An Efficient Projected Minimal Conflict Generator for Projected Prime Implicate and Implicant Generation

by

Paul Harrison Elliott

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Masters of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

© Massachusetts Institute of Technology 2004. All rights reserved.

Author .....
Department of Aeronautics and Astronautics
February 9, 2004

Certified by .....
Brian C. Williams
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

An Efficient Projected Minimal Conflict Generator for Projected Prime Implicate and Implicant Generation

by

Paul Harrison Elliott

Submitted to the Department of Aeronautics and Astronautics
on February 9, 2004, in partial fulfillment of the
requirements for the degree of
Masters of Science in Aeronautics and Astronautics

Abstract

Performing real-time reasoning on models of physical systems is essential in many situations, especially when human intervention is impossible. Since many deductive reasoning tasks take memory or time that is exponential in the number of variables that appear in the model, efforts need to be made to reduce the size of the models used online. The model can be reduced without sacrificing reasoning ability by targeting the model for a specific task, such as diagnosis or reconfiguration. A model may be reduced through model compilation, an offline process where relations and variables that have no bearing on the particular task are removed.

This thesis introduces a novel approach to model compilation, through the generation of *projected* prime implicates and *projected* prime implicants. Prime implicates and prime implicants compactly represent the consequences of a logical theory. Projection eliminates model variables and their associated prime implicates or implicants that do not contribute to the particular task. This elimination process reduces the size and number of variables appearing in the model and therefore the complexity of the real-time reasoning problem.

This thesis presents a minimal conflict generator that efficiently generates projected prime implicates and projected prime implicants. The projected minimal conflict generator uses a generate-and-test approach, in which the candidate generator finds potential minimal conflicts that are then accepted or rejected by the candidate tester.

The candidate generator uses systematic search in combination with an iterative deepening algorithm, in order to reduce the space required by the algorithm to a space that is linear in the number of variables rather than exponential. In order to make the algorithm more time efficient, the candidate generator prunes the search

space using previously found implicants, as well as minimal conflicts, which identify the sub-spaces that contain no new minimal conflicts.

The candidate tester identifies implicants of the model by testing for validity. The tester uses a clause-directed approach along with finite-domain variables to efficiently test for validity. Combined, these techniques allow the tester to test for validity without assigning a value to every variable.

The conflict generator was evaluated on randomly generated models; problems in which models with 20 variables, 5 domain elements each, were projected onto 5 variables. All projected prime implicates were generated from models with 20 clauses within 2 seconds, and from models with 80 clauses within 13 seconds.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank Joelle Brichard, who was always there for me during the writing process. Her patience has been greatly appreciated.

I would like to thank my advisor, Brian Williams, and my co-workers Samidh Chakrabati, Seung Chung, Stanislav Funiak, Mitch Ingham, Mike Pekala, Rob Ragno, Martin Sachenbacher, and Margaret Yoon who made all of this possible.

I would also like to thank my mother, Kristen Zorica, who also took time to read this document.

Contents

List of Figures

List of Tables

Nomenclature

The following is an overview of the symbols that are used throughout this thesis.

A lower case letter, such as v, is used to represent a primitive element, in this case a variable. A capital letter, such as V, represents a set of the elements v, in this case a set of variables. A script letter, such as $\mathcal{V}$, represents a set of sets.

The symbol $\Downarrow_p$ after a clause C or theory $\mathcal{C}$ indicates that the theory is projected. A variable is projected over a subset of the variables, V_p, of the model. Thus, the clause being denoted $C \Downarrow_p$ is only in terms of the variables of V_p. Lack of a projection operator implies that the clause is in terms of all the variables V. The letters p and u are used to represent the projected and unprojected variables, respectively. Thus, $\mathcal{C}$ and $\mathcal{C} \Downarrow_{V_u \cup V_p}$ are equivalent.

V : The set of all variables.

v : A single variable.

$v \in V$

V_p : The set of projected variables.

V_u : The set of unprojected variables.

X : The set of all possible values for all variables.

$D(v)$: The domain $D(v) \subseteq X$ of the variable v.

$D : V \mapsto X$

s : An assignment of a single value to a single variable.

$s = \{< v, x > | v \in V, x \in D(v)\}$

s_p : An assignment of a value to a projected variable.

$s = \{< v, x > | v \in V_p, x \in D(v)\}$

s_u : An assignment of a value to an unprojected variable.

$s = \{< v, x > | v \in V_u, x \in D(v)\}$

c : A literal. It has an assignment and a parity flag, that specifies whether the assignment is either true or false.

$c = \{< a, parity > | parity \in \{Positive, Negative\}\}$

C : A disjunctive clause of literals.

$C = \bigvee c$

$\mathcal{C}$: The constraints of the plant model, represented as a conjunctive sentence of disjunctive clauses.

$\mathcal{C} = \bigwedge C$

A : A conjunction of literals.

$A = \{\bigwedge_{i \in \{1,\ldots,n\}} c_i | \forall c_j, c_k. ((c_j.v = c_k.v) \Leftrightarrow (j = k))\}$

Where $n \in \mathbb{N}$, so this clause can have 0 or more assignments.

A is a valid assignment if it contains at most one assignment per variable, thus A is a tuple over the variables V.

$\mathcal{A}$: A disjunction of conjunctive clauses A; i.e., a constraint.

$\mathcal{A} = \bigvee A$

Chapter 1

Introduction

1.1 Problem Statement

Many problems can be formulated as a set of finite constraints that describe interactions among a set of variables. For example, in model-based reasoning, variables are used to describe components and variable constraints are used to describe the component behaviors. Typically, these constraints are initially specified by a modeler. This user-specified model often has extra variables and constraints, which were introduced to make it easier for humans to write, to re-use, and to understand the model. For example, for portability, each component is often written with an interface, such as the current state of power to the device, which is often provided by some other component, like a power supply. In this simple system, as shown in Figure 1-1, there are already 4 variables: the state of the power supply, the state of the powered device, a variable that describes the power being output by the power supply, and a variable that describes the power being input to the powered device. Given this system, the model can be compiled so as to eliminate both intermediate power variables. In the compiled model, the powered device has power when the power supply is on.

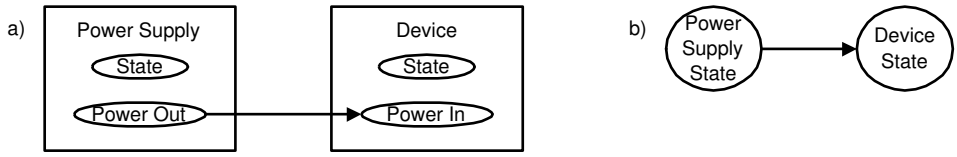

Figure 1-1: (a) A simple model with a power supply, modeled with two variables, and a powered device, also modeled with two variables. Each component is encapsulated with a power in/power out interface variable. (b) shows the same simple model with the interface variables compiled out. In the compiled form, the component distinctions are removed.

Reasoning on constraints, which is equivalent to solving the Constraint Satisfaction Problem (CSP), is intractable; in the worst case, CSPs take an exponential amount of time to solve as a function of the number of variables. Thus, reasoning on the user-specified constraints can take unnecessarily long amounts of time. One effective way of speeding up the response time is to reduce the problem size by eliminating both the extra variables and the extra constraints that are not needed for a specific task. This is referred to in the literature as *knowledge compilation* or *model compilation* [12] [6].

One way to achieve this reduction is through prime implicate and implicant generation [1]. A set of prime implicates is a minimal set of clauses such that each clause is implied by the theory. Prime implicates are useful for diagnosis, where all diagnoses in the theory are also present in the prime implicates. A set of prime implicants is a minimal set of clauses such that each clause implies the theory. Prime implicants are useful for controllers, where only those control actions that have specific effects are present in the prime implicants. Implicates and implicants are explained in greater detail in Section 3.1. Using prime implicates and prime implicants can speed up the reasoning process; however, generating these implicates and implicants is also an NP-complete task. Fortunately, this task can be performed just once and can be performed offline.

Another way of reducing the model is to eliminate variables. Since the model typically contains information suitable for a number of tasks, it also typically contains a number of variables that are not useful for a particular task. Thus, by *projecting* the model on to the set of variables useful for a particular task, the task can be performed more efficiently. Projection generates a new theory tailored to a specific task.

This thesis combines these two methods of reducing the model. This thesis addresses three problems: (1) eliminating unnecessary variables from the model by projecting the constraints onto the set of desired variables, (2) representing the projected model compactly through prime implicates and implicants, and (3) providing a projected minimal conflict generator, where a minimal conflict is a negated prime implicate, that provides an efficient means to simultaneously solve the first two problems.

1.2 Approach

As mentioned above, this thesis solves the two problems of (1) projected prime implicate and (2) projected prime implicant generation. These two problems differ primarily in the way that they treat ambiguous conditions. Ambiguities arise when the value of some variable can take on an arbitrary set of values, either by design, or because some of the variables that would clear up the ambiguity can neither be measured nor estimated. If a variable v is ambiguous, then a theory based on the projected prime implicates will optimistically assume that any assignment to v is correct as long as that assignment is possible. A theory based on the projected prime implicants will pessimistically assume that all assignments to v are inconsistent as long as a different value could have been chosen.

The first problem arises in model-based diagnosis, as it allows for the possibility that information is partial, and hence a hypothesis is a valid diagnosis if it is

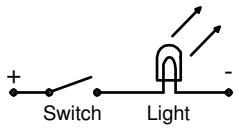

Figure 1-2: A light with a switch.

merely consistent with what is observed, rather than explaining all observations. (1) uses consistency to ensure that the resulting compiled model does not rule out these possible diagnoses.

The second problem arises in model-based control. Unlike diagnosis, where the diagnoses are partial, in model-based control one would like to make sure that the system is guaranteed to be only controlled into particular states. (2) will ensure that the compiled model contains only consistent configurations that entail the goals.

Consider an example of how these two problems are used to address ambiguity. One common type of ambiguity arises when one attempts to model the behavior of a broken component. Typically the behavior of a broken component is unrestricted. For instance, assume that there is a broken light switch in the system shown in Figure 1-2. When estimating the state of the switch, one should allow for the possibility that the light attached to the switch is either on or off. Either is possible, depending on how the switch failed. However, when performing control, one requires that the system be in a certain state. The control side should therefore assume that the light will be off when the switch being broken. If the light needs to be on, the controller should repair the switch, rather than assuming that the light may be on and taking no action.

The projection eliminates variables that do not contribute to the relation of interest, such as the relation between observations and state. Since reasoning with the model is exponential in the number of its variables, reducing the number of variables can lead to dramatic performance improvements.

In this thesis, the problems of generating projected prime implicates and projected prime implicants are reformulated as the problem of generating *projected minimal conflicts*. A conflict is a negated implicate; hence, a conflict of a theory is an assignment that is inconsistent with that theory. A minimal conflict is a conflict that is inconsistent with the theory, but no subset of the conflict is inconsistent with the theory. This is equivalent to the negation of a prime implicate. A projected minimal conflict is a minimal conflict that is restricted to a subset of the variables, namely the projected variables. The generator is implemented as a generate-and-test algorithm that uses iterative deepening search to be space-efficient. The generation algorithm employs pruning using implicants, as well as minimal conflicts, which have been identified thus far by the tester.

Conceptually, pruning eliminates sub-trees of candidates that cannot contain solutions, namely minimal conflicts. While minimal conflicts are necessarily determined, the candidate tester must be able to test for validity in order for the tester to identify implicants.

The algorithm tests each candidate assignment to determine if it is a minimal conflict using a SAT-based algorithm that is able to distinguish between valid, consistent, and inconsistent candidates. Valid candidates are implicants, while inconsistent candidates are conflicts. The tester is able to test for validity efficiently through the use of two concepts. First, the tester is clause-directed, meaning that it assigns values to variables only to satisfy clauses. Second, the tester operates on finite-domain variables directly, rather than representing each finite-domain variable as a set of binary variables. Combined, these two techniques enable the tester to conclude validity, namely when all the clauses have been satisfied, without having to assign values to every variable.

1.3 Thesis Outline

Chapter 2 introduces a model that is common throughout this thesis and the related work upon which this thesis builds. It presents current developments in knowledge compilation and their relation to projected prime implicate generation. A projected prime implicate generator and a projected prime implicant generator is presented in Chapter 3. These algorithms builds upon a projected minimal conflict generator.

The projected minimal conflict generator is presented in Chapter 4. It uses a generate-and-test algorithm. The generator for the generate-and-test algorithm is a memory-efficient iterative deepening algorithm that prunes candidates based on implicates and implicants found so far. The tester uses a SAT engine, augmented to extract information used by the generator to prune the search.

Chapter 5 summarizes the content of this thesis. It also presents a number of potential research areas that may improve upon what this thesis has presented.

Chapter 2

Background and Related Work

This chapter sets the context for the model compilation algorithms developed in this thesis. Section 2.1 introduces a simple pedagogical example, used to demonstrate the concepts introduced by this thesis. Section 2.2 sets this thesis in the context of previous research on model compilation.

2.1 Modeling Physical Systems using Logical Theories

This thesis is primarily motivated by the CSP problems that arise in model-based autonomy. Model-based autonomy [17] is an architecture for robust control of autonomous robots and other embedded systems, by reasoning on the fly from models of that system. Model-based autonomy uses a constraint-based model to describe the behavior of the system, and then uses a general-purpose constraint optimization engine to operate on this model.

The following section introduces an example that will be used throughout this thesis. This example generates a set of projected prime implicates. Intuitively, the

prime implicates are clauses that relate the state variables to the observations.

2.1.1 Example: Propulsion Model

The model has four types of variables: (1) state variables, which describe the persistent state of the system, (2) command variables, which describe the input to the system, (3) observable variables, which describe the measurable outputs of the system, and (4) dependent variables, which describe transient information that cannot be observed or commanded directly. For example, consider the variables associated with a single valve. (1) The current position of a valve is a state variable, (2) the input to the valve actuator is a command variable, (3) the flow rate through the valve is an observable variable (with a suitable sensor), and (4) if the pressure out of the valve cannot be measured directly, it is a dependent variable. The value of a dependent variable is deduced based on the values of the state, observable, and command variables. For the propulsion model in this section, if the observable variable $P2$ is high and the state variable V2 is Open, then the dependent variable $PV2$ is High. For this chapter, the projected variables V_p will be all the state and observable variables.

The simple propulsion model presented in this section is shown in Figure 2-1. The propulsion model has a fuel tank, three valves, and a thruster. The thruster produces thrust whenever fuel can flow from the fuel tank to the thruster. This corresponds to turning on either valve $V2$ or $V3$, together with the first valve $V1$. A model is defined by a set of variables, domains, and clauses. The clauses are broken into two types: state constraints, which describe consistent state configurations, and transitions, which describe how one gets from a previous state to a next state. For example, $(F1 = Filled) \Rightarrow (P1 = High)$ is a state constraint, which says that the pressure $P1$ must be high if the fuel tank $F1$ is full. $(R1 = On) \wedge (CR1 = TurnOff) \Rightarrow (R1' = Off)$ is a transition constraint, which says that the next state

of the thruster $R1'$ is off if the command in $CR1$ is to turn off and the previous state $R1$ was on. Both of these types of clauses are merged into a single theory $\mathcal{C}_\Phi$ by conjoining all of them.

Variables and Domains

This section defines the variables V and domains $D(V)$ of the model. These variables are grouped by their type: state, command, observable, or dependent.

$State : Exp$

$F1 \in \{Filled, Empty\}$

$V1 \in \{Open, Closed\}$

$V2 \in \{Open, Closed\}$

$V3 \in \{Open, Closed\}$

$R1 \in \{On, Off\}$

$Command :$

$CV1 \in \{None, Open, Close\}$

$CV2 \in \{None, Open, Close\}$

$CV3 \in \{None, Open, Close\}$

$CR1 \in \{None, TurnOn, TurnOff\}$

$Observable :$

$P1 \in \{High, Low\}$

$P3 \in \{High, Low\}$

$T1 \in \{Thrust, NoThrust\}$

$Dependent :$

$P2 \in \{High, Low\}$

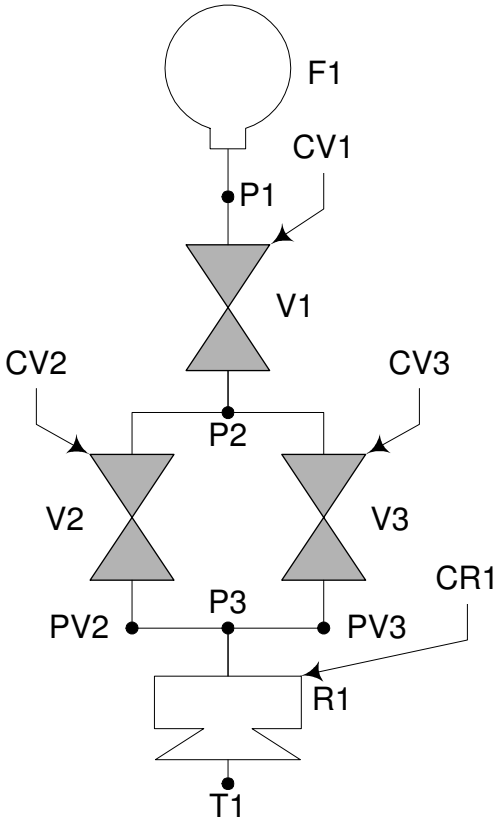

Figure 2-1: A simple propulsion model. The model has five components: a fuel tank $F1$, three valves $V1$, $V2$, and $V3$, and a thruster $R1$. The variables $P1$, $P2$, and $P3$ describe the pressure at their respective points in the system. The variable $T1$ describes the thrust produced by the thruster. $PV2$ and $PV3$ are the pressures out of each valve, $V2$ and $V3$, respectively, and are used to determine the pressure at $P3$. The variables $CV1$, $CV2$, $CV3$, and $CR1$ correspond to the commands into the components $V1$, $V2$, $V3$, and $R1$, respectively.

$PV2 \in \{High, Low\}$

$PV3 \in \{High, Low\}$

For the purpose of this chapter, the projected variables V_p are $\{F1, V1, V2, V3, R1, P1, P3, T1\}$. Other projected variables can be used depending on the compilation task, though these other cases will not be addressed.

State Constraints

Here we define the constraints for each component, relating the component variables at each point in time. These are expressed as the following clauses:

The fuel tank $F1$ outputs pressure only when Filled.

1. $(F1 = Filled) \Rightarrow (P1 = High)$
2. $(F1 = Empty) \Rightarrow (P1 = Low)$

The valves $V1$, $V2$ and $V3$ output their input when Open and output Low when they are closed.

Valve 1:

3. $(V1 = Open) \wedge (P1 = High) \Rightarrow (P2 = High)$
4. $(V1 = Open) \wedge (P1 = Low) \Rightarrow (P2 = Low)$
5. $(V1 = Closed) \Rightarrow (P2 = Low)$

Valve 2:

6. $(V2 = Open) \wedge (P2 = High) \Rightarrow (PV2 = High)$
7. $(V2 = Open) \wedge (P2 = Low) \Rightarrow (PV2 = Low)$
8. $(V2 = Closed) \Rightarrow (PV2 = Low)$

Valve 3:

9. $(V3 = Open) \wedge (P2 = High) \Rightarrow (PV3 = High)$
10. $(V3 = Open) \wedge (P2 = Low) \Rightarrow (PV3 = Low)$
11. $(V3 = Closed) \Rightarrow (PV3 = Low)$

The next constraints state that the pressure $P3$ at the junction is High whenever at least one valve is outputting High. Otherwise, if both are Low, it is also Low.

