output boolean variable `classSent`.

OS specification

The operating system specification, given in Figure 3.3a is like a test harness for our driver. It specifies which requests the driver may receive (e.g. to send a notification) and how it must respond (e.g. by eventually sending the requested notification). It is simply another state machine, like the device specification, but it may also have goals.

The goal state, state `O3`, is the state that a correct driver will eventually force the OS specification to be in. It is possible to specify multiple goal states, in which case the driver must force execution into any goal state.

Our OS state machine starts off in the `osIdle` state. From there, if a `classSent` event is triggered by the device state machine, then the device must have sent a notification without the OS ever having requested one. This is an error, so the OS state machine transitions to the `osError` state. There is no way that the OS could have known that this event happened as it does not monitor the device notification output, so, the OS specification does not represent the way that the OS behaves in reality. These events that do not correspond to real interactions but are used to enforce correctness are called *virtual events*. Responding to virtual events in this way is how the OS spec guarantees correctness of the system.

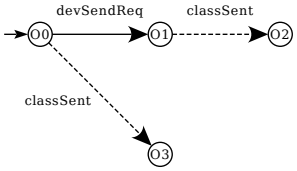
There is nothing special about the error state, however, we have constructed the state machine in such a way that the error state is a dead end and it is not possible to reach the goal from that state.

If a `devSndReq` event happens, the OS transitions into the `devRequested` state. From there, when a `classSent` event happens, the OS specification transitions into the goal state as this time the `classSent` event was expected.

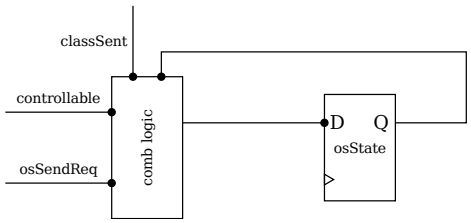
We can see from the OS specification that it is the job of the driver to ensure that a `devSendReq` event is eventually followed by `classSent` event. However, a `classSent` event can only be issued by the device, not the driver. The driver can, however, force the device to issue a `classSent` event indirectly in response to a `devSendReq` event.

In general, it is the job of the driver synthesis algorithm to figure out how to force certain events to happen at the right time (as specified by the OS state machine) by using information from the device state machine.

The symbolic specification is given in Figure 3.3c. After seeing the device specification, it is fairly self explanatory. Note that, on line 21 the class event



(a) OS specification



(b) OS specification

```
State
osState  : {osIdle, osRequested, osDone, osError};

Label
osReqSend    : bool;
controllable : bool;

Init
osState == osIdle

Transitions
osState := case {
    osState == osIdle :
        case {
            classSent                 : osError;
            osReqSend && controllable : osRequested;
            true                      : osState;
        };
    osState == osRequested :
        case {
            classSent : osDone;
            true      : osState;
        };
    osState == osDone  : osDone;
    osState == osError : osError;
};
```

(c) Symbolic OS specification

defined in Figure 3.2c is used.

As with the device specification, the OS specification may be seen as a Mealy machine as shown in Figure 3.3b. Again, there is a single register for storing the current state and this state is updated on each transition using combinational logic. In addition to taking the label variables as input, it also takes the `classSent` boolean signal that was produced by the device Mealy machine as input.

```
State
osState  : {osIdle, osRequested, osDone, osError};
devState : {devIdle, devSending};

Label
osReqSend    : bool;
classSent    : bool;
devSendReq   : bool;
controllable : bool;

Init
osState == osIdle && devState == devIdle

Transitions
devState := case {
    devState == devIdle && devSendReq && controllable  : devSending;
    devState == devSending && devSent && !controllable : devIdle;
    true                                               : devState;
}

define(classSent, (devState == devSending && devSent && !controllable))

osState := case {
    osState == osIdle :
        case {
            classSent                 : osError;
            osReqSend && controllable : osRequested;
            true                      : osState;
        };
    osState == osRequested :
        case {
            classSent : osDone;
            true      : osState;
        };
    osState == osDone  : osDone;
    osState == osError : osError;
};
```

Figure 3.4: Combined ASL specification

Combined Specification

We combine the device and OS specifications by executing them concurrently. We do this by instantiating both Mealy machines in parallel and connecting the signals, as shown in Figure 3.6. In ASL, this is as simple as placing both case statements in the `Transitions` section, combining the state and label declaration sections and combining the initial states. The result is given in Figure 3.4.

Figure 3.5 shows the state machine of the combined specification. This large and cumbersome state machine clearly shows the size and complexity advantages of using separate Mealy machines in parallel as the specification

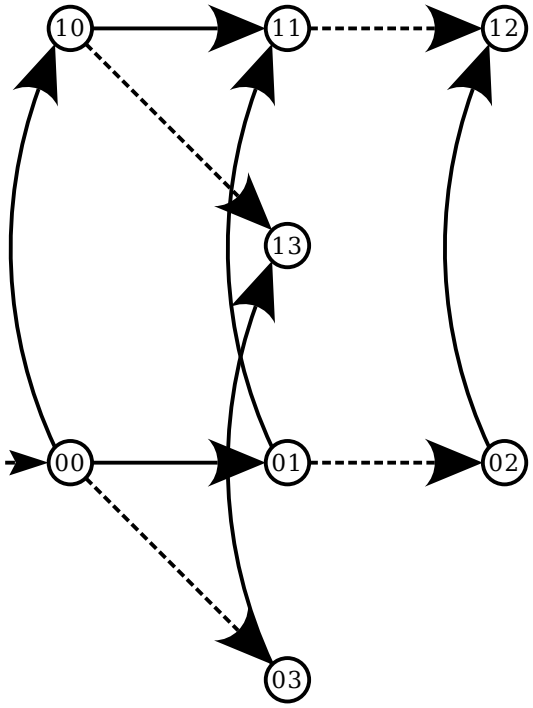


Figure 3.5: Combined specification

language.

3.4 GR(1) based formalism

In the following sections, we attempt to formalise the driver synthesis problem using the simplest game - the reachability game - and analyse its shortcomings. We show that extending our formalism to GR(1) objectives, as used in Termite, overcomes these shortcomings.

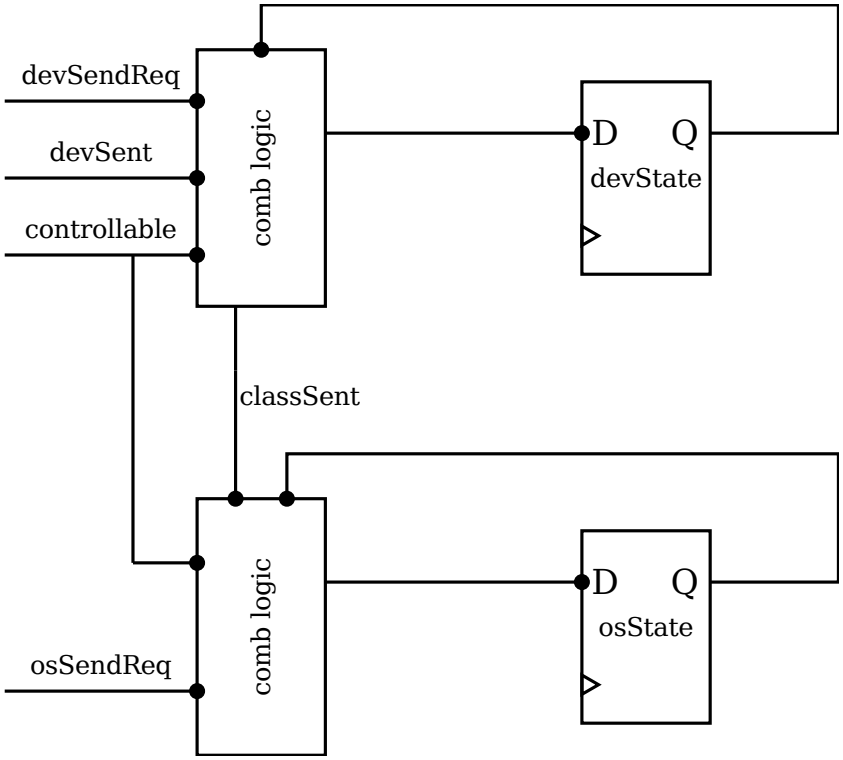


Figure 3.6: Combined specification

Reachability

As a concrete example, we could create a crude formalism for driver synthesis using only a reachability game. Consider, for example, figure 3.7, which shows the state machine for a game to control a hypothetical network controller. Solid lines indicate controllable transitions and dashed lines indicate uncontrollable transitions. Execution begins in the leftmost state where the OS may initiate a network transfer by choosing the 'send' label. The goal of the game is the rightmost state (labelled 'G') as this is the point where player 1 has completed the request. So, to win, player 1 (who controls the transitions with solid lines) must ensure that execution of the state machine reaches the goal.

The network device has two 8-bit registers, command (abbreviated cmd) and data. Writing 0x01 to the command register starts the transfer, and eventually whatever is in the data register gets written out to the network. Note that the actual sending of the data is an uncontrollable event.

The correct sequence to win the game, therefore, is to write the data register and then the control register after the OS performs a send request. This takes

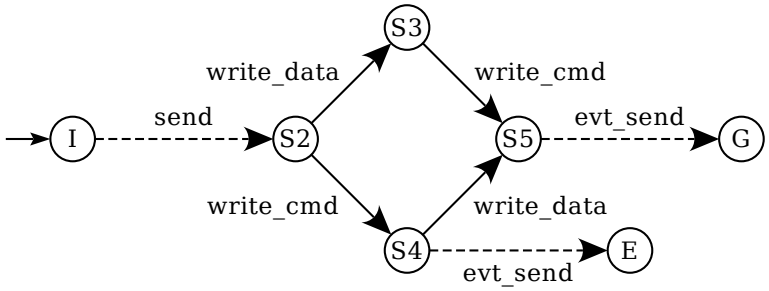


Figure 3.7: Reachability game for simple network device

us to state 'S5' where the only move by player 2 is 'evt_send' taking us to the goal.

If the command register is written first and then the data register there is potential for the environment to play the 'evt_send' label before the data is written, potentially resulting in the wrong data being sent. This is the transition that terminates in the 'E' state (for error). The 'E' state is a dead end, so it is not possible to reach the goal.

So, if player 1 takes the top half of the diamond (i.e. writes data before command) then it will be guaranteed to reach the goal and the reachability game is winning for player 1. The strategy to reach the goal tells us the sequence of labels the driver must play to get to the goal. In principle, this could be turned into a driver for our simple network device.

This simplistic formalism for driver synthesis has several shortcomings that we will deal with in the following sections.

Büchi

Consider a simplified network controller that does not have a command register. Instead, writing to the data register triggers transmission of the byte. However, there are two ways of writing to the data register. One is a standard register write. The other also performs the register write and then schedules a self destruct sequence to happen immediately after the byte is transmitted. The state machine for this device is shown in figure 3.8. The goal, in this case, is the set $S3, S5$ corresponding to the state after completion of the send request. The problem is that, unless you only ever want to send one byte, this goal does not capture the required behavior. One could easily work around this problem by specifying only $S3$ as the goal, but this breaks the compositionality of the

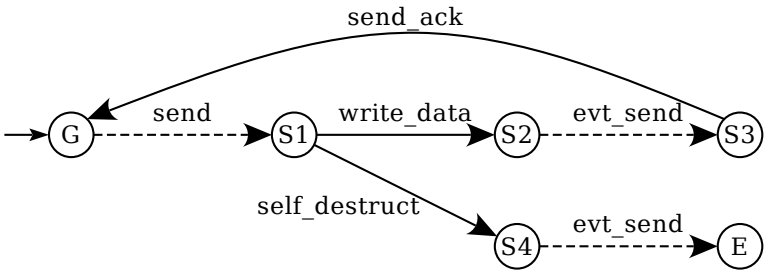


Figure 3.8: Buchi game for simple network device

specifications.

The solution is to modify the objective of the game. Instead of being able to reach the goal once, we want to be able to reach the goal an infinite number of times. Or, equivalently, we want to always be able to reach the goal again. This kind of objective is called a Buchi objective and a game with a Buchi objective is called a Buchi game.

Fairness

Consider a modification of our simplified network device without a self destruct sequence, but with the ability to check that noone is using the communication medium prior to transmitting. The state machine of this device is given in figure 3.9. After the user requests data transmission by writing to the data register, it executes a loop that checks if the medium is free, and if so, it performs the transmission.

If we pose this as a reachability game with goal state G, then the game is not winnable. The device may stay in the loop forever as it is never guaranteed to exit. Such a behavior should not prevent a driver from being synthesized providing that we have good reason to believe that the loop will eventually exit. Looping forever can be seen as a invalid behavior and we want to synthesize a driver for this system providing the invalid behavior does not occur.

In model checking these behaviors are eliminated with fairness conditions. Fairness conditions are sets of states which we guarantee will eventually be left, which we refer to as unfair states. In the example, the unfair states are the set $S2, S3$. The fairness condition says that we will eventually leave the unfair set, and the only way of doing this is through the evt_send transition, and the game becomes winning.

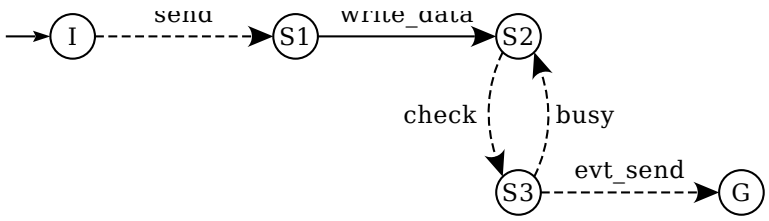


Figure 3.9: Fair reachability game for simple network device

Multiple Goals

Multiple Fairness

GR(1)

The combination of fairness and buchi objectives is called a GR(1) objective. Intuitively a GR(1) objective says that we can always reach some goal state provided that we do not get stuck forever in some unfair set of states. We use GR(1) objectives in Termite as we have found that in practice it is sufficient to express our goals.

4 User guided synthesis

4.1 Introduction

Overview of Termite

Figure 4.1 gives an overview of the driver synthesis process, described in detail in the rest of this chapter. Termite takes three specifications as its inputs: a device model that simulates software-visible device behavior, an OS model that specifies the software interface between the driver and the OS, and a driver template that contains driver entry point declarations and, optionally, their partial implementation to be completed by Termite.

Given these specifications, driver synthesis proceeds in two steps. The first step is carried out fully automatically by the Termite game-based synthesis engine, which computes *the most general strategy* for the driver—a data structure that compactly represents all possible correct driver implementations. This step encapsulates the computationally expensive part of synthesis. At the

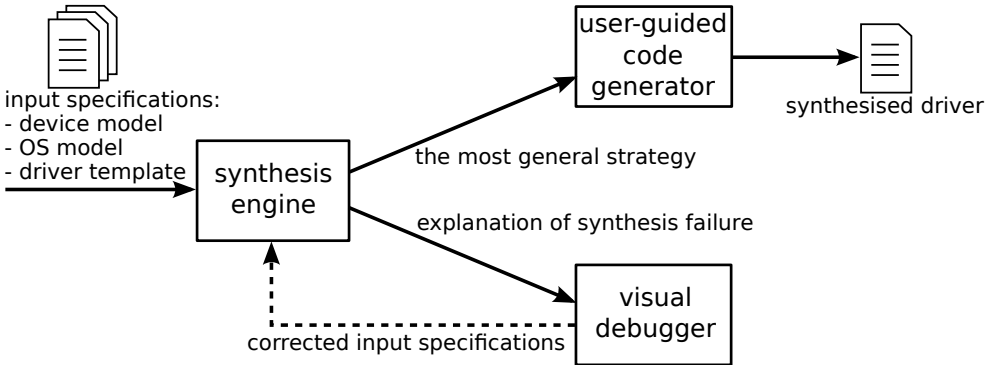


Figure 4.1: Termite synthesis workflow.

second step, the most general strategy is used by the Termite code generator to construct one specific driver implementation in C with the help of interactive input from the user.

The synthesis engine may establish that, due to a defect in one of the input specifications, there does not exist a specification-compliant driver implementation. In this case, it produces an explanation of the failure, which can be analysed with the help of the Termite debugger tool in order identify and correct the defect.

Limitations of Termite

The device driver synthesis technology is still in its early days and, as such, has several important limitations. Most notably, Termite does not currently support synthesis or verification of code for managing direct memory access (DMA) queues. This code must be written manually and is treated by Termite as an external API invoked by the driver. As another example, in certain situations, explained in Section 4.4, Termite is unable to produce correct code without user assistance; however it is able to verify the correctness of user-provided code. We discuss limitations of Termite in more detail in Section 4.7.

4.2 Specifications

Input to Termite consists of the three specifications, which model the complete system consisting of the driver, the device, and the OS, shown in Figure 4.2. The OS and device models simulate the execution environment of the driver and specify constraints on correct driver behavior. The device model simulates software-visible device behavior. The OS model serves as a workload generator that issues I/O requests to the driver and accepts request completions in a way consistent with real OS behavior.

The virtual interface between the device and the OS, shown with the dashed arrow in Figure 4.2, is used by the device model to notify the OS model about important hardware events, such as completion of I/O transactions and error conditions. Methods of the virtual interface do not represent real runtime interactions between the device and the OS, but are used by the OS model to specify correctness constraints for the driver (see Section 4.2).

Finally, the driver template contains a partial driver implementation to be completed by Termite. A minimal template consists of a list of driver

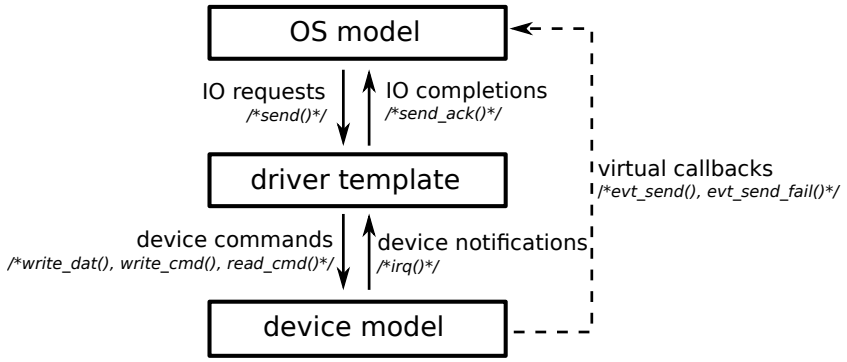


Figure 4.2: Input specifications for driver synthesis. Labels in italics show interfaces from the running example (Figure **??**).

entrypoints without implementation. At the other extreme, it can provide a complete implementation, in which case Termite acts as a static verifier for the driver. By this we mean that it checks that the interactions between the driver software and the hardware ensure that all operating system requests are eventually fulfilled. It does not, for example, check for the absence of null pointer dereferences and similar defects that can be found by tools such as SLAM [**?**]. In fact, TSL is designed to express state machines, not arbitrary programs and such defects are not expressible. Of course, this also limits the drivers that can be expressed.

All specifications are written using the Termite Specification Language (TSL). In line with our goal of making synthesis as close to the conventional driver development workflow as possible, TSL is designed as a dialect of C with additional constructs for use in synthesis. The full specification of the TSL language is given in Appendix A. We introduce relevant features of TSL throughout this section.

We minimize the amount of work needed to develop specifications for every synthesized driver by maximizing the reuse of specifications. In particular, Termite allows the use of existing device specifications developed by hardware designers in driver synthesis. It does this by providing a compiler from the modelling language used for the existing specification to TSL. This is explained in Section 4.2.

Furthermore, the OS specification for the driver can be derived from a generic specification for a class of similar devices (e.g., network or storage). Thus we expect that additional per-driver effort will consist of:

1. inserting device-class callbacks in appropriate locations of the device model and

2. extending the OS specification to support device-specific features missing in the generic OS specification.

Device model

The device model simulates the device operation at a level of detail sufficient to synthesize a correct driver for it. To this end, it must accurately model external device behavior visible to software. At the same time, it is not required to precisely capture internal device operation and timing, as these aspects are opaque to the driver.

Such device models are routinely developed by hardware designers for the purposes of design exploration, simulation, and testing. They are widely used by hardware manufacturers in-house [Intel Corporation] and are available commercially from major silicon IP vendors [Synopsys]. These models are known as *transaction-level models* (TLMs) (in contrast to the detailed register-transfer-level models used in gate-level synthesis) [Cai and Gajski, 2003]. A TLM focuses on software-visible events, or *transactions*, such as a write to a device register or a network packet transmission.

While the transaction-level modeling technology is relatively new, TLMs are already widely used by hardware manufacturers in-house [Intel Corporation] and are available commercially from major silicon IP vendors [Synopsys]. We are optimistic that in the future open-source TLMs will become commonplace, since a TLM does not expose internal implementation details of the device and is therefore less likely to expose sensitive IP.

Existing TLMs created by hardware designers can be used with minor modifications (explained in Section 4.2) for driver synthesis. Model reuse dramatically reduces the effort involved in synthesizing a driver and is therefore crucial to practical success of driver synthesis. By reusing an existing model, we also reuse the effort invested by hardware designers into testing and debugging the model throughout the hardware design cycle, thus making driver synthesis less susceptible to specification bugs. Finally, since TLMs are created early in the hardware design cycle, TLM-based driver synthesis can be carried out early as well, thus removing driver development from the critical path to product delivery.

TLMs are written in high-level hardware description languages like SystemC and DML. In order to use these models in driver synthesis, we need to convert them to TSL. This translation can be performed automatically, and we are currently working on a DML-to-TSL compiler. Since this work is not yet complete, device models used in the experimental section of this paper are either manually translated from existing TLMs or written from scratch using TLM modeling style guidelines [Wind River, 2010].

Unfortunately, few TLMs are publicly available at the moment, which means that driver synthesis must be performed by device vendors in-house. However this is likely to change in the future. A TLM does not expose internal implementation details of the device and is therefore not part of sensitive IP. At the same time, there exist strong incentives for manufacturers to make device TLMs available to third-party vendors to test their software and hardware products for compatibility with the given device.

Running example

Figure 4.3 shows a fragment of a model of a trivial serial controller device used as a running example. The fragment specifies the send logic of the controller, which allows software to send data characters over the serial line. The model is implemented as a TSL *template*. The template encapsulates data and code that manipulates the data, similar to a class in OOP.

The software interface of the device consists of data, command, and status registers declared in line 5. The registers can be accessed from software via the `write_dat`, `write_cmd`, `read_cmd`, and `read_status` methods (lines 8–22). The `controllable` qualifier denotes a method that is available to the driver and can be invoked from synthesized code.

The transmitter logic is modelled in lines 25–41. It is implemented as a TSL *process*. A TSL specification can contain multiple processes. The choice of the process to run is made non-deterministically by the scheduler. The process executes atomically until reaching a `wait` statement or a controllable placeholder (see below).

In line 27, the transmitter waits for a command, issued by the driver by writing value 1 to the command register. Upon receiving the command, it sends the value in the data register over the serial line. The transmission may fail, e.g., due to a serial link problem. The device signals transmission status to

```
1  /* Device model */
2  template dev
3
4      /* Device internal state */
5      uint8 reg_dat, reg_cmd, reg_status = 0;
6
7      /* device commands */
8      controllable void write_dat(uint8 v) {
9          reg_dat = v;
10     };
11
12     controllable void write_cmd(uint8 v) {
13         reg_cmd = v;
14     };
15
16     controllable uint8 read_cmd() {
17         return reg_cmd;
18     };
19
20     controllable uint8 read_status() {
21         return reg_status;
22     };
23
24     /* internal behavior */
25     process ptx {
26         forever {
27             wait (reg_cmd == 1);
28             choice {
29                 {
30                     os.evt_send(reg_dat);
31                     reg_status=0;
32                 };
33                 {
34                     os.evt_send_fail(reg_dat);
35                     reg_status=1;
36                 };
37             };
38             reg_cmd = 0;
39             /*drv.irq(); (see Section 4)*/
40         };
41     };
42 endtemplate
```

Figure 4.3: Trivial serial controller device specifications.

software by setting the status register to 0 or 1. Finally, it clears the command register, thus notifying the driver the request has completed.

Internally, the transmitter circuit consists of a shift register and a baud rate generator used to output data on the serial line. These details are not visible to software and are abstracted away in the model. We use the non-deterministic `choice` construct to choose between successful transmission and failure, without modelling the details of serial link operation. Successful and failed transmissions are modelled using`evt_send` and `evt_send_fail` events, explained in Section 4.2.

OS model

The OS model specifies the API mandated by the OS for all drivers of the given type. For example, any Ethernet driver must implement the interface for sending and receiving Ethernet packets. A separate specification is needed for each supported OS, as different OSs define different interfaces for device drivers.

A high degree of specification reuse can be achieved by creating a library of generic specifications for common types of drivers, e.g., network, storage, or serial drivers. A generic specification describes the API mandated by the OS for all drivers of the given type. For example, any Ethernet driver must implement the interface for sending and receiving Ethernet packets. A separate generic specification is needed for each supported OS, as different OSs define different interfaces for device drivers.

Additionally, each particular device can support non-standard features, e.g., device-specific configuration options or transfer modes. These features must be added as extensions to the generic OS specification in order to synthesize support for them in the driver. TSL supports such extensions in a systematic way via the template inheritance mechanism. This is described in Appendix A.

Running example

Figure 4.4 shows the operating system model. It is written in the form of a test harness that simulates all possible sequences of driver invocations issued by the OS. The `os` template in Figure 4.4 shows the OS model for our running example. The main part of the model is the `psend` process. At every iteration

```
43 /* OS model */
44 template os
45
46     uint8 dat;
47     bool inprogress, acked, success;
48
49     /* driver workload generator */
50     process psend {
51         forever {
52             dat = *; /*randomise dat*/
53             inprogress = true;
54             acked = false;
55             drv.send(dat);
56             wait(acked);
57         };
58     };
59
60     /* I/O completions */
61     controllable void send_ack(bool status) {
62         assert (!inprogress && !acked && status == success);
63         acked = true;
64     };
65
66     /* virtual callbacks */
67     void evt_send(uint8 v) {
68         assert (inprogress && v==dat);
69         inprogress = false;
70         success = true;
71     };
72
73     void evt_send_fail(uint8 v) {
74         assert (inprogress && v==dat);
75         inprogress = false;
76         success = false;
77     };
78
79     /* The goal */
80     goal idle_goal = acked;
81 endtemplate
```

Figure 4.4: Trivial serial controller driver specifications.

of the loop, it non-deterministically chooses an 8-bit value (line 52) and calls the `send` method of the driver, passing this value as an argument. It then waits for the driver to acknowledge the transmission of the byte (line 56) before issuing another request. The driver acknowledges the transmission via the `send_ack` callback (line 61). The callback sets the `acked` flag, which unblocks the `psend` process.

