Constraint-Based Local Search for the Automatic Generation of Architectural Tests

Pascal Van Hentenryck[1], Carleton Coffrin[1], and Boris Gutkovich[2]

[1] Brown University, Providence RI 02912, USA
[2] Intel Corporation, Haifa, Israel

Abstract. This paper considers the automatic generation of architectural tests (ATGP), a fundamental problem in processor validation. ATGPs are complex conditional constraint satisfaction problems which typically feature both hard and soft constraints and very large domains (e.g., all memory addresses). Moreover, the goal is to generate a large number of diverse solutions under tight runtime constraints. To improve solution diversity, this paper proposes a novel approach to ATGPs by modeling them as MaxDiverse$k$Set problems and solving them with constraint-based local search over conditional variables. The paper presents the semantics and implementation of conditional variables in this context and demonstrates the computational benefits of the approach.

1 Background and Motivation

The automatic generation of architectural tests is a fundamental and complex problem in processor design. It consists of generating random sequences of instructions obeying specified constraints. The complexity of this process prevents the problem from being represented and solved globally. Instead the problem is traditionally solved by an incremental process (see Figure 1) which generates one instruction at a time and transforms the constraints on the sequence into constraints on the test generation of single instructions.

This paper considers the main step in this process: the single instruction generator. This automatic test generation problem (ATGP) can be viewed as a constraint satisfaction problem involving three types of constraints:

1. **Architectural Constraints** that specify the instruction set, i.e., which instructions are valid.
2. **Test Scenario Constraints** that specify the intention of a validation engineer.
3. **State Constraints** that specify the current architectural state maintained by the test generator.

The goal is not to find a single solution or to find all solutions, which is impractical due to the size and complexity of modern architectures. Rather, it is to generate diverse tests which exercise the architecture as thoroughly as possible.

To make the problem concrete, consider the simple example depicted in Figure 2. The left table specifies the instruction set. The first two instruction types

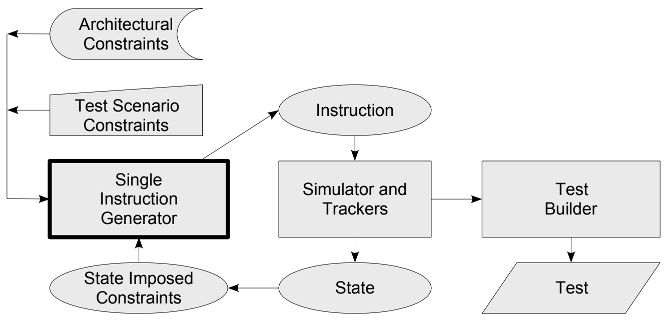
Architectural Constraints
Test Scenario Constraints
Instruction
Single Instruction Generator
Simulator and Trackers
Test Builder
State Imposed Constraints
State
Test

Fig. 1. The Architecture of Automatic Generation of Architectural Tests

Instruction Set				
add	register	r1	r2	r3
sub	register	r1	r2	r3
load	immediate	r1	constant	
load	memory	r1	address	
jump	immediate		constant	
jump	memory		address	

Test Scenario: address = 1234 ∨ constant = abcd			
jump	memory		1234
load	immediate	12	abcd
jump	immediate		abcd
load	memory	4b	1234

Fig. 2. A Simple Processor Instruction Example

are arithmetic operations that operate on registers and have three registers. The third and fourth instruction types are an immediate and a memory load. The last two instruction types are immediate and memory jump. The right table depicts a disjunctive constraint proposed by a validation engineer and the instructions that satisfy it. Observe that the constraint implicitly specifies that the instruction is either a *load* or a *jump*. The CSPs induced by ATGPs are generally quite complex for various reasons:

1. The ATGP is a so-called conditional constraint satisfaction problem [1][3], since the existence of some instruction fields are conditioned on the value of other fields such as the opcode and the addressing mode. In our simple example, the *constant* field is only defined if the instruction is a load or a jump and the mode is immediate.
2. The ATGP typically operates over large instruction sets due to the complexity of modern architectures. Moreover, some of these fields have very large domains, since they represent memory addresses or large constants.
3. The ATGP typically contains both soft and hard constraints, specified by the validation engineer.
4. The validation engineer may specify a desired distribution of instructions to bias toward some specific instructions.