12. $(PV2 = Low) \wedge (PV3 = Low) \Rightarrow (P3 = Low)$
13. $(PV2 = High) \Rightarrow (P3 = High)$
14. $(PV3 = High) \Rightarrow (P3 = High)$

The thruster $R1$ only outputs thrust when it is On and it's input is High. Otherwise, it does not produce thrust.

15. $(R1 = On) \wedge (P3 = High) \Rightarrow (T1 = Thrust)$
16. $(R1 = On) \wedge (P3 = Low) \Rightarrow (T1 = NoThrust)$
17. $(R1 = Off) \Rightarrow (T1 = NoThrust)$

Combined, the above constraints constitute the state constraints of the theory $\mathcal{C}_\Phi$.

Transition Constraints

This section defines the transitions T of the model. A transition describes the evolution of a singe state variable, such as $F1$, from an initial value, such as $Filled$, to a final value, such as $Empty$. It has a guard that specifies the condition under which the transition is enabled. The guard for the transition $(R1 : On \rightarrow Off)$ is $(CR1 = TurnOff)$. A transition is equivalent to the clause $(R1 = On) \wedge (CR1 = TurnOff) \Rightarrow (R1' = Off)$, where $(R1' = On)$ is the next state of the variable[1].

$$F1: \quad Filled \rightarrow Empty \quad | \, Guard(T1 = Thrust)$$

[1]Note that each transition also has a probability associated with it, reflecting the chance that the transition is taken when the transition is enabled. For this model, all transitions have a probability of 1.0 except for the transition $(F1 : Filled \rightarrow Empty)$, whose probability is 0.01. Compilation preserves these probabilities, but does not use them.

The probability of 0.01 indicates that there is a one percent chance per second that the tank will be empty whenever the thruster is producing thrust. Thus, the fuel tank is expected to be empty after about 100 seconds of use, on average.

Each of the three valves closes when it is commanded to close and opens when it is commanded to open.

$$\begin{aligned} V_i: \quad & Open \rightarrow Closed \quad | \, Guard(CV_i = Close) \\ & Closed \rightarrow Open \quad | \, Guard(CV_i = Open) \end{aligned}$$

The thruster turns on when it is commanded to turn on, and turns off when commanded to turn off.

$$\begin{aligned} R1: \quad & On \rightarrow Off \quad | \, Guard(CR1 = TurnOff) \\ & Off \rightarrow On \quad | \, Guard(CR1 = TurnOn) \end{aligned}$$

2.2 Previous Research on Model Compilation

Model compilation is the process of compiling a model from its user-specified representation to a form more suitable for an engine to handle efficiently. Model compilation allows for a compact representation of the model, that facilitates responsiveness. In model-based autonomy [16], model compilation improves the reactiveness of the system.

Extraneous constraints and variables in the model can significantly slow down the constraint solver. The constraints and variables are often extraneous either because they describe behaviours that are impossible to distinguish, due to a lack of observability or controllability, or because they include relations that are not necessary for the particular application, such as estimation or control.

Model compilation involves eliminating both extraneous constraints and variables, as well as generating constraints that represent specialized subsets of the model. These specialized constraints are suited for particular reasoning steps. For example, when estimating the state of the system, one will often wish to determine if the estimated state is consistent. This can be determined by examining which states are allowed, given the observations. If one were to compile a relation that only describes the interaction between observations and states, one can eliminate all the variables related to control and abstract away the variables that describe the interaction between states. Both of these types of variables and their associated constraints do not contribute to the consistency of the estimate with the observations. These specialized constraints reduce the amount of deduction required to infer particular conclusions about the system. Consider the propulsion model from Section 2.1.1. The model has 5 state variables and 3 observable variables; the other 6 variables can be eliminated. Projecting out these 6 variables reduces the model's 17 state constraints down to 10 state constraints.

Johan de Kleer [6], Rina Dechter [10], and Alvaro del Val [8] present approaches to knowledge compilation through the generation of prime implicates or resolution. Prime implicates simplify the constraints into a reduced form; resolution eliminates variables.

De Kleer presents incremental prime implicant generation based on tries. In this scheme, the algorithm generates all possible resolvents, while incrementally pruning all clauses that are not prime. The key contribution of his work is an indexing scheme, called a trie, for efficiently performing the pruning step.

Dechter presents an algorithmic framework called bucket elimination that is a generalization of dynamic programming. She applies bucket elimination to finding optimal candidates based on a weighting function, using directional resolution, which is the fundamental core of DPLL [5] [4]. The directional resolution algorithm is able to decide if a theory is satisfiable by performing resolution on the theory, where bucket elimination is used to speed up the resolution process. Both bucket elimination and a directional resolution algorithm that uses bucket elimination are key contributions provided by Dechter.

Alvaro del Val focuses on consequence-finding using kernel resolution on a compact, symbolic encoding of clauses, called zero-suppressed binary decision diagrams (ZBDD). This approach uses a language $\mathcal{L}$ to specify which sets of prime implicates are desired and then uses bucket elimination in its kernel resolution to generate all the prime implicates that are a member of that language. The approach performs resolution on multiple clauses simultaneously by performing resolution on symbolic ZBDD encodings of sets of clauses. This reduces the space required to encode the clauses, and improves performance, by reducing the number of resolution steps that need to be performed. This approach also uses buckets to specify the order in which variables are processed, where the order in which variables are processed can effect the amount of time it takes to solve the problem. The key contribution of del Val is

a ZBDD-based implementation that performs significantly better than the trie-based prime implicate generator.

The approach pursued in this thesis is to generate prime implicates, but to take also into account the elimination of variables. This will be called *projected* prime implicates. This thesis builds off of the work done by Robert Ragno [14]. Ragno developed an algorithm that generates all full assignments to the projected variables using a clause-directed A* search algorithm. The algorithm was also testing for validity and so was able to prune implicants. Since the algorithm is built upon an A* search, the algorithm prunes implicants by throwing away a search node. The algorithm lacks two essential features for the purpose of model compilation: first, in model compilation, the cost of each assignment is uniform, so the A* search turns into a breadth-first search which requires an exponential amount of memory, and second, the algorithm is designed to generate full assignments to the projected variables, and thus implicates, but not prime implicates. To get prime implicates, and additional step of unifying the implicates into prime implicates needs to be performed.

While none of these algorithms provide a solution to the problem by themselves, there are two straight-forward methods to generate projected prime implicates, by performing projected and implicate generation as two separate steps:

The first method generates all the prime implicates first and then throwing away all the prime implicates that mention unprojected variables. The algorithms provided by de Kleer and del Val are suitable for this method. Independent of the number of variables being eliminated, this scheme has the same cost as generating all the prime implicates. If a number of variables are being eliminated, then many prime implicates will be generated and then thrown away. As stated above, the cost of generating all of the prime implicates is exponential in the number of variables, so this method is impractical for systems of realistic size. More specifically, let p be the number of projected variables and u be the number of unprojected variables. Then the cost is

$C^p C^u$, hence the cost is exponentially more expensive.

The second method of generating projected prime implicates involves projecting the model by using resolution to eliminate each projected variable, one at a time, and then generating the prime implicates of the projected model. This method is suitable for using either Dechter's work or Ragno's work, though it also requires a prime implicate generator. Since resolution is exponential in the number of clauses, and prime implicate generation is exponential in the number of variables, the projected variables in this case, this approach still requires using two different exponential algorithms.

This thesis presents an algorithm that performs the projection and prime implicate generation in a single step. When many variables are eliminated, this algorithm enumerates over a much smaller space of values and can thus solve the problem quickly, unlike the two-step methods. This compilation algorithm has been developed in the context of a model-based autonomy engine called Titan [16] . It can enabled autonomy in a real-time environment [2] [18].

Chapter 3

Model Compilation as Projected Prime Implicate Generation

This chapter presents algorithms to solve two distinct knowledge compilation problems: (1) projected prime implicate generation and (2) projected prime implicant generation. A projected solution is one where only a subset of the variables are used in the solution. Projection is precisely defined in Section 3.1. Both of these problems arise when compiling constraint-based models, where each problem represents a different way of treating model uncertainty. (1) produces a compiled model, where all consistent assignments of the original model are consistent with the compiled model. This is suitable for tasks like consistency-based diagnosis. For example, if a faulty light switch is capable of turning on a light, then the light being on is consistent with the model when the switch is faulty, and hence a valid diagnosis. (2) produces a compiled model where all consistent assignments of the compiled model are consistent with the original, uncompiled theory. This is suitable for planning problems, which involve selecting actions that must have a guaranteed effect. With the faulty switch example above, it is possible for the light to be on or off when the switch is broken.

Thus, to turn the light on, the planner will need to repair the switch; otherwise, when the switch is faulty, the light is not guaranteed to be on.

Note that model compilation involves generating a set of theories, each of which is suitable for specific tasks, such as diagnosis and planning. In each of these cases, the original theory almost always contains three classes of variables: variables that convey state, variables that describe an interaction with the world, both observations and commands, and variables that convey interactions between different state variables, called dependent variables in this thesis. Dependent variables are uniquely determined by the current state, observations, and issued commands. Often, real-world models have many more dependent variables than state variables. Thus, a projected theory that projects dependent variables out of the theory can have significantly fewer variables as compared to the original theory. Since the reasoning tasks performed on the theory are exponential in the number of variables, this compilation can provide a dramatic savings. This savings has been realized for both types of compilation methods in our application context of model-based autonomy [16].

An import feature of the algorithms presented in this thesis is that they do not require first projecting the theory, and they do not require generating all prime implicates or implicants prior to projection. Performed as two steps, two exponential operations are required. Instead, projection and prime implicate or implicant generation are performed within a single step. By performing the task in a single step, the generation process can save significant amounts of time over algorithms that perform the task in two steps.

This chapter begins by defining two supporting concepts, prime implicates and implicants, in Section 3.1 and 3.2, respectively. The algorithm for generating projected prime implicates is presented next in 3.3, and is followed by the algorithm for generating projected prime implicants in 3.4. The solutions to these two problems presented in this chapter are based on a projected prime implicate generator, which

will be detailed in Chapter 4.

3.1 Projected Prime Implicates

Given a theory, an implicate of the theory is a logical consequence of the theory that is represented as a disjunctive clause. Viewing a clause as a set of literals, a *prime* implicate is an implicate such that no proper subset of it is an implicate. The complete set of prime implicates offers a more compact encoding of the theory that is logically equivalent to the complete set of implicates.

More precisely, a theory $\mathcal{C}_\Phi$ is represented as a set of *disjunctive propositional clauses*, which is logically equivalent to a *Conjunctive Normal Form* (CNF) *sentence*. In particular, each clause is a disjunction of literals: $(c_1 \vee c_2 \vee \ldots)$, where a positive literal c_1 is some proposition a and a negative literal c_2 is the negation of some proposition, that is $\neg a$. We develop our compilation methods in the context of propositional state logic. In propositional state logic, a proposition a is an assignment $(v = x)$ of a value x from the domain of the variable v to a variable v. The domain of a variables is written as $D(v)$.

Definition: *Given a theory $\mathcal{C}_\Phi$, C_I is an implicate of $\mathcal{C}_\Phi$ iff C_I is a clause and $\mathcal{C}_\Phi \models C_I$.*

An implicate is true for every assignment in which the original theory is true. For example, consider the theory

$$\begin{aligned}\{&(Switch = Off) \Rightarrow (Power = Low),\\ &(Switch = On) \Rightarrow (Power = High),\\ &(Power = Low) \Rightarrow (Light = Dark),\\ &(Power = High) \Rightarrow (Light = Lit)\}.\end{aligned} \tag{3.1}$$

It is important to note that the algorithms presented in this thesis operate on variables and their domains, not on raw binary-domain literals. For a constraint solver that only operates on binary literals, the mutual exclusion and exhaustion constraints that describe a variable's domain also need to be included in the theory. The mutual exclusion constraints require that every variable have at most 1 value assigned to it. The exhaustion constraints require that every variable have at least 1 value assigned to it. Thus, combined, these constraints require that variables have a single consistent assignment. The algorithms in this thesis operate on the domains of the variables directly, so these domain constraints are enforced by the algorithm instead of by the theory. This substantially reduces the size of the theory.

An implicate of Theory 3.1 is

$$\neg(Switch = On) \vee \neg(Light = Dark). \tag{3.2}$$

The sentence 3.2 is an implicate because it logically follows from the second and fourth clauses of Theory 3.1. The implicate states that the switch must either be off or the room must be lit. The second clause of the theory states that either the switch is off or power is present. The fourth clause states that either there is no power or the light is lit. The implicate follows from the fact that, if there is no power present, then from clause two, the switch is off; however, if power is present, then by clause four the light is lit.

Definition: *A clause C_P is a prime implicate of theory $\mathcal{C}_\Phi$ iff C_P is an implicate of the theory, and there exists no other implicate C_I of the theory such that C_I entails C_P:*

$$(\mathcal{C}_\Phi \models C_P) \wedge \forall C_I.\,((\mathcal{C}_\Phi \models C_I) \wedge (C_I \models C_P) \Rightarrow (C_I = C_P)).$$

This set of prime implicates is also written as $\{C_P | \mathcal{C}_\Phi \models C_P\}_{prime}$.

In the above example, Sentence 3.2 is a prime implicate. Sentence 3.2 is prime because neither of its subsets, $\neg(Switch = On)$ and $\neg(Light = Dark)$, are implicates. For $\neg(Switch = On)$ to be an implicate, it must always be true that the switch is off. However, the Theory 3.1 can consistently have the switch on, namely when the power is high and the light is lit. For $\neg(Light = Dark)$ to be an implicate, it must always be true that the light is lit. Once again, the theory in Equation 3.1 can consistently have the light dark, namely when the power is low and the switch is off. Thus, neither of these subsets are implicates. Conversely, $\neg(Switch = On) \vee \neg(Power = Low) \vee \neg(Light = Dark)$ is an implicate of the theory; however, it is not a prime implicate because it is an implicate of Equation 3.2.

The implicates of a theory make explicit all of the logical consequences of the theory, but often contain a significant level of redundancy. Prime implicates remove much of this redundancy yet are sufficient to encode all logical consequences of a theory. However, the set of prime implicates can still be quite large. This set can be further reduced based on the observation that for a particular task, only a small subset of the prime implicates are relevant. Concepts like theory implicates [15] and task-driven abstraction [13] have been defined in order to specify different concepts of task relevance.

In the model compilation tasks addressed by this thesis we are only interested in implicates that relate a small subset V_S of the system variables V. Implicates relating other, intermediate variables, are superfluous to the task. In particular, given independent knowledge (generally observations) about consistent assignments to some of the variables $V_O \subseteq V_S$, the task is to use $\mathcal{C}_\Phi$ to determine the consistent or necessary assignments to one or more of the other variables in V_S, that is, variables in $V_S \setminus V_O$. In this case, only those implicates involving *only* assignments to V_S are

relevant. That is, the consistent or necessary assignments with respect to $\mathcal{C}_\Phi$ are exactly the same as those with respect to $\mathcal{C}_\Phi \Downarrow_{V_S}$.

Theorem 1 *Let V_S be a subset of all the variables V. Let a subset V_O of V_S be the set of all variables for which independent knowledge (generally observations) is available. Let $\mathcal{C}_\Phi$ be a theory over V. Let the theory $\mathcal{C}_\psi$ be comprised of the implicates of $\mathcal{C}_\Phi$ involving only assignments to the variables V_S. Then an assignment C_A to the variables V_S is consistent with $\mathcal{C}_\psi$ iff it is consistent with $\mathcal{C}_\Phi$.*

To prove this we need this additional lemma:

Lemma 1 *If a clause L is inconsistent with a theory $\mathcal{C}_\Phi$, then $\neg L$ must be an implicate of $\mathcal{C}_\Phi$. (In this case, the clause L is called a conflict.)*

Proof:

Let $\mathcal{C}_F$ be the set of full assignments to the variables V that are consistent with $\mathcal{C}_\Phi$. Since L is inconsistent with $\mathcal{C}_\Phi$, L must be inconsistent with all the elements of $\mathcal{C}_F$, as each one constrains every variable, and each one is consistent with $\mathcal{C}_\Phi$. Thus, if they are all inconsistent with L, they must all be consistent with $\neg L$. This implies that every assignment that is consistent with $\mathcal{C}_\Phi$ is consistent with $\neg L$, which means $\neg L$ is an implicate of $\mathcal{C}_\Phi$. □

Now we are ready to prove Theorem 1.

Proof:

($\Rightarrow$) Assume, for contradiction, that C_A is consistent in the projected model $\mathcal{C}_\psi$, while it is inconsistent in the original model $\mathcal{C}_\Phi$. By Lemma 1, since C_A is inconsistent with $\mathcal{C}_\Phi$, $\neg C_A$ must be an implicate of $\mathcal{C}_\Phi$. C_A, by definition, only contains assignments to variables in V_S, so $\neg C_A$ must also only contain assignments to variables in V_S. Thus, $\neg C_A$ cannot be eliminated from $\mathcal{C}_\Phi$, so it must exist in $\mathcal{C}_\psi$. This implies that

C_A is inconsistent with $\mathcal{C}_\psi$. But this also contradicts our assumption. Thus, if C_A is consistent with $\mathcal{C}_\psi$, it must also be consistent with $\mathcal{C}_\Phi$.
($\Leftarrow$) Since the projection process is removing implicates from the original theory $\mathcal{C}_\Phi$, the new theory $\mathcal{C}_\psi$ is a subset of $\mathcal{C}_\Phi$. Thus, any C_A that is consistent with $\mathcal{C}_\Phi$ must still be consistent with $\mathcal{C}_\psi$. □

For example, for diagnosis, one needs to know only the direct relationship between the variables representing the system observables and the variables representing the system state. Hence, only prime implicates involving these variables are relevant, and any prime implicate that refers to one or more other variables is irrelevant in this context. There are two types of these other variables, variables that command the system into new states, and dependent variables that are determined solely by the other three types of variables. The dependent variables are neither observed nor needed for diagnosing the state. For instance, with the theory in Equation 3.1, "power" is a dependent variable. One can diagnose that there is a problem with the switch if the switch is on and the light is observed dark, without needing to know that the power must also be low. Similarly, with the command variables, one does not need to know how the state is going to evolve, in order to determine whether the current state estimate is consistent or inconsistent with the observations. For example, if a light switch is supposed to be on and the light is off, it is unimportant to know that the light switch is supposed to turn off soon. According to the model, the transition has not yet happened, so the light should still be on and is not.

To describe the prime implicates relevant to a subset of the variables, we introduce the term *projected prime implicate*. We call the interesting subset of the variables the *projected variables*. We use V_p to denote the projected subset of variables V and $\mathcal{C}_\Phi \Downarrow_{V_p}$ to denote the projection of theory $\mathcal{C}_\Phi$ onto V_p.