We keep the specification concise by modeling the state of the driver-OS interface, as opposed to the internal OS state and behavior. For example, the `acked` variable (line 61) serves to model the flow of data between the OS and the driver and is not necessarily present in the OS implementation.

Connecting device and OS models

In addition to simulating I/O requests to the driver, the OS model also specifies the semantics of each request in terms of device-internal events that must occur in order to complete the requested I/O operation. In our running example, after the OS invokes the `send` method of the driver and before the driver acknowledges completion of the request, the device must attempt to send the requested data over the serial line. This requirement establishes a connection between the device and OS models and must be specified explicitly in order to enable Termite to generate a driver implementation that correctly handles the OS request. Note that we only need to specify *which* hardware events must occur, but not *how* the driver generates them.

In order to develop such specifications, we need a way to refer to relevant state and behavior of the device from the OS model. At the same time, in order to maximize specification reuse, we would like to keep the OS specification device-independent. To reconcile these conflicting requirements, we introduce a *virtual interface* between the device and OS model. This interface consists of callbacks used by the device model to notify the OS model about important hardware events. The virtual interface does not represent real runtime interactions between the device and the OS, but serves as part of the correctness specification.

We define a virtual interface for each class of devices. Such *device-class* interfaces are both device and OS-independent. The device-class interface can be extended with additional device-specific callbacks as required to specify a driver for a particular device.

Running example

TODO: device class source

In our example, we define a device-class interface consisting of two virtual callbacks: `evt_send` and `ev_send_failed`, invoked respectively when the device successfully transmits and fails to transmit a byte. These callbacks are invoked in lines 30 and 34 of the device model. The `evt_send` handler is shown in line 67 of the OS model. The assertion in line 68 specifies that the send event is only allowed to occur if there is an outstanding send request in progress and the value being sent is the same as the one requested by the OS. We reset the `inprogress` flag to false in line 69, thus marking the

```
82 /* Driver template */
83 template drv
84
85     void send(uint8 v){
86         ...;
87     };
88
89     /*
90     void irq(){ (see Section 4)
91         ...;
92     };
93     */
94 endtemplate
```

Figure 4.5: Trivial serial controller driver specifications.

current request as completed; line 70 sets the `success` flag to true, thus indicating that the transfer completed without an error. The `evt_send_fail` handler is identical, except that it sets the `success` flag to false. The flags are checked by the `send_ack` method, which asserts that the driver is only allowed to acknowledge a completed request (`!inprogress`) that has not been acknowledged yet (`!acked`) and that the completion status reported by the driver must match the one recorded in the `success` flag.

In this example we use C-style assertions to rule out invalid system behaviors. Assertions alone do not fully capture requirements for a correct driver behavior. For example, a driver that remains idle does not violate any assertions. Hence, we need to specify requirements for the driver to make forward progress. We introduce such requirements into the model in the form of *goal conditions*, that must hold *infinitely often* in any run of the system. For example, a goal may require that the driver is infinitely often in an idle state with no outstanding requests from the OS. The OS can force the driver out of the goal by issuing a new I/O request. To satisfy the goal condition, the driver must return to the goal state by completing the request. Line 80 in Figure 4.4 defines such a goal condition that holds whenever the `acked` flag is set, i.e., the driver has no unacknowledged send requests.

Driver template

Running example

Figure 4.5 shows the driver template for the running example consisting of a single `send` entry point invoked by the OS. The ellipsis in line 86 represent a

location for inserting synthesized code and are part of TSL syntax. We refer to such locations as *controllable placeholders*.

4.3 TSL compiler

In order to compute the most general driver strategy as a solution of a two-player game, we must first convert input TSL specifications into a game automaton. This conversion is performed by the TSL compiler.

Real driver specifications have large state spaces, which cannot be feasibly represented by explicitly enumerating states, as in Figure 4.2. Therefore, in Termite we represent games symbolically (Section 2.7). The state space of the game is defined in terms of a finite set of state variables X, with each state $s \in S$ representing a valuation of variables in X. The TSL compiler introduces a state variable for each TSL variable declared in one of the input templates. In addition, auxiliary state variables are introduced to model the current control location of each TSL process.

We model controllable and uncontrollable actions as valuations of action variables Y_c and Y_u. Transition relations δ_c and δ_u are represented symbolically as formulas over state variables X, action variables Y_c and Y_u, and next-state variables X'.

The TSL compiler splits the input specification into controllable and uncontrollable parts and translates them into controllable and uncontrollable transition relations respectively. The controllable part is comprised of controllable methods that can be invoked by the driver. The controllable transition relation δ_c is computed by rewriting controllable methods in the *variable update form*. Consider, for example, variable `reg_dat` declared in line 2 in Figure **??**. This variable is only modified by the `write_dat` method in line 4. The corresponding fragment of the controllable transition relation in the variable update form is

$$reg\_dat' = \begin{cases} v, & \text{if } tag = write\_dat \\ reg\_dat, & \text{otherwise,} \end{cases}$$

where `reg_dat`′ is the next-state variable representing the value of `reg_dat` after the transition. A special tag variable is used to identify the method being invoked. For example, the specification in Figure **??** has four controllable methods, so tag can take one of the four values $write\_dat$, $write\_cmd$,

$read\_status$ and $send\_ack$. In addition, a separate variable is introduced for each argument of every controllable method (v in this example).

The uncontrollable part of the specification is comprised of TSL processes, which model device and OS behavior. We syntactically decompose each process into atomic transitions. Recall that a process executes atomically until reaching a `wait` statement or a controllable placeholder. Consider the `ptx` process in line 13 in Figure **??**. The process is initially paused in the wait statement. It is scheduled to run when the wait condition holds. It executes the statements in lines 16–22 atomically and stops again in line 15. As part of this atomic transition, the process sets the `reg_cmd` variable to 0 (line 22). This is the only uncontrollable transition that modifies this variable, hence the uncontrollable update function for this variable is defined as follows:

$$reg\_cmd' = \begin{cases} 0, & \text{if } reg\_cmd = 1 \wedge pid = ptx \\ reg\_cmd, & \text{otherwise,} \end{cases}$$

where `pid` is an uncontrollable action variable that models the scheduler's choice of a process to run, and the `reg_cmd` $=$ `1` conjunct corresponds to the wait condition in line 15.

Finally, we need to generate the game objective Φ. In a symbolic representation of the game, goal and fair sets are specified as conditions over state variables that hold for each state in the set. The TSL compiler outputs a goal set B_i for each goal declared in the input specification and a fair set F_i for each `wait` statement. The latter guarantees that every runnable process gets scheduled eventually.

In addition to goal conditions, a TSL specification also contains assertions, which must never be violated. We model assertions using an auxiliary boolean state variable ε, which is set to true whenever an assertion is violated and remains true forever after. We add an extra constraint $\varepsilon = false$ to each accepting set B_i. An assertion violation permanently takes the game out of B_i, and therefore can not occur in any winning run of the game.

4.4 User-guided code generation

The set of input TSL specifications is fed into the Termite synthesis engine, which then automatically computes the most general strategy for the driver. Given a state of the system, the most general strategy determines the set of

all valid driver actions in this state. The most general strategy is used by the Termite code generator to produce a driver implementation in C in a user-guide fashion.

Motivation

Early versions of Termite implemented fully automatic code generation as their only mode of operation. In many cases we found that the tool produced unsatisfactory code, which led to a series of improvements to the code generation algorithm, aimed to generate more compact, user-readable, and efficient implementations. At the same time, we found that many aspects of what is perceived by developers as a good implementation are very hard to formalize, and, even when possible, the effort involved in such a formalization exceeds the effort needed to achieve the desired effect manually.

For example, the interrupt handler logic of most drivers follows the standard structure, where the driver checks every interrupt source in the order of its priority and invokes a separate handler function for every signaled interrupt. Interrupt prioritization and functional decomposition are hard to synthesize automatically without manual guidance.

One might argue that structure and readability are not relevant for synthesized code. In practice, however, if synthesized drivers are to make their way into Linux and other major OSs, they must follow standard coding guidelines adopted by these OSs and be amenable to manual code inspection. Furthermore, human readable code is needed for quality assurance. While automatic synthesis guarantees that the synthesized driver is correct with respect to input specifications, it does not protect against specification defects: despite our best effort to maximize specification reuse and follow good modeling practices, specification defects cannot be avoided altogether. Hence, manual inspection remains an important way to eliminate driver bugs.

In summary, while there exists a potential for further improvement of automatic code generation algorithms, we believe that a truly practical driver synthesis tool must put the user in control of the resulting code and not enforce any particular code structure nor attempt to override user's design decisions.

Workflow

The Termite code generator GUI is similar to a traditional integrated development environment with two additional built-in tools: the *generator* and the *verifier*. The generator works as advanced auto-complete that helps the user to fill the controllable placeholders inside the driver template with code. At any point, the user can invoke the generator to synthesize a single statement or a complete block of code inside a controllable placeholder via a mouse click on the target code location. The user can arbitrarily modify and amend the generated code. However, the generator never modifies user code. Instead it tries to extend it to a complete implementation, which is always possible provided that the existing code is consistent with the most general strategy. The generator currently only allows synthesizing statements after the last control location within a branch. However this restriction is not a conceptual one and will be lifted by ongoing development.

The verifier automatically and on the fly checks that the driver implementation, comprised of a mix of generated and manually written code, is consistent with the most general strategy, thus maintaining strong correctness guarantees that one would expect in automatically synthesized code. The verifier symbolically simulates execution of the system, following the partial driver implementation created so far, and signals the user whenever it encounters a transition that violates the most general strategy.

In the first approximation, the generator algorithm is quite simple: given a source code location, it determines the set of possible system states in this location, picks an action for each state from the most general strategy and translates this action into a code statement. In practice the algorithm uses a number of heuristics to produce compact and human-readable code. In particular, whenever there exists a common action in all possible states in the given location, the algorithm produces straight-line code without branching. These heuristics are described in Section 4.5.

Running Example

When running the generator on our running example, it automatically generates the code in Figure 4.6 for the `send` function (line 85 of the device template, Figure 4.5):

```
1  void send(uint8 v){
2      dev.write_dat(v);
3      dev.write_cmd(1);
4      wait(dev.reg_cmd==0);
5      if (os.success) {
6          os.send_ack(true);
7      } else {
8          os.send_ack(false);
9      };
10 };
```

Figure 4.6: Generated `send` function

This implementation correctly starts the data transfer by writing the value to be sent to the data register and setting the command register to 1. It then waits for the transfer to complete, which is signalled by the device by resetting the command register to 0. Finally, it acknowledges the completion of the transfer to the OS.

Note that the generated code refers to the `dev.reg_cmd` and `os.success` variables. These variables model internal device and OS state respectively and cannot be directly accessed by the driver. This example illustrates an important limitation of Termite—it assumes a white-box model of the system, where every state variable is visible to the driver. Ideally, we would like to synthesize an implementation that automatically infers the values of important unobservable variables. In this case, the value of the command register can be obtained by the driver by executing the `read_cmd` action. Furthermore, the value of the `os.success` variable is correlated with the completion status of the last transfer, which can be obtained by reading the device status register.

While Termite currently cannot produce such an implementation automatically, it implements a pragmatic tradeoff that helps the user build and validate a correct implementation with modest manual effort. The code generator warns the user that the auto-generated code accesses private variables of the device and OS templates. This prompts the user to provide a functionally equivalent valid implementation, replacing the `wait` statement with a polling loop and using the `read_status` method to check transfer status, as shown in Figure 4.7.

The verifier automatically checks the resulting implementation and confirms that it satisfies the input specification.

Note that in this example we have synthesized code that correctly handles device errors. This was possible, as our input device specification correctly

```
1  void send(uint8 v){
2      dev.write_dat(v);
3      dev.write_cmd(1);
4      while(dev.read_cmd()==1);
5      if (dev.read_status()) {
6          os.send_ack(true);
7      } else {
8          os.send_ack(false);
9      };
10 };
```

Figure 4.7: Manually written `send` function

captures device failure modes (namely, transmission failure) and our OS specification describes how the driver must report errors to the OS (via the `status` argument of the completion callback).

In principle, it is also possible to synthesize a driver implementation that handles device and OS failures *not* captured in the specifications: since the synthesis tool knows all possible valid environment behaviors, it can easily detect invalid behaviors and handle them gracefully. Automatic synthesis of such *hardened* device drivers is a promising direction of future research.

The final step of the code generation process translates the synthesized driver implementation to C. This is a trivial line-by-line translation. We expect this translation to become unnecessary in the future as our ongoing work on the TSL syntax aims to make the synthesized subset of TSL a strict subset of C.

Maintaining synthesized code

Device driver development is not a one-off task: following the initial implementation, drivers are routinely modified toimplement additional functionality, adapt to the changing OS interface or support new device features.

The user-guided code generation method naturally supports such incremental maintenance. Since Termite uses TSL as both its input and output language; a completely or partially synthesized driver can be given as input to Termite, along with modified versions of the device and OS models.

A typical maintenance task proceeds in three steps:

1. First, the developer amends device and OS models to reflect the new or changed functionality.

2. Second, they add new methods to the previously synthesized driver, if necessary, and replace existing driver code that is expected to change

with a controllable placeholder.

3. Finally, the user runs Termite to synthesize code for all controllable placeholders.

Termite treats all existing driver code as part of the uncontrollable environment. Hence, if some of the old code is incorrect in the context of the new specifications, this will lead to a synthesis failure, and counterexample-based debugging is used to identify the faulty code, as described in Section 4.6.

Running Example

As an example, we synthesize a new version of the driver for our running example assuming a more advanced version of the serial controller device that uses interrupts to notify the driver on completion of a data transfer. The new device model is obtained by uncommenting line 39 of the device model in Figure 4.3, which invokes the interrupt handler method of the driver after each transfer. The driver template (Figure 4.5) is extended with the `irq` method (line 90). We use the previously synthesized implementation of the `send` method, but manually remove the last two lines, which implement polling, as we want the new implementation to use interrupts instead:

```
1 void send(uint8 v){
2     dev.write_dat(v);
3     dev.write_cmd(1);}
```

Finally, we run Termite on the resulting specifications and use the generator to automatically produce the following implementation of the new `irq` method:

```
1 void irq(){
2     if (os.success) {
3         os.send_ack(true);
4     } else {
5         os.send_ack(false);
6     };}
```

As before, we manually replace the if-condition in the first line with

```
1 if (dev.read_status())
```

This example illustrates how Termite supports incremental changes to the driver by reusing previously synthesized code, while maintaining strong correctness guarantees.

Instrumenting synthesized code

Termite does not automatically instrument synthesized code for debugging, logging, accounting, etc. However, the user can add such instrumentation manually. Termite interprets such code as no-ops and, as with any manual code, never makes any modifications to it.

4.5 Heuristic code generation

Termite acts as an advanced auto-complete by suggesting lines of code for the driver. These suggestions are obtained using the symbolic strategy generated in the synthesis phase. Termite keeps track of the possible set of states that the system may be in at each line of code in the graphical IDE and uses this along with the strategy to generate a suggestion.

Generating a winnng move from an individual state is straightforward. All one has to do is look up the state in the strategy relation and pick any winning label related to that state. Termite's code generator, however, deals in sets of states - the set of states which the system may be in at a particular line - which complicates matters considerably. For example, although every state in the set is winning (an invariant that the code generator enforces), there might not exist a single label that is winning for all of these states. In this case, the code generator uses smarter methods for picking labels or possibly falls back to spliting the current state set in two, as described in the following sections.

Straightford label picking

In the simplest case, there exists a winning action that is available from all states within the current line's state set (denoted $stateSet$). This action will cause all states in $stateSet$ to transition to a state closer to the goal, guaranteeing that if we keep choosing actions in this way we will eventually force execution into the goal. We compute the set of such winning actions using the formula in Equation 4.1.

$$\forall s.\ stateSet \rightarrow strategy \tag{4.1}$$

This returns the set of labels which are part of the strategy for all states in the current state set. If the set is not empty, the Termite IDE the picks one of these at random as the autocompletion suggestion. If it is empty, the Termite

IDE falls back to the more computationally expensive heuristic described in the next section.

Smarter label picking

While the above heuristic is straightforward and cheap, is it often unsatisfactory in practice. As an example, consider a current state set containing two states: s_1 and s_2. Suppose there is no label which is in the winning strategy for both states, but there is a label in the winning strategy for s_1 and it does not take s_2 further away from the goal. Such an action is a good candidate for an auto completion, however it will not be found by the first heuristic.

Our more general label picking algorithm works as follows: we find the greatest distance (N) of any state in $stateSet$ from the goal and then we look up two sets (which were computed and saved during synthesis).

- the set of states at distance N, denoted win_N,
- and, the set of states at distance $N-1$, denoted win_{N-1}.

We define three constraints on the label, all computable symbolically, and pick a label that satisfies all three constraints by taking their conjunction, also symbolically.

When played, the label must:

- keep execution in win_{N-1} for states within $stateSet \cap win_{N-1}$, and
- keep execution in win_N for states within $stateSet \cap win_N$, and
- take at least one state in win_N into win_{N-1}.

These requirements ensure that each time a label is picked in this way, the number of states at the furthest distance to the goal decreases and eventually becomes zero as the number of abstract states in finite, in turn decreasing the maximum distance to the goal, N. Thus, repeatedly picking actions in this way ensures that we eventually get to the goal and is therefore a sound method of picking auto completions. Note that even though execution is guaranteed to reach the goal, the label picked may not be part of the strategy for all states in $stateSet$.

Splitting states

Lastly, suppose that there is no action found by the first two heuristics. In this case we recursively split $stateSet$ by finding a subset $a \subset stateSet$ from which there is a winning action for all of a and splitting this out in C code using an *if-statement*. What remains of $stateSet$ is recursively split enough times that there is a suitable label to play from each partition. This is guaranteed to eventually happen because $stateSet$ in finite.

We describe how the condtion for this if-statement is computed. There exists at least one winning label from each state in the current set, as they are all winning. However, there does not exist a single label which is winning for all states in the current set. Our heuristic to find a subset of $stateSet$ for which there does exist such a label is as follows:

1. Enumerate labels that are winning in some state in $stateSet$.
2. Compute the set where each label is part of the strategy.
3. Compute a small condition over state variables that distinguishes this set from the rest of $stateSet$.
4. Choose the condition that is the simplest and return this and the corresponding lablel.

Enumerating winning labels

In practice, the first step is infeasible, however. Games in Termite have labels over 100 bits in size so this would require enumerating over 2^{100} labels. However, many of these labels are equivalent as certain combinations of bits are dont-care. We use the $availableLabels$ algorithm (Algorithm **??**) to enumerate labels that are equivalent given the current set and transition relation.

This algorithm invokes two other algorithms: $genPair$ (Algorithm **??**) and $enumerate$ (Algorithm **??**), both of which deserve some explanation.

$genPair$, given two sets of variables (X and Y) and a relation, extracts a single valuation of X from the relation, determines the set of valuations of Y that are related to and then determines the set of valuations of X that relate to exactly this set.

$enumerate$, given two sets of variables (X and Y) and a relation repeatedly extracts pairs from the relation using $genPair$, erasing the X valuations from

the relation each time so that they are not chosen again. It computes a set of pairs of sets of X and Y valuations that are related to exactly the other element of the pair.

Finally, $availableLabels$ uses $enumerate$ to enumerate labels with different behaviours in the current $stateSet$. It does this by restricting the transition relation to transitions that originate from states in $stateSet$ and treating this as a relation between $\langle s, s' \rangle$ and l. This works because equivalent labels will result in the same $\langle s, s' \rangle$ relation, i.e. the same state transition, thus calling $enumerate$ with this relation and these variables will enumerate equivalent labels.

---

function GENPAIR(x, y, rel)
  $xMinterm \leftarrow$ EXTRACTMINTERM$(\exists y.\ rel)$
  $img \leftarrow$ SUBSTITUTE$(rel, xMinterm)$
  $genX \leftarrow \forall y.\ reg \leftrightarrow img$
  return $\langle genX, img \rangle$
end function

---

function ENUMERATE(x, y, rel)
  $result = []$
  while $rel \neq False$ **do**
    $\langle genX, img \rangle \leftarrow$ GENPAIR(x, y, rel)
    $result \leftarrow result \oplus \langle genX, img \rangle$
    $rel \leftarrow res \wedge \neg genX$
  end while
  return $result$
end function

---

function AVAILABLELABELS($s, u, l, s', strategy, stateSet$)
  $winning \leftarrow \exists\ s\ u\ s'.\ strategy \wedge stateSet$
  return ENUMERATE$(l, s \cup s', \delta \wedge stateSet \wedge winning)$
end function

Computing the condition

We compute a condition over state variables that distinguishes the states where the label is winning from the rest using the $liCompaction$ function. This function, provided by the CUDD library, minimises a condition BDD given a care set so that it returns the correct value within the care set and is

undefined otherwise. Our condition BDD is the set of states from the winning label is available and our care set is $currentState$. Since we are only applying the if condition in one of the states in $currentState$, this is the only set where it needs to be accurate.

We found that a good heuristic is to choose the label which results in the simplest if condition, where a simple condition has a small BDD representation. The size of a BDD can be conveniently calculated using the function $dagSize$ from the CUDD library.

4.6 Counterexample guided debugging

An important practical issue in game-based synthesis is the complexity of diagnosing synthesis failures due to defects in the input specifications. In the event that Termite fails to solve the game, the user needs to trace the failure back to the specification defect. However, the failure does not carry any information about the defect, which makes the problem harder to resolve.

In Termite we propose a new approach to troubleshooting synthesis failures based on the use of *counterexample strategies*. A counterexample strategy is a strategy on behalf of the environment that prevents the driver from winning the game. It is obtained by solving the *dual game*, where, in order to win, the environment must permanently force the game out of one of the goal regions. A winning strategy in the dual game is guaranteed to exist whenever solving of the primary game fails.

By exploring the counterexample strategy, the user can identify the defect in the input specification. This is similar to the use of counterexamples in software verification, where for each discovered bug the verification tool generates a counterexample trace that triggers the bug. However, a counterexample strategy cannot in the general case be represented by a single execution trace, as the choice of spoiling moves for the environment depends on the actions performed by the driver.

In order to detect and fix the defect in an input specification, the driver developer relies on their understanding of the OS and device logic. The role of the counterexample strategy is to guide the developer towards the defect. To automate this process, we developed a powerful visual debugging tool that allows the user to interactively simulate intended driver behavior and observe environment responses to it. The user plays the game on behalf of the

driver, while the tool responds on behalf of the environment, according to the counterexample strategy.

In a typical debugging session, the debugger, following the counterexample strategy, generates a sequence of requests that are guaranteed to win against the driver. The user plays against these requests by specifying device commands that, they believe, represent a correct way to handle the request. Since this sequence of requests *cannot* be handled correctly given the current input specification, at some point in the game the user runs into an unexpected behavior of one of the players, e.g., one of the user-provided commands does not change the state of the device as expected or the environment performs an uncontrollable transition that violates an assertion. Based on this information, the user can revise the faulty specification.

At every step of the interactive debugging session, the debugger either chooses a spoiling uncontrollable action based on the counterexample strategy or, if the system is inside a controllable placeholder, allows the user to choose a controllable action to execute on behalf of the driver. In the former case the spoiling uncontrollable action corresponds to a transition in one of the TSL processes. The user can explore this transition by stepping through it, exactly as they would in a conventional debugger. In the latter case, the user provides the action that they would like to perform by typing and executing corresponding code statements.

The tool supports a number of features aimed at making the debugging process as simple as possible for the user. We mention two of them here:

- First, the debugger interactively prompts actions available to the driver at each step.
- Second, the debugger keeps the entire history of the game and allows the user to go back to one of previously explored states and try a different behavior from there.

4.7 Limitations

The core of a device driver entails translating I/O requests from the OS into sequences of low-level device commands and responses. We focus on synthesizing driver logic that performs these functions, and this is where game-based synthesis really shines, as it helps to implement tedious and error-

prone logic with minimal manual effort and without bugs. However, there are several real world aspects of device driver creation that are not handled as gracefully by the game based approach.

In Section 4.4, we described one limitation of Termite, namely the lack of support for grey-box synthesis. In this section we discuss other limitations, which, we hope, will help define the agenda for continuing research in driver synthesis.

Direct Memory Access

Most importantly, Termite does not currently support automatic synthesis of direct memory access (DMA) management code. Many modern devices transfer data directly to and from main memory, where it is buffered in data structures such as circular buffers and linked lists. These data structures can have very large or infinite state spaces and cannot be easily modeled within the finite state machine-based framework of Termite. Efficient synthesis for DMA requires enhancing the synthesis algorithm to use a more compact representation of DMA data structures, which is the focus of our ongoing research. At this time, code for manipulating DMA data structures must be written manually. This code is not interpreted or verified by Termite. For example, we use this approach to synthesize a DMA-capable IDE disk driver (Section 4.9).

Boilerplate Code

Device drivers in modern OSs contain a significant amount of boilerplate code that is not directly related to the task of controlling the device. This includes binding the driver to I/O resources (memory mapped regions, interrupts, timers), registering the driver with various OS subsystems, allocating DMA memory regions, creating sysfs entries, etc. While much of this functionality could be synthesized within the game-based framework, we do not believe that this is the correct approach. Previous research has demonstrated that this boilerplate code can be generated in a principled way from declarative specifications of the driver's requirements and capabilities [**?**]. This technique has lower computational complexity than game solving and better captures the essence of the task. A practical driver synthesis tool can combine game-based synthesis of the core driver logic responsible for controlling the device

with declarative synthesis of boilerplate code. As a result, the current version of Termite assumes this boilerplate code is written manually as a wrapper around the synthesized driver.