[3] Conditional CSPs were originally called Dynamic CSPs in [1]. Both terms are heavily overloaded; we use CCSPs in this paper.

Earlier approaches (e.g., [2]) address these difficulties by introducing conditional variables to handle the CCSP and use randomization to obtain diverse solutions.

In this paper, we reconsider the issue of producing diverse solutions for ATGPs. We propose to view the ATGP as a MaxDiverse$k$Set problem in the sense of [3] over the CCSP and to use constraint-based local search to obtain high-quality solutions to the model. We show that the resulting approach produces significant improvements in solution diversity compared to pseudo-random solutions, while the running times remain reasonable. The technical contributions can be summarized as follows:

1. The paper proposes a new approach to producing diverse solutions to ATGPs, exploiting jointly earlier results in constraint programming and constraint-based local search.
2. The paper shows that constraint-based local search finds high-quality solutions to MaxDiverse$k$Set problems, even when the number of required solutions increases significantly. The paper also quantifies the quality loss experimentally.
3. The paper presents a new semantics for conditional variables, which is more suited for ATGPs than the original one which was designed for configuration problems.
4. The paper presents the algorithmic foundations for constraint-based local search over conditional variables and shows how constraint-based local search can accommodate conditional variables naturally and compositionally.

The rest of the paper is organized as follows. The next two sections present the building blocks of our approach. Section 2 presents the approach to generate diverse solutions to CSPs and presents experimental results on some simple problems. Section 3 discusses the modeling of the problem as a CCSP and the semantics of conditional variables and how to perform constraint-based local search over conditional variables. Section 4 shows how to model ATGPs in CBLS and search for diverse solutions. Section 5 reports experimental results of our prototype implementation on some benchmark ATGPs to validate the approach, and Section 6 concludes the paper.

2 Generating Diverse Solutions

Modern approaches to ATGPs (e.g., [2]) use a pseudo-random exploration of the search space to generate diverse solutions. This is often sub-optimal however. The goal of this paper is to provide a more principled approach to diversity for ATGPs. For this reason, ATGPs are modeled as MaxDiverse$k$Set problems, which were studied extensively in [3]. More precisely, given a CSP $\mathcal{P}$, its set of solutions $sol(\mathcal{P})$, and a function δ to measure the distance between solutions, the MaxDiverse$k$Set problem for $\mathcal{P}$ consists in finding a set of solutions $S = \{s_1, \ldots, s_k\}$ maximizing

$$\sum_{1 \leq i < j \leq k} \delta(s_i, s_j)$$

In ATGPs, it is convenient to use the Hamming distance for the distance function δ between two solutions $s_1 = \langle v_1, \ldots, v_n \rangle$ and $s_2 = \langle w_1, \ldots, w_n \rangle$

$$\delta(s_1, s_2) = \sum_{i=1}^{n} (v_i \neq w_i)$$

This problem is in general quite difficult from a practical standpoint, since it requires to search for k solutions simultaneously and produces very large CSPs (See also [3] for the theoretical complexity which is $FP^{NP[logn]}$-complete). Moreover, in this application domain, it is often desirable to produce the solutions incrementally, one at a time. For this reason, the incremental algorithm in [3] is an excellent candidate for finding approximated solution to MaxDiverse$k$Set problems associated with ATGPs. The algorithm can be formalized as follows:

```
IncrementalGeneration($\mathcal{P}$)
1  $S \leftarrow \{\}$;
2  while $|S| \leq k$
3      do find $s \in sol(\mathcal{P})$ maximizing $\sum_{e \in S} \delta(s, e)$;
4          $S \leftarrow S \cup \{s\}$;
5  return $S$;
```

The algorithm generates one solution at a time until a set of the required cardinality is obtained. The core of the algorithm is in Line 3, which generates the solution maximizing the distance to already generated solutions.

To implement line 3, reference [3] proposed a constraint-programming algorithm using a global constraint for enforcing arc consistency on the distance constraint (derived from the objective). The resulting algorithm was applied to the generation of three solutions to a real-life configuration application, where "three" is considered the "optimal" cardinality to present to users in recommender systems. However, in ATGPs, the cardinality is in general much larger and it is typical to generate 50 to 100 instructions. This led us to approximate the computation in Line 3 with constraint-based local search in order to scale the incremental algorithm effectively.