Definition: *Let $\mathcal{C}_\Phi$ be a theory over variables V. The clause C_P is a Projected Prime Implicate of $\mathcal{C}_\Phi$ onto $V_p \subseteq V$ iff C_P is a prime implicate of $\mathcal{C}_\Phi$, and if for all*

assignments $(x_i = v_i)$*, in* C_P*,* $v_i \in V_p$*.*

A variable v_i *of* V *is said to be projected out relative to* C_P *iff* $v_i \in (V \setminus V_p)$*.*

For example, projecting out the variable "power" in the switch example given by the Theory 3.1 results in two projected prime implicates. The first was given in Equation 3.2. The second is the clause $\neg(Switch = Off) \vee \neg(Light = Lit)$. Note that Theory 3.1 has four other prime implicates, all of which involve the eliminated variable power, and hence are unnecessary. An example is the prime implicate $\neg(Switch = On) \vee \neg(Power = Low)$.

3.2 Projected Prime Implicant

Given a theory, an *implicant* of the theory is a conjunctive clause that implies the theory. A *prime implicant* of a theory is an implicant of the theory that has a minimal number of literals. The complete set of prime implicants is logically equivalent to the complete set of implicants and hence represents a more compact encoding of the theory.

More precisely, a theory of prime implicants $\mathcal{C}_N$ is represented as a set of conjunctive propositional clauses, which is logically equivalent to a Disjunctive Normal Form (DNF) sentence. Each clause is a conjunction of literals: $(c_1 \wedge c_2 \wedge \ldots \wedge c_n)$. As with implicates, c_1-c_n are positive or negative literals. The set represents a disjunction, that is at least one of the clauses must hold. Prime Implicants and Prime Implicates are dual representations of the same theory. Without projection, they are equivalent, and the particular theory determines which one is more compact. Implicants are more compact in the case where the theory has few solutions, and implicates are more compact in the case where the theory has many solutions. The choice is also affected by which encoding is more explicit for the task at hand.

Definition: *Given a theory $\mathcal{C}_\Phi$, C_N is an implicant of $\mathcal{C}_\Phi$ iff C_N is a conjunctive clause and $C_N \models \mathcal{C}_\Phi$.*

Note that whenever an implicant is true, the original theory is also true. For example, if the theory is

$$\begin{aligned}
&((Switch = Off) \Rightarrow (Power = Low)) \wedge \\
&((Switch = On) \Rightarrow (Power = High)) \wedge \\
&((Power = Low) \Rightarrow (Light = Dark)) \wedge \\
&((Power = High) \Rightarrow (Light = Lit)) \wedge \\
&((Heater = On) \Rightarrow (Temp = Warm))
\end{aligned} \tag{3.3}$$

then an implicant of the theory is

$$(Switch = Off) \wedge (Power = Low) \wedge (Light = Dark) \wedge (Heater = Off) \tag{3.4}$$

Definition: *The clause C_R is a prime implicant of the theory $\mathcal{C}_\Phi$ iff C_R is an implicant of the theory and there exists no other implicant C_N of the theory such that C_R is an implicant of C_N:*

$$(C_R \models \mathcal{C}_\Phi) \wedge \forall C_N . ((C_N \models \mathcal{C}_\Phi) \wedge (C_R \models C_N) \Rightarrow (C_R = C_N))$$

This set of prime implicants is also written as $\{C_R | C_R \models \mathcal{C}_\Phi\}_{prime}$.

In the above example, Equation 3.4 is a prime implicant. Equation 3.4 is prime because neither of its subsets, $(Switch = Off)$, $(Switch = Off) \wedge (Light = Dark)$, etc. are implicants. For $(Switch = Off)$ to be an implicant, it must be true that when the switch is off, all other variables can take on any value, as they are unconstrained. However, the theory implies that the light must be dark and the power low

when the switch is off. $((Switch = Off) \wedge (Power = Low) \wedge (Light = Dark) \wedge (Heater = Off) \wedge (Temp = Cold))$, for example, is an implicant that is not a prime implicant. This implicant is an implicant of both the theory and Equation 3.4.

All extensions of an implicant are consistent with the theory; hence, there are often many of them, and they often contain a significant amount of redundancy. Prime implicants remove much of this redundancy; however, they still encode all possible assignments that are consistent with the theory.

As with implicates, one desires a relevant sub-set that is suitable for performing a specific task. For example, suppose a reasoning system is given a set of goals and asked to find one or more states in which all consistent states include the goals. The states are such that they all imply that the system has reached its goal with certainty. To perform this task, the reasoning system needs a relation between goals and the set of states. One possible representation for this relation is a set of implicants of a theory that is projected onto the state variables and a goal variable of interest. If the implicants of the theory are also implicants of the goal, then when the system is in the specified state, it is both consistent for the system to be in that state, and it ensures that the goal is reached. Thus, by this design, only the variables describing the state of the system and the single goal of interest are relevant in the final theory, for a particular goal, which is a small subset of the total set of variables. For example, with the Theory 3.3, if the goal is that one wants the light to be lit, then one would like to generate the clause $(Switch = On) \wedge (Light = Lit)$. This clause indicates that the only way to ensure that the light is lit is to ensure that the switch is on. However, this clause is not an implicant of the Theory 3.3. An implicant would also require the power to be high and the heater to be off. However, $(Switch = On) \wedge (Light = Lit)$ is an implicant of Theory 3.3, projected onto the variables switch and light. By projecting the model first, the goals only require that a consistent assignment exists to the variables projected out. This is in contrast to the original problem formulation,

which requires that any assignment to the variables projected out be consistent. Thus, projected implicants are implicants of the projected prime implicates, rather than of the original theory. If one does not project the theory first, then following the same steps will generate implicants that are projected prime implicants of the original theory. As a result, traditional projected prime implicants are a simple variation of the algorithm in this thesis. Section 3.4 presents an algorithm that can generate the kinds of implicants we want, such as $(Switch = On) \wedge (Light = Lit)$.

Definition: *Let $\mathcal{C}_\Phi$ be a theory over variables V, then the clause C_P is a Projected Prime Implicant of $\mathcal{C}_\Phi$ onto $V_p \subseteq V$ iff C_P is a prime implicant of $\mathcal{C}_\Phi$ and if for all assignments $(x_i = v_i)$ in C_P, $v_i \in V_p$.*

If one were to project out the variable temp from Theory 3.3, for example, then Equation 3.4 is one of the implicants that remains. All of the implicants that involve temp would be removed.

For example, model compilation for planning involves reasoning about a relationship between a subset V_S of the variables V that guarantee (entail) a set of goals. In particular, given a goal configuration as a set of assignments C_G to a subset V_G of the variables in V_S, the planning task is to determine the consistent or necessary assignments to one or more of the other variables in V_S, such that the desired configuration C_G is entailed by the assignments. This entailment must hold for any assignments to the remaining variables in V_S. In addition, there must exist a consistent set of assignments to the remaining variables in $V \setminus V_S$. That is, all extensions to the assignment over the variables V_S must be consistent. When all the extensions to a clause of assignments C_A is consistent with the theory $\mathcal{C}_\Phi$, then C_A is called *valid* with respect to $\mathcal{C}_\Phi$. Note that implicates only requires that the candidate be consistent with the theory, meaning it has at least one extension that is also consistent, rather than that they all be consistent.

Theorem 2 *Let V_S be a subset of all the variables V. Let $\mathcal{C}_\Phi$ be a theory, and let the theory $\mathcal{C}_\psi$ be comprised of the implicants of $\mathcal{C}_\Phi$ involving only assignments to the variables V_S. Then an assignment C_A to the variables V_S is valid with $\mathcal{C}_\psi$ iff it is valid with $\mathcal{C}_\Phi$.*

To prove this we need this additional lemma:

Lemma 2 *If a clause C_A is valid with the theory $\mathcal{C}_\Phi$, then C_A must be an implicant of $\mathcal{C}_\Phi$.*

Proof:

Let $\mathcal{C}_F$ be the set of full-assignment extensions to C_A over the variables V that are consistent with $\mathcal{C}_\Phi$. Since C_A is valid with the theory $\mathcal{C}_\Phi$, $\mathcal{C}_F$ must be the set of all full-assignment extensions to C_A, and each of these is consistent with $\mathcal{C}_\Phi$. For C_A to be an implicant of $\mathcal{C}_\Phi$, C_A must entail $\mathcal{C}_\Phi$. This is equivalent to requiring that all extensions of $\mathcal{C}_F$ entail $\mathcal{C}_\Phi$. Since they are all full extensions, this simplifies into requiring that all extensions of $\mathcal{C}_F$ be consistent with $\mathcal{C}_\Phi$. But we already know this is true, so C_A must be an implicant of $\mathcal{C}_\Phi$. □

Now we are ready to prove Theorem 2.

Proof:

($\Rightarrow$) Assume, for contradiction, that C_A is valid with the projected model $\mathcal{C}_\psi$, while it is not valid with the original model $\mathcal{C}_\Phi$. By Lemma 2, since C_A is valid with the projected model $\mathcal{C}_\psi$, C_A must be an implicant of $\mathcal{C}_\psi$, and therefore be an element of $\mathcal{C}_\psi$. Since $\mathcal{C}_\psi$ is a subset of $\mathcal{C}_\Phi$, C_A must also be an element of $\mathcal{C}_\Phi$. Since C_A is an element of $\mathcal{C}_\Phi$, it must be valid with $\mathcal{C}_\Phi$, contradicting the assumption.

($\Leftarrow$) Assume, for contradiction, that C_A is not valid with the projected model $\mathcal{C}_\psi$, while it is valid with the original model $\mathcal{C}_\Phi$. Since C_A is valid with $\mathcal{C}_\Phi$, it must be an implicant in $\mathcal{C}_\Phi$. C_A only contains the variables V_S, so it must be the case that it was

not removed from $\mathcal{C}_\Phi$. But then C_A must be an implicant of $\mathcal{C}_\psi$, so it must be valid with $\mathcal{C}_\psi$. □

Projecting the Theory 3.3 over just the switch and light variables, thus allowing the heat, power, and temp variables to take on any consistent value, creates a theory with only two prime implicants: $(Switch = On) \wedge (Light = Lit)$ and $(Switch = Off) \wedge (Light = Dark)$. These describe all the consistent combinations of these two variables. Thus, the switch being on implies that the light is lit, as desired.

Consider again the difference between prime implicates and prime implicants. For example, consider the Theory 3.3. The set of prime implicates of this theory are

$$\begin{aligned}
&\neg(Switch = Off) \vee \neg(Power = High),\\
&\neg(Switch = On) \vee \neg(Power = Low),\\
&\neg(Power = Low) \vee \neg(Light = Lit),\\
&\neg(Power = High) \vee \neg(Light = Dark),\\
&\neg(Switch = Off) \vee \neg(Light = Lit),\\
&\neg(Switch = On) \vee \neg(Light = Dark), \text{ and}\\
&\neg(Heater = On) \vee \neg(Temp = Cool).
\end{aligned}$$

The set of prime implicants of the theory are

$$\begin{aligned}
&(Switch = Off) \wedge (Power = Low) \wedge (Light = Dark) \wedge (Heater = Off),\\
&(Switch = On) \wedge (Power = High) \wedge (Light = Lit) \wedge (Heater = Off),\\
&(Switch = Off) \wedge (Power = Low) \wedge (Light = Dark)\\
&\qquad \wedge (Heater = On) \wedge (Temp = Warm), \text{ and}\\
&(Switch = On) \wedge (Power = High) \wedge (Light = Lit)\\
&\qquad \wedge (Heater = On) \wedge (Temp = Warm).
\end{aligned} \tag{3.5}$$

Consider the case where the variable temp is eliminated. The projected theory has the two prime implicants, the first two: $(Switch = Off) \wedge (Power = Low) \wedge (Light = Dark)$ and $(Switch = On) \wedge (Power = Low) \wedge (Light = Dark)$. The variable heater has also been eliminated as its value no longer matters when the variable temp is ignored.

3.3 Projected Prime Implicate Generation

In Chapter 4, a conflict-directed enumeration algorithm for generating all projected prime implicates is developed. This algorithm, called *primeImplicates*, takes four inputs: the theory $\mathcal{C}_\Phi$, the set of projected variables V_p, a set of implicants $\mathcal{A}_P$ and a set of conflicts $\mathcal{A}_F$. The algorithm returns a set of projected prime implicates $\mathcal{C}_P$ onto the projected variables V_p. Here the elements of $\mathcal{C}_P$ are projected prime implicates, as defined in Section 3.1. $\mathcal{A}_P$ and $\mathcal{A}_F$ extend the basic projected implicate equation from $\{C_P | \mathcal{C}_\Phi \Downarrow_{V_p} \models C_P\}_{prime}$ to the equation $\{C_P | (\mathcal{C}_\Phi \vee \mathcal{A}_P) \Downarrow_{V_p} \models C_P, \neg(\neg\mathcal{A}_F \models C_P)\}_{prime}$. The addition of implicants $\mathcal{A}_P$ into the first part of the equation adds the implicants $\mathcal{A}_P$, and all extensions to them, to the set of consistent solutions of the projected prime implicates $\mathcal{C}_P$. The second part of the equation states that the projected prime implicates $\mathcal{C}_P$ must not contain all of the same solutions described by the implicates $\neg\mathcal{A}_F$. Thus, the implicants $\mathcal{A}_P$ add solutions to $\mathcal{C}_P$, while the conflicts $\mathcal{A}_F$ constrain the solutions to lay outside of region defined by the conflicts.

3.4 Projected Prime Implicant Generation As Projected Prime Implicate Generation

This section describes how to generate all projected prime implicants C_R that satisfy the theory $\mathcal{C}_\Phi$ and the additional constraints $\mathcal{C}_\rho$:

$$C_R \models (\mathcal{C}_\rho \wedge \mathcal{C}_\Phi) \Downarrow_{V_p} \tag{3.6}$$

The additional constraints $\mathcal{C}_\rho$ is used to specify a simple, easily inverted set of constraints.

Proposition 1 *The implicate generation problem $C_I \models \mathcal{C}_\Phi$ is equivalent to the implicant generation problem $\neg\mathcal{C}_\Phi \models \neg C_I$. Both are reduced to the same problem, namely $\forall C_I.\mathcal{C}_\Phi \wedge \neg C_I$ is inconsistent.*

Thus Equation 3.6 is equivalent to

$$(\neg\mathcal{C}_\rho \vee \neg\mathcal{C}_\Phi) \Downarrow_{V_p} \models \neg C_R \tag{3.7}$$

The theory in this case is $\mathcal{C}_\rho \wedge \mathcal{C}_\Phi$, which allows for the specification of a common theory and some additional constraints $\mathcal{C}_\rho$, as needed. Note that this approach assumes that $\mathcal{C}_\Phi$ is a large theory that one does not wish to invert, namely the $\neg\mathcal{C}_\Phi$ in Equation 3.7, while $\mathcal{C}_\rho$ is a small theory that is easily inverted. The cost of inverting a theory is worst case exponential in the number of variables of the theory, both in terms of time and space. The exponent results from applying the distribution law. This cost makes inversion intractable for large models.

A specific goal relation is typically represented as $\mathcal{C}_\rho$. For example, in the context of a simple switch, $(Power = On)$ is a goal relation $\mathcal{C}_\rho$ whose inverse $\neg\mathcal{C}_\rho$ is $\neg(Power = On)$. The model for the simple switch is represented in $\mathcal{C}_\Phi$.

Without $\mathcal{C}_\rho$, one could generate projected prime implicants for $\mathcal{C}_\Phi$ by projecting $\mathcal{C}_\Phi$ and then generating the implicants. However, since $\neg\mathcal{C}_\rho$ may not be expressed in terms of just the projected variables V_p, it is necessary to solve the problem in the context of the theory $\mathcal{C}_\Phi$, which contains the relationship between the variables of V_p and the variables used in $\mathcal{C}_\rho$. We derive the new problem formulation by conjoining to Equation 3.7 the term $\mathcal{C}_\Phi \vee \neg\mathcal{C}_\Phi$:

$$((\mathcal{C}_\Phi \vee \neg\mathcal{C}_\Phi) \wedge (\neg\mathcal{C}_\rho \vee \neg\mathcal{C}_\Phi)) \Downarrow_{V_p} \models \neg C_R. \tag{3.8}$$

Applying distribution, this expands to:

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \vee (\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\Phi) \vee (\neg\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \vee (\neg\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\Phi)) \Downarrow_{V_p} \models \neg C_R. \tag{3.9}$$

The second term $(\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\Phi)$ simplifies to $false$ and is eliminated. The fourth term $(\neg\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\Phi)$ simplifies to $\neg\mathcal{C}_\Phi$. The first term $(\neg\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho)$ is then subsumed by the fourth term, $\neg\mathcal{C}_\Phi$, and is eliminated, resulting in:

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \vee (\neg\mathcal{C}_\Phi)) \Downarrow_{V_p} \models \neg C_R. \tag{3.10}$$

Notice that Equation 3.10 is in the form $(\mathcal{C}'_\Phi \vee \mathcal{A}_P) \Downarrow_{V_p} \models C_P$. Thus, if we let $\mathcal{C}'_\Phi = (\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho)$ and $\mathcal{A}_P = (\neg\mathcal{C}_\Phi)$, the projected prime implicate generator that will be presented in Chapter 4 is capable of solving this problem. Notice also that this problem can also be solved by instead letting $\mathcal{C}'_\Phi = ((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \vee (\neg\mathcal{C}_\Phi))$ and $\mathcal{A}_P = \{\}$; however, $\mathcal{C}'_\Phi$ must be in conjunctive normal form. We make the assumption that converting $\mathcal{C}'_\Phi$ back into conjunctive normal form in this latter case is more expensive than treating $\neg\mathcal{C}_\Phi$ as a set of implicants $\mathcal{A}_P$ in the generation process, as per the former option. This former approached is represented by Equation 3.11.

The above derivation results in the following problem formulation

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \Downarrow_{V_p} \models \neg C_R) \text{ with Implicants } \neg\mathcal{C}_\Phi \tag{3.11}$$

or, equivalently,

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \vee \neg\mathcal{C}_\Phi) \Downarrow_{V_p} \models \neg C_R. \tag{3.12}$$

As was suggested in Section 3.2, for planning problems, the theory should be projected prior to implicant generation. Projecting the theory first results in a two-step solution:

$$\mathcal{C}_\Phi \Downarrow_{V_p} \models \mathcal{C}_\psi \tag{3.13}$$

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \Downarrow_{V_p} \models \neg C_R) \text{ with Implicants } \neg\mathcal{C}_\psi \tag{3.14}$$

This is equivalent to:

$$\mathcal{C}_\Phi \Downarrow_{V_p} \models \mathcal{C}_\psi \tag{3.15}$$

$$((\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho) \Downarrow_{V_p} \vee\neg\mathcal{C}_\psi) \models \neg C_R. \tag{3.16}$$

The two steps exactly correspond to the two lines of the following generatePrimeImplicants algorithm.