Concurrency

Drivers execute in a concurrent OS environment and must handle invocations from multiple threads, as well as asynchronous hardware interrupts. We separate synthesis for concurrency into a separate step. Drivers synthesized by Termite are correct assuming a sequential environment, where driver entry points are invoked atomically. The resulting sequential driver is then processed by a separate tool that performs a sequence of transformations of the driver source code, which preserve the driver's sequential behavior, while making the driver thread-safe. Such transformations include adding locks around critical code sections, inserting memory barriers, and reordering instructions to avoid race conditions. Concurrency synthesis is still work in progress and is not the subject of this thesis. Preliminary results are published in [Cerny et al., 2013, 2014].

Real time synthesis

Termite does not explicitly support specification and synthesis of timed behaviors. Instead, it uses a pragmatic approach that allows it to synthesize time-sensitive behavior without having to explicitly reason about time. To this end, Termite conservatively approximates timed operations by fairness constraints: it ignores the exact duration of each device operation, but keeps the knowledge that the operation will complete *eventually*, and synthesizes a driver that waits for the completion. Termite is also able to handle time-out conditions, modeled as external events. However, at this time it is not capable of generating device drivers for hard real-time systems, where the driver must guarantee completion of I/O operations by a certain deadline.

4.8 Implementation

We implemented all components of the Termite toolkit, including the TSL compiler, the game solver, the counterexample debugger, the user-guided code generator, and the TSL-to-C compiler, in the Haskell programming

language. Our implementation uses the CUDD BDD library for efficient symbolic manipulations over boolen relations, and the Z3 SMT solver for satisfiability queries over the theory of bit vectors, as described in [Walker and Ryzhyk, 2014]. Termite is publicly available under the BSD license and can be downloaded from the project website `http://termite2.org`. The version of Termite presented here consists of 30,000 lines of code. The estimated overall project effort is 10 person years.

4.9 Evaluation

We evaluate Termite by synthesizing drivers for eight I/O devices. Specifically, we synthesized drivers for a UVC-compliant USB webcam, the 16550 UART serial controller, the DS12887 real-time clock, and the IDE disk controller for Linux, as well as bare metal drivers (which, for example, would run on seL4 [Klein et al., 2009]) for I2C, SPI, and UART controllers on the Samsung exynos 5 chipset[1] and SPI controller on the STM32F10 chipset. With the exception of the IDE disk, these devices are representative of peripherals found in a typical embedded platform, such as a smartphone. Our synthesized drivers implement data transfer, configuration and error handling. The main barrier to synthesizing drivers for more advanced devices, e.g., high-performance network controllers, is the current lack of support for synthesis of DMA code in the current version of Termite.

Modelling complexity

Models of UART and DS12887 devices were developed based on existing publicly available device models [WindRiver; OpenCores]. Models of other devices were derived from their vendor-provided documentation, following standard TLM modeling guidelines [Wind River, 2010]. OS models for the relevant device classes were created based on Linux kernel documentation and source code.

Table 4.1 summarises the size, in lines of code, of device and OS models in our case studies. Developing a complete set of specifications for each driver took approximately one week, of which only one to three days were

[1] At the time of writing, the exynos drivers have not yet been tested due to hardware availability issues; however we confirmed via manual inspection that they implement the same device control sequences as existing manually developed drivers.

	input spec		driver	
	OS	device	synthesized	native
webcam	102	385	113	307
16450 UART	122	167	74	261
exynos UART	128	252	37	166
STM SPI	73	244	24	64
exynos SPI	88	239	40	183
exynos I2C	146	180	79	211
RT clock	118	252	84	183
IDE	188	480	94[a]	474

[a]Excluding 36 lines of manually written code that manipulates the DMA descriptor table.

Table 4.1: Size (in lines of code) of input specifications and of synthesized and equivalent manually written drivers.

spent building the models and the rest of the time was spent studying device and OS documentation. This efficiency can be attributed to the choice of the right level of abstraction and modeling language. In particular, the use of transaction-level device modeling abstracts away complicated internal device machinery by focusing on high-level events relevant to driver synthesis, while the TSL language allows modeling the driver environment using standard programming techniques, as illustrated by our running example.

Interestingly, we found the most error-prone step in developing specifications for driver synthesis to be defining correct relative ordering of OS-level and device-level events with the help of the virtual interface (Section 4.2). Naïve specifications tend to be either too restrictive, leading to synthesis failures, or too liberal, leading to incorrect synthesized drivers. As we gained more experience synthesizing different types of drivers, we identified common modeling patterns that help avoid errors in virtual interface specifications.

As a common example, most virtual interfaces contain callbacks that signal a change to one of the device configuration parameters, e.g., transfer speed, parity, etc. A naïve OS model may only allow such a callback to be triggered when the OS has requested a change to the corresponding device setting. However, many devices only allow setting multiple configuration parameters simultaneously, so that setting any individual parameter triggers multiple callbacks, thus making the specification non-synthesizable. The problem can be rectified by changing the device specification to only trigger callbacks if the new value of the parameter is different from the old one; however this

	vars(bits)	refine-ments	predi-cates	synt. time (s)	verif. time (s)
webcam	128 (125565)	47	192	215	794
16450 UART	81 (407)	65	128	210	464
exynos UART	80 (1185)	54	111	645	82
STM SPI	68 (389)	29	63	67	31
exynos SPI	83 (933)	31	72	25	44
exynos I2C	65 (303)	21	56	45	96
RT clock	92 (810)	25	74	56	127
IDE	114 (1333)	42	105	285	778

Table 4.2: Performance of the Termite game solver.

bloats the device model due to the extra checks. A better solution, used in all our models, is to design the OS specification to allow configuration callbacks to be triggered at any time, provided that the new value of the parameter is equal to the last value requested by the OS.

Synthesis time

Table 4.2 summarises the performance of the Termite game solver in our case studies. The second column of the table characterises the complexity of the two-player game constructed by the TSL compiler from the input specifications in terms of the number of states variables and the total number of bits in these variables. The third column shows the number of iterations of the abstraction refinement loop required to solve the game. The next column shows the size of the abstract game at the final iteration, in terms of the number of predicates in the abstract state space of the game. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The second-last column shows that the Termite game solver was able to find the most general winning strategy within a few minutes in all case studies.

We compared the performance of the Termite game solver against a state-of-the-art abstraction refinement algorithm for games [de Alfaro and Roy, 2007] as well as against the standard symbolic algorithm for solving games without abstraction [Piterman et al., 2006]. In all case studies, the Termite solver was the only one to find a winning strategy within a two-hour limit. We refer the reader to [Walker and Ryzhyk, 2014] for a more detailed performance

analysis of the Termite synthesis algorithm.

The final column of Table 4.2 shows the time that it took Termite to verify a complete driver. Recall that the Termite synthesis algorithm doubles as a verification algorithm and can be used to verify drivers written in TSL. We used complete synthesized drivers containing a combination of manual and automatically generated code as inputs to Termite. We have been able to successfully verify all of our drivers. We also experimented with introducing faults to synthesized drivers. Termite was able to detect these faults and produce correct counterexample strategies. In most cases verification took longer than synthesis. The reason for this is that Termite has not yet been optimized for verification workloads. This is one area for future improvement.

User-guided code generation and debugging

We evaluate the key contribution of this paper, namely the user-guided debugging and code generation technique. Each line of code in a Termite-generated driver originates from one of three sources: it can be

1. synthesized automatically by the tool,
2. developed offline and given to Termite as part of the driver template, or
3. added or modified by the user during an interactive code generation session.

A perfect synthesis tool, capable of generating a complete driver fully automatically while producing code that meets all non-functional requirements, would eliminate the need for manual code altogether. We do not believe that such a tool is feasible in the near future. We therefore explore the tradeoffs that arise when using our current, imperfect, tool. In particular, we would like to empirically characterize situations when the user can rely on the synthesizer to automatically produce near-optimal code, and when they are better off completely or partially implementing certain functionality manually. These tradeoffs are likely to change as the tool improves.

Based on our experience so far, automatic synthesis is most helpful in generating code that performs device configuration or starts a data transfer. This code may involve a long sequence of commands to the device, which must be issued in the right order and with correct arguments. The synthesis

algorithm of Termite proved more effective at doing this than human developers, producing correct code that only requires minimal cosmetic changes in most cases. For example, Figure 4.8 shows a screenshot of Termite with a synthesized implementation of the IDE driver `write()` function, which starts a data transfer to the device. The function writes request parameters into appropriate device data registers and sets bit fields in command registers to prepare the device for data transfer. One deficiency in this auto-generated implementation is that it uses absolute values instead of symbolic constants for bit fields.

As another example of suboptimal synthesized code, consider the following synthesized fragment:

```
1  void packet_received() {
2      if (((packet_data[9:9] == 1) && (packet_data[14:14] == 1))) {
3          os.ack_packet(1,1,packet_data[16:32]);
4      } else if ((dev.packet_data[9:9] == 1)) {
5          os.ack_packet(1,0,packet_data);
6      } else if ((dev.packet_data[14:14] == 1)) {
7          os.ack_packet(0,1,packet_data[16:32]);
8      } else {
9          os.ack_packet(0,0,packet_data[16:32]);
10     };
11 };
```

which can be replaced by an equivalent one-liner:

```
1  os.ack_packet(packet_data[9:9],
2      packet_data[14:14],packet_data[16:32]);
```

While both issues can, and will, be addressed by an improved code generation algorithm, our experience shows that unaccounted corner cases will arise occasionally. Therefore, the ability to manually modify synthesized code without sacrificing correctness is crucial for a practical synthesis tool.

Limitations of Termite are most noticeable in synthesizing interrupt handler code responsible for processing I/O completions. This involves querying device state to determine which operations completed and with what status, reporting results to the OS, and clearing interrupt status registers. Since Termite does not support grey-box synthesis, it can not generate this code automatically and instead produces code that directly accesses device-internal state (see Section 4.4). Termite correctly reports such situations and allows

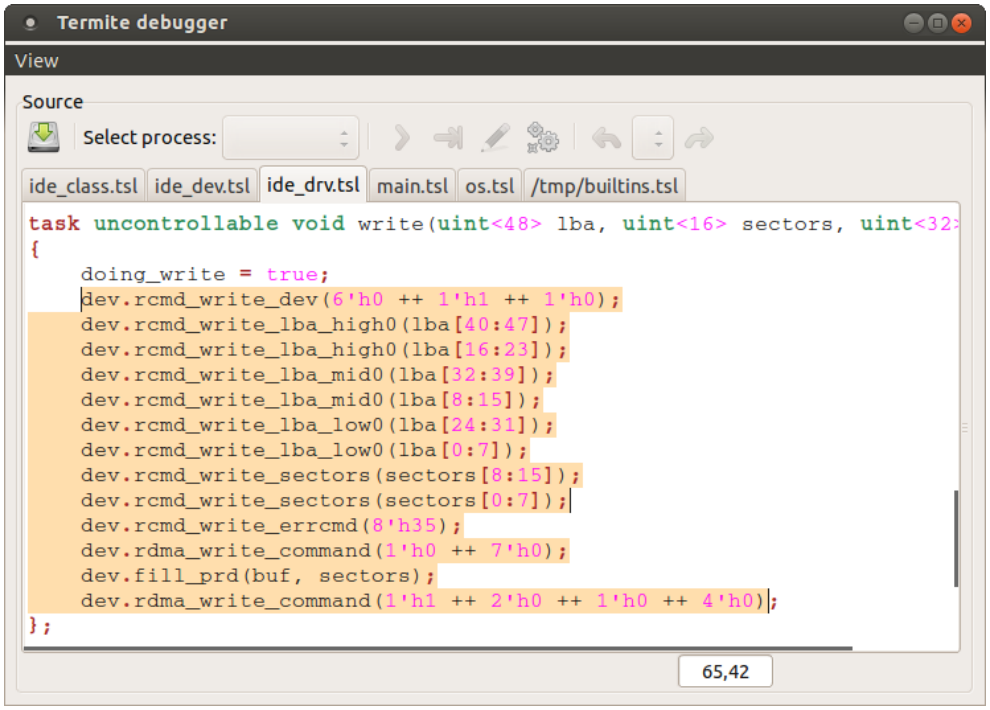


Figure 4.8: Screenshot of Termite with a synthesized implementation of the IDE driver. Automatically generated code is highlighted.

the user to mitigate them by manually editing synthesized code. In practice, however, we found it easier to develop most of the interrupt handler logic offline, as part of the driver template, and rely on Termite to (a) establish correctness of this code and (b) extend it to a complete implementation.

In our case studies, 60% to 90% of the code was generated fully automatically, with the rest of the code produced in a user-guided fashion. Once an initial version of device and OS specifications was ready, it took us several hours to generate the driver implementation for each of our case studies. Three quarters of this time was spent debugging the input specifications, with the rest of it spent generating driver source code with the help of the user-guided code generation GUI.

We found counterexample-driven debugging to be crucial to the productivity of synthesis-based development. Before the debugger was available, we had to rely on code inspection to identify defects in the input specifications, which proved to be a frustrating and unpredictably long process. The Termite debugger streamlines this process, giving us the confidence that any failure

can be localised by following well-defined steps. A typical debugging session takes a few minutes and involves entering only a few commands manually before the defect is localised.

Size of synthesized code

The last two columns of Table 4.1 compare the size of synthesized drivers to existing manually developed drivers. Synthesised drivers are significantly more compact than conventional drivers for two main reasons. First, as explained in Section 4.7, we only synthesize the driver logic directly responsible for controlling the device. Conventional drivers typically contain a large amount of boilerplate code managing various OS resources. We believe that this code can and should be synthesized using complementary techniques. At the moment we implement this functionality manually as a wrapper around the synthesized driver.

Second, conventional device drivers are often designed to support multiple similar devices with slightly different interfaces and capabilities. This leads to code bloat, as the driver must implement multiple versions of various operations, as well as logic to dynamically discover device capabilities and choose the right implementation to use. In contrast, every Termite driver supports one specific device model with a fixed set of features. Drivers for similar devices can share common specification code, but are synthesized as separate source code modules. This approach leads to simpler code and is preferable for platforms with a fixed set of peripheral devices, such as smartphones, where shipping drivers that support only the required devices enables smaller system image.

Specification reuse

Our specification methodology ensures mutual independence of device and OS specifications, and thus facilitates their reuse. We have not yet carried out a substantial evaluation of such reuse; however we report our limited experience based on synthesizing two SPI drivers for the seL4 OS. The corresponding OS specification was initially developed during the work on the SPI driver for the exynos chipset. It was later used to synthesize a driver for the STM32F10 chipset. We were able to reuse most of the original specification. Minor changes (8 lines of code) were required in the part of the

specification describing configuration functionality of the driver, since the STM SPI controller supports a number of ad hoc transfer modes. We expect to observe similar pattern for other devices and operating systems: generic OS specifications can be reused with localized, device-specific changes required to support non-standard device features.

Performance of synthesized drivers

Our synthesized drivers implement effectively identical device control logic to their conventional counterparts and therefore have similar performance. We benchmarked the USB webcam driver, which is the most performance-critical one among our case studies. We measured CPU load and data throughput generated by the conventional and synthesized drivers for varying bitrates. We obtained identical results, modulo measurement errors, for both drivers in all cases.

5 Solving Games Efficiently

We have a formalism for the driver synthesis problem as a game. A practical driver synthesis tool using the game formalism must be able to solve and find strategies for these games for real device and operating system specifications in a reasonable amount of time. The principle challenge of this work is creating a synthesis algorithm that scales well enough to handle the large state machines of real device and operating system specifications.

The straightforward symbolic solver that uses BDDs as the symbolic data structure is remarkably efficient. In fact, it is the current state of the art in reactive synthesis. I will use this as the starting point for my description of Termite's game solver.

I begin by describing my entry to the reactive synthesis competition in 2014, appropriately named "Simple BDD Solver". This solver won the sequential realizability category.

Next, I introduce abstraction, a technique to increase the scalability of the basic symbolic algorithm by reducing the effective size of the state space that the algorithm must operate on. The cost of reducing the size of the state space is that the abstracted game is represented with less precision and is often not precise enough to solve the original game. To counter this, the abstraction needs to be *refined*, i.e. the precision of the abstraction needs to be increased. This is performed in an *abstraction-refinement loop*. Ideally, this loop results in an abstraction just precise enough to solve the game, but coarse enough that solving it is tractable.

I give an abstraction-refinement algorithm that uses *variable abstraction*, a technique for reducing the state space of the game by eliminating a subset of the state variables from the game. I then build on this to arrive at an algorithm that performs *predicate abstraction*, a technique that, instead of representing

the state space using state variables, represents relationships between state variables that capture key properties that are likely to be of importance in solving the game. This representation is usually far more compact. This is the synthesis algorithm that Termite uses.

5.1 Synthesis Competition

The Reactive Synthesis Competition is a competition for reactive synthesis tools inspired by competitions in other fields such as the SAT competition and the Hardware Model Checking Competition. The competition had four tracks:

- Sequential realizability
- Parallel realizability
- Sequential synthesis
- Parallel synthesis

The tools were required to solve safety games given in an extension of the AIGER format [Jacobs, 2014].

Entrants in the synthesis categories were required to produce an implementation of a controller that enforced the safety condition, also given in extended AIGER format. Entrants in the realizability category were only required to determine if the safety game was winnable.

The Synthesis Competition was run as a satellite event to the CAV conference and Kurt Godel medals in silver were awarded to the winners of each category.

Overview

Simple BDD Solver is a substantial simplification of the solver that was developed for the Termite adapted to safety games given in the AIGER format. It performs realizability checking and does not synthesize a strategy. It uses the standard backwards symbolic algorithm to compute the winning set of states as a greatest fixed point. Binary decision diagrams (BDDs) are used to symbolically represent sets and relations. In particular, it computes:

$$\nu X.\forall U.\exists C.\forall S'.((\delta(S,U,C,S') \rightarrow X') \wedge \sigma(S,U,C)) \tag{5.1}$$

where U is the set of valuations of uncontrollable inputs, C is the set of valuations of controllable inputs, and S is the set of valuations of state variables. Given a set X, X' denotes the next state copy of X. σ is the safety condition and δ is the transition relation.

Implementation

The solver is written in the Haskell functional programming language. It uses the CUDD [Somenzi] package for binary decision diagram manipulation and the Attoparsec Haskell package for fast parsing. Altogether, the solver, AIGER parser, compiler and command line argument parser are just over 300 lines of code. The code is available online at: `https://github.com/adamwalker/syntcomp`.

Optimizations

The optimizations that we have used, in approximate order of importance, are:

- dynamic variable reordering using the sifting algorithm [Rudell, 1993]
- partitioned transition relations [Burch et al., 1991]
- direct substitution with CUDD_VECTORCOMPOSE
- dereference unused BDDs as soon as possible
- rearrange the computed formula to avoid creating large BDDs
- simultaneous conjunction and quantification with CUDD_BDDANDABSTRACT
- terminate early where possible
- an abstraction-refinement loop

The optimized algorithm is given in algorithm 12. It calls the optimized controllable predecessor defined in algorithm 11.

Algorithm 11 Controllable predecessor

function CPRE($target$)
  $substituted \leftarrow$ CUDD_VECTORCOMPOSE($target, \delta$)
  $safeSub \leftarrow$ CUDD_BDDANDABSTRACT($C, \sigma, substituted$)
  $winning \leftarrow$ CUDD_BDDUNIVABSTRACT($U, safeSub$)
  return $winning$
end function

Algorithm 12 Simple BDD Solver

function SOLVE($\sigma, init, \delta, C, U$)
  $win \leftarrow$ CUDD_READLOGICONE
  loop
    $res \leftarrow$ CPRE(res)
    $win \leftarrow$ CUDD_BDDLEQ($init, res$)
    if $\neg win$ **then**
      return $False$
    end if
  end loop
end function

Dynamic variable ordering

We do not try to find a good static variable ordering at the start and instead rely on the sifting algorithm provided by the CUDD package for finding good variable orderings dynamically. In our experience, the sifting algorithm provides the best tradeoff between the quality of the resulting ordering and time taken to find it. We enable sifting at the start so that is is active during both compilation and solving. We did not modify any of the default parameters to the sifting algorithm.

Partitioned transition relations

We do not compute the transition relation as a monolithic BDD defined over current state, input variables and next state. This would likely be very large and slow down the algorithm considerably. Instead, we keep it in a conjunctively partitioned form with one partition for each next state variable. We can do this because the next state value of any state variable depends only on the current and input variables and not any other next state variables.

Furthermore, the next state value of any state variable is deterministic. This means that we can represent it directly as a function of the current state

and input variables. We use a BDD defined over current state and input variables to represent this function. Our transition relation becomes a list of BDDs, one for each state variable, each of which only depends on current state and input variables.

To compute the implication, $\forall S'.(\delta(S, U, C, S') \rightarrow X')$, we use the CUDD function CUDD_VECTORCOMPOSE as shown in algorithm 11 on line 1 to substitute each update function into X. This avoids building the monolithic transition relation and, importantly, it avoids having to ever declare a next state copy of each state variable in the BDD manager.

Dereferencing dead BDDs

We dereference BDDs that are no longer needed as soon as possible. Each live BDD node is processed during reordering and counted when the algorithm checks to see if the total BDD size in the manager is reduced. These unused BDDs should not count toward the total node count that is used to evaluate an ordering, and, the time that is spent reordering them and counting them is wasted. In the interest of clarity, BDD dereferencing is not shown in algorithms 11 and 12.

Rearranging the formula

$$\nu X.\forall U.\exists C.(\forall S'.(\delta(S, U, C, S') \rightarrow X') \wedge \sigma(S, U, C)) \tag{5.2}$$

We can arrange equation 5.1 to move the conjunction with the safety condition outside of the innermost universal quantification, as shown in equation 5.2, as the safety condition does not depend on the next state variables. This avoids building the potentially large BDD of the conjunction.

Simultaneous conjunction and quantification

We perform simultaneous conjunction and quantification wherever possible. The existential quantification is performed at the same time as conjunction with the safety condition using CUDD_BDDANDABSTRACT on line 2 of algorithm 11. This avoids building the potentially large BDD representing the conjunction.

Early termination

We terminate early when possible. As we are computing a greatest fixed point, we start with the universal set and progressively shrink it to find the winning region. Each time we shrink the winning region, we check that is it still a superset of the initial set. If it is not, we know there is no way we can win as the winning set only shrinks as the algorithm progresses. We use the function CUDD_BDDLEQ on line 4 of algorithm 12 for this purpose as it allows us to efficiently check that one BDD implies another without constructing the BDD of the implication.

Evaluation

Simple BDD solver failed when the BDDs representing the winning sets, or the intermediate BDDs in the controllable predecessor computation grew too large. Additionally, it failed when a large number of iterations was required to determine the outcome, such as the benchmarks with a large counter.

5.2 Three Valued Abstraction-Refinement

An abstraction is a simplification of the original transition system. An abstraction is used when the game is too large to be solved. Ideally, an abstraction is both small enough to be solved and detailed enough to gain some additional information about the properties of the system.

One common use of an abstraction is in an abstraction-refinement loop. In an abstraction refinement loop, an initial simple abstraction is found and is solved. Then, the results of the abstraction are used to refine the abstraction, ie. to build another system model that contains slightly more detail than the original abstraction. This is repeated in a loop until the original game is solved. It is often possible to solve the original game with a far less detailed abstraction that the original system.

Finding an abstraction that is simultaneously small and useful for making progress in solving the game is a difficult task and is what will be dealt with in the following sections. We start with earlier work on three valued abstraction refinement.

Three valued abstraction refinement

The idea is that given an abstraction, we classify states into one of three categories: winning, losing, and unknown. If we discover that the entire initial set is winning, we know that the original game is winning and we can terminate. Dually, if we discover any initial state that is losing, we know that the entire initial set can never be winning, hence the game is losing and we can terminate.

At termination, either

- all of the initial states are classified as winning (but the other states need not be classified), or
- one of the initial states is classified as losing (again, no other states need to be classified)

This additional imprecision often allows us to use a less precise abstraction compared to the original algorithm where all states are exactly classified. The use of a coarser abstraction is usually computationally more efficient. If none of the termination conditions are met then we need to refine the abstraction. A correct abstraction refinement scheme will guarantee that one of the termination conditions is eventually met after enough refinements are performed. A good abstraction refinement scheme will ensure that when the algorithm terminates the abstraction is not unnecessarily fine.

Abstraction

An abstraction of a game structure G is a tuple $\langle V, \downarrow \rangle$, where

- V is a finite set of abstract states and
- $\downarrow : V \to 2^S$ is the *concretisation function*, which takes an abstract state and returns the possibly empty set of concrete states that the abstract state corresponds to.

We require that

- $\bigcup_{v \in V} v\downarrow = S$, ie. the abstraction covers the entire state space.
- $v_1\downarrow \cap v_2\downarrow = \emptyset$ for any v_1 and v_2, $v_1 \neq v_2$, ie. the abstraction partitions the state space.

In the case when $v\downarrow = \emptyset$ the abstract state v is said to be *inconsistent*. We extend the $\downarrow$ operator to sets of abstract states as follows: for $U \subseteq V$: $U\downarrow = \bigcup_{u \in U} u\downarrow$.

Algorithm

In this section we present a modified version of the three-valued abstraction refinement technique of de Alfaro and Roy [de Alfaro and Roy, 2007]. To simplify the presentation, we focus on solving reachability games.

We start by defining two versions of the abstraction operator: the *may-abstraction* $\uparrow^m$ and the *must-abstraction* $\uparrow^M$. For a set of concrete states $T \subseteq S$:

$$T\uparrow^m = \{v \in V \mid v\downarrow \cap T \neq \emptyset\} \tag{5.3}$$

Intuitively, this returns the set of abstract states which, when concretised, overlap some element of the set to be abstracted.

$$T\uparrow^M = \{v \in V \mid v\downarrow \subseteq T\} \tag{5.4}$$

Intuitively, this returns the set of abstract states which, when concretised, are contained within the set to be abstracted.

We say that an abstraction is *precise* for a set $T \subseteq S$ if :

$$(T\uparrow^m)\downarrow = (T\uparrow^M)\downarrow \tag{5.5}$$

Intuitively, this means that there are no abstract states that, when concretised, contain states that within T and states that are outside T.