Figures 3 and 4, depict experimental results on the allinterval series of size 10 and the 100-queens problem (with 100 variables). The figures report the average Hamming distance between two generated solutions (y-axis) in sets of increasing cardinality (x-axis) for three methods: the incremental CBLS algorithm (Incremental-CBLS), the incremental CP algorithm (Incremental-CP), and a control CBLS algorithm (Control-CBLS), which generates pseudo-random solutions using CBLS. The CBLS techniques are averaged over ten runs. Incremental-CP uses a global constraint maintaining arc consistency for the Hamming distance and the variable/value heuristic proposed in [3]. Incremental-CBLS uses a generic min-conflict search and a global constraint for the Hamming distance, with value-based violations [4] enabling move evaluation in constant time. The average distance of two solutions is calculated by dividing the MaxDiverse-kSet value by the number of solution pairs in the set S. Specifically, given a

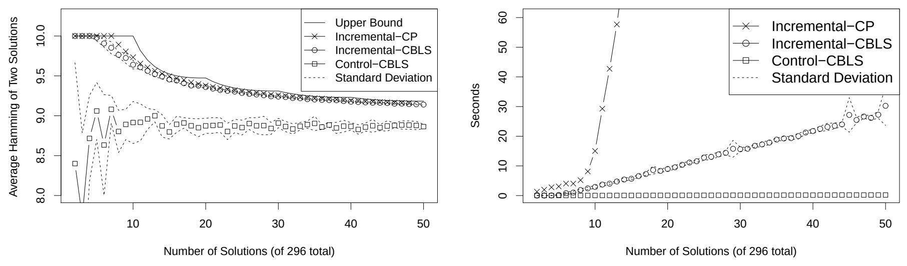

Fig. 3. Diversity Results and Computation Times for the Allinterval Series of Size 10.

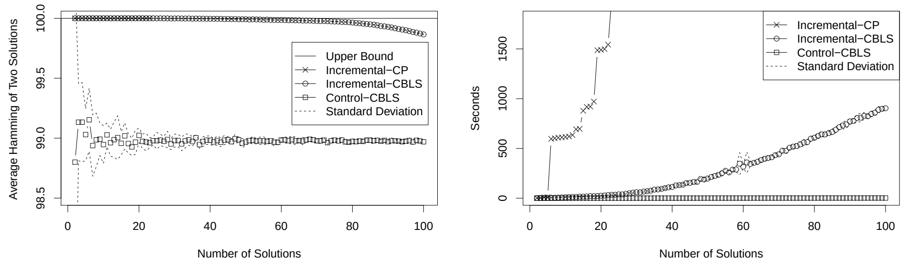

Fig. 4. Diversity Results and Computation Times for the Queens Problem of Size 100.

distance function δ and a set of solutions $S = \{s_1, \ldots, s_k\}$ the average distance is

$$\frac{\displaystyle\sum_{1 \leq i < j \leq k} \delta(s_i, s_j)}{\binom{k}{2}}$$

The figures also depict a simple upper bound on the maximal diversity and the standard deviation. The upper bound ignores the problem constraints and uses only the domains of the variables, shifting the assignment by one for each successive solution. Assuming that the problem variables are $v_1, \ldots, v_m$ and that $D(v_i)_n$ denotes the n-th value of domain of v_i, the upper bound is computed as follows:

```
HammingUpperBound($k$)
1  for $i \in 1..m$
2      do for $j \in 1..k$
3            do $s_{i,j} \leftarrow D(v_i)_{((j-1)\%|D(v_i)|)+1}$ return $\sum_{1 \leq i < j \leq k} \sum_{a=1}^{m} s_{a,i} \neq s_{a,j}$
```

opcode	mode	r1	r2	r3	address	constant
add	register	0..0xff	0..0xff	0..0xff		
sub	register	0..0xff	0..0xff	0..0xff		
load	immediate	0..0xff				0..0xffff
load	memory	0..0xff			0..0xffff	
jump	immediate					0..0xffff
jump	memory				0..0xffff	