3.4.1 Algorithm

generatePrimeImplicants(V_p, $\mathcal{C}_\Phi$, $\neg\mathcal{C}_\rho$)

$\mathcal{C}_\psi \leftarrow \text{primeImplicates}(V_p, \mathcal{C}_\Phi, \{\}, \{\})$

$\mathcal{C}_R \leftarrow \neg\text{primeImplicates}(V_p, (\mathcal{C}_\Phi \wedge \neg\mathcal{C}_\rho), \mathcal{C}_\psi, \{\})$

return $\mathcal{C}_R \Downarrow_{V_p}$

3.5 Summary

This chapter has shown how to solve the projected prime implicate and prime implicant generation problems using a projected prime implicate generator. From a general-purpose model, both of these algorithms generate a smaller task-specific model. Both generators project the theory onto a subset of its variables; eliminating extraneous variables, such as dependent variables that relate states to each other. Projected prime implicates provide a compiled theory that can test consistency in place of the original model; projected prime implicants provide a compiled theory that can test validity in place of the original model. Generating projected prime implicants can be reduced to generating projected prime implicates. The next chapter will provide the details on how to generate projected prime implicates, as well as on how to incorporate conflict and implicant pruning.

Chapter 4

Prime Implicate Generation

This chapter describes a fast prime implicate generation algorithm that performs the model compilation tasks of Chapter 3. The prime implicate generation algorithm exploits both conflicts and implicants to perform pruning of the search space, allowing for up to two orders of magnitude improvements over an algorithm that does not use these methods.

This chapter presents a novel algorithm for pruning based on implicants. In order to make this algorithm efficient, this chapter also presents an efficient means for testing for validity, which allows the tester to identify implicants. Validity testing is fast through the combined use of a clause-directed search strategy and the use finite-domain variables within the algorithm.

Recall that the prime implicate generator finds all of the projected prime implicates $\mathcal{C}_I$, from the set of all projected partial clauses $\mathcal{C} \Downarrow_{V_p}$, of a theory $\mathcal{C}_\Phi$ over projected variables V_p:

$$\mathcal{C}_I = \{C_I | C_I \in \mathcal{C} \Downarrow_{V_p}, \mathcal{C}_\Phi \models C_I\}_{prime}.$$

Since $p \models q$ iff $p \wedge \neg q$ is inconsistent, this problem is equivalent to finding all minimal

clauses C_I such that $\mathcal{C}_\Phi \wedge \neg(C_I \Downarrow_{V_p})$ is inconsistent:

$$\mathcal{C}_I = \{C_I | C_I \in \mathcal{C} \Downarrow_{V_p}, \mathcal{C}_\Phi \wedge \neg C_I \text{ is inconsistent}\}_{prime}.$$

Using this equation, the prime implicate generator is implemented as a propositional unsatisfiability algorithm, which is comprised of a candidate generator and a candidate tester.

The candidate generator generates the negation of prime implicate candidates $\neg C_C \Downarrow_{V_p}$, that is, each candidate $\neg C_C$ is a conjunctive clause over V_p. Hence, the generator generates minimal conflicts $\neg C_I$, not prime implicates C_I. The prime implicates are recovered by negating the discovered conflicts. For simplicity, we call the candidate conflicts A_C in place of $\neg C_C$, where A_C denotes a set of inconsistent assignments.

The candidate tester checks each candidate partial assignment A_C to see if it is a conflict. It accomplishes this by checking to see if $\mathcal{C}_\Phi \wedge A_C$ is unsatisfiable. If it is unsatisfiable, then $C_C \equiv \neg A_C$ is a prime implicate C_I. The input to the prime implicate generator is the projected variables V_p, the theory $\mathcal{C}_\Phi$, an initial set of implicants $\mathcal{A}_P$, and an initial set of conflicts $\mathcal{A}_F$. The generator returns a set of projected minimal conflicts $\mathcal{A}_I \Downarrow_{V_p}$, the negated set of all the projected prime implicates of $\mathcal{C}_\Phi$.

Determining unsatisfiability can be a computationally expensive operation; the problem is co-NP complete and the search space associated with the problem is worst-case exponential in $|V|$, the number of variables in $\mathcal{C}_\Phi$. In particular, the candidate generator may generate an exponential number of candidate assignments to the variables in V_p, and to prove inconsistency, the tester may need to search an exponential number of assignments to the remaining variables, $V_u = V \setminus V_p$.

The key to the algorithms detailed in this chapter is that they are able to reduce the number of tests and the size of the search space employed during generation. This is accomplished through the use of two sets of information that are determined in the testing process: candidates that are implicants of $\mathcal{C}_\Phi$ and candidates that are minimal conflicts of $\mathcal{C}_\Phi$. These two sets are used to prune the candidate space by bounding the set of viable candidates. Implicants provide a lower bound on what can be an implicate. Conflicts provide an upper bound above which candidates will be implicates, but will not be minimal. The candidate generation algorithm is designed to generate candidates within the two bounds, without creating candidates outside these bounds. Figure 4-1 shows these different bounds.

For tests on randomly generated problems, this approach has been shown to reduce the number of candidates generated, in comparison to the complete candidate space, by up to two orders of magnitude.

The first section of this chapter introduces a compilation example, used to demonstrate the prime implicate generation process. Section 4.1 introduces the top-level algorithm that coordinates the generator and tester. Section 4.2 introduces the candidate generator, while Section 4.3 introduces the candidate tester. Finally, Section 4.4 performs an empirical evaluation of the overall projected prime implicate generator on randomly generated problems.

4.1 Prime Implicate Generator

This section presents the high level flow of data between the candidate generator and candidate tester, thus providing the top-level algorithm of the prime implicate generator. The candidate generator generates candidate minimal conflicts $\mathcal{A}_C \Downarrow_{V_p}$, which are partial assignments to the projected variables. Each candidate is evaluated by the candidate tester for consistency with $\mathcal{C}_\Phi$. The tester returns whether $\mathcal{C}_\Phi \wedge A_C \Downarrow_{V_p}$

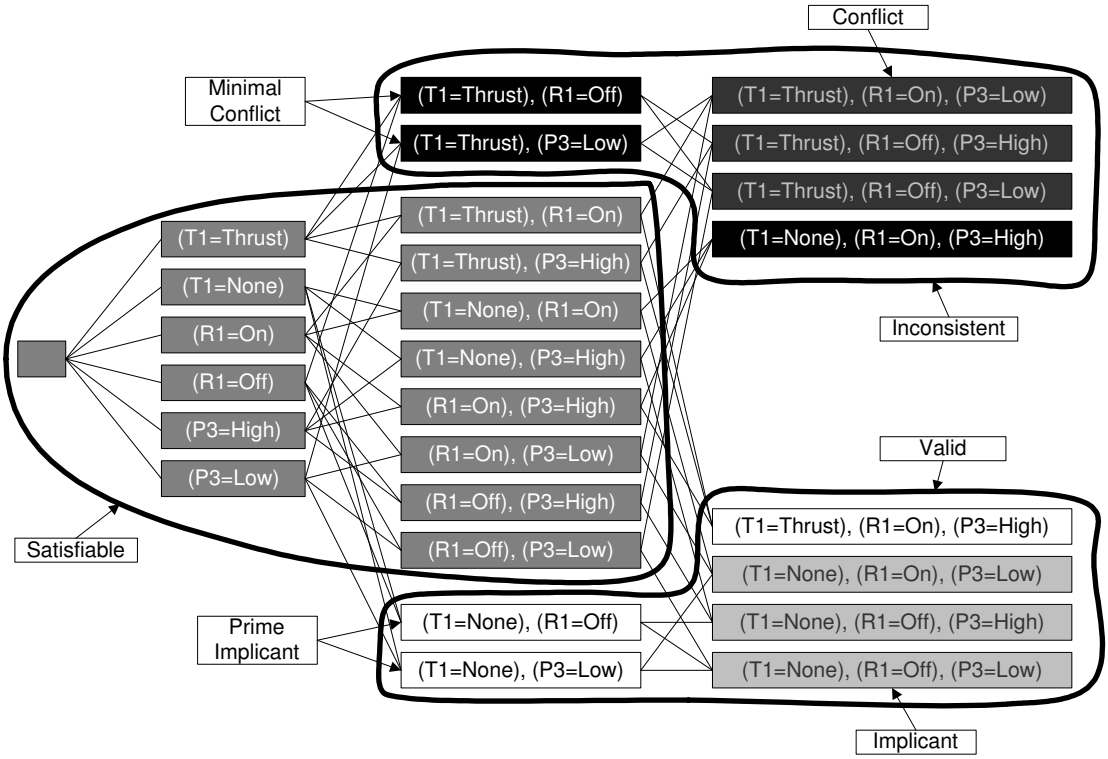

Figure 4-1: Implicant and conflicts of the model $\mathcal{C}_\Phi$ are used to filter candidates $A_C \Downarrow_{V_p}$. This lattice represents all possible partial assignments. All extensions to a valid partial assignment are also valid. All extensions to an unsatisfiable partial assignment are also unsatisfiable. Thus, sub-lattices are formed of each of these types. The root of a valid sub-lattice is a prime implicant. The root of an unsatisfiable sub-lattice is a minimal conflict. The remaining nodes above the sub-tree roots are all satisfiable, they have a path to both a conflict and an implicant. This lattice represents the relationship between the thruster, its pressure input, and it thrust output.

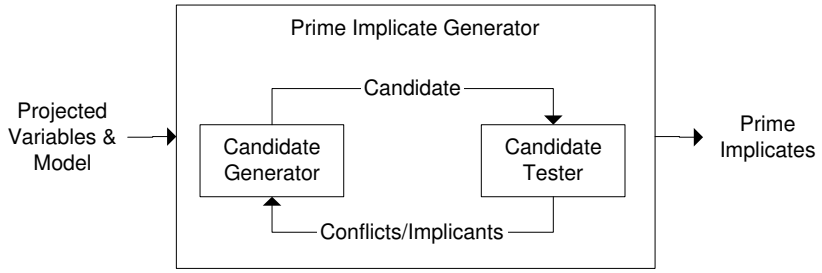

Figure 4-2: The architecture of the prime implicate generator. The candidate generator generates candidate prime implicates and the candidate tester determines if the candidate is a solution as well as identifying implicants and conflicts.

is *valid*, *unsatisfiable*, or *satisfiable*. In addition, when satisfiable is returned, it also returns a projected implicant of the model $\mathcal{C}_\Phi$, which it found when testing satisfiability. *Valid* indicates that a consistent assignment exists to the non-projected variables V_u for all projected variable extensions to the candidate. *Unsatisfiable* indicates that the candidate has no consistent extension. If a candidate is neither valid nor unsatisfiable, the tester returns *satisfiable*; it has already found an extension that is classified as unsatisfiable and an extension that is classified as valid. The projected implicant returned in this case is the projection of the valid-classified extension. The architecture of the prime implicate generator is shown in Figure 4-2.

As mentioned before, the generator can generate an exponential number of candidates to test. In an effort to reduce the number of candidates generated, the generator employs three pruning rules.

1) **Conflict Pruning**:

The first rule prunes all candidates that are extensions of known minimal conflicts. These candidates must not be minimal, so they can be ignored.

2) **Implicant Pruning**:

The second rule prunes all candidates that are extensions of known implicants. Since

an implicant is consistent for all extensions, and a conflict is inconsistent for all extensions, there cannot exist a minimal conflict that is an extension of a known implicant.
3) **Skip Satisfiable Candidates**:
The final rule prunes all candidates that can be extended to a superset of a known implicant. These extensions must be satisfiable, so their corresponding candidates can not be conflicts.

For the first two pruning rules, no extensions to pruned candidates need to be examined, since all extensions to conflicts are conflicts, and all extensions to implicants are implicants. Thus, they too can all be pruned. In the case of the third pruning rule, some extension to the pruned candidate may be a minimal conflict, so the extensions to the candidate still need to be examined. Hence, the first two pruning rules allow for the elimination of complete sub-trees of candidates. The third rule only allows the generator to ignore a particular candidate, saving the time of testing the candidate. These three rules are detailed in figure 4-1.

For example, assume $(V1 = Open) \wedge (F1 = Full)$ is a known implicant of $\mathcal{C}_\Phi$ and $(F1 = Empty)$ is a known minimal conflict of $\mathcal{C}_\Phi$. The first pruning rule eliminates all extension of $(F1 = Empty)$, such as $(F1 = Empty) \wedge (V1 = Open)$. Every extension to a minimal conflict must also be a conflict, and must not be minimal. The second pruning rule eliminates all extensions of $(V1 = Open) \wedge (F1 = Full)$, such as $(V1 = Open) \wedge (F1 = Full) \wedge (V2 = Open)$. Every extension to an implicant must also be an implicant. The third rule prunes all candidates that can be extended to include the implicants. For example, $(V1 = Open)$, $(F1 = Full)$, and $(V2 = Open) \wedge (V1 = Open)$ can all be pruned as they can all be extended to be extensions of the known implicant $(V1 = Open) \wedge (F1 = Full)$. These must all be implicants, so they cannot be conflicts.

In the satisfiable case, the tester must find a projected implicant. The tester, in order to determine that the candidate is neither valid nor unsatisfiable, must find

both a valid partial assignment and an unsatisfiable partial assignment. The valid assignment is an implicant; the unsatisfiable assignment is a conflict. The tester returns the projected implicant to the generator.

The algorithm linking the generator and tester is responsible for passing information between the two components and collecting the solutions. This algorithm simply gets a candidate from the generator and then tests the candidate in the tester. If the candidate is valid, it is inserted as an implicant into the generator. If the candidate is satisfiable, the implicant found is inserted into the generator. If the candidate is unsatisfiable, the candidate generator is informed of the minimal conflict and the minimal conflict is added to the set of solutions. This process repeats until either there are no more candidates or a sufficient number of minimal conflicts (prime implicates) have been found, as specified by the user.

4.1.1 Example

This example demonstrates the passing of information between the generator and the tester on a sample run. The run uses the model in Section 2.1.1, which describes a simple propulsion model with three valves, a fuel tank, and a thruster. In this example, the projected variables V_p are the state and observation variables $\{F1, V1, V2, V3, R1, P1, P3, T1\}$. All of these variables have two-element domains.

The generator generates the empty assignment $\{\}$, which is equivalent to $true$, as the first candidate conflict. The tester determines that the first candidate is satisfiable, and returns the implicant 4.1:

$$(F1 = Empty) \wedge (P1 = Low) \wedge (V1 = Closed) \wedge (V2 = Closed) \wedge$$
$$(V3 = Closed) \wedge (P3 = Low) \wedge (R1 = Off) \wedge (T1 = NoThrust) \qquad (4.1)$$

This implicant is fed back to the generator.

The generator constructs the candidate $\{(F1 = Empty)\}$ and prunes it by the third rule, because it can be extended to be a superset of the implicant 4.1. The generator then constructs the candidate $\{(F1 = Filled)\}$, and returns this as the second candidate. This candidate is also satisfiable, and the tester returns the implicant 4.2:

$$(F1 = Filled) \wedge (P1 = High) \wedge (V1 = Closed) \wedge (V2 = Closed) \wedge$$
$$(V3 = Closed) \wedge (P3 = Low) \wedge (R1 = Off) \wedge (T1 = NoThrust) \qquad (4.2)$$

This implicant is also added to the generator.

The next generated candidate is $\{(V1 = Open)\}$, followed by the candidates $\{(V2 = Open)\}$,$\{(V3 = Open)\}$, $\{(P3 = High)\}$, $\{(R1 = On)\}$, and $\{(T1 =$

Table 4.1: A list of the candidates and their corresponding implicants from the tester.

$\{(V1 = Open)\}$:	$(F1 = Empty) \wedge (P1 = Low) \wedge (V1 = Open) \wedge$
	$(V2 = Closed) \wedge (V3 = Closed) \wedge (P3 = Low) \wedge$
	$(R1 = Off) \wedge (T1 = NoThrust)$
$\{(V2 = Open)\}$:	$(F1 = Empty) \wedge (P1 = Low) \wedge (V1 = Closed) \wedge$
	$(V2 = Open) \wedge (V3 = Closed) \wedge (P3 = Low) \wedge$
	$(R1 = Off) \wedge (T1 = NoThrust)$
$\{(V3 = Open)\}$:	$(F1 = Empty) \wedge (P1 = Low) \wedge (V1 = Closed) \wedge$
	$(V2 = Closed) \wedge (V3 = Open) \wedge (P3 = Low) \wedge$
	$(R1 = Off) \wedge (T1 = NoThrust)$
$\{(P3 = High)\}$:	$(F1 = Filled) \wedge (P1 = High) \wedge (V1 = Open) \wedge$
	$(V2 = Open) \wedge (V3 = Closed) \wedge (P3 = High) \wedge$
	$(R1 = Off) \wedge (T1 = NoThrust)$
$\{(R1 = On)\}$:	$(F1 = Empty) \wedge (P1 = Low) \wedge (V1 = Closed) \wedge$
	$(V2 = Closed) \wedge (V3 = Closed) \wedge (P3 = Low) \wedge$
	$(R1 = On) \wedge (T1 = NoThrust)$
$\{(T1 = Thrust)\}$:	$(F1 = Filled) \wedge (P1 = High) \wedge (V1 = Open) \wedge$
	$(V2 = Open) \wedge (V3 = Closed) \wedge (P3 = High) \wedge$
	$(R1 = On) \wedge (T1 = Thrust)$

$Thrust)\}$. All of these are satisfiable and generate implicants. These implicants are shown in Table 4.1.

The generator then generates the candidate $\{(F1 = Empty) \wedge (P1 = High)\}$, which is unsatisfiable. Since it is unsatisfiable, it is a minimal conflict, and is added to $\mathcal{C}_{solutions}$. It is also added to the generator's list for pruning non-minimal conflicts.

This process continues until all potential candidates have been pruned, generating a total of 9 additional minimal conflicts, which are summarized in Table 4.2. These are the projected prime implicates of the state and observable variables. The prime implicants show again in a more readily understood form in Table 4.3. Notice that our model previously had 17 state constraints and now only has 10 state constraints. In addition, 3 variables were eliminated. As is shown in Figure 4-3, for the first two search groups, only 8 candidates are tested, 5 of which are minimal conflicts.

Table 4.2: The projected prime implicates generated for the propulsion system example.

1. $\neg(F1 = Filled) \vee \neg(P1 = Low)$
2. $\neg(F1 = Empty) \vee \neg(P1 = High)$
3. $\neg(P1 = Low) \vee \neg(P3 = High)$
4. $\neg(V1 = Closed) \vee \neg(P3 = High)$
5. $\neg(V2 = Closed) \vee \neg(V3 = Closed) \vee \neg(P3 = High)$
6. $\neg(P1 = High) \vee \neg(V1 = Open) \vee \neg(V2 = Open) \vee \neg(P3 = Low)$
7. $\neg(P1 = High) \vee \neg(V1 = Open) \vee \neg(V3 = Open) \vee \neg(P3 = Low)$
8. $\neg(P3 = Low) \vee \neg(T1 = Thrust)$
9. $\neg(R1 = Off) \vee \neg(T1 = Thrust)$
10. $\neg(P3 = High) \vee \neg(R1 = On) \vee \neg(T1 = NoThrust)$

Internally, the generator generates an additional 23 candidates, but uses Rule 3 to prune these candidates, hence avoiding 23 calls to the tester. The generator uses Rule 1 to avoid generating 2 additional candidates and to prune one additional candidate that it generated.