Next, we define may and must versions of the abstract controllable predecessor operator:

$$Cpre_i^m(U) = Cpre_i(U\downarrow)\uparrow^m \tag{5.6}$$

Intuitively, this returns the set of abstract states such that, when concretised, we can force execution from at least one concrete state into the concretisation of U.

$$Cpre_i^M(U) = Cpre_i(U\downarrow)\uparrow^M \tag{5.7}$$

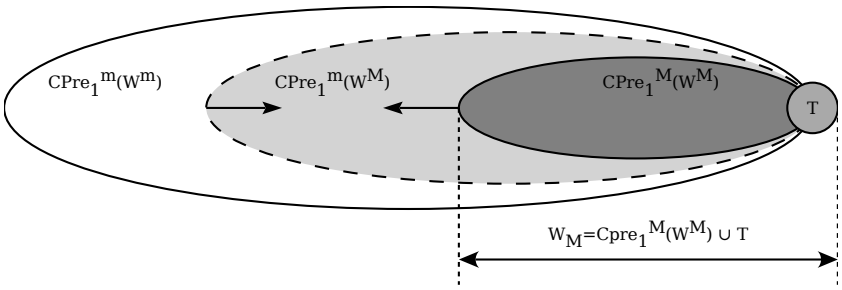


Figure 5.1: Overview of three valued abstraction-refinement

Intuitively, this returns the set of abstract states such that, when concretised, we can force execution from all of the concrete states into the concretisation of U.

These operators have the property:

$$Cpre_i^M(U)\downarrow \subseteq Cpre_i(U\downarrow) \subseteq Cpre_i^m(U)\downarrow \tag{5.8}$$

And therefore:

$$\textsc{Reach}(T\uparrow^M, Cpre_i^M)\downarrow \subseteq \textsc{Reach}(T, Cpre_i) \subseteq \textsc{Reach}(T\uparrow^m, Cpre_i^m)\downarrow \tag{5.9}$$

We denote $\textsc{Reach}(T\uparrow^M, Cpre_i^M)$ as W^M and refer to it as the *must-winning* region and we denote $\textsc{Reach}(T\uparrow^m, Cpre_i^m)$ as W^m and refer to it as the *may-winning* region.

Figure 5.1 illustrates the main idea of three valued abstraction refinement, which is presented in algorithm 13. The winning region, in grey, is overapproximated by the may-winning set in white and underapproximated by the must-winning set in dark grey. The abstraction-refinement loop progressively brings W^m and W^M closer together until the game outcome can be determined.

At every iteration, the algorithm computes the must-winning set W^M that underapproximates, and the may-winning set W^m that overapproximates the true winning set (lines 2–3). The algorithm terminates if the must-winning set contains the entire initial set or the may-winning set has shrunk beyond the initial set (lines 4–5). Otherwise, the algorithm attempts to refine the abstraction in a way that the must-winning set will be expanded in the next iteration when the game is re-solved.

Algorithm 13 Three-valued abstraction refinement for games.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow \rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \text{REACH}(T, Cpre_1)$, and *No* otherwise.

```
1: function SOLVE(transitionRelation, goal)
2:     loop
3:         W^M ← REACH(T↑^M, Cpre_1^M)
4:         W^m ← REACH(T↑^m, Cpre_1^m)
5:         if I↑^M ⊆ W^M then
6:             return Yes
7:         else if I↑^M ⊈ W^m then
8:             return No
9:         else
10:            REFINEABSTRACTION(W^M)
11:        end if
12:    end loop
13: end function
```

To expand the must-winning set we attempt to perform a refinement of the abstraction. This means that we split one or more of the abstract states in V in two.

We observe that if there exists a winning concrete state c then there also exists a chain of winning concrete states from c to the goal. If c is not part of $W^M{\downarrow} \cup goal$, then, some point, this chain must enter $W^M{\downarrow} \cup goal$. Therefore:

1. If there exists some concrete winning state that is not part of $W^M{\downarrow} \cup goal$ then there also exists one with a transition into $W^M{\downarrow} \cup goal$.

We also have the contrapositive:

2. If there does not exist a concrete winning state with a transition into $W^M{\downarrow} \cup goal$ then there exist no further concrete winning states that are not part of W^M.

Thus, we can narrow down our search for abstract states to split to those that have at least one concrete state with a transition into W^M, excluding W^M as these states are already must-winning so splitting them will not achieve anything. Thus, we look in $Cpre_1^m(W^M \cup goal) \setminus W^M$, i.e. the set of all may-predecessors of the must-winning set.

If we find an abstract state within this set for which from a subset of concrete states it is possible to force execution into $W^M{\downarrow}$ then we can partition

this abstract state in two such that those winning concrete states form one partition while the remaining states for the other partition.

The former partition, p, consists entirely of winning concrete states. Furthermore, it is on the boundary of the must-winning set so it will become part of W^M when the game is solved again with the refined abstraction.

Each time a refinement is made, $W^M{\downarrow}$ grows. If the concrete state space is finite $W^M{\downarrow}$ can only grow so much so it must reach a fixed point (that may contain the entire concrete state space). When W^M can no longer grow it is because there are no winning states at the boundary. Thus there are no winning states that are not part of W^M and W^M must be the exact winning set. Furthermore, W^m must be equal to W^M ***TODO: why***. Therefore one of the two termination conditions (lines 5 and 7) must have already happened. Therefore the algorithm terminates.

5.3 Variable Abstraction

We now apply the three valued abstraction-refinement scheme to symbolic games (Section 2.7). We describe an efficient instantiation of the three valued abstraction-refinement scheme for variable abstraction, a technique for reducing the state space of symbolic games by eliminating a subset of the state variables.

The description of three valued abstraction refinement leaves a lot unspecified. In particular

- How is the initial abstraction specified?
- How are the controllable predecessors computed?
- How is the abstraction refined?

For clarity, I have created a running example which I will refer to throughout the description of the symbolic algorithm. The example is given in figure 5.2.

Variable Abstraction

In variable abstraction, the abstract state space is created by dropping a subset of the state variables. The abstraction, α, is defined by the state variables that remain. We denote this set V_α.

```
Goal: X==True ∧ Y==True
Init: X==False ∧ Y==False

a1: X := True;
a2: Y := U;
a3: U := True;
a3: V := True;
```

Figure 5.2: The game specification for our running example

Given the set V_α, the corresponding abstraction (see Section 5.2) is $\langle V, \downarrow \rangle$, where:

- V, the abstract state space, is the Cartesian product of the domains of the variables in V_α.
- $a\downarrow$ is the set of concrete states for which the variables in V_α take the same values in both the concrete state and a and all other variables are free to take any value.

This abstraction satisfies our two requirements for a valid abstraction. ***TODO: do i need to prove this?***

Initial abstraction

The initial abstraction, V_α, for a boolean reachability game is created from only the variables that are mentioned in the goal. This ensures that the initial abstraction is *precise* for the goal set.

The initial abstraction of our running example is illustrated in figure 5.3. The abstract states are illustrated by the solid squares. There is one state for each valuation of X and Y, the variables that occur in the goal in our example. Each abstract state contains four concrete states, one for each valuation of the concrete variables that were dropped from the abstraction, U and V. $\downarrow$ is a function from valuations of U and V to the powerset of the set of valuations of all state variables. For example:

$$\langle T, T \rangle\downarrow = \{\langle T, T, F, F \rangle, \langle T, T, F, T \rangle, \langle T, T, T, F \rangle, \langle T, T, T, T \rangle\} \tag{5.10}$$

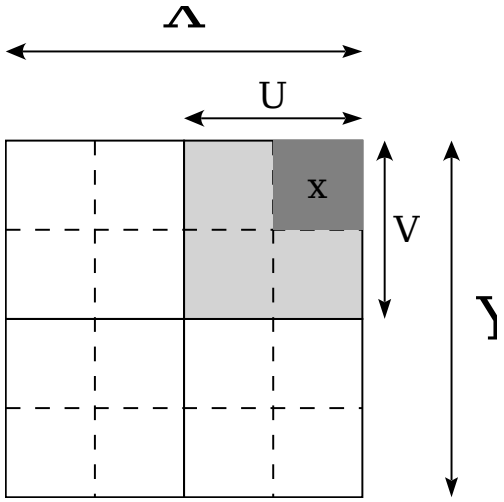


Figure 5.3: Abstract state space

Where the variables in each concrete tuple are ordered $\langle X, Y, U, V \rangle$. Note that X and Y are both True in all concrete state tuples whereas U and V are free to take any value.

It is clear from the figure that the abstraction forms a partition of the concrete state space and that it covers the entire concrete state space.

Controllable predecessor

The controllable predecessor is constructed using the transition relation. For efficiency, the transition relation will be constructed incrementally. As the state variables in our initial abstraction are only the variables in G, these are the only ones we need to compute the transition relation for.

Assuming that the transition relation for the concrete system is given as update functions, one for each concrete variable, then computation of the abstract transition relation is straightforward. We just compile the update functions for only those variables that appear in G.

However, there is the complication that these update functions may depend on additional variables that are not in G. Consider the update function for Y in the running example. The next state value of Y depends on U which is not part of the abstract state space.

To handle this, we introduce an additional set of variables, denoted F (for free, for reasons that will become apparent) for variables that are needed

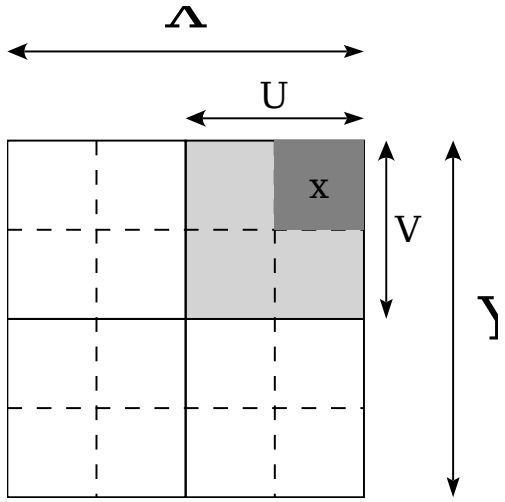


Figure 5.4: Partitioning induced by F

when computing the update functions but are not part of the abstract state space.

We define the two controllable predecessors:

$$Cpre_1^M(X) = \forall F.\exists L.\forall n.TRANS \rightarrow X \tag{5.11}$$

$$Cpre_1^m(X) = \exists F.\exists L.\forall n.TRANS \rightarrow X \tag{5.12}$$

Though the F variables are really part of the state, we treat them as free input variables. Intuitively, their values are chosen by player 2 when computing $Cpre_1^M$ and by player 1 when computing $Cpre_1^m$. This makes it more difficult, and respectively, easier for player 1 to control execution into the target set.

Given a valuation of a subset of the concrete state variables $V \subseteq C$ we say that the valuation *agrees* with a concrete state c if the value assigned to each of the variables in V equals the value assigned to the same variable by c. A subset of concrete state variables defines a partitioning of the concrete state space:

$$\{s \in 2^C | \exists v \in Valuations(V).\forall c \in s.\ c \text{ agrees with } v\} \tag{5.13}$$

We refer to each set of states in the partitioning as a *sub-state*. Figure 5.4 shows the partitioning induced by $F \cup V$.

$Cpre_1^M$, by Equation 5.11, is the set of abstract states from which we can force execution into the target set from all sub-states induced by F. Since no variables in $C \setminus (V \cup F)$ can influence the next state of variables in V *we can ignore them* and conclude that $Cpre_1^M$ computes the set of abstract states such that for all concrete states within the abstract state, player 1 can force execution into the concretisation of the target set. This agrees with the definition of the abstract must controllable predecessor in Equation 5.7.

Dually, $Cpre_1^m$ computes the set of abstract states such that for some concrete state within the abstract state, player 1 can force execution into the concretisation of the target set. This agrees with the definition of the abstract may controllable predecessor in Equation 5.6.

TODO: running example

Solving the game

Now that we have an abstraction and a definition of the controllable predecessor we can solve the game. We first solve the game using $Cpre_1^M$ and denote the winning region W^M, or the *must* winning region. We also solve the game using $Cpre_1^m$ and denote the winning region W^m or the *may* winning region.

Since W^M is a subset of the true winning region, if $I \rightarrow W^M$ then we may terminate as the game is surely winnable. As W^m is a superset of the true winning region, if $I \not\rightarrow W^m$ then we may terminate as the game is surely not winnable.

If none of the above termination conditions held then it is because our abstraction is not precise enough and must be refined.

TODO: running example

Refinement

Algorithm 14 Pseudocode of REFINEABSTRACTION

```
1: function REFINEABSTRACTION(W^M)
2:     U^M ← CpreU_1^M(W^M) ∧ ¬W^M
3:     toPromote ← ω⃗ ∩ SUPPORT(SHORTPRIME(U^M))
4:     PROMOTE(toPromote)
5: end function
```

As explained in Section 5.2, when refining the abstraction of a reachability game, we can limit our search for states to split to the *may-must boundary*. We

split an abstract state in two such that from one of the new states a it is possible to force execution into W^M from every state in $a\downarrow$. Instead of searching each state at the boundary in turn for a candidate state to split as was implied in section 5.2, we find a candidate state using an efficient symbolic calculation.

Moreover, this symbolic calculation finds a state to split that is heuristically good.

We define the function $Cpre_1^F$ that returns the set of sub-states in the partition induced by the variables in $V_\alpha \cup F$ that are winning:

$$Cpre_1^F(X) = \exists L.\forall N.TRANS \rightarrow X \tag{5.14}$$

Note that this differs from Equations 5.11 and 5.12 in that is never quantifies out F. Thus, it returns the set of $\langle S, F\rangle$ tuples that are winning.

We compute:

$$CPre_1^F(W^M) \wedge \neg W^M \tag{5.15}$$

to find the $\langle S, F\rangle$ tuples that are not must winning but have a transition to the must winning set. This is the characteristic formula of the states on the may-must boundary and are thus candidates for splitting.

We extract a large prime implicant from this characteristic function in line 3, an efficient operation with BDDs. We also extract the variables that occur in this prime and call them V_{pi}. $V_{pi} \cap V_\alpha$ defines a set of abstract states, all of which contain both winning and non-winning concrete states and are this refinement candidates. Furthermore, $V_{pi} \cap V_F$???.

Optimisation

There are two straightforward optimisations we can perform to speed up the abstraction refinement loop. The first is critical in practice as it drastically improves the performance of the algorithm.

Reuse W^M from the last abstraction-refinement iteration

W^M is, by definition, the set of states from which we know we can win with the current abstraction. If we refine the abstraction, this set can only grow, so it must at least include W^M from the last iteration. This means that, after refinement when we solve the game again, we may begin iterating the controllable predecessor from the previously found W^M. This effectively

saves us from having to discover again that this set is winning with the new abstraction. The modified algorithm is given in algorithm 15.

Algorithm 15 Three-valued abstraction algorithm optimised to reuse previously discovered winning regions.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow \rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \text{REACH}(T, Cpre_1)$, and *No* otherwise.

```
1:  function SOLVE(transitionRelation, goal)
2:      W^M ← ∅
3:      loop
4:          W^M ← REACH(T↑^M ∨ W^M, Cpre_1^M)
5:          W^m ← REACH(T↑^m ∨ W^M, Cpre_1^m)
6:          if I↑^M ⊆ W^M then
7:              return Yes
8:          else if I↑^M ⊈ W^m then
9:              return No
10:         else
11:             REFINEABSTRACTION(W^M)
12:         end if
13:     end loop
14: end function
```

Do not compute W^m

If we expect the game to be winning, and we are only interested in solving the game to compute the strategy, we may avoid computing W^m entirely. The purpose of computing W^m is to terminate early if our abstraction is precise enough to determine that we cannot win. If we already know that we can win or we expect it is likely that we can win, then it is not worth computing. The modified algorithm is given in algorithm 16. It terminates when it discovers that the game is winning or when it finds that there are no refinements that guarantee that a new winning state will be found. The second termination condition happens when we were wrong and, in fact, the game was not winning. If we had computed a may winning set in addition, we would have discovered this much earlier so this algorithm is not a good choice when there is a reasonable possibility that the game is not winning.

Algorithm 16 Three-valued abstraction refinement for games optimised to not compute W^m

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow \rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \text{REACH}(T, Cpre_1)$, and *No* otherwise.

```
1: function SOLVE(transitionRelation, goal)
2:     loop
3:         W^M ← REACH(T↑^M, Cpre_1^M)
4:         if I↑^M ⊆ W^M then
5:             return Yes
6:         else
7:             res ← REFINEABSTRACTION(W^M)
8:             if res == False then
9:                 return No
10:            end if
11:        end if
12:    end loop
13: end function
```

Summary

- The algorithm categorizes as many states as it can while the abstraction is still simple.
- The transition relation is compiled incrementally on demand.
- The algorithm reuses earlier work by reusing the winning set.

Safety games

Safety games are solved dually to reachability games. The algorithm with the optimisation where the computed winning set is reused is given in algorithm 17.

Arbitrary ω-regular games

We generalise the algorithm to games specified by μ-calculus formulas in prefix normal form, ie, formulas of the form:

$$\nu X.\mu Y. \ldots \phi(X, Y, \ldots) \tag{5.16}$$

or

Algorithm 17 Three-valued abstraction refinement for safety games.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow \rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \text{SAFE}(T, Cpre_1)$, and *No* otherwise.

```
1:  function SOLVE(transitionRelation, goal)
2:      W^M ← ∅
3:      loop
4:          W^M ← SAFE(T↑^M ∧ W^m, Cpre_1^M)
5:          W^m ← SAFE(T↑^m ∧ W^m, Cpre_1^m)
6:          if I↑^M ⊆ W^M then
7:              return Yes
8:          else if I↑^M ⊈ W^m then
9:              return No
10:         else
11:             REFINEABSTRACTION(W^m)
12:         end if
13:     end loop
14: end function
```

Algorithm 18 Pseudocode of REFINEABSTRACTION FOR SAFETY GAMES

```
1: function REFINEABSTRACTION(W^m)
2:     U^m ← ¬CpreU_1^m(W^m) ∧ W^m
3:     toPromote ← ω⃗ ∩ SUPPORT(SHORTPRIME(U^m))
4:     PROMOTE(toPromote)
5: end function
```

$$\mu X.\nu Y. \ldots \phi(X, Y, \ldots) \tag{5.17}$$

where the fixed point quantifiers strictly alternate.

The algorithm begins in the same way as for safety and reachability. We solve the game with the current abstraction to find W^m and W^M on lines 3 and 4 and terminate if these allow us to determine the outcome on lines 6 and 8. We then refine and solve again. Again, as in the reachability and safety cases, we aim to grow W^M and shrink W^m so that, if we do not terminate early, they will eventually become the same set. This guarantees termination through one of the if conditions. Predicate promotion is the same as before but operates on different boundary states. We describe the algorithm to find these boundaries.

Algorithm 19 Three-valued abstraction refinement for μ-calculus games

Input: **Input:** A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a specification formula in prefix normal form ϕ, and an initial abstraction $\alpha = \langle V, \downarrow \rangle$ that is precise for each set that occurs in ϕ, I, and τ_i.

Output: **Output:** *Yes* if $I \subseteq \text{WIN}(\phi, Cpre_1)$, and *No* otherwise.

function SOLVE($transitionRelation$, $goal$)
  loop
    $W^M \leftarrow \text{SOLVE}(\phi, Cpre_1^M)$
    $W^m \leftarrow \text{SOLVE}(\phi, Cpre_1^m)$
    if $I\uparrow^M \subseteq W^M$ **then**
      return Yes
    else if $I\uparrow^M \not\subseteq W^m$ **then**
      return No
    else
      REFINEABSTRACTION(W^M)
    end if
  end loop
end function

Algorithm 20 Pseudocode of REFINEABSTRACTION for μ-calculus games

function REFINEABSTRACTION(ϕ)
  if $\phi = \nu X.\psi$ **then**
    $W^m \leftarrow \overline{\text{SOLVE}(\phi, Cpre_1^m)}$
    $U^m \leftarrow \overline{CpreU_1^m(W^m)} \wedge W^m$
    if $U^m \neq False$ **then**
      $toPromote \leftarrow \vec{\omega} \cap \text{SUPPORT}(\text{SHORTPRIME}(U^m))$
      PROMOTE($toPromote$)
    else
      REFINEABSTRACTION($\psi[X = W^m]$)
    end if
  else if $\phi = \mu X.\psi$ **then**
    $W^M \leftarrow \text{SOLVE}(\phi, Cpre_1^M)$
    $U^M \leftarrow CpreU_1^{M-}(W^M) \wedge \overline{W^M}$
    if $U^m \neq False$ **then**
      $toPromote \leftarrow \vec{\omega} \cap \text{SUPPORT}(\text{SHORTPRIME}(U^M))$
      PROMOTE($toPromote$)
    else
      REFINEABSTRACTION($\psi[X = W^M]$)
    end if
  end if
end function

Correctness

Assuming correctness of the $Solve$ function, W^m and W^M will always contain an underapproximation, and overapproximation of the true winning region respectively. The algorithm only terminates through one of the if conditions on lines 6 and 8. If it terminates through the first one, then the initial state is a subset of W^M and thus certainly a subset of the true winning region, so the algorithm correctly returns Yes. If it terminates through the second if condition, then the initial set is not a subset of W^m and thus is certainly not a subset of the true winning region, so the algorithm correctly returns No. Thus the algorithm returns the correct answer at all termination conditions.

Termination

Suppose the specification formula is of the form $\nu X.\psi$, ie. it has a greatest fixed point at its outermost level. We have already calculated the may winning region and we denote this X^m as it is the final value that the X variable takes when solving the may game. We attempt to directly shrink X^m by reconsidering the last application of $Cpre^m$ that yielded X^m and looking for refinements that cause some of the may winning states found by this last iteration to become losing. This amounts to checking the X^m-lose boundary for additional losing states, in the same way safety games are refined, which also happens to be specified with a greatest fixed point.

We redo the last $CPre$ application as follows. We evaluate $\phi(X, Y, Z, ...)$ with each fixed-point-quantifier variable substituted as X^m, ie. $X = X^m$, $Y = X^m$, ... as these are the values that the fixed point variables had in the last iteration when the game was solved. This happens because the mu-calculus formula is in prefix normal form. We then use this value as the target and refine states and consistency relations as described previously. We re-solve if we succeed.

Making refinements only as described above does not guarantee that eventually $W^m = W^M$. We refine recursively as follows. We define a new objective: $\mu Y., ..., cpre(\phi(X = X^m, Y, ...))$, ie. we drop the $\nu X.$ quantifier and replace X by X^m and refine recursively with this. Note that $Y^m \neq Y^M$ as otherwise X^m would equal X^M and we would have terminated. Conceptually, we are trying to either grow Y^M to X^m ($= Y^m$) through repeated refinement, proving that $X^M = X^m$ (and terminating with an answer somewhere along the way),

or find a reason why Y^M does not equal X^m (finding this is equivalent to shrinking Y^m, and hence X^m) and continue, having achieved our goal of bringing X^m closer to X^M. One of the two outcomes (growing Y^M to X^m or shrinking $Y^m = X^m$) must happen because they are not equal initially and (by structual induction on the mu-calculus formula, assuming the algorithm is correct for shorter formulas, with safety and reachability as the base cases) must meet somewhere in the middle.

Note, that any refinements found in some step would have been found in a subsequent step had that step been skipped. We find that giving priority to the outermost fixed point results in better abstractions and as refinements for outer fixed points are cheaper to compute it makes sense to prioritise them.

Every recursive call drops one fixed point quantifier, so eventually we reach a formula of the form $QX.\phi(X)$ where X is either μ or ν and proceed as in the safety or reachability case. Termination is guaranteed for these, and, by induction for the rest of the specification.

GR(1) games

GR(1) games are a specific case of the above where the formula is:

$$\nu X.\mu Y.\nu Z.Cpre((U \wedge Z) \vee (G \wedge X) \vee Y) \tag{5.18}$$

Thus, the variable abstraction algorithm is correct and terminates for GR(1) games.

5.4 Predicate abstraction

Predicate abstraction has proved to be a particularly successful technique in model checking [Graf and Saïdi, 1997]. Predicate abstraction partitions the state space of the game based on a set of predicates, which capture essential properties of the system. States inside a partition are indistinguishable to the abstraction, which limits the maximal precision of solving the game achievable within the given abstraction but greatly improves the scalability of the synthesis algorithm. As with variable abstraction, if the abstraction is too coarse, it is iteratively refined by introducing new predicates.

The key difficulty in applying predicate abstraction to games is to efficiently solve the abstract game arising at every iteration of the abstraction

refinement loop. This requires computing the abstract *controllable predecessor* operator efficiently. This involves enumerating concrete moves available to both players in each abstract state, which can be prohibitively expensive.

We address the problem by further approximating the expensive controllable predecessor computation and refining the approximation when necessary. To this end, we introduce additional predicates that partition the set of actions available to the players into *abstract actions*. The controllable predecessor computation then consists of two steps:

1. computing abstract actions available in each abstract state
2. and, evaluating controllable predecessor over abstract states and actions

The first step involves potentially expensive analysis of concrete transitions of the system and is therefore computed approximately. More specifically, solving the abstract game requires overapproximating moves available to one of the players, while underapproximating moves available to the other [**?**]. The former is achieved by allowing an abstract action in an abstract state if it is available in at least one corresponding concrete state, the latter allows an action only if it is available in all corresponding concrete states. We compute the overapproximation by initially allowing all actions in all states and gradually refining the abstraction by eliminating spurious actions. Conversely, we start with an empty underapproximation and add available actions as necessary.

We build on the three valued variable abstraction-refinement scheme already developed.

Running Example

We introduce our running example, where we aim to synthesise a driver for an artificially trivial I/O device. This example also motivates the need for predicate abstraction when synthesizing device drivers. The device contains 32 bits of non-volatile memory, which can be accessed from software via the data register. The task of the driver is to transfer a data value from the main memory to the device memory.

We set up a game between the driver (player 1) and the device (player 2). Device and driver internal state is modelled using state variables (Figure **??**). The player who makes the next move is determined by the value of the bsy flag inside the device. When the flag is set to 0, the device remains idle and the

Figure 5.5: Game variables

Var	Type	Description
State variables (X)		
mem	$int32$	Device memory
dat	$int32$	Data register
bsy	$bool$	Device busy bit
req	$int32$	Value to write to mem
Label variables (Y)		
val	$int32$	Value to write to dat
$write\_en$	$bool$	Perform a write on this transition

driver may perform an action, such as a write to the data register. When the flag is set to 1, the device is performing an internal operation and no actions of the driver will have any effect.