Table 1. The Variables in The Simple ATG Problem

For consistency with the MaxDiverse$k$Set value, the upper bound is also divided by the number of solution pairs in the set S, i.e. $\binom{k}{2}$, to produce a bound on the average distance between two solutions. The results are particularly interesting. They indicate that Incremental-CBLS is near-optimal in quality on both benchmarks, since the distance to the upper bound is minimal, and outperforms Control-CBLS significantly and asymptotically (the two curves never converge). The quality of Incremental-CP is of course at least as good as the quality of Incremental-CBLS. However, the computational results also show that Incremental-CP does not scale well once the number of solutions is increased. Note also that the Control-CBLS is extremely fast, since it simply finds pseudo-random solutions at each step and never tries to optimize the diversity.

3 Conditional Variables

When modelling an ATGP, it is traditional to introduce a variable for each field which appears in some instruction. As an illustration, Table 1 describes the variables and their domains in our simple ATG example, as well as the instructions they are used in. The first row depicts the variables, while subsequent ones depict the instructions, the variables they use, and the domains of these variables. For instance, the *add* and the *sub* instructions do not use the *address* and *constant* variables. As a result, the ATGP gives rise to a Conditional Constraint Satisfaction Problem (CCSP) in the sense of [1], as observed in [5]. Mittal and Falkenhainer showed how to transform a CCSP into a CSP by introducing additional variables denoting whether a variable is *active*, i.e., whether its condition holds. Subsequent work produces new reformulation techniques (e.g., introducing a dummy value in the domain to express whether the variable is active) and dedicated algorithms to produce significant improvements in efficiency [6, 7].

In the context of ATGPs, Moss [2] extended constraint-programming solvers with the concept of conditional variables. A conditional variable y is a pair (x, C), in which x is a regular variable and C is a constraint. Figure 5 depicts the modeling of the simple ATG problem with conditional variables. The variable section declares the variables, their domains, and possibly a condition. The last 5 variables are conditional and depend on the values of the opcode or mode variables. The second section depicts the architectural constraints which, together with

Variables:

$\text{opcode} \in \{\text{add}, \text{sub}, \text{load}, \text{jump}\}$
$\text{mode} \in \{\text{register}, \text{immediate}, \text{memory}\}$
$r_1 \in \{0..0\text{xff}\}$ if $\text{opcode} = \text{add} \vee \text{opcode} = \text{sub} \vee \text{opcode} = \text{load}$
$r_2 \in \{0..0\text{xff}\}$ if $\text{opcode} = \text{add} \vee \text{opcode} = \text{sub}$
$r_3 \in \{0..0\text{xff}\}$ if $\text{opcode} = \text{add} \vee \text{opcode} = \text{sub}$
$\text{address} \in \{0..0\text{xffff}\}$ if $\text{mode} = \text{memory}$
$\text{constant} \in \{0..0\text{xffff}\}$ if $\text{mode} = \text{immediate}$

Architectural Constraints:

$(\text{opcode} \in \{\text{add}, \text{sub}\} \wedge \text{mode} = \text{register}) \vee$
$(\text{opcode} = \text{load} \wedge \text{mode} \in \{\text{immediate}, \text{memory}\}) \vee$
$(\text{opcode} = \text{jump} \wedge \text{mode} \in \{\text{immediate}, \text{memory}\}$

Test Scenario:

$\text{address} = 1234 \vee \text{constant} = abcd$

Fig. 5. Modeling the Simple ATG Example with Conditional Variables

the domains, specify the legal instructions. The test scenario is depicted in the third section.

The motivation for introducing conditional variables was twofold. First, Moss argued that the reformulation techniques are not necessarily feasible in ATGPs, since the domain can already take the entire memory word. She also argued that the more specialized techniques are not general enough for ATGPs. Second, the availability of conditional variables at the modeling level makes it possible to design search algorithms exploiting the semantics of conditional variables in the ATGP context. In particular, the search procedure in [2] nondeterministically decides the active status of each variable and enforces the condition or its negation by adding a new constraint.