Table 4.3: The projected prime implicates generated for the propulsion system example rewritten to be human readable. Notice that our model previously had 17 state constraints and now only has 10 state constraints. 3 variables were also eliminated.

1. $(F1 = Filled) \Rightarrow (P1 = High)$
2. $(F1 = Empty) \Rightarrow (P1 = Low)$
3. $(P1 = Low) \Rightarrow (P3 = Low)$
4. $(V1 = Closed) \Rightarrow (P3 = Low)$
5. $(V2 = Closed) \wedge (V3 = Closed) \Rightarrow (P3 = Low)$
6. $(P1 = High) \wedge (V1 = Open) \wedge (V2 = Open) \Rightarrow (P3 = High)$
7. $(P1 = High) \wedge (V1 = Open) \wedge (V3 = Open) \Rightarrow (P3 = High)$
8. $(P3 = Low) \Rightarrow (T1 = NoThrust)$
9. $(R1 = Off) \Rightarrow (T1 = NoThrust)$
10. $(P3 = High) \wedge (R1 = On) \Rightarrow (T1 = Thrust)$

4.1.2 Algorithm

This section presents the algorithm just described for projected prime implicate generation. Lines 1 and 11 request new candidates from the generator. Line 2 checks to be sure that a new candidate was available. Line 3 tests the candidate, both classifying the candidate in f_{status} as well as return an implicant in $A_{implicant}$ when $f_{status} = Satisfiable$. Lines 4 and 5 handle the case when the candidate is valid. In this case, the candidate is an implicant, so the generator is given the new implicant. Lines 6, 7, and 8 handle the case when the candidate is unsatisfiable. In this case, the candidate is a minimal conflict, so it is added to the solutions and added to the generator. Lines 9 and 10 handle the case when the candidate is neither valid nor unsatisfiable. In this case, a implicant $A_{implicant}$ is returned, and is given to the generator. Line 12 returns the solutions generated.

primeImplicates(V_s, $\mathcal{C}_\Phi$, $\mathcal{A}_P$, $\mathcal{A}_F$)

0a. $\forall A_P \in \mathcal{A}_P \cdot$ generator.addImplicant(A_P)

0b. $\forall A_F \in \mathcal{A}_F \cdot$ generator.addConflict(A_F)

1. $A_C \leftarrow$ generator.getNextCandidate(V_s)

2. While $A_C \neq$ AllDone Do

3. $\quad \{f_{status}, A_{implicant}\} \leftarrow$ tester.testCandidate(A_C)

4. $\quad$ If $f_{status} = Valid$ Then

5. $\quad\quad$ generator.addImplicant(A_C)

6. $\quad$ Else If $f_{status} = Unsatisfiable$ Then

7. $\quad\quad$ generator.addConflict(A_C)

8. $\quad\quad \mathcal{C}_{solutions} \leftarrow \mathcal{C}_{solutions} \cup \{\neg A_C\}$

9. $\quad$ Else If $f_{status} = Satisfiable$ Then

10. $\quad\quad$ generator.addImplicant($A_{implicant}$)

11. $\quad A_C \leftarrow$ generator.getNextCandidate()

12. Return $\mathcal{C}_{solutions}$

4.2 Candidate Generator

This section explains the functionality of the candidate generator. The candidate generation function is called $getNextCandidate$. It takes as input a set of variables over which it should generate candidates and remembers, between function calls, the candidate that it previously generated. Each call to the function returns the candidate that successively follows the previously returned candidate, starting with the candidate $\{\}$, which represents a candidate that evaluates to $true$.

The candidate generator implements two novel concepts: an iterative deepening search algorithm and a set of prune rules that prune based on conflicts and implicants.

The generator generates candidate minimal conflicts over the space of all partial assignments to the variables it takes as its input. The algorithm is systematic, hence each candidate is generated at most once. The candidates are generated in increasing length, meaning number of literals, to ensure that the first conflict found is a minimal conflict of the theory $\mathcal{C}_\Phi$. If $(F1 = Filled) \wedge (P1 = Low)$ is a minimal conflict, then $(F1 = Filled) \wedge (P1 = Low) \wedge (V1 = Open)$ must also be a conflict, but is not minimal, thus the latter need never be generated. The algorithm used for the generation of candidates as well as the search tree associated with the generation process is elaborated in Section 4.2.1. As mentioned in Section 4.1, there are three pruning rules used in the candidate generator. The minimal conflicts found are used to prune candidates that are non-minimal conflicts (Rule 1); implicants are used to prune all candidates that must be implicants, and therefore can not be conflicts (Rule 2); and implicants are used to prune all candidates that must be satisfiable with the theory $\mathcal{C}_\Phi$ and therefore can not be conflicts (Rule 3). These three pruning rules will be further developed in Section 4.2.2. The algorithm is presented in Section 4.2.3.

For the example in this section, there are three different variable sets: 1) $\{F1, P1\}$, 2) $\{P1, V1, V2, V3, P3\}$, and 3) $\{P3, R1, T1\}$. These come from a cut-set based

decomposition of the model, not presented in this thesis. The purpose of the decomposition is to reduce the exponent of the space being searched over. From each set, a set of minimal conflicts and implicants is generated. These minimal conflicts and implicants carry over between the candidate generation processes for each set. Thus, candidates are generated locally from each set, and pruning is applied globally across all sets. Locally, the candidates returned to the tester can be selected from among any one set of variables. $\{(F1 = Full) \wedge (P1 = High)\}$ is a candidate, as is $\{(P3 = Low)\}$; however, $\{(F1 = Full) \wedge (P3 = Low)\}$ is not a candidate since it does not come from a single set of variables. Note that $\{(P3 = Low)\}$ is a candidate from two different sets; it is only generated once because any duplicate candidate is discarded by one of the pruning rules.

4.2.1 Search Tree and Iterative Deepening Search

The search tree is organized in such a way that all nodes at a particular depth d correspond to all partial candidates of length d. A sample tree is shown in Figure 4-3. The tree is constructed by first defining an ordering on the variables and their assignments. The algorithm assigns variables and values according to this ordering. For example, in Figure 4-3, the variable $F1$ is ordered before the variable $P1$; the value $Filled$ of the variable $F1$ is ordered before the value $Empty$. Thus, the assignment $(F1 = Filled)$ is ordered before the assignment $(F1 = Empty)$, which in turn is ordered before $(P1 = High)$. The important property of the tree is that all the children of a variable contain variables that have an ordering greater than that variable. This ordering ensures that the partial candidates are never duplicated. For example, for the initial branch of $R1$ and the value On in the figure, the only variable after $R1$ is $P3$, so this is the only variable that can be selected. $P3$ in turn has two values $high$ and low. Thus, $(R1 = On) \wedge (P3 = High)$ is a valid candidate, while the

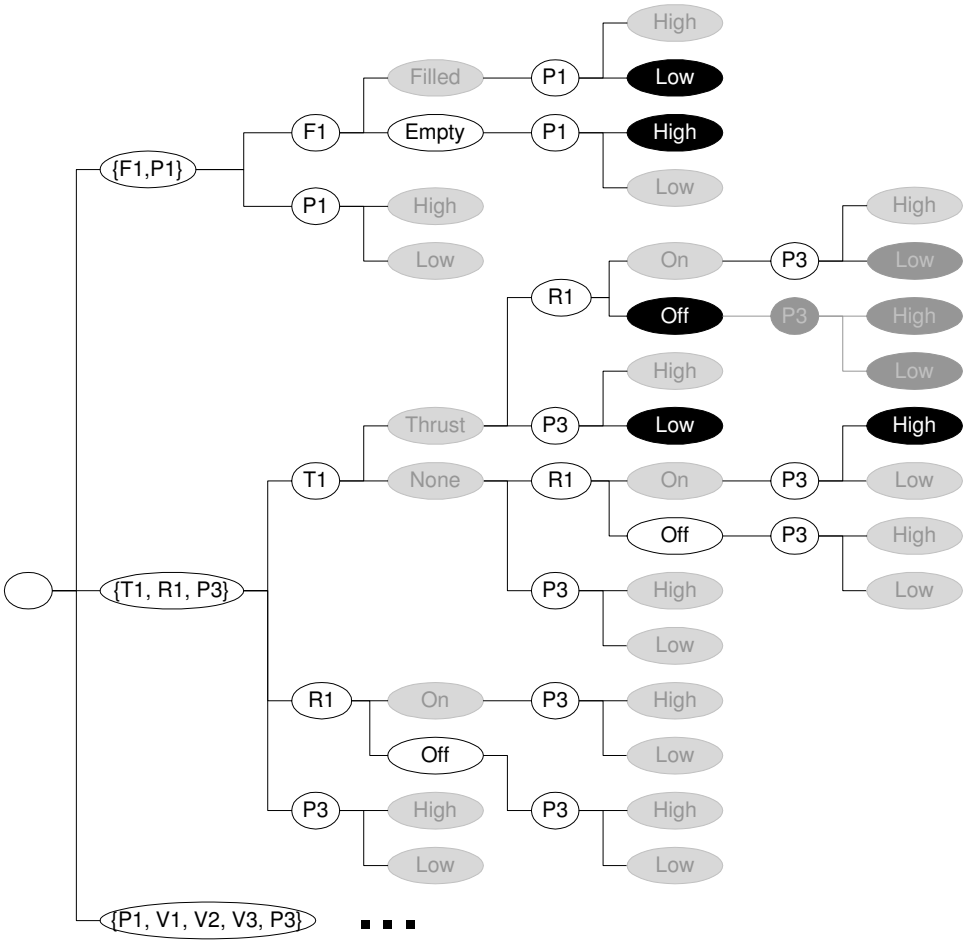

Figure 4-3: The search tree associated with a complete candidate generator run for the example in Section 4.2. The nodes with the initial sets represent the current group being examined. A path from the root to a node collectively represents the current group and candidate. For example, the top-most black node *Low* has the path $(\{F1, P1\}, F1, Filled, P1, Low)$. This corresponds to a current group of $\{F1, P1\}$ and a current candidate of $\{(F1 = Filled), (P1 = Low)\}$. Light grey nodes were pruned by Rule 3. Dark grey nodes were pruned by Rule 1 and Rule 2. The white nodes were tested and were satisfiable. The black nodes were tested and were unsatisfiable. As the figure illustrates, only the root and three other nodes were tested and were not solutions. The other five nodes tested were all solutions. A total of 26 nodes were pruned, of which 23 were generated.

permutation $(P3 = High) \wedge (R1 = On)$ is not valid, as $R1$ comes before $P3$.

The candidate generation algorithm selects candidates from this tree using an iterative deepening algorithm[11] to generate candidates with increasing length. Iterative deepening is a search method that uses a depth-first search algorithm[3], but limits the depth of the search at each round r to a maximum depth of r. Using iterative deepening, the candidate generation algorithm will generate all candidates of length r in round r before generating any candidate of depth $r+1$ in round $r+1$. If only the leaves of length r are considered as candidates in round r, the algorithm simulates a breadth-first search[3], returning all candidates of length r before allowing for any candidates of length greater than r. However, iterative deepening has a much smaller memory bound than breadth-first search, namely the same as depth-first search. A breadth-first search uses $\mathcal{O}(b^d)$ space and time, where b is the average domain size, and d is the number of search variables. Iterative deepening increases the runtime by a constant factor, so it is still $\mathcal{O}(b^d)$; however, it only uses $\mathcal{O}(d)$ space. The constant factor is almost always less than 1.5 and can be much closer to one, as the average domain size per variable grows above two. Thus, iterative deepening does not significantly increase the run-time of the generator, while the dramatic savings in space can allow the generator to run on substantially larger problems. In other words, the limit becomes the amount of time one wishes to spend, rather than the amount of memory available.

For the candidate generation algorithm, a node represents an assignment, and a search path corresponds to a list of assignments. Thus, searching deeper involves adding an assignment to the list, and going up a level in the search tree corresponds to removing an assignment. The iterative deepening portion of *getNextCandidate* is on lines 3 to 12, 14 to 17, and 20 to 21 (Section 4.2.3).

Intuitively, the iterative deepening portion of the algorithm is trying to walk vertically down the search tree shown in Figure 4-3, and when it reaches the bottom,

it starts again at the top at one level deeper. Lines 3 to 12 handle the case when walking down the tree at the current depth just involves switching the last assignment to the next one in the ordering. For example, the candidate $\{(T1 = Thrust), (R1 = On)\}$ is followed by $\{(T1 = Thrust), (R1 = Off)\}$, which is in turn followed by the candidate $\{(T1 = Thrust), (P3 = High)\}$. In the search tree, to get from $\{(T1 = Thrust), (P3 = Low)\}$ to $\{(T1 = None), (R1 = On)\}$, one needs to first walk down one node at the previous level and then can select a variable and value at the current depth. In this case, at the previous level, the node $\{(T1 = Thrust)\}$ is followed by $\{(T1 = None)\}$. Line 4 walks down the values assigned to the same variable at the same depth. For example, the candidate $\{(R1 = On)\}$ will be followed by the candidate $\{(R1 = Off)\}$. Line 6 selects the next variable according to the ordering. For example, the candidate $\{(T1 = None)\}$ is followed by $\{(R1 = On)\}$. If line 8 is reached, the algorithm must first walk down the tree at one level lower to select the next candidate. Thus, it removes the last assignment in the list and walks down the sub-tree formed by the shorter list. If line 10 is reached, then the algorithm has walked down the entire tree at the current depth, so there are no more candidates of length d. Thus, it increments the length of the candidates it is searching for to $d+1$. In either case, reaching line 8 or 10, the algorithm recursively attempts to find a successor candidate.

Once the algorithm has found the successor partial candidate, potentially through the removal of any number of assignments from the list, lines 13 through 17 verify that the partial candidate has enough assignments corresponding to the desired depth; if not, the same steps as line 8 through 11 are performed. For example, if the candidate is $\{(R1 = Off), (P3 = Low)\}$ and the depth is 2, then line 8 removes $(P3 = Low)$, because it has no successor value or variable. Upon recursing, $\{(R1 = Off)\}$ is followed by $\{(P3 = High)\}$. Line 13 compares the number of variables required, in this case 1, and the number of variables remaining, in this case 0. It concludes that

$\{(P3 = High)\}$ does not have enough variables remaining in its sub-tree to generate a candidate of length 2. Line 14 deletes the last assignment in the partial candidate, so line 16 increases the depth to 3.

If the candidate makes it past lines 13 through 17, then the candidate once again needs to be extended to the desired depth. This last step is performed by lines 20 to 23. For example, when finding the candidate after $\{(T1 = Thrust), (P3 = Low)\}$, lines 3 through 12 stripped off $(P3 = Low)$ and then found that $\{(T1 = None)\}$ was the successor of $\{(T1 = Thrust)\}$. $\{(T1 = None)\}$ is then passed to lines 20 to 23, which, with the goal of walking down the tree, needs to assign the first value of the first available variable, $(R1 = On)$. Thus, the successor candidate of $\{(T1 = Thrust), (P3 = Low)\}$ has successfully been found, $\{(T1 = None), (R1 = On)\}$.

For the correctness of this algorithm, it is essential to store the previously generated candidate. This previous candidate is stored in the variable A_C. This is a convenience, as the previous candidate is transformed into the current candidate, so when the new candidate is returned, A_C once again contains the current candidate. For the next function call, this will be the previous candidate. The pseudo-code given in Section 4.2.3 assumes that the first candidate $\{\}$ is returned by some mechanism prior to calling the function *getNextCandidate*, thus A_C will always have a defined value, initially $\{\}$. This value can trivially be returned by initially having the depth be 0 and simultaneously increasing the depth by 1 and returning $\{\}$.

Putting the whole example together, consider the variable set $\{T1, R1, P3\}$ from Figure 4-3. Assume that the candidate A_C is of length 2 and is given as $\{(T1 = Thrust), (R1 = On)\}$ and that the desired candidate length is also 2. The candidate is succeeded by $\{(T1 = Thrust), (R1 = Off)\}$, as Off is the next value of $R1$. This new candidate is succeeded by $\{(T1 = Thrust), (P3 = High)\}$, as $R1$ no longer had any values, and the variable $P3$ is after $R1$. This is then succeeded by $\{(T1 = Thrust), (P3 = Low)\}$. Since $(P3 = Low)$ does not have a successor value or variable,

it is removed, leaving $\{(T1 = Thrust)\}$. This is then succeeded by $\{(T1 = None)\}$. Lines 20 to 23 then add $(R1 = On)$, making the next candidate $\{(T1 = None), (R1 = On)\}$. This will progress until the candidate $\{(R1 = Off), (P3 = Low)\}$ is reached. $(P3 = Low)$ will be removed, as before, and $\{(R1 = Off)\}$ is succeeded by $\{(P3 = High)\}$. However, there are no longer any remaining variables, so the desired depth will be increased to 3 and $(P3 = High)$ will be removed from the candidate, leaving the candidate empty. This corresponds to walking vertically down the $\{T1, R1, P3\}$ sub-tree shown in Figure 4-3, starting in the second column of values. Since the candidate is empty, the algorithm will fill in all the values, thus the next candidate is $\{(T1 = Thrust), (R1 = On), (P3 = High)\}$, which corresponds to restarting the walking function at the top of the third column in Figure 4-3.

Without pruning, everything behaves as described. The purpose of pruning is to avoid generating candidates that need not be generated and avoid testing candidates which are definitely not conflicts. Section 4.2.2 describes how the rules for pruning are incorporated into this algorithm, so as to prevent the generation of sub-trees that do not contain any minimal conflicts, and to eliminate all candidates that cannot be conflicts.

4.2.2 Pruning Rules

As mentioned in Section 4.1, there are three types of pruning rules performed by the candidate generator. These rules use the conflicts and implicants found thus far in the search for minimal conflicts. For the problem shown in Figure 4-3, these three types of pruning reduce the number of tested candidates down from 35 to 9. They also reduce the number of nodes generated from 35 to 32. While the latter improvement seems less significant, note that the implicants and conflicts in the example are rather long – the shortest has a length of two, while there are only three variables in the

largest grouping shown. Hence, fewer sub-trees can be pruned. This pruning rule is quite effective in the empirical data, when run on random problems. The empirical data can be found in Section 4.4. This rule can reduce the overall time to find the solution by a factor of five. The novel contribution of this section is a set of pruning rules that utilize implicants. For minimal conflict generation, implicants serve the purpose of specifying sets of assignments that can never be conflicts.

The first rule, conflict pruning, eliminates all sub-trees starting at a conflict. This rule is motivated by the fact that every superset of a conflict must be a non-minimal conflict. Thus, the first rule can prune sub-trees of candidates representing conflicts as each extension to the candidate cannot be a minimal conflict. For example, assume that $(T1 = Thrust) \wedge (R1 = Off)$ is a minimal conflict, then $(T1 = Thrust) \wedge (R1 = Off) \wedge (P3 = High)$ cannot be a minimal conflict, the assignment $(P3 = High)$ is extraneous.