The argument of the write is modelled by the val label variable. When the $write\_en$ label variable is set to true and the bsy flag is false, a write operation happens on the next transition. The write operation flips the bsy flag to 1. This triggers a device transition at the next round of the game, when it is player 2's turn, which copies the value in the data register to memory. The objective of the game on behalf of player 1 is to reach the target set $T = (req = mem)$, i.e., the device memory must store the requested value req (Figure **??**). We require that the game is winnable from any state, hence $I = \top$.

The winning strategy for player 1 in this example is to write the value of req in the first transition (by setting $val = req$), thus forcing the device to copy this value to memory at the second transition.

Figure **??** specifies the transition relation δ of the game in the form of variable update functions $x' = t_x(X, Y)$, one for each variable $x \in X$. Consider the update function for bsy as an example. The variable switches to $true$ if the device is not currently busy and the driver requests a write by setting $write\_en$. It switches back to $false$ on the next transition when the memory write operation completes.

Definitions

We instantiate the three-valued abstraction refinement scheme for predicate abstraction instead of simple boolean variables. Consider a symbolic game $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$ defined over state variables X and label variables Y. Let

Figure 5.6: Game specification

(a) Turn functions, initial and target sets

$\tau_1 = (bsy = false)\ \ \tau_2 = (bsy = true)\ \ I = \top\ \ T = (req = mem)$

(b) Variable update functions

$$dat' = \begin{cases} val, & \text{if } \neg bsy \wedge write\_en \\ dat, & \text{otherwise} \end{cases}$$

$$bsy' = \begin{cases} true, & \text{if } \neg bsy \wedge write\_en \\ false, & \text{if } bsy \end{cases}$$

$$mem' = \begin{cases} dat, & \text{if } bsy \\ mem, & \text{otherwise} \end{cases}$$

$$req' = req$$

$\Sigma \subseteq \mathcal{F}(X)$ be a finite set of boolean predicates over the state variables. We refer to Σ as *state predicates*. We introduce boolean variables $\vec{\sigma} = (\sigma_1 \dots \sigma_n)$ to represent values of predicates Σ. Given a boolean variable σ, $\|\sigma\|$ denotes its corresponding state or label predicate. $\|\vec{\sigma}\|$ denotes the vector of all state predicates in Σ.

The state space V of the abstract game is defined as $V = \mathbb{B}^n$, where each abstract boolean state vector $v \in V$ represents a truth assignment of variables $\vec{\sigma}$. The concretisation function $\downarrow$ from Section 5.2 can be expressed as:

$$v{\downarrow} = \bigwedge_{i=1..n} \|\sigma_i\| = v_i \tag{5.19}$$

which maps an abstract state v into the set of concrete states such that each predicate in Σ evaluates to true or false depending on the value of the corresponding element of v.

Example. Consider an abstraction of the running example game induced by abstract variables σ_1, σ_2 and corresponding predicates: $\|\sigma_1\| = (\texttt{req} = \texttt{dat})$, $\|\sigma_2\| = (\texttt{req} = \texttt{mem})$. Consider an abstract state $\texttt{v} = (\texttt{true}, \texttt{false})$. We compute $\texttt{v}{\downarrow} = ((\texttt{req} = \texttt{dat}) = \texttt{true} \wedge (\texttt{req} = \texttt{mem}) = \texttt{false})$ or equivalently $\texttt{v}{\downarrow} = (\texttt{req} = \texttt{dat} \wedge \texttt{req} \neq \texttt{mem})$. Hence v represents the set of all concrete states where conditions $(\texttt{req} = \texttt{dat})$ and $(\texttt{req} \neq \texttt{mem})$ hold for concrete state variables mem, req, and dat. □

Figure 5.7: Abstract variables and corresponding predicates

a.var	predicate
state predicates	
σ_1	$req = dat$
σ_2	$req = mem$
untracked predicates	
ω_1	$bsy = false$
ω_2	$req = 5$
label predicates	
λ_1	$val = req$
λ_2	$val = 5$

5.5 Approximate Three Valued Abstraction Refinement

The goal of the following sections is to apply the machinery developed in the previous sections for variable abstraction to predicate abstraction. There are several pitfalls, so we must identify these pitfalls and take them into account.

Motivation

The abstraction-refinement scheme described in Section 5.2 is very general. It makes the assumption that both of the controllable predecessors, $CPre^m$ and $CPre^M$, can be computed precisely and efficiently.

While it was straightforward to compute them when performing variable abstraction on a boolean game, it is not clear that they can be computed efficiently when performing predicate abstraction.

Example. Consider the label abstraction induced by abstract label variables λ_1 and λ_2 and the corresponding label predicates $\|\lambda_1\| = (\texttt{val} = \texttt{req})$ and $\|\lambda_2\| = (\texttt{val} = \texttt{5})$.

When performing variable abstraction, we would have treated λ_1 and λ_2 as independent variables. Each player was able to set them to any value in any state. However, when performing predicate abstraction, they are not independent, as they share the `val` variable. Suppose player 1 tries to set them both to true. This can only be performed when `req` $=$ `5`. This additional state

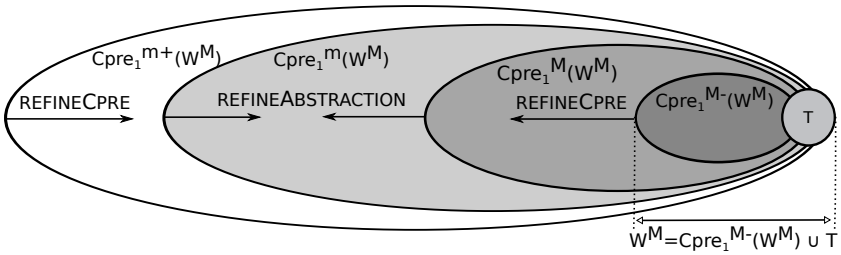


Figure 5.8: Overview of approximate three valued abstraction-refinement

predicate is needed to keep track of which label predicate combinations are valid. □

This is just one of the potential pitfalls of handling predicate abstraction in the same way as variable abstraction.

Algorithm

Following [**?**], we modify the algorithm to work on approximate versions of the controllable predecessor operators, $CPre^{m+}$ and $CPre^{M-} :: 2^V \rightarrow 2^V$. In the following sections we will show how to efficiently compute these operators using techniques similar to variable abstraction.

We lose precision when approximating the controllable predecessors, just as we do when approximating the state space with standard three valued abstraction-refinement. Thus, we need to introduce another form of refinement. This form of refinement refines the controllable predecessor operators themselves. This process is shown graphically in Figure 5.8. Conceptually, refining the controllable predecessors shrinks the gap between $CPre^M$ and $CPre^{M-}$ or the gap between $CPre^m$ and $CPre^{m+}$. We also perform refinement of the abstraction as before.

We require that, for all $U \subseteq V$ we have:

$$CPre^m \subseteq CPre^{m+} \tag{5.20}$$

and

$$CPre^{M-} \subseteq CPre^M \tag{5.21}$$

With these operators we create a new approximate abstraction-refinement scheme given in Algorithm 21.

Algorithm 21 Approximate three-valued abstraction-refinement

Input: **Input:** A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow, Cpre_1^{m+}, Cpre_1^{M-} \rangle$ that is precise for T, I, and τ_i.

Output: **Output:** *Yes* if $I \subseteq \text{REACH}(T, Cpre_1)$, and *No* otherwise.

```
1: function SOLVE(transitionRelation, goal)
2:     loop
3:         W^M ← REACH(T↑^M, Cpre_1^{M-})
4:         W^m ← REACH(T↑^m, Cpre_1^{m+})
5:         if I↑^M ⊆ W^M then return Yes
6:         else if I↑^M ⊈ W^m then return No
7:         else
8:             refined ← REFINECPRE(W^M)
9:             if (¬refined) REFINEABSTRACTION(W^M) endif
10:        end if
11:    end loop
12: end function
```

Correctness

If the algorithm terminates then it returns the correct answer. By equations 5.20 and 5.21,

$$\text{REACH}(T{\uparrow}^M, CPre^{M-}) \subseteq \text{REACH}(T{\uparrow}^M, CPre^M) \tag{5.22}$$

and

$$\text{REACH}(T{\uparrow}^m, CPre^m) \subseteq \text{REACH}(T{\uparrow}^m, CPre^{m+}) \tag{5.23}$$

thus, by Equation 5.9:

$$W^M{\downarrow} = \text{REACH}(T{\uparrow}^M, CPre^{M-}){\downarrow} \subseteq \text{REACH}(T, CPre) \subseteq \text{REACH}(T{\uparrow}^m, CPre^{m+}){\downarrow} = W^m{\downarrow} \tag{5.24}$$

Thus, if we terminate on lines 5 or 6 then we terminate with the right answer.

Termination

TODO: Do I just handwave? I need some properties of refineCpre and refineAbstraction

5.6 Abstraction-Refinement for Predicate Abstraction

The goal of the following sections is to create a symbolic version of algorithm 21 adapted to predicate abstraction.

Abstract Transition Relations

Following the three-valued algorithm presented in Section 5.2, we would like to find an efficient way to compute over- and under-approximations $Cpre^{m+}$ and $Cpre^{M-}$ of the abstract controllable predecessor operators. Recall that computing $Cpre^{m}$ and $Cpre^{M}$ precisely is expensive, as it requires applying the controllable predecessor operator to the concrete transition relation δ. We approximate this costly computation by computing the controllable predecessor over the *abstract transition relation* instead. The abstract transition relation of the game is defined over boolean predicate variables and therefore can be manipulated much more efficiently than the concrete one.

We construct the abstract transition relation via efficient syntactic analysis of the concrete transition relation δ. We present the construction assuming that δ is given in the variable update form, as in Figure **??**b. A similar construction is possible for specifications written in real-world hardware and software description languages.

For each state predicate in Σ, we compute the update function by replacing concrete variables in the predicate with their corresponding update functions. We then transform the resulting formula into a boolean combination of atomic predicates over concrete state and label variables.

Example. Let us compute the update function for abstract variable σ_1 (Figure **??**d). Using update functions for req and dat variables (Figure **??**c), we obtain:

$$\sigma_1' = (req' = dat') = \neg(bsy = false) \wedge (req = dat) \\ \vee (bsy = false) \wedge (val = req) \quad (5.25)$$

This equation contains three atomic predicates: in addition to the existing predicate $\sigma_1 \leftrightarrow (req = dat)$, it introduces new predicates $(bsy = false)$ and $(val = req)$.

The first two predicates correspond to existing state variables σ_1 and σ_2. The last predicate is new; hence it is added to set Ω and a new untracked variable ω_1 is created for it. By substituting predicates in the equation with corresponding abstract variables, we obtain the following abstract transition relation for σ_1 in line 11 of the algorithm: $\sigma_1' = (\overline{\sigma_2} \wedge \omega_1) \vee (\sigma_2 \wedge \sigma 1)$ □

In the general case, the syntactically computed update function for a predicate may depend on existing state predicates in Σ as well as new predicates that are not yet part of the abstraction. The new predicates are partitioned into *untracked predicates* defined over concrete state variables (e.g., `bsy` $=$ `false` in the above example) and *label predicates* that involve at least one concrete label variable (e.g., `val` $=$ `req`). The term "untracked predicate" indicates that these predicates are not part of the abstract state space of the game. Untracked predicates can be seen as partitioning abstract states in V into smaller *untracked sub-states*, as illustrated in Figure **??**.

By substituting untracked and label predicates with fresh boolean variables, $\vec{\omega}$ and $\vec{\lambda}$ respectively, we obtain the abstract transition relation Δ in the form:

$$\vec{\sigma}' = \Delta(\vec{\sigma}, \vec{\omega}, \vec{\lambda})$$

This syntactically computed transition relation contains two sources of imprecision.

- First, untracked variables $\vec{\omega}$ are not part of the abstract state space Σ and are therefore treated as external inputs.
- Second, not all abstract labels are available in all abstract states and hence not all transitions in Δ correspond to a feasible concrete transition. For example, given the set of predicates shown in Figure **??**d, the abstract label $\lambda_1 = true, \lambda_2 = true$ is only available in concrete states that satisfy the condition $req = 5$. In general, given a state-untracked-label tuple $\langle v, u, l \rangle$, the abstract label l may be available in all, some, or none of the concrete states consistent with v and u.

Consistency Relations

We formalise this by introducing *consistency relations* C^m and C^M that over- and under-approximate available abstract labels. A state-untracked-label

tuple $\langle v, u, l \rangle$ is *may-consistent* if the abstract label l is available in *at least one* concrete state consistent with v and u:

$$C^m(v, u, l) = \exists X, Y. \|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l. \tag{5.26}$$

The tuple $\langle v, u, l \rangle$ is *must-consistent* if l is available in *any* concrete state consistent with v and u:

$$C^M(v, u, l) = \forall X. ((\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u) \rightarrow \exists Y. \|\vec{\lambda}\| = l) \tag{5.27}$$

Computing C^m and C^M can be prohibitively expensive. Therefore we use approximations C^{m+} and C^{M-} such that

$$C^m \subseteq C^{m+} \tag{5.28}$$

and

$$C^{M-} \subseteq C^M \tag{5.29}$$

Initially we assign $C^{m+} = \top$ and $C^{M-} = \bot$. Approximations are refined lazily as part of the abstraction refinement process, as explained below.

Example. To illustrate the above definitions, we introduce two label predicates to our running example: $\|\lambda_1\| = (\texttt{val} = \texttt{req})$, $\|\lambda_2\| = (\texttt{val} = \texttt{5})$. Consider the state-untracked-label tuple $\texttt{v} = (\texttt{true}, \texttt{false})$, $\texttt{u} = (\texttt{true})$, $\texttt{l} = (\texttt{true}, \texttt{true})$, which corresponds to the following assignment to abstract variables: $\sigma_1 = \texttt{true} \wedge \sigma_2 = \texttt{false} \wedge \omega_1 = \texttt{true} \wedge \lambda_1 = \texttt{true} \wedge \lambda_2 = \texttt{true}$. It is easy to see that this condition is satisfied for example by the following concrete variable valuation: $\texttt{mem} = \texttt{5}$, $\texttt{dat} = \texttt{5}$, $\texttt{bsy} = \texttt{true}$, $\texttt{req} = \texttt{5}$, $\texttt{val} = \texttt{5}$, hence $\langle \texttt{v}, \texttt{u}, \texttt{l} \rangle$ is may-consistent: $\texttt{C}^{\texttt{m}}(\texttt{v}, \texttt{u}, \texttt{l}) = \texttt{true}$. However, it is not must-consistent:

$$\texttt{C}^{\texttt{M}}(\texttt{v}, \texttt{u}, \texttt{l}) = \forall \texttt{mem}, \texttt{dat}, \texttt{bsy}, \texttt{req}. (((\texttt{req} = \texttt{mem}) \wedge (\texttt{bsy} = \texttt{true}) \wedge (\texttt{req} = \texttt{dat})) \rightarrow \exists \texttt{val}. (\texttt{val} = \texttt{req}) \wedge (\texttt{val} = \texttt{5}))$$

There exist concrete state variable assignments (e.g., $\texttt{mem} = \texttt{1}$, $\texttt{dat} = \texttt{1}$, $\texttt{bsy} = \texttt{true}$, $\texttt{req} = \texttt{1}$) that satisfy state and untracked predicates in the left-hand side of the implication but that can not be extended with a label variable assignment that satisfies the right-hand side, hence $\texttt{C}^{\texttt{M}}(\texttt{v}, \texttt{u}, \texttt{l}) = \texttt{false}$. □

Abstract Controllable Predecessor

We compute over- and under-approximations of the controllable predecessor operator by resolving the two sources of imprecision in favour of one of the players. In particular, we compute $Cpre_i^{m+}$ by (1) allowing player i to pick assignments to untracked predicates, (2) over-approximating consistent labels available to i, and (3) under-approximating consistent labels available to the opponent player $\bar{i}$:

$$\begin{aligned} Cpre_i^{m+}(\phi) = \exists\vec{\omega}.\ \tau_i{\uparrow^M} \wedge \exists\vec{\lambda},\vec{\sigma'}.((C^{m+} \wedge \Delta) \wedge \phi')\ \vee \\ \tau_{\bar{i}}{\uparrow^M} \wedge \forall\vec{\lambda},\vec{\sigma'}.((C^{M-} \wedge \Delta) \rightarrow \phi') \end{aligned} \tag{5.30}$$

This formula has a similar structure to the definition of the concrete controllable predecessor operator (Equation **??**). It replaces the concrete transition relation δ with the abstract transition relation Δ restricted with consistency relations (C^{m+} and C^{M-}). In addition, it existentially quantifies untracked variables $\vec{\omega}$, i.e., an abstract state v is a may-predecessor of ϕ if at least one of its untracked sub-states is a may-predecessor of ϕ.

Dually, we compute $Cpre_i^{M-}$ by (1) allowing the opponent player $\bar{i}$ to pick values of untracked predicates, (2) under-approximating labels available to i and (3) over-approximating labels available to $\bar{i}$:

$$\begin{aligned} Cpre_i^{M-}(\phi) = \forall\vec{\omega}.\ \tau_i{\uparrow^M} \wedge \exists\vec{\lambda},\vec{\sigma'}.((C^{M-} \wedge \Delta) \wedge \phi')\ \vee \\ \tau_{\bar{i}}{\uparrow^M} \wedge \forall\vec{\lambda},\vec{\sigma'}.((C^{m+} \wedge \Delta) \rightarrow \phi') \end{aligned} \tag{5.31}$$

Note that the use of C^{m+} and C^{M-} in (Equation 5.30) and (Equation 5.31) under-constrains moves available to player i and over-constrains moves available to the opponent. The formula for $CpreU_i^{M-}(\phi)$ is analogous, except that it over-constrains moves available to i and under-constrains moves available to $\bar{i}$.

Equations 5.30 and 5.31 suggest two possible abstraction refinement tactics, which correspond to the two types of refinement used in Algorithm 13. First, we can refine C^{m+} and C^{M-} by removing spurious transitions from C^{m+} or adding new consistent transitions to C^{M-}. Such a refinement increases the precision of controllable predecessor computation without introducing new state predicates, which corresponds to the REFINECPRE operation in the algorithm. Second, we can add some of the untracked predicates to the set of state predicates Σ, thus reducing the imprecision introduced by treating them

as external inputs. This refinement increases the precision of the abstraction, which corresponds to the REFINEABSTRACTION function in the algorithm.

In summary, we solve the abstract game by decomposing potentially expensive computations into four types of light-weight operations performed on demand, as required to improve the precision of the abstraction:

- Computing the abstract transition relation Δ via light-weight syntactic analysis of the concrete game
- Computing consistency relations C^{m+} and C^{M-} by iteratively identifying spurious and consistent transitions
- Iteratively refining the abstraction used in the syntactic computation of the transition relation
- Solving the abstract game using abstract controllable predecessor operators (Equation 5.30) and (Equation 5.31)

The computational bottleneck in this method can arise either from having to perform an excessive number of refinements or if abstractions generated by the algorithm are too complex. Our refinement procedures, described below, are designed to avoid such situations by heuristically picking refinements that are likely to speed up the convergence of the algorithm.

Consistency Refinement

Figure **??** illustrates the main idea of the consistency refinement algorithm. It shows an abstract state v (Figure **??**a) at the may-must boundary whose untracked substates u_1, u_2, and u_3 have C^{m+}-consistent transitions to the must-winning set W^M, but none of these transitions is consistent with C^{M-}. The REFINECPRE algorithm attempts to precisely categorise these substates as must-winning or must-losing.

Since all untracked substates of v are either must-losing or may-winning but not must-winning, it is impossible to split out a must-winning subset of v purely by predicate promotion; hence a consistency refinement is needed.

The consistency refinement algorithm proceeds in one of three ways:

- In Figure **??**b, the algorithm identifies the abstract transition $\langle v, u_1, l_1 \rangle$ as spurious and eliminates it from C^{m+}, thus making the u_1 sub-state must-losing.

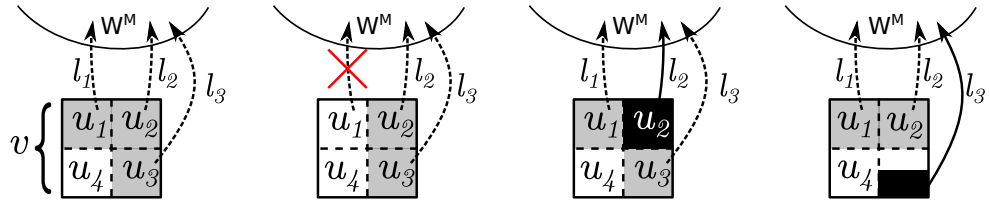


Figure 5.9: Different types of consistency refinements. White, grey, and black background is used to mark respectively must-losing, may-winning, and must-winning untracked substates. Dashed and solid arrows show C^{m+} and C^{M-}-consistent abstract transitions.

- Alternatively, it may detect that abstract transition $\langle v, u_2, l_2 \rangle$ is available in all concrete states in u_2 and thus add this transition to C^{M-}, making the u_2 sub-state must-winning (Figure **??**c).

- Finally, it may determine that abstract transition $\langle v, u_3, l_3 \rangle$ is available in some, but not all, concrete states in u_3, i.e., $\langle v, u_3, l_3 \rangle \in C^m \setminus C^M$. It then partitions u_3 into two or more subsets, exactly one of which has a C^{M-}-consistent transition to W^M, by introducing new untracked predicates (Figure **??**d).

In all three cases, further refinement via untracked predicate promotion becomes possible. Such refinement is performed by the REFINEABSTRACTION function described in Section **??**.

Note that in the special case when, after performing consistency refinement, all untracked substates of v become must-winning or must-losing, the entire state v can be removed from the boundary region without performing predicate promotion. This corresponds to the two types of refinement labelled as REFINECPRE in Figure **??**.

For this reason, Algorithm 13 recomputes W^M and W^m after each successful consistency refinement, and only calls REFINEABSTRACTION once no more consistency refinements are possible.

Consistency Refinement Algorithm

Algorithm 22 shows the pseudocode of REFINECPRE. Lines 3–6 compute the set of candidate tuples $\langle v, u, l \rangle \in C^m \setminus C^M$. Note that for player i states we

Algorithm 22 Pseudocode of the REFINECPRE function

```
1:  function REFINECPRE(W^M)
2:                                          ▷ player i may-winning transitions
3:      T_i ← τ_i↑^M ∧ C^{m+} ∧ ¬C^{M−} ∧ ∀σ⃗'.(Δ → (W^M)')
4:                                          ▷ player ī may-spoiling transitions
5:      T_ī ← τ_ī↑^M ∧ C^{m+} ∧ ¬C^{M−} ∧ ∃σ⃗'.(Δ ∧ ¬(W^M)')
6:      T ← T_i ∨ T_ī
7:      if T = ⊥ then return false          ▷ no refinement is possible
8:      else
9:          choose ⟨v, u, l⟩ ∈ T
10:         F ← (‖σ⃗‖ = v ∧ ‖ω⃗‖ = u ∧ ‖λ⃗‖ = l)
11:         if SATISFIABLE(F) then
12:             A ← ELIMINATEQUANTIFIERS(∃Y.‖λ⃗‖ = l)
13:             A ← MASSAGE(A)
14:             P ← atomic predicates in A
15:             ω⃗ ← ω⃗ ∪ (fresh variables for predicates in P \ Ω)
16:             Ω ← Ω ∪ P
17:             Â ← A[‖x‖ | x, for all x ∈ Ω ∪ Σ]
18:             Â ← replace atomic predicates in A with boolean
                  vars, introducing fresh vars when necessary
19:             C^{M−} ← C^{M−} ∨ (Â ∧ λ⃗ = l)
20:         else
21:             C^{m+} ← C^{m+} ∧ ¬UNSATCORE(F)
22:         end if
23:         return true
24:     end if
25: end function
```

consider may-consistent transition to W^M, whereas for player $\bar{i}$ states we consider spoiling transitions to $V \setminus W^M$.

Line 9 picks a single refinement candidate $\langle v, u, l\rangle$ from the set. By construction we know that $\langle v, u, l\rangle \in C^{m+}$. Since C^{m+} is an overapproximation of C^m, we check whether $\langle v, u, l\rangle \in C^m$, i.e., whether v, u, and l satisfy Equation **??**. To this end, in line 11 we invoke a decision procedure for the underlying theory to check satisfiability of the formula:$(\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l)$.If the formula is unsatisfiable, then $\langle v, u, l\rangle$ is a spurious transition that must be eliminated from C^{m+}. Furthermore, by extracting an unsatisfiable core of the formula,we obtain an inconsistent subset of its conjuncts ($\bigwedge \|\alpha_i\| = c_i$), $\alpha_i \in \vec{\sigma} \cup \vec{\omega} \cup \vec{\lambda}$,which represents a potentially large set of similar spuriousstransitions. We eliminate all of these transitions from C^{m+} inline 16.

If, on the other hand, the formula is satisfiable, then there exists a concrete state-label pair consistent with $\langle v, u, l\rangle$. In this case we want to precisely characterise the set of states where label l is available, so that we can either add $\langle v, u, l\rangle$ to C^{M-} (as in Figure **??**c) or refine it with additional untracked predicates (as in Figure **??**d).

Line 12 computes the set of concrete states where abstract label l is available by performing quantifier elimination from formula $(\exists Y.\|\vec{\lambda}\| = l)$, resulting in a quantifier-free formula A over concrete state variables X. We assume that the underlying theory supports quantifier elimination, which is the case for many practically relevant theories, including the theory of fixed-size bit vectors supported by our tool. In line 13, the resulting formula A is decomposed into atomic predicates possibly introducing new untracked and label predicates. By replacing all atomic predicates in A with corresponding boolean variables, we obtain a formula $\hat{A}$ that describes the set of all state-untracked pairs must-consistent with the abstract label l. Line 14 refines C^{M-} with the set of newly discovered must-consistent transitions.