This research follows a similar path but for constraint-based local search instead of constraint programming. It uses conditional variables as first-class modeling objects and uses their semantics to guide the search, albeit in a fundamentally different way. The rest of this section will specify the semantics of conditional variables which is only defined informally in [2] and extends the concept of constraint violations in CBLS to conditional variables.

The Semantics of Conditional Variables There are many possible semantics for conditional variables, each of which may be appropriate for a particular application domain. Mittal and Falkenhainer use what we call a *lenient* semantics in which a constraint holds as soon as one of its conditional variables is inactive. The lenient semantics are appropriate for the configuration problems they consider, but is not suited for ATGPs. Consider the instruction set proposed earlier and the test scenario

$$constant > 10.$$

Using the lenient semantics, the ATGP problem admits as solutions, all the intructions that do not include a constant, i.e., the arithmetic instructions and the memory load and jump instructions, as well as all those for which the constant is greater than 10. Indeed, if the constant variable is inactive, the constraint is ignored in the lenient semantics.

The semantics we propose are strict on the basic constraints: A constraint only holds if all its variables are active. However, the strictness requirement does not carry over logical or threshold connectives. Consider the test scenario

$$address = 1234 \vee constant = abcd.$$

If we require strictness on the disjunction, the resulting ATGP has no solution, since no instruction has both an address and a constant. The intended semantics here is to generate instructions which have either an address with value "1234" or a constant whose value is "abcd". Finally, consider a Hamming distance constraint

$$\sum_{i=1}^{n}(v_i \neq w_i) \geq d$$

which involves reification. The semantics cannot be strict over the entire constraint or it would never be instrumental in comparing two solutions. Rather the reified constraint should only return 1 (true) when it is satisfied and all its variables are active and 0 (false) otherwise. In other words, the strictness is limited to the reified constraint and not the enclosing expression. Note also that a lenient semantics does not make sense for this constraint, since the constraint would hold as soon as a variable is not active. Even a lenient semantics on the reified constraints is not desirable, since similar instructions would have a positive score when many of their variables are undefined.

Figures 7, 8, and 9 describe the semantics of the small language given in Figure 6. The figures use *var(y)* and *cond(y)* to denote the variable and condition part of a conditional variable. The semantics are given for an assignment α of values to the variables. The figures also give the invariants which maintain the truth values of all constraints, showing that the semantics can be implemented compositionally and maintained incrementally, as was the case for differentiable invariants [8]. This indicates that our approach does not require any program transformation and leverages all the functionalities of CBLS. Figure 7 gives the semantics for the evaluation of expressions and should not raise any issue. Observe that the condition of a conditional variable is ignored and is handled at a different level. Figure 8 gives the semantics of constraints. The primitive constraints hold when their traditional semantics hold and when their expressions are well-defined, meaning that their variables are active. The logical connectives simply apply the semantics recursively on their subexpressions. By definition, a conjunction is always strict: all its variables must be active. The rest of the figure specifies when an expression is well-defined. Figure 9 depicts how to handle reification, which is important to give the semantics to the Hamming distance over conditional variables. The evaluation of a reified constraint simply calls the semantic definition for constraints and uses the Kronecker symbol δ to convert Boolean values into 0/1 values:

$$\delta(b) = \begin{cases} 1 & \text{if } b = true; \\ 0 & \text{otherwise.} \end{cases}$$

$v \in \mathcal{N}$; $x \in$ *Variable*; $y \in$ *ConditionalVariable*; $e \in$ *Expression*; $c \in$ *Constraint*.
$e ::= v \mid x \mid y \mid e + e \mid e - e \mid e \times e \mid c$
$c ::= e = e \mid e \leq e \mid c \vee c \mid c \wedge c$

Fig. 6. The Syntax of Expressions and Constraints (Partial Description).

$$\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{E}}[v] & = v & i_v & \leftarrow v \\
\underset{\alpha}{\mathbb{E}}[x] & = \alpha(x) & i_x & \leftarrow x \\
\underset{\alpha}{\mathbb{E}}[y] & = \alpha(var(y)) & i_y & \leftarrow i_{var(y)} \\
\underset{\alpha}{\mathbb{E}}[e_1 + e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] + \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1+e_2} & \leftarrow i_{e_1} + i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 - e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] - \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1-e_2} & \leftarrow i_{e_1} - i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 \times e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \times \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1 \times e_2} & \leftarrow i_{e_1} \times i_{e_2}
\end{array}$$

Fig. 7. The Evaluation of Expressions and their Underlying Invariants.