The second rule, implicant pruning, eliminates all sub-trees starting at an implicant. As with conflict pruning, once a candidate is a superset of an implicant, every extension is also going to be an implicant. Since any candidate that is an implicant can not be a conflict, it is unnecessary to test such a candidate when looking for minimal conflicts. The second rule also prunes sub-trees for which the candidate at the root of the sub-tree is a superset of an implicant that has been discovered. For example, assume that $(F1 = Filled)$ is an implicant. Then $(F1 = Filled) \wedge (P1 = High)$ must also be an implicant.

The third rule, skip satisfiable candidates, eliminates candidates that must be consistent and therefore can not be conflicts given the implicants that have been found. Any candidate that can be extended to be a superset of an implicant, by assigning at most one value per variable, must have an extension that is an implicant. Since implicants and conflicts are disjoint, if the candidate has an extension that is an implicant, then the candidate can not be a conflict. Thus, the candidate need not

be tested. However, unlike the previous two cases, this third type of pruning only prevents one from returning a specific candidate. It is possible that an extension to the candidate is a conflict; it is only known that the candidate itself is not a conflict. For example, assume that $(F1 = Filled)$ is an implicant. The third rule will prune the candidate $(P1 = High)$ because it can be consistently extended to $(F1 = Filled) \wedge (P1 = High)$. This extension is an implicant, as its a superset of $(F1 = Filled)$, so $(P1 = High)$ must not be a conflict.

Since the first two pruning rules are both superset tests, and have the same effect of pruning the tested sub-tree, the two tests are folded into the same test routine, *pruneSupersets*. The *pruneSupersets* routine is described in Section 4.2.4. Pruning is checked after each successor is generated, every time the partial candidate changes. Rules 1 and 2 are checked on lines 18 and 22.

To understand why pruning occurs on lines 18 and 22, consider Figure 4-3. Assume that the previous candidate was $\{(T1 = Thrust), (R1 = On), (P3 = Low)\}$. As specified by the search process in Section 4.2.1, the algorithm determines that $(P3 = Low)$ has no successor and removes it from the candidate list. The algorithm then tries to find the successor of $\{(T1 = Thrust), (R1 = On)\}$. In this case the successor is $\{(T1 = Thrust), (R1 = Off)\}$. The black node for this successor in the figure indicates that it is a minimal conflict. Thus, the algorithm prunes the sub-tree of this candidate. In this example, having found a successor of $\{(T1 = Thrust), (R1 = On)\}$, the algorithm is on line 13. This line, as mentioned before, makes sure that the partial candidate $\{(T1 = Thrust), (R1 = Off)\}$ has a deep enough sub-tree (enough variables left) to generate a suitable candidate. Since this sub-tree does have enough elements, the algorithm makes it to line 18. At this point the algorithm verifies that $\{(T1 = Thrust), (R1 = Off)\}$ is a superset of a known minimal conflict, in this case equal to the minimal conflict. Step 19 causes the algorithm to immediately find the successor of $\{(T1 = Thrust), (R1 = Off)\}$, in this case $\{(T1 = None), (R1 = On)\}$.

Hence, the whole sub-tree of $\{(T1 = Thrust), (R1 = Off)\}$ was pruned in a single step, due to the pruning check on line 18. Similarly, on line 22, the algorithm verifies that the candidate has not become a superset of a conflict or implicant, as it adds assignments to the candidate. Thus, the candidate and its sub-tree is pruned as soon as possible.

The third type of pruning is checked on line 24 of the *getNextCandidate* routine by the *pruneByImplicants* routine. The *pruneByImplicants* routine is described in Section 4.2.5. At this point in the *getNextCandidate* routine, the candidate is of the correct length and is otherwise ready to be returned as the next candidate. Thus, checking the candidate here ensures that only full candidates are pruned when they are not conflicts. This has the effect of skipping over the candidate, selecting instead the successor of the candidate, saving an unnecessary test in the candidate tester.

4.2.3 Algorithm

Variables

$d_{current}$	The current iterative deepening depth
V_s	The search variables. Candidates are selected from these variables.
A_C	The working candidate. This variable starts out as the previous candidate in *getNextCandidate*.

Initially, A_C is set to the first candidate $\{\}$, which is a candidate of length zero. $d_{current}$ is set to one. This procedure handles all candidates past the first candidate, $\{\}$. Note that A_C is a clause, represented as a list of assignments. The list is, by virtue of the way this algorithm works, sorted based on the assignment ordering. Thus, the first candidate is the assignment with the earliest ordering and the last candidate is the assignment in A_C with the latest ordering. Note that an assignment c is a pair of terms, a variable v and a value x. $c.v$ refers to the variable of the assignment c.

Note that $d_{current} - size(A_C)$ is the number of assignments needed at the current depth $d_{current}$, given how many have already been assigned $size(A_C)$. $size(V_s) - orderingOf(lastAssignmentOf(A_C).v)$ is the number of free variables remaining. A variable is free if the variable's ordering is greater than the variable of last assignment of A_C. The variables are ordered starting at 1, where 0 denotes that there are no assignments.

getNextCandidate(V_s)

1. If $d_{current} \leq size(V_s)$ Then
2. If $size(A_C) > 0$ Then
3. $c_p \leftarrow$ last assignment of A_C
4. $c'_p \leftarrow (c_p.v, x')$ where $x' \in D(c_p.v)$ and $x' =$ next value after $c_p.x\}$
5. If (no next value) Then
6. $c'_p \leftarrow (v', x')$ where $v' \in V_s$ and $v' =$ next variable after $c_p.v$, $x' \in D(v'), x' =$ first value of $D(v')\}$
7. If (no next variable) Then
8. Delete last assignment of A_C
9. If $(size(A_C) = 0)$ Then
10. $d_{current} \leftarrow d_{current} + 1$
11. Return getNextCandidate(V_s)
12. Replace last assignment of A_C with c'_p
13. If $d_{current} - size(A_C) > size(V_s) - orderingOf(lastAssignmentOf(A_C).v)$ Then
14. Delete last assignment of A_C
15. If $(size(A_C) = 0)$ Then
16. $d_{current} \leftarrow d_{current} + 1$
17. Return getNextCandidate(V_s)

18. If $pruneSuperset(A_C)$ Then
19. Return getNextCandidate(V_s)
20. While $size(A_C) < d_{current}$ Do
21. $c_p \leftarrow$ last assignment of A_C
Add (v, x) where $v \in V_s, v =$ next variable after $c_p.v$,
$x \in D(v)$, and $x =$ first value of $D(v)\}$ to A_C
22. If $pruneSuperset(A_C)$ Then
23. Return getNextCandidate(V_s)
24. If $\neg pruneByImplicants(A_C)$ Then
25. Return A_C
26. Return getNextCandidate(V_s)
27. Return AllDone

4.2.4 Pruning Supersets of Conflicts and Implicants

This section presents the routine $pruneSuperset(A_C)$ for testing whether a candidate A_C is the superset of the implicants and conflicts that have been found, Rule 1 and Rule 2. It also presents the routine $addConflict$ for adding conflicts to the data structure P used in the superset test. Conflicts are only used in this test, so they are presented here. Implicants are added in the same way as conflicts for the purpose of this test, but are also used in the $pruneByImplicants$ test, so are presented in the next section.

The $pruneSuperset(A_C)$ routine determines if the current partial candidate should be pruned, because it has become a superset of an implicant or conflict. The implicants and conflicts are stored as lists. If the partial candidate is a superset of an element of either of these lists, then the candidate must be eliminated.

The data structure P used for this test is a list of elements, called Trigger objects, which store two values, the number of assignments that have matched the candidate and the number of assignments in this conflict or implicant. These are both integers. The data structure also keeps a list of all assignments and associates with each assignment a list of these Triggers. A trigger is in the list of an assignment if it's corresponding conflict or implicant contained that assignment.

With this data structure P, there is a three step process to determine if the candidate is a superset of one of the implicants or conflicts. The algorithm first needs to record that each Trigger has not yet been accessed, corresponding to lines 1 and 2. It then needs to retrieve the list of Triggers for each of its assignments. For each of the elements in the list, the algorithm needs to add one to the Trigger's count. This corresponds to lines 3 to 6. Finally, the algorithm checks to see if any Trigger has been accessed as many times as it has elements. If so, then the candidate must be a superset of the Trigger's corresponding implicant or conflict. This is checked on lines 7 to 9. If none of them have enough assignments in common, then the routine indicates that the candidate should not be pruned on line 10.

For i implicants and conflicts, with an average length of d, and a candidate of length l, the algorithm requires $O(i)$ time perform the first phase. For the second phase, if $l < d$, the algorithm requires, in worst case, $O(l \cdot i)$ time, otherwise, in worst case, $O(d \cdot i)$ time. The final phase requires $O(i)$ time.

For example, if the implicant $(F1 = Empty) \wedge (P1 = Low)$ is represented in the data structure P, then it has a corresponding Trigger in the data structure that requires two accesses to activate and is in the list returned by $getPruneList()$ for the assignments $(F1 = Empty)$ and $(P1 = Low)$. If the candidate ever has both of these elements in it, the Trigger is accessed twice, and prune returns true, indicating that the item should be pruned, as desired. Otherwise, the Trigger must have been accessed less than twice, and the routine will not return true on account of this

Trigger, as desired.

The $addConflict$ routine adds a new conflict S as a Trigger to the data structure P. As mentioned above, implicants are added in the same way. Adding a conflict involves creating a new Trigger p with a trigger count of $size(S)$. This corresponds to lines 1 and 2 of $addConflict$, respectively. This new Trigger p is then added to P, the list of all Triggers, on line 3. Lines 4-6 add the new Trigger p to the trigger list of every assignment s in S.

Variables

P: The set of all prune entries

P_s: The set of prune entries that the assignment s is a part of

addConflict(S)

1. $p \leftarrow$ New Trigger
2. $p.total \leftarrow size(S)$
3. $P.push(p)$
4. For $\forall s \in S$ Do
5. $\quad P_s \leftarrow s.getPruneList()$
6. $\quad P_s.push(p)$

pruneSuperset(A_C)

1. For $\forall p \in P$ Do
2. $p.trigger \leftarrow 0$
3. For $\forall c \in A_C$ Do
4. $P_c \leftarrow c.getPruneList()$
5. For $\forall p_c \in P_c$ Do
6. $p_c.trigger \leftarrow p_c.trigger + 1$
7. For $\forall p \in P$ Do
8. If $p.trigger = p.total$ Then
9. Return $true$
10. Return $false$

4.2.5 Pruning Satisfiable Candidates Using Implicants

This section presents the routine $pruneByImplicants(A_C)$ for testing whether a candidate is inconsistent with all of the implicants that have been found, according to Rule 3. It also presents the routine *addImplicant* for adding implicants to the data structure P used in the superset test and in the inconsistency pruning. When the candidate is inconsistent with every implicant that has been found, an extension does not exist that will make the candidate a superset of the known implicants. Since both the candidate and the implicants are a conjunction of assignments, a candidate and implicant is inconsistent whenever both have an assignment to the same variable where the value of the two assignments differ.

To test for this condition, the data structure consists of a set of lists, one per assignment, where each list maps the assignment to a set of elements, called Implicant objects, with which the assignment is inconsistent. Each Implicant corresponds to an implicant that has been found. The list of Implicants associated with each assignment

is call the Inconsistent list. As stated in the pruning section, Section 4.2.4, implicants are also used in pruning, thus an Implicant is a subclass of a Trigger.

Using this data structure, testing to see if a candidate is inconsistent with every implicant involves three steps. First, the algorithm initializes the set of Implicants so that they all indicate that they are still consistent with the candidate. This corresponds to lines 1 and 2. Second, for each assignment in the candidate, the algorithm gathers the list of all implicants that the assignment is inconsistent with and marks them as inconsistent. This corresponds to lines 3 through 6. Finally, the algorithm checks to see if any implicant is still consistent. If so, the candidate must be consistent with this implicant and must therefore be consistent with the theory $\mathcal{C}_\Phi$; otherwise, the algorithm returns false. This part of the algorithm corresponds to lines 7 through 10.

For example, consider the implicant $(F1 = Empty) \wedge (P1 = Low)$[1]. The implicant indicates that there can never be a minimal conflict that involves having both $F1$ be $Empty$ and $P1$ be Low. Thus, to be a conflict the candidate must contradict one of the two assignments. Any candidate that does not contradict one of the implicant's assignments can be consistently extended by this candidate and the resulting candidate is an implicant. For example, the candidate $(P3 = Low)$ will be pruned, as it can be extended to $(F1 = Empty) \wedge (P1 = Low) \wedge (P3 = Low)$, which is an implicant. The candidate $(P3 = Low) \wedge (F1 = Filled)$ is a valid candidate, as $(F1 = Filled)$ contradicts $(F1 = Empty)$.

Adding a new implicant S for use in the implicant and superset checks involves creating a new Implicant l with a trigger count of $size(S)$. This corresponds to lines 1-2. This Implicant l is then added to the Prune list P and Implicant list L on lines 3-4. Lines 6-7 insert l into the appropriate assignment prune list P_s for each

[1]Note that $(F1 = Empty) \wedge (P1 = Low)$ is not an implicant for the above thruster example, as $(P1 = Low)$ is part of a minimal conflict that does not involve $F1$

assignment for the superset test. Lines 8-10 insert l into the inconsistent lists L_s of every assignment to the current variable $s.v$ of s that contradicts the assignment's value $s.x$.

Variables

P: The set of all prune entries

P_s: The set of prune entries that the assignment s is a part of

L: The set of all implicant entries

L_s: The set of prune entries that s satisfies

addImplicants(S)

1. $l \leftarrow$ New Implicant
2. $l.total \leftarrow size(S)$
3. $P.push(l)$
4. $L.push(l)$
5. For $\forall s \in S$ Do
6. $\quad P_s \leftarrow s.getPruneList()$
7. $\quad P_s.push(l)$
8. $\quad$ For $\forall(s_l \in \{(s.v, x)|x \in (D(s.v) \setminus s.x)\})$ Do
9. $\quad\quad L_s \leftarrow s_l.getSatisfyList()$
10. $\quad\quad L_s.push(l)$

pruneByImplicants(A_C)

1. For $\forall l \in L$ Do
2. $\quad l.consistent \leftarrow True$
3. For $\forall c \in A_C$ Do
4. $\quad L_c \leftarrow c.getSatisfyList()$
5. $\quad$ For $\forall l_c \in L_c$ Do
6. $\quad\quad l_c.consistent \leftarrow False$
7. For $\forall l \in L$ Do
8. $\quad$ If $l.consistent$ Then
9. $\quad\quad$ Return $true$
10. Return $false$

4.3 Candidate Tester

This section explains the functionality of the candidate tester. The tester distinguishes between valid, satisfiable, and unsatisfiable candidates, with respect to a theory and a set of projected variables. The candidate C_C is valid whenever all the extensions to each of the projected variables are consistent. In other words, the candidate is valid whenever all unassigned projected variables can take on any of their possible domain values. A candidate is satisfiable when it is neither valid nor unsatisfiable. A candidate is unsatisfiable whenever there exists no extension to any of the variables that is consistent with the theory. The candidate tester takes as input a candidate and returns the classification of the candidate as well as an implicant when the candidate is classified as satisfiable.

This section provides a novel approach for efficiently testing for validity. This approach is able to determine valid candidates without assigning a value to every variable. Validity testing allows the tester to identify implicants, namely a valid extension to the candidate.

In general, testing for validity can be computationally expensive, however the test provides useful information. This is especially true for the typical under-specified model that has a much larger space of satisfiable assignments than unsatisfiable assignments. In such situations, the generator will explore the set of valid candidates and all of their extensions, testing every one of these candidates. For each of these candidates, the tester will need to repeat much of its work to determine once again that the candidate is still satisfiable. This search will never turn up a conflict and can add significant amounts of time to the search, much more than testing for validity. Therefore, testing for valid is highly advantageous.

A straight-forward method for testing the validity of a candidate is to verify that all full extensions to the candidate are consistent. If so, the candidate is valid.

However, this approach requires enumerating all possible extensions. This has an exponential complexity that is a function of the number of unassigned variables. The approach that the tester of this thesis uses is based on the realization that a candidate is known to be valid as soon as all the clauses of the model have been satisfied. This holds because any combination of the remaining variables must be consistent. Thus, for theories with few clauses, the tester need only assign a few variables, while complex theories will require searching over more variables.

This thesis uses a clause-directed approach [14], which directs its search towards satisfying clauses. In particular, a clause-directed algorithm selects assignments by choosing them from clauses that have not yet been satisfied. A clause is satisfied when an assignment has been selected that makes one of its literals true. For example, the clause $\neg(F1 = Full) \vee (P1 = High)$ is satisfied by making one of the assignments $(F1 = Empty)$, $\neg(F1 = Full)$ or $(P1 = High)$. If none of these assignments has already been selected, then to satisfy the clause the tester will select either $\neg(F1 = Full)$ or $(P1 = High)$, since these two choices each provide a minimal constraint on the domains of the variables while still satisfying the clauses. In this particular example, the choice of $(F1 = Empty)$ is equivalent to $\neg(F1 = Full)$; however, if the variable had three values in its domain, $\neg(F1 = Full)$ would constrain the domain from three to two values, while $(F1 = Empty)$ would constrain the domain from three to one value.

To be systematic, the tester must make sure that the set of extensions are only examined once. To accomplish this the tester first chooses an assignment from a clause and searches the extensions including the assignment. The tester then chooses the negation of the assignment that it chose first, and searches the extensions including that negation. For example, the tester will examine the extension of $\neg(F1 = Full)$ as its first branch and $(F1 = Full)$ as its second branch. The effect is to split the search space into two disjoint sets, ensuring that the search is systematic. The tester

performs unit propagation after choosing each assignment, in order to quickly deduce assignments to unassigned variables that are implied by the chosen assignment. This is similar to DPLL [5] [4]. This branching process continues until all clauses have been satisfied or until a clause is found unsatisfiable, meaning that all of the assignments in the clause are inconsistent with respect to the set of extensions that have been made thus far.

Since we allow for non-binary variable domains, the assignment operation, for a positive assignment, constrains the domain of the variable to the value of the assignment, while a negative assignment removes the value of the assignment from the domain of the variable. In the former case, if the value was no longer in the domain of the variable, then the extensions are inconsistent. In the latter case, the extension has no effect.

The branching process can be thought of as constraining the flexibility of a variable. The algorithm maintains a count of how many values remain in the domain of the variable. If all the values are removed from the variable's domain, then the branch is inconsistent. If a value is removed from the domain of a projected variable, then the value removed, along with the extended candidate, must be classified as unsatisfiable, which is to say that it is a conflict. Hence, the extension just prior to removing the value from the domain of the projected variable must be satisfiable or unsatisfiable; it cannot be valid.

For example, consider the task of generating prime implicates for the fuel tank of the propulsion example and the tank pressure. This corresponds to clauses 1 and 2 from the model in Section 2.1.1. If the candidate conflict being considered is that the fuel tank is $Filled$, $\{(F1 = Filled)\}$, then Constraint 1, $(F1 = Filled) \Rightarrow (P1 = High)$, constrains the pressure to $High$ by unit propagation. If the pressure is also a projected variable, then $(F1 = Filled) \wedge (P1 = Low)$ is inconsistent with the theory, and hence a conflict. If $\{(F1 = Filled)\}$ is satisfiable, as opposed to unsatisfiable,

then there must exist some extension that is valid, and hence an implicant. For example, if $(F1 = Filled) \wedge (P1 = High)$ is valid, then this extension can be returned as an implicant.