Example. Assume that in line 9 the algorithm picks a tuple $\langle \mathtt{v}, \mathtt{u}, \mathtt{l}\rangle$ where $\mathtt{l} = (\mathtt{true}, \mathtt{true})$. Line 12 performs quantifier elimination from the formula $\exists \mathtt{val}.(\|\lambda_1\| = \mathtt{true} \wedge \|\lambda_2\| = \mathtt{true}) = \exists \mathtt{val}.(\mathtt{val} = \mathtt{req} \wedge \mathtt{val} = 5) = (\mathtt{req} = 5)$, i.e., conditions $(\mathtt{val} = \mathtt{req})$ and $(\mathtt{val} = 5)$ can only hold simultaneously if $(\mathtt{req} = 5)$. We have discovered a new predicate $\mathtt{req} = 5$ that must hold in states where abstract label $\mathtt{l}$ is available. We introduce a new untracked variable ω_2, $\|\omega_2\| = (\mathtt{req} = 5)$ and refine $\mathtt{C}^{\mathtt{M}-}$ with a new consistent transition: $\mathtt{C}^{\mathtt{M}-} \leftarrow \mathtt{C}^{\mathtt{M}-} \vee (\omega_2 \wedge \lambda_1 \wedge \lambda_2)$. □

Refining the Abstraction

The REFINEABSTRACTION function is invoked by the abstraction refinement algorithm when no further consistency refinements are possible. At this point, every untracked sub-state of the boundary region is either must-winning or must-losing, i.e., can be coloured white or black using notation of Figure **??**. REFINEABSTRACTION promotes a subset of untracked predicates making sure that the winning region W^M expands after re-solving the game in line 2 of Algorithm 13.

The procedure is identical to the refinement procedure for variable abstraction in Section 5.3 so it is not given in detail here.

Algorithm 23 Pseudocode of REFINEABSTRACTION

```
1: function REFINEABSTRACTION(W^M)
2:     U^M ← CpreU_1^{M−}(W^M) ∧ ¬W^M
3:     toPromote ← ω⃗ ∩ SUPPORT(SHORTPRIME(U^M))
4:     PROMOTE(toPromote)
5: end function
```

Algorithm 23 shows the pseudocode of REFINEABSTRACTION. Line 2 computes all untracked boundary substates that are must-predecessors of W^M. Here, $CpreU^{M-}$ is the same as $Cpre^{M-}$ (Equation 5.31), but without untracked variable quantification:

$$\begin{aligned} CpreU_i^{M-}(\phi) =& \tau_i{\uparrow^M} \wedge \exists \vec{\lambda}, \vec{\sigma'}.((C^{M-} \wedge \Delta) \wedge \phi') \ \vee \\ & \tau_{\bar{i}}{\uparrow^M} \wedge \forall \vec{\lambda}, \vec{\sigma'}.((C^{m+} \wedge \Delta) \rightarrow \phi') \end{aligned}$$

We aim to grow W^M by promoting as few untracked predicates as possible. To this end, we extract a short prime implicant from U^M and promote the untracked variables in the support of the prime implicant (line 3). This has the effect of adding a large cube over state and untracked predicates to W^M. The PROMOTE function invoked on line 4 moves the selected untracked predicates to the set of state predicates Σ and recomputes the abstraction transition relation Δ for the new state predicates. This can lead to the introduction of new untracked and label predicates, which can serve as refinement candidates in the future.

We must be careful when refining the abstraction. Do our consistency constraints remain valid after the abstraction has been refined? The following example shows that in some cases they do not.

Example. Consider an abstraction of the running example game induced by abstract variable σ_1 with corresponding predicate $\|\sigma_1\| = (req = dat)$ as well as the abstract label variable λ_1 with corresponding predicate $\|\lambda_1\| = (val = req)$. There are no untracked variables.

Our must consistency constraint, C^{M-}, is a relation over the variables σ_1 and λ_1. For example, the tuple $v = (true), l = (true)$ is must consistent because it is always possible to assert $\lambda_1 = true$ when $\sigma_1 = true$ by choosing val appropriately.

Suppose, when computing the update function for some variable that has just been promoted we introduce the abstract untracked variable ω_2 with

corresponding predicate $\|\omega_2\| = (req = 5)$. This partitions each sub-state into two. The definition of C^{M-} (Equation 5.27) ensures that if a label is must consistent in a given state before the variable promotion then it will be must consistent in each sub-state after the promotion. In this example, the tuples $v = (true), u = (false), l = (true)$ and $v = (true), u = (true), l = (true)$ are both must consistent. Notice that, if we are symbolically representing C^{M-}, then the new C^{M-} does not depend on the newly introduced untracked variable. Thus, we may keep the same C^{M-} from before the promotion.

The situation is not so good for promotions that introduce new label variables. We cannot simply reuse C^{M-} from before the variable promotion as we do for untracked variables. Consider what happens when, continuing on from before, we add λ_2 with corresponding predicate $\|\lambda_2\| = (val = 5)$. The tuple $v = (true), u = (true), l = (true)$ is must consistent. Extending the tuple to $v = (true), u = (true), l = (true, true)$ is fine, but $v = (true), u = (true), l = (true, false)$ is not. The latter tuple is not must-consistent. Therefore, we may not simply reuse C^{M-} as in the case where an untracked variable is added.

We can rebuild C^{M-} by considering all possible tuples with the additional label. In this case as there are few variables this would work and we would discover that the second tuple above is not part of C^{M-}. However, when there are more variables this is infeasible. □

From this example, we make two important observations. When compiling the update function for a untracked predicate that is being promoted to a state predicate when refining the abstraction,

- if the compilation only introduces new untracked predicates, then we can keep the consistency relations from the previous iteration.

- if the compilation introduces new label predicates, then we cannot reuse C^{M-} from the previous iteration without modification.

We resort to conservatively resetting C^{M-} to $false$ when a new label variable is added and incrementally rebuilding C^{M-} lazily as before. This is a severe performance impediment as C^{M-} may have to be rebuilt many times. We present a pragmatic solution to this problem that is used by Termite in Section 5.7.

Checking Containment of the Initial Set

We create an abstraction of the initial set by introducing a boolean state variable for each predicate that appears in the expression for the initial set. We then replace each predicate by the corresponding boolean variable.

The approximate three valued abstraction-refinement algorithm must check whether the winning set contains the initial set. This check is performed for both winning sets in Algorithm 21. Again, predicate abstraction complicates things as the following example shows.

Example. Consider an abstraction of the running example induced by the abstract variables σ_1, σ_2 and σ_3 and corresponding predicates $\|\sigma_1\| = (req = dat)$, $\|\sigma_2\| = (req = mem)$ and $\|\sigma_3\| = (dat = mem)$. The initial set is specified as $req = dat \wedge req = mem$ which is abstracted to $\sigma_1 \wedge \sigma_2$. Suppose that the may winning set of states, W^{m+}, that the game solving algorithm returns is $\{v = (true, true, true)\}$. Importantly, this set does not contain $v = (true, true, false)$. Naively, we might check if the initial set implies the winning set, i.e., if $(\sigma_1 \wedge \sigma_2) \rightarrow W^{m+}$. However, as $\sigma_1 \wedge \sigma_2 \wedge \neg\sigma_3$ is not part of W^{m+}, this implication is false i.e. $(\sigma_1 \wedge \sigma_2) \not\rightarrow (\sigma_1 \wedge \sigma_2 \wedge \sigma_3)$.

We should have ignored the losing state $v = (true, true, false)$ when checking if the winning set consumes the initial set as it is inconsistent and does not correspond to any real states in the concrete game. □

As the example shows, checking if the initial set implies each of the winning sets is not correct. Instead, we find a witness losing state and we then check consistency of this state. The algorithm is given in Algorithm 24.

In line 2 we check if the abstraction of the initial set is a subset of the abstraction of the given winning set. If so, we return True.

If it is not, we extract a witness state on line 6. We check consistency of the state on line 7. If the state is consistent, then we have found at least one losing concrete state. Otherwise, we extract an unsatisfiable core on line 10 and add this to a consistency constraint that prevents this and other states from being selected as witnesses in the future. We then recurse with the updated consistency constraint on line 11. Effectively, we have another refinement loop that iteratively refines a consistency relation over states.

Algorithm 24 Checking inclusion of the initial set

```
1: function REFINEINIT(W, Init, inconsistent)
2:     if Init → W then
3:         return True
4:     else
5:         witnesses ← Init ∧ ¬W ∧ ¬inconsistent
6:         implicant ← PRIMEIMPLICANT(witnesses)
7:         if SATISFIABLE(implicant) then
8:             return False
9:         else
10:            inconsistent ← inconsistent ∨ UNSATCORE(implicant)
11:            REFINEINIT(W, Init, inconsistent)
12:        end if
13:    end if
14: end function
```

Initial Abstraction

We obtain the initial abstraction by extracting atomic predicates from expressions T, and τ_i, which guarantees that the abstraction is precise for T, and τ_i. While this property is not essential for our approach, we will rely on it to simplify the presentation of the algorithm.

The initial state and label predicates are those that are created during compilation of the update functions for these predicates.

Finally, we assign $C^{m+} = \top$ and $C^{M-} = \bot$ initially.

Putting it Together

The Algorithm for three valued predicate abstraction-refinement is given in Algorithm 25. In lines 3 and 4 we compute the must and may winning sets. We then check if the initial set is a subset of the must winning set using the procedure described in Section 5.6 on line 5. If it is, we terminate as the game is winning. We then check if the initial set is a subset of the may winning set using the same procedure on line 7. If it is not, we terminate as the game is losing. Finally, if none of the termination conditions are met, we refine. First, we attempt to refine the consistency constraints on line 10 using the procedure described in Section 5.6. If that fails, we refine he abstraction on line 12 using the procedure described in Section 5.6.

Finally, to arrive at the Termite game solver, we extend the algorithm to GR(1) games in the same way we did for variable abstraction.

Algorithm 25 Three-valued abstraction refinement for games.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow, Cpre_1^{m+}, Cpre_1^{M-} \rangle$ that is precise for T, I, and τ_i.

Output: *Yes* if $I \subseteq \text{REACH}(T, Cpre_1)$, and *No* otherwise.

```
1:  function SOLVE(transitionRelation, goal)
2:      loop
3:          W^M ← REACH(T, Cpre_1^{M-})
4:          W^m ← REACH(T, Cpre_1^{m+})
5:          if REFINEINIT(W^M, I, inconsistentInit) then
6:              return Yes
7:          else if REFINEINIT(W^m, I, inconsistentInit) then
8:              return No
9:          else
10:             refined ← REFINECPRE(W^M)
11:             if (¬refined) then
12:                 REFINEABSTRACTION(W^M)
13:             end if
14:         end if
15:     end loop
16: end function
```

Correctness

The correctness and termination theorems of [de Alfaro and Roy, 2007] hold for Algorithm 13 with REFINECPRE and REFINEABSTRACTION functions defined above.

Theorem 1. If Algorithm 13 terminates, it returns the correct answer.

Proof. By construction, $Cpre_i^{m+}$ and $Cpre_i^{M-}$ over- and under-approximate abstract controllable predecessor operators, i.e., $Cpre_i^m(\phi)\downarrow \subseteq Cpre_i^{m+}(\phi)\downarrow$ and $Cpre_i^{M-}(\phi)\downarrow \subseteq Cpre_i^M(\phi)\downarrow$, for any set ϕ. Hence, winning sets $W^m = \text{REACH}(T\uparrow^m, Cpre_1^{m+})$ and $W^M = \text{REACH}(T\uparrow^M, Cpre_1^{M-})$ computed using these operators over- and under-approximate the winning set W of the concrete game: $W^M\downarrow \subseteq W \subseteq W^m\downarrow$.

If the algorithm returns *Yes* then the initial set of the game is a subset of the must-winning region ($I \subseteq W^M\downarrow$) and hence $I \subseteq W$. Likewise, if the

algorithm returns *No* then $I \not\subseteq W^m{\downarrow}$ and hence $I \not\subseteq W$. In both cases the answer produced by the algorithm is correct.

□

Theorem 2. If there exists a finite region algebra $\mathcal{A}$ such that all abstractions $\langle V, \downarrow\rangle$ produced by Algorithm 13 are contained in $\mathcal{A}$ then the algorithm terminates.

Proof outline. Let W^M and $\hat{W}^M$ be must-winning sets computed at two subsequent iterations of Algorithm 13.

We first show that refinement procedures REFINECPRE and REFINEABSTRACTION guarantee that the must-winning set computed at every iteration of the refinement loop grows monotonically, i.e., $W^M{\downarrow} \subseteq \hat{W}^M{\downarrow}$. This follows from the soundness of the refinement procedures, which improve the precision of $Cpre_i^{M-}$ at every iteration.

Next we show that the algorithm is guaranteed to make forward progress, i.e., after a finite number of refinements it either terminates or discovers new must-winning states ($W^M{\downarrow} \subset \hat{W}^M{\downarrow}$). Consider the consistency refinement procedure REFINECPRE first. Every invocation of this procedure classifies some of the untracked substates at the may/must boundary as either must-winning or must-losing (see Figure **??**). Eventually, it will either classify all boundary states as must-losing, in which case $W^m{\downarrow} = W = W^M{\downarrow}$, and the algorithm terminates, or find at least one must-winning sub-state (as in Figures **??**c and **??**d). In the latter case, a subsequent invocation of the abstraction refinement procedure REFINEABSTRACTION is guaranteed to partition one of the boundary states so that one of the resulting abstract states is must-winning. This state will be discovered at the next run of the reachability algorithm, thus expanding the must-winning set.

Since, by the assumption of the theorem, all must-winning sets W^M generated by the algorithm belong to a finite region algebra, the algorithm is guaranteed to terminate after a finite number of iterations.

□

The theory of fixed-size bit vectors supported by our current implementation satisfies the premise of Theorem 2, which guarantees the termination of the algorithm.

5.7 Optimisation

Algorithm 22 has two important performance issues:

- Every time an untracked variable is promoted, if a new label variable is created in the process, C^{M-} must be reset to False.
- In lines 10–16 it analyses and adds to C^{M-} a single abstract label l. The set of all abstract labels is exponential in size in the number of label predicates and can be very large in practice, making explicit enumeration infeasible.

To address the first issue, we modify the algorithm to handle a set of abstract labels at every iteration. To this end, in line 7, instead of choosing a complete assignment to state, untracked, and label predicates, we compute a *prime implicant* of set T, i.e., an assignment to a subset of variables in $\vec{\sigma} \cup \vec{\omega} \cup \vec{\lambda}$ such that any extension of this assignment to the remaining variables satisfies T:

$$\langle v, u, l \rangle \leftarrow \text{PRIMEIMPLICANT}(T),$$

where v, u, and l are partial valuations of abstract variables, which compactly represent a potentially large set of abstract transitions. In practice, we typically discover prime implicants that only constrain few of the abstract variables, meaning that other predicates are irrelevant for the outcome of the transition.

Given this modification, the C^{m+} refinement case of the algorithm (lines 18–19) is still correct and does not require any changes. However, changes are needed in the C^{M-} refinement logic. The set A computed in lines 10–1 contains all concrete states where *at least one* abstract label from the set characterised by the partial assignment l of label predicates is available. It does not guarantee the availability of any particular label from this set. To model this constraint, we would have to change C^{M-} to be a relation over sets of labels. Every element of the relation would describe a set of labels, one of which is guaranteed to be available for the given state and untracked predicate assignment.

This introduces a new form of imprecision to the consistency relation: rather than categorising each abstract label as available or unavailable in the given state, we record a set of labels, one of which is available. However, such a relation would be hard to represent and manipulate efficiently in

the symbolic form. Therefore, we propose a different approach that allows symbolic implementation.

We transform the game (without changing its winning set) in order to solve it more efficiently. The idea of our solution is to model the new form of imprecision as non-determinism in the abstract transition relation Δ. For each abstract label variable λ_i, we introduce an auxiliary *enabling variable* ε_i and transform the transition relation Δ as follows:

$$\Delta \leftarrow (\varepsilon_i \wedge \Delta) \vee (\overline{\varepsilon_i} \wedge \exists \lambda_i . \Delta).$$

When $\varepsilon_i = true$, Δ behaves exactly as before. In case $\varepsilon_i = false$, the value of λ_i chosen by the player is ignored and the environment non-deterministically selects next-state variables assignment that is consistent with *some* assignment of λ_i.

We can now modify line 16 of Algorithm 22 as follows:

$$C^{M-} \leftarrow C^{M-} \vee (\hat{A} \wedge \bigwedge_{i \in j_1 \ldots j_p} \lambda_i = l_i \wedge \bigwedge_{i \notin j_1 \ldots j_p} \varepsilon_i = false),$$

where $j_1 \ldots j_p$ are indices of variables assigned by l. The above statement refines C^{M-} by allowing the player to assign a subset of label variables in accordance with l and disabling other label variables not constrained by l. The environment will non-deterministically pick arbitrary values for these variables. In this way we precisely capture what we currently know about consistent predicate assignments in a symbolic form, at the cost of introducing extra variables ε_i.

5.8 Implementation

We implemented this abstraction refinement algorithm as part of Termite. Termite currently supports specifications in its own domain-specific Termite Specification Language (TSL), which will be described in Chapter **??**.

We extended our abstraction-refinement algorithm to handle GR(1) games as outlined in Section 5.3.

Termite currently supports input specifications over the concrete domain of fixed-size bit vectors and arrays. We use the Z3 SMT solver to check satisfiability and retrieve unsatisfiable cores of formulas over concrete variables (lines 11 and 21 of Algorithm 22). Quantifier elimination (line 12) over bit

	Statistic	Case study									
		IDE	RTC	UART-1	UART-2	I2C-1	I2C-2	SPI	UVC	simple SPI	si
1	concrete state vars (bits)	83 (952)	64 (624)	61 (335)	65(896)	64 (458)	50(222)	66(644)	95 (75908)	7 (46)	
2	concrete label vars (bits)	27 (389)	24 (199)	20 (86)	15(289)	25 (199)	15(81)	24(384)	33 (49657)	9 (58)	
3	consistency refinements	11	9	42	4	12	4	6	22	0	
4	state refinements	18	16	18	50	15	17	26	25	11	
5	state predicates	31	25	33	58	24	24	31	30	14	
6	label predicates	57	41	40	53	36	32	28	130	19	
7	untracked predicates	7	4	35	2	5	1	6	32	0	
8	run time (s)	71	74	309	603	39	43	14	190	1	
9	peak BDD size	864612	515088	907536	1142596	440482	688828	324996	785918	87892	
Performance of the de Alfaro and Roy algorithm [de Alfaro and Roy, 2007]											
10	run time (s)	∞	∞	∞	∞	∞	∞	∞	∞	865	
11	peak BDD size	-	-	-	-	-	-	-	-	400624	

Table 5.1: Summary of experimental case studies.

vector formulas is performed using our custom implementation of the decision procedure for bit vectors by Barrett et al. [Barrett et al., 1998]. Termite interacts with the theory solver through a well-defined interface and hence can be readily extended with additional theories. All computations over the abstract domain are performed symbolically using the CUDD BDD package.

In addition to the techniques described for variable and predicate abstraction implemented the optimisations described in Section 5.1.

We also implemented an optimised version of the original algorithm by de Alfaro and Roy, as described in Section 5.3, which enables a direct comparison of the two techniques.

The core abstraction-refinement algorithm consists of 1800 lines of Haskell code took approximately 10 person-years to develop.

5.9 Evaluation

We evaluate our synthesis algorithm by synthesising drivers for several real-world I/O devices, including an IDE hard disk, a real-time clock, two UART serial controllers, two I2C bus controllers, an SPI bus controller, and a UVC webcam. We developed corresponding device and OS models using the Termite Specification Language (TSL) by following the common methodology used by hardware developers in building high-level device models. We refer the reader to Chapter **??** for a description of the TSL language and the modelling methodology. The source code of the case studies is available as part of the Termite distribution [Termite].

Table 5.1 summarises our experiments. The first two rows characterise the

complexity of the input models in terms of the number of variables and the total number of bits used in the concrete specification of the game. Concrete state variables model internal device state, as well as the state of the driver-OS interface; label variables model commands and responses exchanged by the driver, the device, and the OS.

Rows 3 and 4 show the number of iterations of the abstraction refinement loop required to solve the game. Rows 5 through 7 show the size of the abstract game at the final iteration, when a winning strategy for the driver was obtained, in terms of the number of state, label, and untracked predicates. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The resulting abstract games are still too complex to solve using explicit state enumeration, hence the use of symbolic techniques is essential. In all case studies, Termite was able to find the winning strategy within 11 minutes running on a 2.9GHz Intel Core i7 laptop (row 8), with peak BDD size under one million nodes (row 9).

The two final rows show the performance of the original three-valued variable abstraction refinement algorithm of de Alfaro and Roy on our benchmarks. As expected, the algorithm does not terminate on any of the real-world driver benchmarks within a two-hour time limit. We therefore developed simplified versions of two of the benchmarks (SPI and I2C-2) with significantly reduced state spaces. As shown in the last two columns of the table, the de Alfaro and Roy algorithm terminates on these benchmarks; however it takes several orders of magnitude longer than our new algorithm, which uses predicate abstraction. These results show that predicate abstraction is essential to solving complex real-world games.

6 Conclusions

Device driver synthesis is a radical alternative to traditional driver development. It has the potential to drastically reduce the work required to create device drivers and improve their reliability.

This dissertation has presented the design and implementation of the Termite driver synthesis tool. Termite is the first tool to marry automatic game-based synthesis with conventional manual development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications.

Based on our experimental results, we consider Termite to be an important step towards truly practical device driver synthesis. In particular, our synthesis algorithm is able to efficiently handle real-world device specifications, while the user-guided approach reliably leads to high-quality code.

In order to develop Termite, we created a rigorous game-based formalism of device driver synthesis. This allowed us to apply pre-existing theory and results from the field of reactive synthesis. We also presented and evaluated a practical predicate-based abstraction refinement algorithm for solving games. To the best of our knowledge, this is the first such algorithm described in the literature. We addressed key performance bottlenecks involved in applying predicate abstraction in game settings and demonstrated that our algorithm performs well on real-world reactive synthesis benchmarks.

Further research is needed to solve the key remaining problems described in Section 4.7, primarily the DMA problem, which poses the main obstacle to synthesis of more complex drivers, and the grey-box synthesis problem, which limits the degree of automation achieved by Termite.

Additional research may explore ways to improve the quality of automatically generated code and thus further reduce the need for user involvement.

This includes performance- and power-aware synthesis. Lastly, our approach may be used for the automatic synthesis of hardened device drivers, i.e., drivers that gracefully handle misbehaving devices [**?**], by detecting behaviour that deviates from the specification.

A TSL Language Reference

A.1 Overview

Static and dynamic namespaces

TSL supports two namespaces: the *static namespace* and the *dynamic namespace*. The static namespace is populated with compile-time objects: *types* and *constants*. The dynamic namespace is populated with runtime objects: *processes*, *variables*, *methods*, and *wires*. Static objects are uniquely identified by their name and syntactic scope. In contrast, runtime objects can be instantiated multiple times within the specification and hence must be referred to relative to their runtime scope.

Templates

Templates are the principal mechanism for managing both static and dynamic namespaces. A template models an entity, such as a device, an OS, or a device driver. It declares a set of static objects (types and constants) and a set of runtime objects. Static objects declared inside a template can be referenced from any part of the specification via the template name. Runtime objects are instantiated together with the template and can be accessed via a reference to a template *instance*.

The following template declares type `word` (static object) and variable `x` (runtime object):

```
template A
  // type declaration
  typedef uint<16> word;
```

```
  // variable declaration
  export word x;
endtemplate
```

The `word` type is globally visible via the `::`-notation as `A::word`. It can be used even if template `A` is never instantiated. In contrast, variable `x` can only be accessed via an instance of `A`.

A template is instantiated inside another template. The only exception is the `main` template, which is implicitly instantiated in the top-level scope. Every complete TSL specification must contain a template called `main`.

In the following example, the `main` template creates an instance of `A`, making its variables accessible from `main` via instance name:

```
template main
  instance A a;

  process pmain {
    // assigning variable x of template instance a.
    a.x = 16'd0;
  }
endtemplate
```

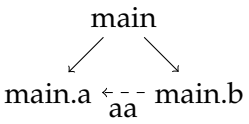
The template instantiation mechanism gives rise to an *instance tree* with the `main` template as its root. Dynamic objects within the tree are accessed using hierarchical identifiers such as `a.x`. This mechanism allows accessing objects down the branch of the instance tree, starting at the local template. In practice, it is often necessary to access objects instantiated in other parts of the tree. This is achieved in a structured way using *template ports*.

A template port is an alias to a template of a given type bound at the time of instantiation. For example, template `B` below declares port `aa` of type `A`, which makes runtime objects in the scope of `A` visible from `B`. Both templates are then instantiated in the `main` template, with the instance of `A` connected to port `aa` of `B`.

```
// template B with port aa of type A
template B(A aa)
  process proc {
    aa.x = 16'd0;
  };
endtemplate
```

```
template main
  // create instances of A and B; connect port aa of
  // B to the instance of A.
  instance A a;
  instance B b(a);
endtemplate
```

The following diagram illustrates the resulting instance tree and the link between different branches of the tree via port `aa`.



Another mechanism for managing name spaces is *template inheritance*. Using inheritance, one can create generic templates that capture common properties of a family of entities, leaving some of the properties underspecified. The generic template can be specialised by a child template that fills in the missing details. For example, the following template models common device-class callbacks that must be implemented by any OS specification for the IDE device class. Note that callbacks are defined without bodies, as the exact behaviour is OS-specific.

```
template ide_os
  procedure void write_sectors(uint<48> lba,
    uint<16> sectors, uint<32> buf, bool xfer_error);
  procedure void read_sectors(uint<48> lba,
    uint<16> sectors, uint<32> buf, bool xfer_error);
  procedure void reset();
  ...
endtemplate
```

This template is specialised by the `l4_ide_os` template, that describes the IDE driver interface defined by the seL4 OS.

```
template l4_ide_os(l4_ide_drv drv)
  // the derive statement is used to establish the
  // inheritance relation
  derive ide_os;

  // additional specification items can be declared in the
```

```
// child template
export iostatus reset_status = ionone;

// the child template implements methods inherited from
// the parent.
procedure void write_sectors(uint<48> lba,
  uint<16> sectors, uint<32> buf, bool xfer_error)
{
  assert (lba == r_lba);
  ...
}
```

A template that has part of its functionality, namely method body or wire assignment, underspecified is called *pure template*. Pure templates can be used in derive statements and in port declarations, however they cannot be instantiated.

Template inheritance is subject to the following rules:

- Derived template must re-declare all ports of all its parent templates with the same names and types. It can also declare additional ports not found in any of its parents.
- If a template has multiple parent templates, then the namespaces of parent templates (consisting of variable, wire, method, goal, and process names) must not overlap.
- If the derived template re-declares a wire or method declared in one of its parents, the new declaration must have the same signature as the parent declaration.
- The child template can override wire and method declarations of its parent templates. Other parts of the parent template, including variable, goal, and process declarations, are inherited by the child template and cannot be overriden.