The rule of well-definedness of a reified constraint simply returns true, meaning that the definedness is local to the reified constraints and does not propagate to the enclosing expression.

It is worth highlighting that the lenient semantics can be obtained in a very similar way: just replace the conjunction by disjunction and negate the well-definedness condition in the first two lines of Figure 8 and include a recursive call in the definition of well-definedness for reified constraints. So it is possible to accommodate easily both the lenient and the strict semantics in the same system. Observe also that the generated invariants are acyclic by construction since the condition in a conditional variable can only use previously declared variables. Acyclicity is in fact always assumed in CCSPs and is natural in their application domains.

The Definition of Violations Figure 10 depicts the violations of constraints over conditional variables, as well as the invariants to maintain them. Several points deserve to be highlighted. First, the definition of violations capture the importance of conditions in conditional variables. The violations of a constraint $c(y_1, \ldots, y_n)$ over conditional variables is expressed directly in terms of the violations of the same constraint over traditional variables $c(var(y_1), \ldots, var(y_n))$ but it adds a penalty ϕ for each of its conditional variables whose condition does not hold. The expression $\underset{\alpha}{\mathbb{U}}[e]$ computes the number of inactive conditional variables in e. The penalty is large to focus the search on making the conditional variables active before considering the other violations. Second, observe that conditional variables in reifed constraints are not counted, reflecting the semantics of reification in this context too. Finally, the invariants are once again computed naturally, showing the compositional nature of the implementation.

4 Modeling and Solving The ATGP Problem

We now describe how to model and solve ATGPs.

$$\begin{array}{ll|l}
\underset{\alpha}{\mathbb{B}}[e_1 = e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] = \underset{\alpha}{\mathbb{E}}[e_2] \wedge \underset{\alpha}{\mathbb{D}}[e_1] \wedge \underset{\alpha}{\mathbb{D}}[e_2] & b_{e_1=e_2} \leftarrow i_{e_1} = i_{e_2} \wedge d_{e_1} \wedge d_{e_2} \\
\underset{\alpha}{\mathbb{B}}[e_1 \leq e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \leq \underset{\alpha}{\mathbb{E}}[e_2] \wedge \underset{\alpha}{\mathbb{D}}[e_1] \wedge \underset{\alpha}{\mathbb{D}}[e_2] & b_{e_1 \leq e_2} \leftarrow i_{e_1} \leq i_{e_2} \wedge d_{e_1} \wedge d_{e_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \vee r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \vee \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \vee r_2} \leftarrow b_{r_1} \vee b_{r_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \wedge r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \wedge \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \wedge r_2} \leftarrow b_{r_1} \wedge b_{r_2}
\end{array}$$

$$\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{D}}[v] & = \mathit{true} & d_v & \leftarrow \mathit{true} \\
\underset{\alpha}{\mathbb{D}}[x] & = \mathit{true} & d_x & \leftarrow \mathit{true} \\
\underset{\alpha}{\mathbb{D}}[y] & = \underset{\alpha}{\mathbb{B}}[\mathit{cond}(y)] & d_y & \leftarrow d_{\mathit{cond}(y)} \\
\underset{\alpha}{\mathbb{D}}[e_1 + e_2] & = \underset{\alpha}{\mathbb{D}}[e_1] \wedge \underset{\alpha}{\mathbb{D}}[e_2] & d_{e_1+e_2} & \leftarrow d_{e_1} \wedge d_{e_2} \\
\underset{\alpha}{\mathbb{D}}[e_1 - e_2] & = \underset{\alpha}{\mathbb{D}}[e_1] \wedge \underset{\alpha}{\mathbb{D}}[e_2] & d_{e_1-e_2} & \leftarrow d_{e_1} \wedge d_{e_2} \\
\underset{\alpha}{\mathbb{D}}[e_1 \times e_2] & = \underset{\alpha}{\mathbb{D}}[e_1] \wedge \underset{\alpha}{\mathbb{D}}[e_2] & d_{e_1 \times e_2} & \leftarrow d_{e_1} \wedge d_{e_2}
\end{array}$$