The algorithm works by first propagating the initial assignments on lines 1-5 of *testCandidate*. If the solution is known at this point, there is nothing more to do. If not, then the algorithm assigns the first assignment of the first clause as either true or false on lines 2-3 and 5 of *evaluate*. For example, if the current candidate is $\{(F1 = Filled) \wedge (P1 = High)\}$, and the first literal of the first clause is $\{\neg(V1 = Open)\}$, then the algorithm checks the two sub-trees, corresponding to the extensions $\{(F1 = Filled) \wedge (P1 = High) \wedge \neg(V1 = Open)\}$ and $\{(F1 = Filled) \wedge (P1 = High) \wedge (V1 = Open)\}$.

Starting with the first branch, the algorithm then recursively checks how the candidate extended with the positive literal is classified (valid, satisfiable, or inconsistent), and how the candidate extended with the negated literal is classified, on lines 4 and 6.

Lines 7-12 of *evaluate* recursively classify the candidate according to the classification of its two branches. If the branches agree, the candidate is classified the same as the classification of its two branches; however, if an assignment inconsistent with the candidate has been found through unit propagation, when both of the candidate's branches were valid, then the candidate is classified as satisfiable instead of valid[2]. Otherwise, the candidate is classified as satisfiable. The classification based on the two branches is summarized in Table 4.4.

The tester makes an additional optimization: if the first branch examined is found to be satisfiable but not valid, then the other branch does not need to be examined; both a consistent assignment and an inconsistent assignment has been found. Hence, the classification of the other branch is irrelevant as the candidate is known to be

[2]The *notValid* flag indicates that an inconsistent assignment exists for the candidate.

Table 4.4: This table summarizes how a candidate is classified depending on how the two branches are classified and which flags are set. The candidate is valid if the generator is unable to generate an extension to the candidate that is inconsistent. A candidate is inconsistent if no consistent extension to the candidate exists. Otherwise, the candidate is satisfiable. The classification depends on the *notValid* flag as well as whether the branch variable is projected or unprojected.

notValid is false

projected variable		2nd Branch unsat	sat	valid
1st Branch	unsat	unsat	sat	sat
	sat	sat	sat	sat
	valid	sat	sat	valid

unprojected variable		2nd Branch unsat	sat	valid
1st Branch	unsat	unsat	sat	valid
	sat	sat	sat	valid
	valid	valid	valid	valid

notValid is true		2nd Branch unsat	sat	valid
1st Branch	unsat	unsat	sat	sat
	sat	sat	sat	sat
	valid	sat	sat	sat

satisfiable. The conditions under which the second branch need not be examined is summarized in Table 4.5.

Note that this algorithm is sound but not complete in classifying valid candidates. The algorithm is sound in that when a candidate is classified as valid or unsatisfiable, the classification is guaranteed to be correct. A candidate that is classified as satisfiable, however, may in fact be valid. The incompleteness of the algorithm does not affect the completeness of the conflict generation algorithm since a candidate classified as valid is only used for pruning in the candidate generator.

The algorithm may misclassify a valid candidate as satisfiable when it finds an inconsistent assignment that contains an unprojected variable through unit propagation. If an inconsistent assignment contains unprojected variables, the subset of the assignments that correspond to the projected variables may have a consistent extension. By the definition of a valid candidate, as defined at the beginning of this

Table 4.5: This table summarizes the conditions under which the second branch does not need to be examined. The second branch needs to be examined if the candidate has the potential of being classified as something other than satisfiable, as per Table 4.4. The decision to check the second branch depends on the classification of the first branch as well as whether the branch variable is projected or unprojected.

projected variable	1st Branch unsat	sat	valid
Explore 2nd Branch	yes	no	*

unprojected variable	1st Branch unsat	sat	valid
Explore 2nd Branch	yes	*	no

*If $notValid$ is false, then yes, otherwise no.

section, such an assignment should be classified as valid.

The candidate tester uses a flag called $notValid$ to indicate when it has found an inconsistent assignment. The tester sets the $notValid$ flag when the domain of a projected variable is reduced by unit propagation.

For example, consider the variables $P1$ and $F1$ from the propulsion model. Assume that $F1$ is projected and $P1$ is unprojected. In the model, pressure $P1$ is determined by the tank's state $F1$, and visa-versa. Since the a candidate is valid if all extensions to the projected variables are consistent, all candidates of this model are valid; $F1$ can be either $Filled$ or $Empty$ and there is always an assignment to $P1$ that is consistent with the chosen assignment to $F1$. If the tester selects $P1$ first, however, the algorithm always constrain $F1$ by unit propagation, and thus the candidate will be categorized as satisfiable, even though it is valid. The tester selects satisfiable, instead of valid, because it does not know that there exists a consistent extension to any assignment to $F1$.

Consider testing the candidate ($F1 = Filled$) with the propulsion model. The search tree generated for this example is shown in Figure 4-4. As can be seen, the search for this example did not require any backtracking, due to the specific selection of assignments. Upon reaching the assignment of $NoThrust$ to $T1$, node 17, the search

has satisfied all clauses, and in this case, also assigned all variables. The algorithm classifies node 17 as valid. At the previous node, the algorithm detects that it has constrained the value of the projected variable $T1$ by unit propagation, so it marks itself as satisfiable, instead of valid. The algorithm knows that the assignment $(F1 = Filled) \wedge (P1 = High) \wedge (V1 = Closed) \wedge (P2 = Low) \wedge (V2 = Closed) \wedge (PV2 = Low) \wedge (V3 = Closed) \wedge (PV3 = Low) \wedge (P3 = Low) \wedge (R1 = Off) \wedge (T1 = Thrust)$ is inconsistent with the theory, thus, with $(T1 = Thrust)$ removed from the assignment, back to node 15, the resulting assignment is satisfiable. One extension is valid, node 17, and the other is unsatisfiable, node 16. Additionally, since the current node, node 15, is a satisfiable node, the algorithm records its valid child, node 17, as the shortest implicant found thus far. All of the previous nodes also constrain projected variables, so they all return without checking their second branch. Thus, the candidate $(F1 = Filled)$ is classified as satisfiable with the implicant 4.2.

If one of the branches did not assign a projected variable, the other branch can be examined to determine if the other branch was valid. If the other branch were valid, then node branching on the unprojected variable is classified as valid, instead of satisfiable, as there exists one assignment to the unprojected variable that leaves all the projected variables free to take on any value. This one assignment will always be consistent, for any candidate generated with all of its preceding extensions. However, the generator need not generate any further extensions.

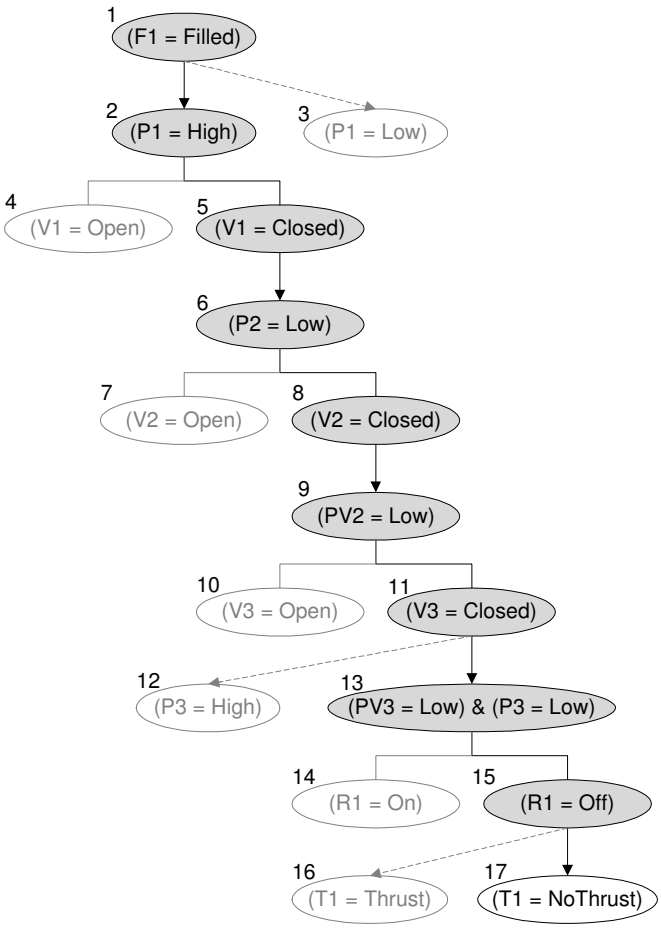

Figure 4-4: The search tree associated with a tester run, with ($F1 = Filled$) as a candidate. White nodes with black text are valid. The grey nodes are satisfiable. White nodes with grey text are untested branches. White nodes with grey, dashed lines connecting them are cases where the search constrained a variable to a value and the grey node represents the alternative that was ruled out. The nodes with arrows are assignments determined by unit propagation, while the ones without arrows are nodes where a choice was made.

testCandidate(C_C)

1. For $\forall(s_c, p_c) \in C_C$ Do
2. If (s_c, p_c) *is the last element in* C_C Then
3. $n.propagate(s_c, p_c, True, False)$
4. Else
5. $n.propagate(s_c, p_c, True, True)$
6. Return $n.evaluate()$

evaluate()

1. If $solution = Unknown$ Then
2. $(s_c, positive_c) \leftarrow \mathcal{C}_\Phi.C_0.first()$
3. $node_p.propagate(s_c, positive_c, True, False)$
4. $solution_p \leftarrow node_p.evaluate()$
5. $node_n.propagate(s_c, \neg positive_c, True, False)$
6. $solution_n \leftarrow node_n.evaluate()$
7. If $(solution_p = solution_n) \wedge (\neg notValid \vee \neg(solution_p = Valid))$ Then
8. $solution \leftarrow solution_p$
9. Else If $((solution_p = Valid) \vee (solution_n = Valid)) \wedge \neg(s_p.v \in V_p)$ Then
10. $solution \leftarrow Valid$
11. Else
12. $solution \leftarrow Satisfiable$
13. Return solution

4.3.1 Propagate

This section presents the algorithm responsible for applying both positive and negative assignments to the clauses, and to the domains of the variables. The algorithm also performs unit propagation. The function *propagate*() first makes the requested assignment, by appropriately restricting the variable's domain, and then by determining which clauses have become satisfied as a result of the assignment. Propagate then performs unit propagation on the remaining clauses.

Propagate takes, as its input, an assignment and three flags. The first flag *positive* indicates whether to remove this assignment from the domain (false) or to constrain the domain to the specified value (true). For example, $(F1 = Full)$ is an assignment. If the flag is false, then it is equivalent to the literal $\neg(F1 = Full)$.

The second flag, *ignoreConstraint*, is true whenever *propagate* should not set the $notValid$ flag, due to restricting the domain of a projected variable. Normally, propagate sets $notValid$ whenever the assignment constrains a projected variable. *ignoreConstraint* is only set to *true* by *evaluate* when both branches are being examined, and thus the restriction of the domain of the projected variable is expected and does not indicate that the algorithm has found an inconsistent assignment.

The third flag, *multiSet* is only used when applying the constraints from the initial candidate. This flag has the effect of delaying unit propagation until all of the initial candidate's constraints have been added. Since the candidate can contain multiple assignments, it would be incorrect to set the $notValid$ flag if one assignment in the candidate constrained the value of some other assignment in the candidate, which would be discovered before the second assignment has been propagated, as both are already set to some value. Unit propagation is delayed until the last assignment of the candidate has been added. For example, if the candidate was $(F1 = Full) \wedge (P1 = High)$, then adding either of these implies the other one. Thus, if $(F1 = Full)$ were

allowed to cause unit propagation, then it would constrain ($P1 = High$). Since $P1$ is a projected variable, this would cause the algorithm to believe that the candidate is at best satisfiable, but the constraint imposed is part of the candidate, so the imposed constraint can be ignored.

The main body of the algorithm has five parts. The first part checks the consistency of the new assignment s_p with the variables that have already been assigned a value. If the variable in s_p is already assigned a value, the new assignment must be consistent with the previous value assigned to the variable. Otherwise, this propagation is inconsistent. For example, if the assignment is $\neg(F1 = Full)$, and $F1$ has been assigned a single value because its domain was reduced to that single value, then this check will verify that the value assigned is consistent with $\neg(F1 = Full)$. So long as the value assigned is not $Full$, the propagation is successful and nothing more needs to be done. Otherwise, the candidate is inconsistent. This corresponds to lines 1-4.

The next two parts have a positive and a negative counterpart, lines 6-22 and 24-40, respectively. The first part modifies the domain of the variable to match the new assignment, lines 6-12 and 24-29. The second part satisfies or constrains all remaining clauses so that they reflect how the new assignment has changed the theory, lines 13-22 and 30-40. The positive counterpart sets the variable to the specified value. The negative counterpart removes the value from the domain of the variable.

For example, if the assignment is $(F1 = Full)$ and the *positive* flag is true, then the domain of $F1$ is set to $Full$. Any clause mentioning $(F1 = Full)$ or an element of $\{\neg(F1 = x)|x \in [D(F1) \setminus Full]\}$ is satisfied, and is removed from the list of unsatisfied clauses. Any clause with $\neg(F1 = Full)$ or an element of $\{(F1 = x)|x \in [D(F1) \setminus Full]\}$ has that literal removed from the clause since the literal is false. When the clause becomes empty, the clause, and hence the theory, is unsatisfiable.

The fourth part is responsible for propagating unit clauses, lines 41-46. If there is only one literal left in a clause, the clause can only be satisfied by one specific assignment. Unit propagation makes this remaining assignment. For example, if a clause was reduced to $\neg(F1 = Full)$, then the current node's assignments are extended by $\neg(F1 = Full)$ by calling propagate on this assignment. This satisfies the clause in the only way possible.

The last part, lines 47-52, is responsible for determining whether all the clauses have been assigned. When this happens, the candidate can be classified as Valid, or if the $notValid$ flag is set, as Satisfiable.

Thus, this algorithm can handle both types of assignments, positive and negative, and can perform unit propagation. All of these operations involve modifying the domains of the variables as well as the set of satisfied clauses and inconsistent literals.

4.3.2 Algorithm

propagate(s_p**,** *positive***,** *ignoreConstraint***,** *multiSet***)**

checkExistingAssignment

1. If $s_p.v \in V_{assigned}$ Then
2. If $\neg((s_p.x \in D(s_p.v))$ xor *positive*) Then
3. $solution \leftarrow Unsatisfiable$
4. Return

positiveAssignment

5. If *positive* Then

constrainDomain

6. If $s.x \in D(s_p.v)$ Then
7. $D(s_p.v) \leftarrow s.x$
8. If $s_p.v \in V_{search} \wedge \neg ignoreConstraint$ Then
9. $notValid \leftarrow True$
10. Else
11. $solution \leftarrow Unsatisfiable$
12. Return

updateClauses

13. For $\forall C \in \mathcal{C}_\Phi$ Do
14. For $\forall (s_c, positive_c) \in C$ Do
15. If $s_p.v = s_c.v$ Then
16. If $(s_p.x = s_c.x)$ xor $positive_c$ Then
17. $\mathcal{C}_\Phi \leftarrow \mathcal{C}_\Phi \setminus C$
18. Else
19. $C \leftarrow C \setminus (s_c, positive_c)$
20. If $C = \{\emptyset\}$ Then
21. $solution \leftarrow Unsatisfiable$
22. Return

negativeAssignment

23. Else

constrainDomain

24. If $s.x \in D(s_p.v)$ Then
25. $\quad D(s_p.v) \leftarrow D(s_p.v) \setminus s.x$
26. $\quad$ If $s_p.v \in V_{search} \wedge \neg ignoreConstraint$ Then
27. $\quad\quad notValid \leftarrow True$
28. Else
29. $\quad$ Return

updateClauses

30. For $\forall C \in \mathcal{C}_\Phi$ Do
31. $\quad$ For $\forall (s_c, positive_c) \in C$ Do
32. $\quad\quad$ If $s_p.v = s_c.v$ Then
33. $\quad\quad\quad$ If $s_p.x = s_c.x$ Then
34. $\quad\quad\quad\quad$ If $positive_c$ Then
35. $\quad\quad\quad\quad\quad C \leftarrow C \setminus (s_c, positive_c)$
36. $\quad\quad\quad\quad$ Else
37. $\quad\quad\quad\quad\quad \mathcal{C}_\Phi \leftarrow \mathcal{C}_\Phi \setminus C$
38. $\quad$ If $C = \{\emptyset\}$ Then
39. $\quad\quad solution \leftarrow Unsatisfiable$
40. $\quad\quad$ Return

unitPropagate

41. If $\neg multiSet$ Then
42. For $\forall C \in \mathcal{C}_\Phi$ Do
43. If $size(C) = 1$ Then
44. $(s_c, positive_c) \in C$
45. $propagate(s_c, positive_c, False, False)$
46. Return

identifySolutionStatus

47. If $size(\mathcal{C}_\Phi) = 0$ Then
48. If $solution = Unknown$ Then
49. If $notValid$ Then
50. $solution \leftarrow Satisfiable$
51. Else
52. $solution \leftarrow Valid$

This section has shown a set of algorithms capable of classifying a candidate as either valid, satisfiable, or unsatisfiable. The addition of the classification of valid has allowed this algorithm to perform implicant extraction during the test process. These implicants can then be used to prune the search space in an effort to find minimal conflicts.

4.4 Empirical Evaluation

The prime implicate generation engine described in this chapter was bench marked with and without validity testing enabled. When valid testing is enabled, the algorithm works as described above. More specifically, to test for validity, like unsatisfiable, one must prove that all extensions to the candidate are classified the same way. When validity testing is disabled, the tester need only check another extension when it finds an unsatisfiable extension. Otherwise the algorithm knows that at least one solution exists and can safely classify the candidate as satisfiable. In this mode, the tester may still return valid, but only if this can be determined through unit propagation.

The implementation of the prime implicate generator outlined in this chapter is called CompileSAT. When comparing the performance of CompileSAT with and without validity testing, having validity testing enabling showed an improvement on random problems in which in ratio of clauses to variables was lower than four. In this region, where there tends to be more valid candidates than unsatisfiable candidates, the engine testing for validity was capable of finding all of the prime implicates up to five times faster. In regions greater than four, the validity testing engine demonstrated equivalent performance; it was within 15% of CompileSAT with valid testing disabled. Note that it only rarely took longer to solve the problem with validity testing enabled. Figure 4-5 also shows the performance of the candidate generator when the valid-testing algorithm from Section 4.3 is replaced by a straight DPLL algorithm. The generator remains the same. The DPLL-based algorithm takes an order of magnitude longer to run, at best, and at worst, several orders of magnitude longer. This is in large part due to pruning done by implicants, as is done in a number of modern SAT solvers, such as Chaff [19].