Execution model

All state transitions in TSL occur in the context of *processes*. Multiple processes can be enabled at the same time; however exactly one process participates in each individual transition. Process transitions are atomic with respect to other

processes. Process that perform the next transition is chosen by the scheduler among enabled processes. Scheduling is fair, i.e., if a process stays enabled sufficiently long, it will eventually get scheduled.

Processes are declared inside templates using the `process` keyword. Such processes become runnable in the initial state of the specification. Additional processes can be spawned at runtime using the `fork` construct. Syntactic constraints of the language (namely, the lack of support for recursion) guarantee that only a bounded number of processes can be spawned at any time.

In the following example, template `A` contains a single static process `psndrcv`, which spawns two subprocesses `psend` and `preceive`.

```
template A

process psndrcv {
  fork {
    psend:    forever send();
    preceive: forever receive();
  };
  shutdown();
};
...

endtemplate
```

A process state transition starts in the current program location and stops at the next *pause location*. Pause locations can be explicit or implicit. Explicit pause locations are declared using `pause`, `wait`, and `stop` constructs. Implicit pause locations are introduced automatically by the compiler in the following cases:

- Before `fork` statements
- Before magic blocks A.1
- On entries to and exits from controllable and uncontrollable tasks (see below)

Process behaviour can be factored into *methods*. A process can invoke methods declared inside its own template as well as in other template instances, via hierarchical identifiers discussed in the previous section. There are three types

of methods in TSL: *functions*, *procedures*, and *tasks*. Functions and procedures must complete instantaneously, i.e., they are not allowed to contain explicit or implicit pauses. In addition, functions are not allowed to have side effects, i.e., they cannot modify any global variables, perform pointer dereferences, or contain assertions (pointer dereferences and assert statements can have the side effect of taking the system into an error state).

Tasks can take time to execute. A task can have an optional `controllable` or `uncontrollable` qualifier. Controllable tasks are the only kind of task that can be invoked from synthesised code; although they can also be called from manually written code. They are used to model the device register interface and OS callbacks. In the current implementation of the TSL compiler, controllable tasks are subject to additional constraint: they are not allowed to contain internal pause locations, i.e., a controllable task must always complete in a single transition.

Uncontrollable tasks represent driver methods invoked by the OS or the device. Uncontrollable tasks are the only kind of task that can contain automatically generated code, i.e., a magic block can only be placed inside an uncontrollable task.

Controllable and uncontrollable task invocations introduce implicit pause locations on entry and return from the task. In the following example execution of process `p1` will consist of three transitions.

```
template A
  uint<16> x;

  process p1 {
    x=1;
    t1(x);
  };

  task uncontrollable void t1(uint<16> arg) {};
endtemplate
```

The first transition assigns global variable `x` and sets the argument of task `t1` to be equal to `x`. An implicit pause location is inserted at this point. The second transition executes the body of task `t1`, which completes instantaneously, as it does not contain any pause locations. Another implicit pause is inserted at the task return location. The remainder of the process is executed in the third

transition. Note that other processes can run and potentially modify variable values in between transitions 1 and 2 and transitions 2 and 3.

Tasks without `controllable` and `uncontrollable` qualifiers behave as if the body of the task was inlined at the call location. No additional pauses are inserted before or after the task.

Variables and wires

Variables are used to store state that persists for the lifetime of the variable. Variables in TSL can be declared in the template, process, or method scope. Template-scope variables, also called *global variables*, are instantiated together with the template and are visible from anywhere inside the template. In addition, variables declared with the `export` qualifier can be accessed from other templates via their hierarchical identifiers. Process and method variables are only visible within the syntactic scope of the process or method where they are declared.

In contrast to variables, *wires* are simply aliases to expressions defined over global variables and are not allocated their own storage. Wires keep their values throughout a transition and are updated at the end of the transition. Initial value of a wire is computed based on initial values of global variables.

```
template A
  uint<16> x = 0;
  wire uint<16> w = x + 1;
endtemplate
```

Correctness specifications

TSL provides several mechanisms for specifying correctness conditions over system behaviour:

- *Goals.* A goal is a side-effect-free boolean expression over global variables that must hold infinitely often in any infinite run of the system. A template can declare any number of goals. In addition, TSL defines two implicit goals. The first one requires the system to be outside of a magic block infinitely often; in other words, the system cannot stay inside a magic block forever. The second implicit goal requires that the system never enters an *error state* (see below).

- *Assertions.* An `assert` statement can be placed anywhere inside processes, tasks, and procedures. It defines a condition whose violation immediately transitions the system to an error state.
- *Postconditions*. Magic block postconditions are discussed in Section A.1 above.

Magic blocks

A *magic block* is a place holder for automatically generated code. TSL supports two types of magic blocks. Magic blocks with *postconditions* specify a condition that must hold upon returning from the magic block.

```
template main
  task uncontrollable bool probe() {
    // Magic block with postcondition
    {...} post((os.reset_status == iosuccess) ||
               (os.reset_status == ioerror));
    if (os.reset_status == iosuccess) {
      return true;
    } else {
      return false;
    };
};
endtemplate
```

The second type of magic blocks are magic blocks with *goals*. These are needed because not all correctness conditions can be captured using postconditions and assertions. For correctness conditions specified using goals (see Section A.1), the synthesis algorithm computes a strategy for each goal; however the scheduling of strategies is left to the user, i.e., the user decides when to execute each strategy by assigning goals to magic blocks.

A.2 Syntax reference

Literals

TSL supports boolean literals "`true`" and "`false`" and Verilog-style binary, octal, decimal and hexadecimal integer literals. The exact number of bits can be specified for each integer literal.

```
<intLit> := <decNumber>
          | [<width>] "'b"  <binNumber>
          | [<width>] "'sb" <binNumber>
          | [<width>] "'o"  <octNumber>
          | [<width>] "'so" <octNumber>
          | [<width>] "'d"  <decNumber>
          | [<width>] "'sd" <signedDecNumber>
          | [<width>] "'h"  <hexNumber>
          | [<width>] "'sh" <hexNumber>
<width> := <decNumber>
```

Examples of integer literals:

```
uint<8> x;
sint<8> y;

x = 255;
x = 8'd35;
x = 8'b01010101;
x = 8'b1;

y = 8'sb11111111;
y = 8'sd-6;
```

In case literal width is not specified explicitly, the compiler assumes width sufficient to encode the given integer value, for example literal 5 is assumed to be 3 bits wide, as 3 bits are sufficient to encode values between 0 and 5. The compiler does not perform automatic truncation or extension of integers. For example, the following assignment statement is invalid, as the left and the right-hand sides of assignment have different width:

```
uint<8> x = 0; // error: x has width 8, while
               // literal 0 has widt 1

uint<8> y = 8'd0; // ok
```

Identifiers

Identifiers are used throughout TSL specification to refer to static and runtime objects (Section A.1). TSL supports three forms of identifiers:

Simple identifiers. A simple identifier is a name of a static or runtime object visible within the current syntactic scope. This includes objects declared within the current scope or in one of its parent scopes. For example, simple identifier `x` used in a method body can refer to a local variable or argument of the method, template-global variable, or a type or constant declared in the template or top-level scope.

```
<ident> := (<letter> | "_") (<letter> | <digit> | "_")*
```

Static scoped identifiers. These identifiers refer to static objects declared within template scope. They can be used to refer to static objects declared in templates other than the local template.

```
<staticIdent> := <ident> "::" <ident>
```

Hierarchical identifiers. These identifiers can only be used to refer to runtime objects. A dynamic identifier is a dot-separated sequence of literals that traverses the instance tree (Section A.1) via port and instance names:

```
<dynIdent> := <ident> ["." <ident>]*
```

The following example illustrates the use of different types of identifiers.

```
template A
  typedef uint<16> word;
  export word x;
endtemplate

template B(A aa)
  // Reference to the word type declared in template A
  // via static scoped identifier;
  A::word y;

  process proc {
    // In the following statement:
    // * y is a simple identifier that refers to a variable
    //   declared in the local template
    // * aa.x is a hierarchical identifier that refers to
    //   variable x declared in template A.
    y = aa.x;
```

```
  };
endtemplate

template main
  // create instances of A and B; connect port aa of
  // B to the instance of A.
  instance A a;
  instance B b(a);
endtemplate
```

Types

Type expressions can occur in various contexts in a TSL specification, including variables, method, wire, type, and constant declarations. The language currently supports arbitrary fixed-width signed and unsigned integers, booleans, enums, structs, arrays, and pointers.

```
<typeSpec>   := ( <sintType>    // signed int
                | <uintType>    // unsigned int
                | <boolType>    // boolean
                | <userType>    // user-defined type name
                | <enumType>    // enumeration
                | <structType>) // struct
                <typeModifier>* // type modifiers

<sintType>   := "sint" "<" <decimalNumber> ">"
<uintType>   := "uint" "<" <decimalNumber> ">"
<boolType>   := "bool"
<userType>   := <staticIdent>
<enumType>   := "enum" "{" (<ident> ",")*  <ident> "}"
<structType> := "struct" "{" (<typeSpec> <ident> ";")+ "}"

// A type modifier is either array dimension or the pointer
// modifier ("*")
<typeModifier> := ("[" <expr> "]")
                | "*"
```

TSL enum's are different from C-style enum's in that they are not integers. In particular, a variable of enum type cannot be cast to an integer. Such a variable can only take one of the values in the enumeration and not any other

arbitrary integer value. Furthermore, an enum declaration cannot assign integer values to enumerators.

Type expressions are subject to the following constraints:

- Enumerations can only be declared in `typedef` statements, e.g., they cannot be used in variable or method declarations. This ensure that every enum type has a name.
- Variable-size arrays are not supported: array dimensions must be compile-time constants.

Examples of type expressions:

```
typedef uint<16> t1;
typedef sint<13> t2;
typedef enum {e1, e2, e3} t3;
typedef struct {t1 f1; t2 f2;} t4;
const t1 c1 = 16'd5;
typedef t4[c1] arrtype;
typedef struct {bool f1; uint<16>[10] f2;} * structptr;
```

Expressions

TSL expressions are constructed from identifiers (Section A.2), literals (Section A.2), and method invocations A.2 using operators summarised in Table A.1 in the order of decreasing precedence.

In addition, TSL supports three kinds of conditional expressions: `if-else` expressions (or ternary expressions), `case`-expressions, and `cond`-expressions.

```
<ternExp> := "if" <expr> <expr> "else" <expr>
<caseExp> := "case" "(" <expr> ")" "{" (<expr>":"<expr>";")*
                 (<expr> ":" <expr> ";")*
                 ["default" ":" <expr> ";"]
           "}"
<condExp> := "cond" "{"
                  (<expr>    ":" <expr> ";")*
                  ["default" ":" <expr> ";"]
             "}"
```

A `case`-expression chooses a value based on the value of its key expression, whereas a `cond`-expression chooses a value to return by evaluating a series

operator	syntax	arg type	res type	comment
`[:]`	`e[l:h]`	integer	unsigned int	bit slice
`[ ]`	`e[i]`	array	-	array index
`.`	`e.f`	struct	-	struct field
`->`	`e->f`	struct pointer	-	struct dereference
`!`	prefix	bool	bool	boolean negation
`~`	prefix	integer	integer	bit-wise negation
`-`	prefix	integer	integer	unary minus
`*`	prefix	pointer	-	pointer dereference
`&`	prefix	any	pointer	address-of
`==, !=`	infix	any	bool	
`<, <=, >, >=`	infix	integer	bool	
`&`	infix	integer	integer	bit-wise and
`|`	infix	integer	integer	bit-wise or
`^`	infix	integer	integer	bit-wise xor
`&&`	infix	bool	bool	boolean and
`||`	infix	bool	bool	boolean or
`=>`	infix	bool	bool	boolean implication
`*`	infix	integer	integer	multiplication
`%`	infix	integer	integer	residue
`+`	infix	integer	integer	plus
`-`	infix	integer	integer	minus

Table A.1: TSL operators

of conditions in order. Note that `if-else` expressions are different from `if` statements described in Section A.2.

Finally, TSL supports struct expressions with explicit or implicit field names:

```
<structExp>   := <staticIdent> "{"
                     (<namedFields> | <anonFields>)
                 "}"
<namedFields> := "." <ident> "=" <expr>
                 [("," "." <ident> "=" <expr>)*]
<anonFields> := <expr>
                 [("," <expr>)*]
```

Note that TSL does not provide type casting operations. In particular, it is impossible to convert a pointer to an integer or an integer to a pointer. In addition, the TSL compiler enforces the following type and memory safety rules:

- No arithmetic operations are allowed on pointers

- Bit-wise operators must be applied to integer operands of the same width
- `==` and `!=` operations can only be applied to operators of identical types (including width and signedness)
- Labels in a `case`-expression and conditions in a `cond`-expression must be side-effect free
- All branches of a `case`, `cond`, or `if-else` expression must return values of the same type
- Lower and upper bounds of a bit slice must be constant expressions, lower bound must be less than or equal to the upper bound, and both bound must be smaller than the width of the argument.
- The address-of operator (`&`) can only be applied to memory variables (see Section A.2).

Non-deterministic expression

Special expression `*` is used to generate non-deterministic values of arbitrary type. The exact type is derived from the context, as illustrated in this example:

```
x = *;  // Generate random value of the same type as x.
f(*,*); // Generate random values that match argument
        // types of f.
```

Note that `*` cannot be used as an atom in a bigger expression, for example, the following is not valid:

```
x = * + 2; // Invalid use of *
```

L-expressions

L-expressions are a subset of TSL expressions that can be used as the left-hand side of assignments statement and as output arguments of methods. L-expressions are defined by the following rules:

1. Names of variables visible in the current scope, including global variables, local variables, and function arguments are L-expressions

2. All valid expressions of the form `*e` and `e->f` are L-expressions.
3. If `e` is an L-expression then the following are also L-expressions:
   - `e.f`
   - `e[i]`
   - `e[l:h]`

Expression evaluation

Expression evaluation order matters in cases when expression operands contain method calls, which may have side effects. All expression operands are evaluated before the expression itself is evaluated. In particular, if the expression contains method calls, then all of these calls are performed before the expression is evaluated.

Statements

Statements are used to describe process behaviour and are used in process and method bodies and always-blocks (Section A.2).

```
<statement> := <varDecl>  // local var declaration
             | <sseq>     // sequential block
             | <spar>     // parallel blocks
             | <sforever> // forever loop
             | <sdo>      // do loop
             | <swhile>   // while loop
             | <sfor>     // for loop
             | <sbreak>   // break out of a loop
             | <schoice>  // nondeterministic choice
             | <spause>   // pause
             | <swait>    // wait on a condition
             | <sstop>    // stop
             | <sassert>  // assertion
             | <sassume>  // assumption
             | <site>     // if-then-else statement
             | <scase>    // case statement
             | <sinvoke>  // method invocation
             | <sassign>  // variable assignment
             | <sreturn>  // return statement
             | <smagic>   // magic block
```

Variable declarations

Variable declaration statements are used to declare local variables visible within the syntactic scope of a process, method or always-block. A local variable is visible everywhere within the process, task, or always-block regardless of the exact location where it has been declared. A variable declaration consists of an optional `mem` qualifier, variable type and name and optional initial assignment:

```
<varDecl> := ["mem"] <typeSpec> <ident> ["=" <expr>]
```

The `mem` qualifier labels the variable as in-memory variable, which allows the address-of operator to be applied to this variable or any of its fields. In addition, in performing pointer analysis of a TSL specification, one can assume that a pointer can only point to an in-memory variable of a matching type.

In case variable declaration does not specify an initial value for the variable, the value for the variable is chosen non-deterministically.

Sequential blocks

C-style sequential blocks; nothing fancy here:

```
<sseq> = "{" (<statement> ";")* "}"
```

Parallel blocks

The `fork` construct is used to spawn several child processes.

```
<spar> := "fork" "{"
                (<ident> ":" <statement> ";")*
          "}"
```

Each child process is assigned a unique label, which helps identify processes during debugging. The spawning process blocks waiting for all forked processes to terminate. Forked processes terminate when all of them have reached a *final* state (i.e., a process cannot terminate without waiting for its siblings to be in final states as well). A process is in a final state when it reaches either the end of its execution (i.e., executes its last instruction) or a `stop` statement (Section A.2).

In the following example two dynamically spawned processes send and receive data in an infinite loop.

```
process psndrcv {
  fork {
    psend:    forever {
                  stop;   // final state
                  send();
              };
    preceive: forever {
                  stop;   // final state
                  receive();
              };
  };
  shutdown();
};
```

At each iteration of the loop, they go through final states by `stop`. When both processes are in their respective final states, the entire `fork` block *may* terminate, with the parent process moving on to the next statement; however, it is not required to terminate and can continue executing `forever` loops. The choice between terminating and continuing execution is performed nondeterministically by the environment. In contrast, in the following example, once both processes reach their final control locations, the fork block terminates instantaneously, as neither process can execute any more transitions:

```
process psndrcv {
  fork {
    psend:    send();
    preceive: receive();
  };
  shutdown();
};
```

Loops

In addition to conventional do, while, and for loops, TSL supports forever loops, which are equivalent to `while(true)`.

```
<sdo>      := "do" <statement> "while" "(" <expr> ")"
<swhile>   := "while" "(" <expr> ")" <statement>
```

```
<sfor>     := "for" "(" [<statement>] ";"
                        <expr> ";"
                        <statement> ")"
                  <statement>
<sforever> := "forever" <statement>
```

At the moment TSL does not allow *instantaneous loops*, i.e., every possible path through the loop body must contain an explicit or implicit pause location (Section A.1). This restriction is enforced by the compiler via a simple static check. Note that there is no implicit pause before the body of the loop, i.e., execution enters the loop instantaneously and continues until reaching a pause location inside the loop. For example, in the following example, process `foo` executes two transitions: (1) `x=16'd0; (x<1) == true; x=x+16'd1`, and (2) `(x<1) == false; y=16'd0`.

```
process foo {
    x = 16'd0;
    while (x < 1) {
        x = x + 16'd1;
        pause;
    };
    y = 16'd0;
};
```

The `break` statement can be used anywhere inside a loop to transfer control to the first instruction following the body of the innermost.

Non-deterministic choice

The `choice` construct allows the environment to non-deterministically choose between two or more actions:

```
<schoice> := "choice" "{"
               (<statement> ";")*
             "}"
```

Pause statements

TSL offers three ways to insert an explicit pause location (Section A.1) in the control flow of a process or method: `wait`, `pause`, and `stop` statements.

```
<swait>  := "wait" "(" <expr> ")"
<spause> := "pause"
<sstop>  := "stop"
```

The `wait` statement inserts a pause location and disables the current process until the wait condition becomes true. If the wait condition stays true for sufficiently long time, the process is guaranteed to eventually leave the pause location. Note that if the wait condition is already true when the process enters the pause location, the current transition terminates anyway.

The `pause` statement is a shortcut for `wait(true)`. `stop` behaves like `pause`, but additionally marks the current process state as final (Section A.2).

Assertions

An assertion behaves as a no-op if its argument evaluates to true, and causes a transition to an error state otherwise.

```
<sassert> := "assert" "(" <expr> ")"
```

The argument of an `assert` statement must be a side-effect-free expression.

Assumptions

```
<sassume> := "assume" "(" <expr> ")"
```

Assumptions are used to constrain possible system behaviours. Similar to assertions, they specify constraint that must hold for any valid execution of the system. However, while an assertion causes transition to an error state if its condition is violated, the `assume` statement prunes all transitions that violate the assumption.

Assumptions are particularly useful in imposing restrictions on randomly generated values, as illustrated by the following example.

```
// assign random values for stopbits and data vars
stopbits  = *;
data      = *;
// only certain combinations of stopbits and data values
// are valid
assume(((stopbits==UART_STOP_BITS_15) && (data==4'd5)) ||
        ((stopbits==UART_STOP_BITS_2) && (data!=4'd5)));
case (data) {
```

```
    4'd5:       data_bits = CUART_DATA5;
    4'd6:       data_bits = CUART_DATA6;
    4'd7:       data_bits = CUART_DATA7;
    4'd8:       data_bits = CUART_DATA8;
    default: assume(false); // other values are not allowed
};
```

Conditional statements

TSL supports C-style `if-else` statements and `case` statements:

```
<site>  := "if" "(" <expr> ")" <statement>
             ["else" <statement>]
<scase> := "case" "(" <expr> ")" "{"
             (<expr> ":" <statement> ";")*
             ["default" ":" <statement> ";"
           "}"
```

Labels of the `case` statement can be arbitrary side-effect-free deterministic expressions of a matching type. At runtime, the first matching label is selected. At most one branch of a case statement is executed; execution does not fall through to the next label automatically. Unlike in C, the `break` statement cannot be used to break out of a case clause. While `break` is allowed inside the body of a `case`, it has the effect of breaking out of the innermost loop.

Method invocation

Method invocations can occur as atoms in expressions as well as standalone statements. Method name is a hierarchical identifier that refers either to a method declared in the local template or exported from another template.

```
<sinvoke> := <dynIdent> "(" [<expr> [("," <expr>)*]] ")"
```

Method arguments must match the number and types of formal arguments in the method declaration. Output arguments A.2 must be L-expressions A.2.

Assignment statements

```
<sassign> := <expr> "=" <expr>
```

The left-hand side of an assignment statement must be an L-expression (Section A.2). Types of left- and right-hand side expressions must match, including sign and width for integer types.

Return statements

```
<sreturn> := "return" [<expr>]
```

Return statements are only allowed in method bodies. If the method is a `void` method, return must not have an argument; otherwise it must have an argument whose type matches the return type of the method. Every path through a non-void method must end with a `return` statement. Statements following a `return` statement are ignored and control transfer to the location immediately following the call site.

If the method is a controllable or uncontrollable task then an implicit pause location (Section A.1) is introduced at the return location. In this case, if the value returned by the task is used as the right-hand side of an assignment, the value of the left-hand side is modified at the beginning of the next transition performed by the process.

Magic blocks

```
<smagic> := "{" "..." "}" ("using" <ident> | "post" <detexpr>)
```

See section A.1.

TSL file structure

A TSL file consists of *import* statements, type declarations, constant declarations, and template declarations.

```
<tslFile>  := <specItem>*
<specItem> := <import>
            |  <typeDecl>
            |  <const>
            |  <template>
```

Import statements

Import statements are used to combine multiple TSL files into a single specification. They are only allowed in the top-level syntactic scope, i.e., they are

illegal inside template of type declarations. An import statement consists of the `import` keyword followed by file path in angle brackets:

```
import<ide_dev.tsl>
import<os/ide_tsl2/l4_ide.tsl>
import<../../os/ide_tsl2/ide_class.tsl>
```

The TSL compiler appends this path to each import directory, specified via the `-I` command line switch, in order, until a file with this name is found.

Type declarations

Type declarations can be placed in the top-level scope or in a template scope.

```
<typeDecl> := "typedef" <typeSpec> <ident>
```

Constant declarations

Constant declarations can be placed in the top-level scope or in a template scope. The value of a constant is an expression that can be evaluated at compile time.

```
<const> := "const" <typeSpec> <ident> "=" <expr>
```

Examples:

```
typedef struct {bool f1; uint<16> f2;} stype;

const bool     b = true;
const uint<16> u = 16'd5;
const stype    s = stype {.f1 = b, .f2 = u};
```

Templates

Templates must be declared within the top-level scope, i.e., nested template declarations are not allowed. Template declaration has the following syntax:

```
<template> := "template" <ident> [(<portDeclarations>)]
                  (<templateItem> ";")*
              "endtemplate"
```

Here, `<portDeclarations>` is a comma-separated list of port declarations, described in Section A.1. A template item is one of:

- Derive statement
- Instance declaration
- Type declaration
- Constant declaration
- Global variable declaration
- Init block
- Always block
- Process
- Method
- Goal
- Wire declaration

Derive statements and instance declarations were considered in Section A.1. Type declarations and constant declarations were considered in Sections A.2 and A.2. We describe the remaining types of template items below.

Global variable declarations

Global variables (Section A.1) are declared in the template scope. The declaration syntax is the same as for local variables (Section A.2) with the optional `export` qualifier that indicates that the variable can be accessed from outside the template:

```
<gvarDecl> := ["export"] <varDecl>
```

Similar to local variable declarations, if the declaration does not specify an initial value for the variable, the value for the variable is chosen non-deterministically. The initial value can be further constrained by `init` blocks, as described below.

Init blocks

An `init` block defines a constraint over initial assignment of global veriables.

```
<initBlock> := "init" <expr>
```

The body of an `init` block is a boolean expression over global variables visible from the current template, including variables exported from other templates.

The following example constrains initial values of `config_in_progress` and `sendq_head` variables:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress;
    uint<16> sendq_head;
    init (config_in_progress == true) &&
         (sendq_head == 16'd0);
endtemplate
```

Note that the same result can be achieved using initial variable assignments:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress = true;
    uint<16> sendq_head = 16'd0;
endtemplate
```

In general, however, `init` blocks are a more general mechanism for constraining initial state, for example the following condition cannot be captures using initial variable assignments:

```
template linux_uart_drv(uart_dev dev)
    init (config_in_progress == true) ||
         (sendq_head == 16'd0);
endtemplate
```

A template can contain multiple `init` blocks. In using `init` blocks, one must make sure that different `init` blocks do not contradict each other and initial variable assignments, leading to an empty initial set, as in the following example:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress = true;
    init config_in_progress == false;
```

```
endtemplate
```

Always blocks

An always block is an arbitrary statement that is automatically prepended to all transitions of the system. It is intended as a low-level mechanism that allows implementing certain behaviours that are tricky to achieve without it. It should be used with care and should probably be hidden behind syntactic sugar.

```
<always> := "always" <statement>
```

Processes

See Section A.1.