In Figure 4-5, the DPLL algorithm tends to perform badly on the left because it is unable to prune anything from the search space, while the CompileSAT algorithms are able to prune some implicants, when validity testing is disabled, and all implicants when validity-testing is enabled. When searching for conflicts in this region, the candidate tester need not examine very many extensions, as there are very few clauses, but the generator needs to generate all possible partial conflicts. Towards a ratio of 2-4 clauses/variable, the tester needs to examine additional extensions to classify the candidates. The generator also needs to examine more candidates, as fewer are eliminated by implicants and there are still few minimal conflicts. At higher ratios, the generator prunes a number of candidates that are unsatisfiable, and so the curve starts dropping. For CompileSAT, when valid testing is disabled, the algorithm still gets a reduced number of candidates generated due to the number of candidates that are easily classified as valid. It also benefits from having implicants, though the implicants are not very good as the tester can only find implicants through unit propagation. Towards the right, the algorithm not testing for validity tends to have many more unsatisfiable candidates than valid candidates and so begins performing much like the DPLL algorithm, performing better than it due to use of implicants and because it operates in a clause-directed manner. The valid-testing algorithm benefits significantly in low clause/variable ratios as the tester can quickly determine that most candidates are valid, and so the generator prunes large sub-trees of its search space. This benefit is present only at the beginning where large sub-trees can be pruned this way, where the implicants are short. Otherwise, the validity-testing algorithm finds mostly unsatisfiable branches, so it does not do any more work than when valid-testing is disabled.

In figure 4-6, the problem is slightly harder because it has an additional projected variable with a domain size of 5. One will note that the problem takes about twice as long to solve, on average, with this extra variable, and that the full search space

is five times larger. The trends between this example and the previous example are approximately the same. The graph has what appears to be a random bump towards 8 clauses/variable. This is apparently due to noise based on the number of samples. As the number of clauses/variable get close to 10, the hardness of the problem is highly variable, so there are too few samples in this region to get a smoother curve.

As the data shows, testing for validity is always advantageous. Combined with implicant extraction, the prime implicate generator performs much better on problems with fewer clauses per variable. For even highly constrained problems such as a more complicated propulsion model similar to the one in this thesis, testing validity is beneficial. This more complicated model has 8 projected variables and 14 unprojected variables. Each variable has a domain size of 3 to 5 values. The DPLL-based algorithm took 568 seconds to project the model over the state and observation variables. The non-valid testing algorithm took 194 seconds, and the valid-testing algorithm took 170 seconds. The validity-testing algorithm took 12% less time then the algorithm that did not test for validity. It took 70% less time than the DPLL-based algorithm.

4.5 Summary

This chapter has shown how to implement a prime implicate generator as an unsatisfaction engine based on the algorithms from the previous chapter. The generator uses iterative deepening on a systematic tree of candidates such that each candidate is generated only once and sub-trees of candidates can be eliminated through the application of three different pruning rules. These rules are based on implicants and conflicts extracted by the candidate tester. The candidate tester, in turn, is able to extract implicants through the use of an efficient routine for determining if a candidate or one if its extensions is valid.

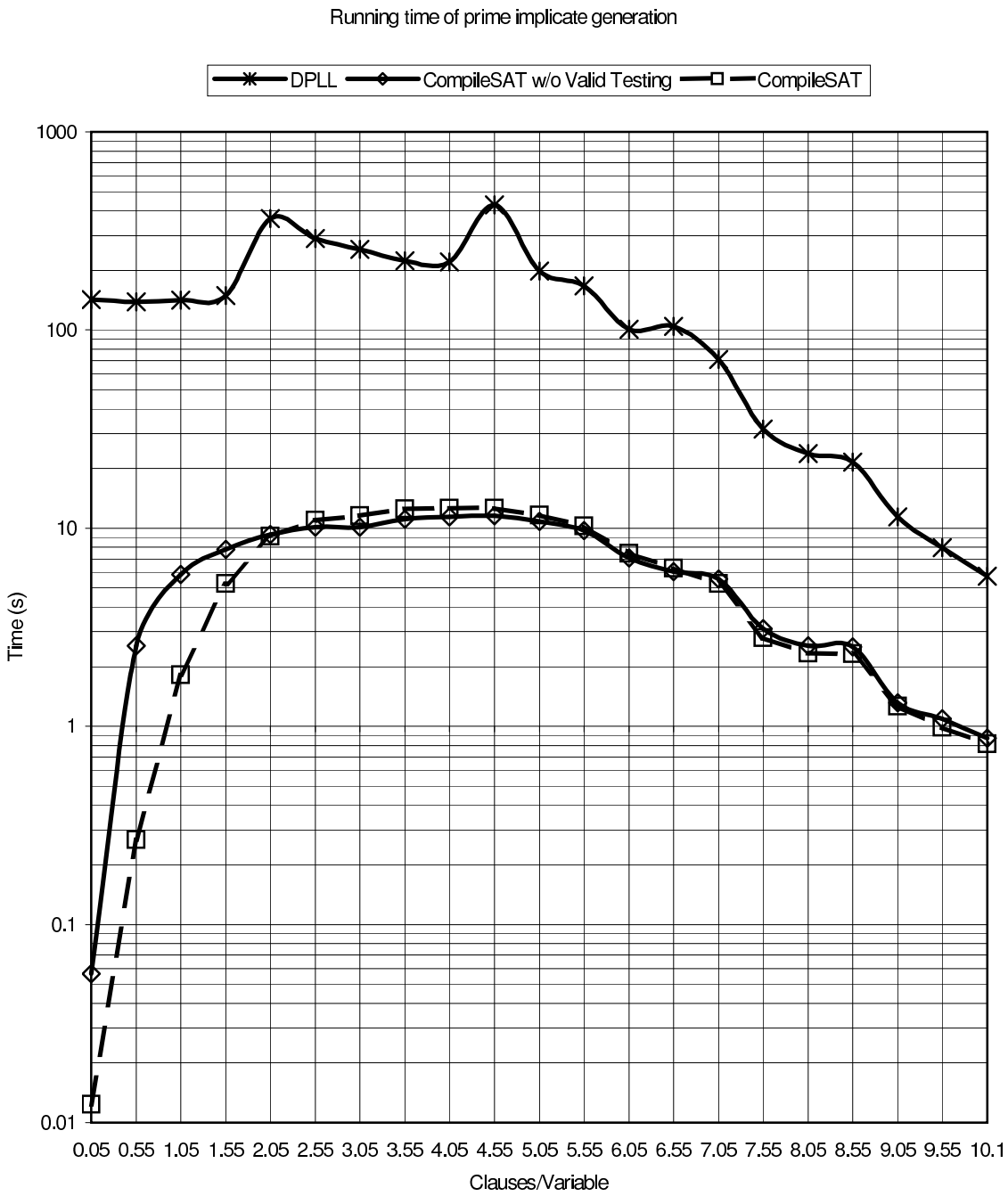

Figure 4-5: This graph shows the performance benefit of using a SAT engine capable of detecting Valid candidates and using decomposition. Each data point represents the average of 100 test cases. Each test ran on 20 variables, 5 of which were projected variables. Each clause had 3 literals. Each variable had 5 domain elements. These tests were run on a 733 MHz Pentium III processor.

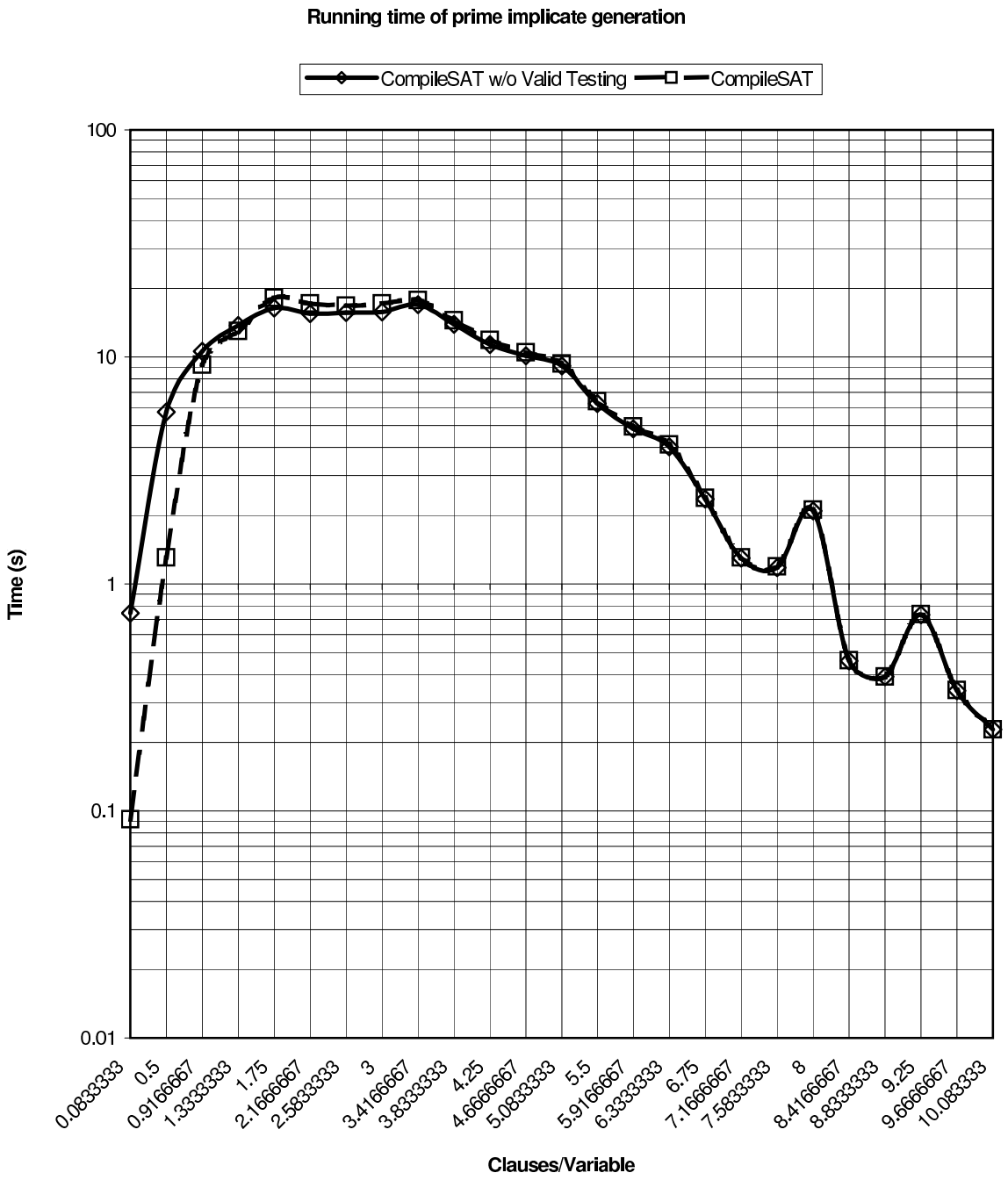

Figure 4-6: This graph shows the performance benefits on a harder problem, as it has more projected variables. Each data point represents the average of 100 test cases. Each test ran on 12 variables, 6 of which were projected variables. Each clause had 3 literals. Each variable had 5 domain elements. These tests were run on a 733 MHz Pentium III processor.

The tester utilizes unit propagation to focus its search quickly. The tester also partitions its search space, so it only searches a subspace of assignments once. The tester is also clause-directed with non-binary domain variables, hence it only assigns values that are necessary to satisfy its clauses. The algorithm examines many fewer extensions than a non-clauses directed algorithm.

Chapter 5

Summary and Future Work

5.1 Conclusion

In an effort to reduce the runtime computation required for model-based reasoning in real-time systems, this thesis develops the algorithms necessary to perform pre-runtime model compilation. Two types of compilation algorithms are required: projected prime implicate generation for the estimation of the system, and projected prime implicants for the control of the system. This thesis presents algorithms for both problems and implements them using a projected minimal conflict generator as the core algorithm. The projected prime implicates are obtained as negated projected minimal conflicts of the original model. The projected prime implicants are obtained as projected minimal conflicts of the negated model.

The projected minimal conflict generator is a sophisticated generate-and-test algorithm that employs a number of optimizations that improve the compilation process. In order to keep the memory bound of the generation process small, the candidate generator uses an iterative deepening algorithm on a systematic search tree. Thus, the compilation process can be allowed to run for as long as desired, limited only

by processing time, and does not suffer from space-explosion as would be the case for breadth-first search. The candidate generator employs three pruning rules that significantly reduce the number of candidates generated and tested. The first rule eliminates all non-minimal conflicts by pruning supersets of minimal conflicts. The second and third rules are based on implicants that are extracted during the testing process. The second rule prunes supersets of implicants. Because they must also be implicants, they cannot be conflicts. The third rule prunes candidates that can be extended to be supersets of implicants, since they also cannot be conflicts.

The candidate tester identifies a candidate as inconsistent or valid. There are two key concepts that make testing efficient: 1) the tester uses a clause-directed search and 2) the tester operates directly on the original finite-domain variables of the model, as it treats the problem as a CSP. This approach ensures that only those variables that are necessary to determine the validity of the candidate are assigned values. In contrast, encoding the variables using sets of binary variables would require domain axioms, mutual exclusion and exhaustion clauses, which would force the tester algorithm to assign a value to every binary variable.

The generate-and-test approach of the projected minimal conflict generator allows the algorithm to perform projection and minimal conflict generation in a single step. This approach can be significantly better than approaches that must perform these two steps separately, especially when a large number of variables are projected out.

5.2 Contributions

This thesis contributes a model compilation algorithm that generates projected prime implicates and implicants based on a generate-and-test algorithm.

The main contribution of the thesis lies in the development of the algorithm for the projected minimal conflict generator. This contribution has two parts: a candidate generator that efficiently prunes sub-trees of candidates based on implicants and a candidate tester that efficiently identifies implicants by testing for validity. For the candidate generator, we presented a method for creating a systematic search tree of all partial assignments over a set of finite-domain variables. We also presented a method for performing pruning while searching the tree with an iterative deepening algorithm. Previous approaches were only designed to generate complete assignments, as a tree, and would generate them in a breadth-first manner, thus incurring a significant memory cost.

This thesis also contributes improvements to the candidate tester, specifically converting the clause-directed A* algorithm presented by Ragno's thesis [14] into a validity-testing classification algorithm, and extended the algorithm by adding rules to classify candidates that were not directly valid or inconsistent, but whose children are valid or inconsistent. Finally, the candidate tester has been extended to support implicant extraction.

5.3 Future Work

This thesis demonstrates a method for projected prime implicate generation that scales to many real world problems. However, improved efficiency would expand the scope of the problems that could be solved. This section proposes several future extensions that promise to improve efficiency, in particular, for improving the candidate

tester:

Dynamic Variable Re-Ordering for the Candidate Generator Using a fixed variable ordering can lead to inefficient pruning in the generator. If the variable ordered last in the variable ordering prunes a specific value, the generator will only test and prune candidates including this assignment as leaves of the generator's search tree. If the variables were reordered so that the last variable was first, the whole generator sub-tree with that assignment as its root could be pruned. Thus, the generator should benefit from dynamic variable reordering. This reordering could be based on the number of conflicts/implicants that the variable appears in as well as their length. For example, a heuristic could be to reorder the variables so they are ordered based on how often they appear in conflicts and implicants and/or based on the length of the conflicts/implicants in which they appear.

That is, based on the consideration that if a variable appears in many conflicts and implicants, it is likely that many extensions of the candidate, and all of their sub-trees can be pruned. Similarly, if the conflicts and implicants are short, then it requires fewer extensions before the candidate can be pruned.

A Complete Candidate Tester The tester is currently sound but not complete with respect to determining validity. It is possible that changing the criteria by which the algorithm chooses its next assignment can make the validity testing complete. Currently the tester chooses the first assignment of the first clause. The tester is not complete when there are projected variables that are constrained to the same value as an unprojected variable. If the tester chooses to assign the unprojected variable first, the algorithm will unit propagate the value of the projected variable and the algorithm is forced to conclude it found a conflict, but it has not (Note that this conflict makes the candidate satisfiable. If the conflict were tested, it would be determined to be either valid or satisfiable). This problem may be resolved if the tester always choses to assign projected variables before assigning any unprojected variables, other than

through unit propagation.

Improved Implicant Extraction in the Candidate Tester Implicants generated by the candidate tester are not always minimal. They often contain a few extra variables that got set to values but were actually unconstrained once other values were chosen. They are also non-minimal if assigning a value to one or more unprojected variables make the variable unconstrained. The tester should be able to determine which variables have this property and generate smaller implicants by eliminating these variables from the implicant. It is uncertain if it is worth the additional computation required to create a smaller implicant, in particular if it is expected to be only one or two assignments shorter.

Use an Better Data Structure For Propagation in the Candidate Tester The candidate tester is fairly inefficient in how it handles unit propagation as well as branching. The tester would benefit from using a better data structure for managing its assignments and clauses. Recent advances in SAT engine data structures and techniques should be easy to incorporate into the validity tester, as it has a similar underlying algorithm.

5.4 Summary

This thesis shows that the generate-and-test method utilized in the minimal conflict generator allows for the efficient generation of projected prime implicates and projected prime implicants. The candidate generator used in the minimal conflict generator uses a space space-efficient search algorithm for generating all partial candidates. The candidate tester uses a classification algorithm that can distinguish between valid, satisfiable, and unsatisfiable.

Bibliography

[1] Robert L. Causey. *Logic, Sets, and Recursion*. Jones & Bartlett, January 2001.

[2] S. Chung, J. Van Eepoel, and B. C. Williams. Improving Model-based Mode Estimation through Offline Compilation. In *Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2000.

[4] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[5] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[6] J. de Kleer and R. Reiter. *An improved incremental algorithm for generating prime implicates*, chapter 1. http://www2.parc.com/spl/members/dekleer/, Xerox Palo Alto Research Center, 1999.

[7] Johan de Kleer. An assumption-based tms. *Artif. Intell.*, 28(2):127–162, 1986.

[8] Alvaro del Val. Approximate knowledge compilation: The first order case. In *Proc. Thirteenth (U.S.) National Conf. on AI*, number 13 in AAAI, pages 498–503, Portland, Oregon, 1996. AAAI.

[9] D. Bernard *et al.* Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, 1999.

[10] Kalev Kask, Rina Dechter, and Javier Larrosa. Unifying cluster-tree decompositions for automated reasoning. In *Proceedings of AIJ*. AIJ, June 2003. http://www.ics.uci.edu/ dechter/.

[11] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[12] Hector Levesque and Gerhard Lakemeyer. *The Logic of Knowledge Bases*. MIT Press, February 2001.

[13] J. Pineau and S. Thrun. An integrated approach to hierarchy and abstraction for pomdps. Technical Report CMU-RI-TR-02-21, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, August 2002.

[14] Robert Ragno. Solving Optimal Satisfiability Problems Through Clause-Directed A*. Master's thesis, Massachusetts Institute of Technology, MIT Space Engineering Research Center, June 2002. SSL #17-02.

[15] Laurent Simon and Alvaro del Val. Efficient consequence finding. In *IJCAI*, pages 359–370, 2001.

[16] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE*, volume 9, pages 212–237, Jan 2003.

[17] Brian C. Williams and Michel D. Ingham. Model-based Programming: Controlling Embedded Systems by Reasoning About Hidden State. In *Eighth Int. Conf. on Principles and Practice of Constraint Programming*, Ithaca, NY, September 2002.

[18] Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *In Proceedings of IJCAI-97*, 1997.

[19] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.