```
<processDecl> := "process" <ident> <statement>
```

Methods

A method declaration consists of

- optional `export` qualifier that indicates whether the method can be invoked from outside its template
- method category specifier that labels the method as function, procedure, or task, and in the last case optionally as a controllable or uncontrollable task (Section A.1)
- return type or `void` if the method does not return a value
- argument list
- method body

```
<methodDecl> := ["export"]                // Exported method?
                <methCateg>               // Method category
                ("void" | <typeSpec>)        // Return type
                <ident> "("                  // Method name
                    [<arg> ("," <arg>)*]     // Arguments
                ")"
```

```
                    (                          // Partial body:
                     ["before" <statement>]    // preamble
                     ["after" <statement>]     // epilogue
                    ) |
                    <statement>                // complete form

<methCateg> := "function"
             | "procedure"
             | "task" [ "controllable"
                      | "uncontrollable"]

<arg> := ["out"] <typeSpec> <ident>
```

Method arguments are used to pass data both to and from the method. Each individual argument can be declared as an input or an output argument, but not both. Syntactically, input and output arguments are distinguished using the `out` qualifier.

Both input and output arguments can be read and modified in the body of the method; however the initial value of an output argument is non-derministic. The final value of an input argument is dropped, while the final value of an output argument is propagated to the L-expression (Section A.2) passed as the actual argument to the method. Note that for controllable and uncontrollable tasks the output value is propagated to the argument at the beginning of the next transition performed by the process after the implicit pause location following the return statement (Section A.2).

Method body Method body declaration can be written in the *complete* or *partial* form. A complete declaration is simply a TSL statement. A partial declaration consists of an optional preamble and an optional epilogue that are intended to execute respectively before and after the main body of the method declared in the child template. If neither the preamble not the epilogue are specified, then the body of the method remains empty.

The full method body is constructed by recursively merging the method body provided in the method declaration with the overloaded method declaration in the parent template (if one exists) using Algorithm 26. The following examples illustrate the algorithm.

```
// global variable
bool x;
```

Algorithm 26

Input: Method body M, where M is either a complete body (b) or partial body (p, e).

Output: Method body merged with the parent body (if one exists)

```
1:  function FULLBODY(M)
2:      // Find previous method declaration in one of parent templates
3:      M' ← PARENTMETHOD(M)
4:      if M' = nil then
5:          // M is not an overloaded method
6:          return M
7:      end if
8:      case (M, FULLBODY(M')) of
9:          //Parent method has a complete body: override it
10:         (M, (b')) : return M
11:         // Both parent and child have partial bodies: merge
12:         // preambles and epilogues using sequential composition
13:         ((p, e), (p', e')) : return ((p'; p), (e; e'))
14:         // Child has a complete body, parent has partial body: prepend
15:         // and append parent's preamble and epilogue to the child
16:         ((b), (p', e')) : return (p'; b; e')
17:         otherwise : return error
18: end function
```

```
// parent declaration
procedure void p(uint<16> arg)
before{
    assume(arg != 0);
};
after{
    assert(x);
};

// child declaration
procedure void p(uint<16> arg)
{
    x = (arg > 5);
};

// full child method body generated by the compiler
procedure void p(uint<16> arg)
```

```
{
    assume(arg != 0);
    x = (arg > 5);
    assert(x);
};
```

```
// parent declaration
procedure void p(uint<16> arg)
before{
    assume(arg != 0);
};
after{
    assert(x);
};

// child declaration
procedure void p(uint<16> arg)
before{
    x = (arg > 5);
};

// full child method body generated by the compiler
procedure void p(uint<16> arg)
before{
    assume(arg != 0);
    x = (arg > 5);
};
after{
    assert(x);
};
```

Goals

```
<goalDecl> := "goal" <ident> "=" <expr>
```

See Section A.1.

Wire declarations

Wire declaration (Section A.1) consists of wire type, wire name, and optional wire expression.

```
<wire> := ["export"] "wire" <typeSpec> <ident> ["=" <expr>]
```

The wire expression is defined over global variables and wires of the current template and other templates accessible via hierarchical identifiers. However, circular dependencies among wire expressions are forbidden. If the wire expression is omitted, the wire is a *pure wire* that must be re-defined in child templates. In merging template with its parents, the TSL compiler picks the last wire declaration in the inheritance hierarchy.

B | User Guided Synthesis of an I2C Driver

```
1  // @LICENSE(NICTA, 2014)
2
3  typedef enum {
4      slave_receive, slave_transmit, master_receive, master_transmit
5  } mode_t;
6
7  typedef enum {
8      clocks0, clocks5, clocks10, clocks15
9  } sdaod;
10
11 //internal state
12 typedef enum {
13     idle,
14     transmitting_address_t,
15     addr_transmitted_t,
16     transmitting_data,
17     transmitting_address_r,
18     addr_transmitted_r,
19     receiving_data,
20     transmitting_stop
21 } master_transmit_st_t;
22
23 template i2c_dev
24
25 function bool ack_enabled();
26 function bool clk_src();
27 function bool tx_int();
28 function uint<4> clk_prescalar();
29 function mode_t mode_val(uint<8> val);
30 function mode_t the_mode();
31 function bool val_enabled(uint<8> val);
32 function bool enabled();
33 function uint<7> slave_address();
34 function bool filter_enable();
35 function sdaod sda_output_delay();
```

```
36
37 //internal state
38 export master_transmit_st_t master_transmit_st = idle;
39
40 endtemplate
41
42 template i2c_drv
43
44 task uncontrollable void configure();
45 task uncontrollable void send_address();
46 task uncontrollable void send_data();
47 task uncontrollable void send_stop();
48 task uncontrollable bool send_address_read();
49 task uncontrollable uint<8> read_data(bool ack);
50
51 endtemplate
52
53 template i2c_os
54 export uint<7> os_addr;
55 export uint<8> os_data;
56 export uint<8> os_read_data;
57 export bool sent;
58
59 export uint<7> os_read_addr;
60
61 procedure void address_written(uint<7> addr);
62 task void data_sent(uint<8> data);
63 task void stop_sent();
64 procedure void address_written_read(uint<7> addr, bool acked);
65 task void data_received(uint<8> data, bool acked);
66 procedure void config_updated();
67
68 //internal state
69
70 typedef enum {
71     master_idle,
72     address_pending,
73     data_pending,
74     stop_pending
75 } master_state_t;
76
77 export master_state_t master_state = master_idle;
78
79 typedef enum {
80     master_read_idle,
81     master_read_address_pending,
82     master_read_data_pending,
83     master_read_stop_pending
84 } master_read_state_t;
```

```
85
86 export master_read_state_t master_read_state = master_read_idle;
87
88 export bool was_sent;
89
90 export bool was_recvd;
91
92 endtemplate
```

```
1  // @LICENSE(NICTA, 2014)
2
3  import <class.tsl>
4  import <i2c.tsl>
5  import <os.tsl>
6
7  template i2c_linux_drv(i2c_dev_m3 dev, i2c_linux os)
8
9  derive i2c_drv;
10
11 task uncontrollable void configure(){
12     ...;
13 };
14
15 task uncontrollable void send_address(){
16     ...;
17 };
18
19 task uncontrollable void send_data(){
20     ...;
21 };
22
23 task uncontrollable void send_stop(){
24     ...;
25 };
26
27 task uncontrollable bool send_address_read(){
28     ...;
29     uint<8> res = dev.read8_i2cstat();
30     return (res[0:0] == 1);
31 };
32
33 task uncontrollable uint<8> read_data(bool ack){
34     ...;
35     return dev.read8_i2cds();
36 };
37
38 endtemplate
```

```
1  // @LICENSE(NICTA, 2014)
2
```

```
3  import <class.tsl>
4  import <i2c.tsl>
5  import <os.tsl>
6
7  template i2c_linux_drv(i2c_dev_m3 dev, i2c_linux os)
8
9  derive i2c_drv;
10
11 task uncontrollable void configure(){
12     //address = 0x54
13     dev.write8_i2cadd(/*any value*/1'h0 ++ 7'h54); //why 54? Handwritten driver does t
14
15     //clc = SDA_DELAY15CLK | FILE_EN
16     dev.write8_i2clc(2'h3 ++ 1'h1 ++ /*any value*/5'h0);
17
18     //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
19     dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
20 };
21
22 task uncontrollable void send_address(){
23     /*
24         Below order of operations differs from the handwritten driver, though it seems
25         We do not write to the status register twice as the handwritten driver does.
26         This needs to be tested.
27     */
28
29     //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
30     dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
31
32     //data = os_addr (&~ 1)
33     dev.write8_i2cds(1'h0 ++ os.os_addr);
34
35     //stat = MODE_MTX | ENABLE | BUSY
36     dev.write8_i2cstat(/*any value*/4'h0 ++ 1'h1 ++ 1'h1 ++ 2'h3);
37
38     //wait(control & IRQ_PENDING)
39     uint<8> stat;
40     do {
41         stat = dev.read8_i2ccon();
42     } while(stat[4:4] == 0);
43
44     //We do not check for ACK/NACK of address as is done in send_address_read() becaus
45 };
46
47 task uncontrollable void send_data(){
48     //data = os_data
49     dev.write8_i2cds(os.os_data);
50
51     //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
```

```
52      dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
53
54      //wait(control & IRQ_PENDING)
55      uint<8> stat;
56      do {
57          stat = dev.read8_i2ccon();
58      } while(stat[4:4] == 0);
59
60      //The handwritten driver aborts if the device is not busy. I have no idea how the device c
61  };
62
63  /*
64      This differs from the handwritten driver, but agrees with the datasheet.
65      The handwritten driver sets ~BUSY in the stat register before clearing the pending bit of
66      The flowchart on page 711 of the documentation clears the pending bit before setting ~BUSY
67      This needs to be tested.
68  */
69  task uncontrollable void send_stop(){
70      //stat = ANY_MODE | ENABLE &~ BUSY //TODO: is it ok to use any mode?
71      dev.write8_i2cstat(/*any value*/4'h0 ++ 1'h1 ++ 1'h0 ++ /*any value*/2'h0);
72
73      //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
74      dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
75
76      //wait(control & IRQ_PENDING)
77      uint<8> stat;
78      do {
79          stat = dev.read8_i2cstat();
80      } while(stat[5:5] == 1); //we are waiting for the wrong thing
81  };
82
83  task uncontrollable bool send_address_read(){
84      /*
85          Below order of operations differs from the handwritten driver, though it seems to agre
86          We do not write to the status register twice as the handwritten driver does.
87          This needs to be tested.
88      */
89
90      //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
91      dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
92
93      //data = os_addr (| 1)
94      dev.write8_i2cds(1'h1 ++ os.os_read_addr);
95
96      //stat = MODE_MRX | ENABLE | BUSY
97      dev.write8_i2cstat(/*any value*/4'h0 ++ 1'h1 ++ 1'h1 ++ 2'h2);
98
99      //wait(control & IRQ_PENDING)
100     uint<8> stat;
```

```
101     do {
102         stat = dev.read8_i2ccon();
103     } while(stat[4:4] == 0);
104
105     uint<8> res = dev.read8_i2cstat();
106     return (res[0:0] == 1);
107 };
108
109 //TODO: We dont disable acks at the end
110 task uncontrollable uint<8> read_data(bool ack){
111     //control = ACK_EN | CLK_SRC | PRESCALE(3) | IRQ_EN &~ IRQ_PENDING
112     if(ack){
113         dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h1);
114     } else {
115         dev.write8_i2ccon(4'h3 ++ 1'h0 ++ 1'h1 ++ 1'h1 ++ 1'h0);
116     };
117
118     //wait(control & IRQ_PENDING)
119     uint<8> stat;
120     do {
121         stat = dev.read8_i2ccon();
122     } while(stat[4:4] == 0);
123
124     //wait(os.was_recvd);
125     return dev.read8_i2cds();
126 };
127
128 endtemplate
```

```
1  // @LICENSE(NICTA, 2014)
2
3  import <class.tsl>
4
5  template i2c_dev_m3(i2c_os os)
6
7  derive i2c_dev;
8
9  const uint<8> I2CCON  = 8'h00;
10 const uint<8> I2CSTAT = 8'h04;
11 const uint<8> I2CADD  = 8'h08;
12 const uint<8> I2CDS   = 8'h0C;
13 const uint<8> I2CLC   = 8'h10;
14
15 //registers
16 uint<8> i2ccon  = 8'h0;
17 uint<8> i2cstat = 8'h0;
18 uint<8> i2cadd;
19 uint<8> i2cds;
20 uint<8> i2clc   = 8'h0;
```

```
21
22 //master_transmit_st_t master_transmit_st = idle;
23
24 function bool ack_enabled(){
25     return i2ccon[7:7] == 1;
26 };
27
28 function bool clk_src(){
29     return i2ccon[6:6] == 1;
30 };
31
32 function bool tx_int(){
33     return i2ccon[5:5] == 1;
34 };
35
36 function uint<4> clk_prescalar(){
37     return i2ccon[0:3];
38 };
39
40 function mode_t mode_val(uint<8> val){
41     case(val[6:7]) {
42         2'h0 : return slave_receive;
43         2'h1 : return slave_transmit;
44         2'h2 : return master_receive;
45         2'h3 : return master_transmit;
46     };
47     return slave_receive; //why?
48 };
49
50 function mode_t the_mode(){
51     return mode_val(i2cstat);
52 };
53
54 function bool val_enabled(uint<8> val){
55     return val[4:4] == 1;
56 };
57
58 function bool enabled(){
59     return val_enabled(i2cstat);
60 };
61
62 function uint<7> slave_address(){
63     return i2cadd[1:7];
64 };
65
66 function uint<8> shift_data(){
67     return i2cds;
68 };
69
```

```
70  function bool filter_enable(){
71      return i2clc[2:2] == 1;
72  };
73
74  function sdaod sda_output_delay(){
75      case(i2clc[0:1]) {
76          2'h0 : return clocks0;
77          2'h1 : return clocks5;
78          2'h2 : return clocks10;
79          2'h3 : return clocks15;
80      };
81      return clocks0; //why?
82  };
83
84  task controllable uint<8> read8_i2ccon(){
85      return i2ccon;
86  };
87
88  task controllable uint<8> read8_i2cds(){
89      return i2cds;
90  };
91
92  task controllable uint<8> read8_i2cstat(){
93      uint<8> tmp = i2cstat;
94      if(master_transmit_st == idle){
95          tmp[5:5] = 0;
96      } else {
97          tmp[5:5] = 1;
98      };
99      return tmp;
100 };
101
102 task controllable void write8_i2ccon(uint<8> wval){
103     i2ccon = wval;
104
105     //transmit state machine
106     //clearing pending
107     if(master_transmit_st == addr_transmitted_t && wval[4:4] == 0){
108         if(i2cstat[5:5] == 0){
109             master_transmit_st = transmitting_stop;
110         } else {
111             master_transmit_st = transmitting_data;
112         };
113     };
114     if(master_transmit_st == addr_transmitted_r && wval[4:4] == 0){
115         if(i2cstat[5:5] == 0){
116             master_transmit_st = transmitting_stop;
117         } else {
118             master_transmit_st = receiving_data;
```

```
119         };
120     };
121 
122     os.config_updated();
123 };
124 
125 
126 task controllable void write8_i2cstat(uint<8> wval){
127     i2cstat[6:7] = wval[6:7];
128     i2cstat[4:4] = wval[4:4];
129     i2cstat[5:5] = wval[5:5];
130 
131     //transmit state machine
132     //writing busy
133     if(master_transmit_st == idle && val_enabled(wval)){
134         if(wval[5:5] == 1){
135             //set busy
136             i2cstat[5:5] = 1;
137             case (mode_val(wval)) {
138                 master_transmit : master_transmit_st = transmitting_address_t;
139                 master_receive  : master_transmit_st = transmitting_address_r;
140             };
141         };
142     };
143 };
144 
145 task controllable void write8_i2cadd(uint<8> wval){
146     //I2CADD  : if(enabled()) {
147         i2cadd  = wval;
148     //};
149 };
150 
151 task controllable void write8_i2cds(uint<8> wval){
152     //I2CDS   : if(enabled()) {
153         i2cds   = wval;
154     //};
155 };
156 
157 task controllable void write8_i2clc(uint<8> wval){
158     I2CLC   : i2clc   = wval;
159 };
160 
161 //transmit state machine
162 process pmaster {
163     forever {
164         pause;
165         case(master_transmit_st){
166             transmitting_address_t : {
167                 //address is a write address
```

```
168                 assert(i2cds[0:0] == 0);
169
170                 master_transmit_st = addr_transmitted_t;
171                 i2ccon[4:4] = 1;
172                 os.address_written(i2cds[1:7]);
173             };
174             transmitting_data      : {
175                 master_transmit_st = addr_transmitted_t;
176                 i2ccon[4:4] = 1;
177                 os.data_sent(i2cds);
178             };
179             transmitting_address_r : {
180                 //address is a read address
181                 assert(i2cds[0:0] == 1);
182
183                 master_transmit_st = addr_transmitted_r;
184                 i2ccon[4:4] = 1;
185                 i2cstat[0:0] = *;
186                 os.address_written_read(i2cds[1:7], i2cstat[0:0]==1);
187             };
188             receiving_data : {
189                 master_transmit_st = addr_transmitted_r;
190                 i2ccon[4:4] = 1;
191                 i2cds = *;
192                 os.data_received(i2cds, i2ccon[7:7]==1);
193             };
194             transmitting_stop      : {
195                 //i2ccon[4:4] = 1;
196                 master_transmit_st = idle;
197                 os.stop_sent();
198             };
199         };
200     };
201 };
202
203 endtemplate
```

```
1  // @LICENSE(NICTA, 2014)
2
3  import <os.tsl>
4  import <i2c.tsl>
5  import <drv.tsl>
6
7  template main
8
9  instance i2c_linux           os  (drv, dev);
10 instance i2c_dev_m3          dev (os);
11 instance i2c_linux_drv       drv (dev, os);
12
```

```
endtemplate
```

```
// @LICENSE(NICTA, 2014)

import <class.tsl>

template i2c_linux(i2c_drv drv, i2c_dev dev)

derive i2c_os;

procedure void config_updated(){
    if(inited){
        //assert(dev.ack_enabled());
        assert(dev.clk_src());
        assert(dev.clk_prescalar() == 4'h3);
        assert(dev.tx_int());
    };
};

procedure void configured(){
    assert(dev.ack_enabled());
    assert(dev.clk_src());
    assert(dev.clk_prescalar() == 4'h3);
    assert(dev.tx_int());
    assert(dev.sda_output_delay() == clocks15);
    assert(dev.filter_enable());
    assert(dev.slave_address() == 7'h54);
};

bool inited = false;

process pos {
    drv.configure();
    configured();
    inited = true;
    forever {
        choice {
            master_write();
            master_read();
            {};
        };
        pause;
    };
};

goal g1 = (master_state == master_idle) || sent;

//uint<7> os_addr;
//uint<8> os_data;
```

```
48
49 task void master_write(){
50
51     os_addr = *;
52     master_state = address_pending;
53     drv.send_address();
54     assert(master_state != address_pending);
55
56     forever {
57         if (master_state == data_pending) {
58             os_data = *;
59             was_sent = false;
60             drv.send_data();
61             assert(was_sent);
62         } else if (master_state == stop_pending) {
63             drv.send_stop();
64             assert(master_state == master_idle);
65         } else if (master_state == master_idle) break;
66
67         pause;
68     };
69 };
70
71 procedure void address_written(uint<7> addr){
72     assert(master_state == address_pending);
73     assert(addr == os_addr);
74     master_state = data_pending;
75 };
76
77 bool decis;
78 //bool sent;
79 prefix sent = false;
80
81 task void data_sent(uint<8> data){
82     assert(master_state == data_pending);
83     assert(data == os_data);
84     sent = true;
85     was_sent = true;
86     decis = *;
87     if(decis){
88         master_state = stop_pending;
89     } else {
90         master_state = data_pending;
91     };
92 };
93
94 task void stop_sent(){
95     assert(master_state == stop_pending || master_read_state == master_read_stop_pendi
96     master_state = master_idle;
```

```
97      master_read_state = master_read_idle;
98  };
99
100 goal g2 = (master_read_state == master_read_idle) || recvd;
101
102 bool readAddrAcked;
103 bool ack_this_read;
104
105 task void master_read(){
106     os_read_addr = *;
107     master_read_state = master_read_address_pending;
108     bool acked = drv.send_address_read();
109     assert(master_read_state == master_read_data_pending);
110     assert(acked == readAddrAcked);
111
112     forever {
113         if(master_read_state == master_read_data_pending){
114             was_recvd = false;
115             ack_this_read = *;
116             uint<8> retnd = drv.read_data(ack_this_read);
117             assert(was_recvd);
118             //assert(retnd == os_read_data);
119         } else if(master_read_state == master_read_stop_pending) {
120             drv.send_stop();
121             assert(master_read_state == master_read_idle);
122         } else if(master_read_state == master_read_idle) break;
123
124         pause;
125     };
126 };
127
128 procedure void address_written_read(uint<7> addr, bool acked){
129     assert(master_read_state == master_read_address_pending);
130     assert(os_read_addr == addr);
131     master_read_state = master_read_data_pending;
132     readAddrAcked = acked;
133 };
134
135 bool decis2;
136 bool recvd;
137 prefix recvd = false;
138
139 task void data_received(uint<8> data, bool acked){
140     assert(master_read_state == master_read_data_pending);
141     //assert(ack_this_read == acked);
142     recvd = true;
143     was_recvd = true;
144     decis2 = *;
145     os_read_data = data;
```

```
146     if(decis2){
147         master_read_state = master_read_stop_pending;
148     } else {
149         master_read_state = master_read_data_pending;
150     };
151 };
152
153 endtemplate
```

C | A specification Language for Symbolic Games

We describe a simple specification language for symbolic games. This language is used to illustrate, in the simplest possible way, how Termite's specifications work. Furthermore, I have created a compiler for this language as an alternate frontend for Termite and it is available as part of Termite. As a result, all of the examples given in this chapter will compile and synthesize with Termite.

Fundamentally, symbolic transition systems are specified by defining how each state variable is updated on each transition, as a function of the other state variables and the label variables. We call these functions update functions and there is one update function for each state variable.

Our simple language consists of four sections for defining a transition system:

- The state variable declaration
- The label variable declaration
- The set of initial states
- Variable update functions, one for each state variable

Variables in both the state and label sections may be declared as one of three types:

- Fixed width integer
- Boolean
- Enumeration

The transition relation is specified using nested case statements.
The BNF grammar is given below.

Identifiers

An identifier is a name of a variable or a constant declared as part of an enumeration.

```
<ident>    := <letter> (<letter> | <digit> | "_")*
```

Numbers

Numbers may be specified in hexadecimal, octal or decimal.

```
<number>    := "0x" <hexNumber>
             | "0o" <octNumber>
             | <decNumber>
```

Types

Our specification language supports three different types: Booleans, fixed width bit vectors, and enumerations.

```
<boolTyp>  := "bool"
<intTyp>   := "uint" "<" <digit>* ">"
<enumTyp>  := "{" identifier* "}"
```

Variables

Variables are declared by giving a name of the variable and its type.

```
<type>     := <boolTyp> | <intTyp> | <enumTyp>
<varDecl>  := <identifier> ":" <type>
```

Boolean Expressions

Boolean expressions consist of boolean literals, value expressions (to follow) combined with the equality operator, and other boolean expressions combined with logical connectives.

```
<boolLit>  := "true" | "false"
<binExpr>  := <boolLit>
            | <valExpr> "==" <valExpr>
            | <binExpr> "||" <binExpr>
            | <binExpr> "&&" <binExpr>
            | "!" <binExpr>
```

Value Expressions

Each value expression is a number, an identifier or a case statement. A case statement consists of a list of cases. Each case consists of a boolean condition followed by a value expression. Thus, case statements may contain further case statements. The outcome of a case statement is the value of the first case whose boolean expression evaluates to true.

```
<caseStmt> := "case" "{"
                  (<binExpr> ":" <valExpr> ";")*
              "}"
<valExpr>  := <caseStmt> | <number> | <identifier>
```

Variable updates

Update functions are specified by assigning the next state value of a state variable to a value expression.

```
<update>   := <identifier> ":=" <valExpr>
```

Transition System

Transition systems are defined by giving the set of state variables, the set of label variables, the initial states and the variable update functions. The set of initial states is specified by the characteristic formula of the set.

```
<spec>     := "State"       <varDecl>*
              "Label"       <varDecl>*
              "Init"        <binExpr>
              "Transitions" <update>*
```

Bibliography

Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, San Francisco, California, USA, June 1998.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland, 1991.

Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *"1st International Conference on Hardware/Software Codesign and System Synthesis"*, pages 19–24, Newport Beach, CA, USA, 2003.

Pavol Cerny, Thomas Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *25th International Conference on Computer Aided Verification*, pages 1–16, Saint Petersburg, Russia, July 2013.

Pavol Cerny, Thomas Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. Vienna, Austria, July 2014.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, Chicago, IL, USA, July 2000.

Luca de Alfaro and Pritam Roy. Solving games via three-valued abstraction refinement. In *CONCUR*, pages 74–89, Lisboa, Portugal, September 2007.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, Haifa, Israel, June 1997.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, Portland, Oregon, January 2002.

Intel Corporation. Cofluent technolofy. `http://www.intel.com/content/www/us/en/cofluent/cofluent-difference.html`.

Swen Jacobs. Extended AIGER format for synthesis. *CoRR*, abs/1405.5793, 2014.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. pages 207–220, Big Sky, MT, USA, October 2009.

OpenCores. 16550 UART core. `http://opencores.org/project,a_vhd_16550_uart`.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. pages 364–380, Charleston, SC, USA, January 2006.

Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47, Santa Clara, CA, USA, 1993.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Eurosys 2009*, pages 275–288, Nuremberg, DE, April 2009a.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. pages 73–86, Big Sky, MT, US, October 2009b.

Leonid Ryzhyk, Adam Christopher Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. pages 661–676, Broomfield, CO, USA, October 2014.

Fabio Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

Synopsys. Virtual prototyping models. `http://www.synopsys.com/Systems/VirtualPrototyping/VPModels`.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

Termite. Termite 2 driver synthesis tool. `http://www.termite2.org`.

Wolfgang Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.

Adam Christopher Walker and Leonid Ryzhyk. Predicate abstraction for reactive synthesis. Lausanne, Switzerland, October 2014.

Wind River. Wind River Simics Model Builder user guide. version 4.4, September 2010.

WindRiver. WindRiver Simics DS12887 Model. `http://www.windriver.com/products/simics`.

Raj Yavatkar. Era of SoCs, presentation at the Intel Workshop on Device Driver Reliability, Modeling and Synthesis, March 2012.