Fig. 8. The Evaluation of Constraints and their Corresponding Invariants.

$$\begin{array}{l|l}
\underset{\alpha}{\mathbb{E}}[c] = \delta(\underset{\alpha}{\mathbb{B}}[c]) & i_c \leftarrow \delta(b_c) \\
\underset{\alpha}{\mathbb{D}}[c] = \mathit{true} & d_c \leftarrow \mathit{true}
\end{array}$$

Fig. 9. The Evaluation of Reified Constraints and their Corresponding Invariants.

The Model An ATGP consists of four different components: the objective function to achieve diversity, a hard constraint system, a soft constraint system, and a probabilistic constraint system. The hard constraint system contains the architectural constraints, as well as the hard constraints in the test scenario. The soft constraint system contains the soft constraints of the test scenario. The probabilistic constraint system allows the validation engineer to bias the generated sequence toward some instructions. An entry in a probabilistic constraint system is a tuple $\langle (c_1, p_1), \ldots, (c_k, p_k) \rangle$ where c_i are mutually exclusive constraints and p_i are probabilities satisfying $\sum_{i=1}^{k} p_i = 1$. The intention is to generate a sequence of instructions which satisfy c_i with probability p_i. A typical example would be

$$(\mathit{opcode} = \mathit{add}, 0.7), (\mathit{opcode} = \mathit{jump}, 0.2), (\mathit{opcode} = \mathit{load}, 0.1)$$

which would generate *add*, *jump*, and *load* instructions 70%, 20%, and 10% of the time respectively.

The Search We experimented with various search procedures sharing a common core. The core has four main features. First, the hard and soft constraint systems H and S are combined into a single constraint system C through weights, i.e., $C = w_h * H + S$. The resulting constraint system is then reified into the objective function, once again using weights

$$O = w_d * \mathit{HammingDistance} - w_c * C$$

$$
\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{V}}[e_1 = e_2] & = \underset{\alpha}{\mathbb{E}}[abs(e_1 - e_2)] + \phi \underset{\alpha}{\mathbb{U}}[e_1] + \phi \underset{\alpha}{\mathbb{U}}[e_2] & v_{e_1 = e_2} & \leftarrow i_{abs(e_1 - e_2)} + \phi u_{e_1} + \phi u_{e_2} \\
\underset{\alpha}{\mathbb{V}}[e_1 \leq e_2] & = \underset{\alpha}{\mathbb{E}}[\max(e_1 - e_2, 0)] + \phi \underset{\alpha}{\mathbb{U}}[e_1] + \phi \underset{\alpha}{\mathbb{U}}[e_2] & v_{e_1 \leq e_2} & \leftarrow i_{max(e_1 - e_2, 0)} + \phi u_{e_1} + \phi u_{e_2} \\
\underset{\alpha}{\mathbb{V}}[c_1 \wedge c_2] & = \underset{\alpha}{\mathbb{V}}[c_1] + \mathbb{V}[c_2] & v_{c_1 \wedge c_2} & \leftarrow v_{c_1} + v_{c_2} \\
\underset{\alpha}{\mathbb{V}}[c_1 \vee c_2] & = \min(\underset{\alpha}{\mathbb{V}}[r_1], \underset{\alpha}{\mathbb{V}}[r_2]) & v_{c_1 \wedge c_2} & \leftarrow \min(v_{c_1}, v_{c_2})
\end{array}
$$

$$
\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{U}}[v] & = 0 & u_v & \leftarrow 0 \\
\underset{\alpha}{\mathbb{U}}[x] & = 0 & u_x & \leftarrow 0 \\
\underset{\alpha}{\mathbb{U}}[y] & = \delta(\neg \underset{\alpha}{\mathbb{B}}[cond(y)]) & u_y & \leftarrow \delta(\neg d_{cond(y)}) \\
\underset{\alpha}{\mathbb{U}}[e_1 + e_2] & = \underset{\alpha}{\mathbb{U}}[e_1] + \underset{\alpha}{\mathbb{U}}[e_2] & u_{e_1 + e_2} & \leftarrow u_{e_1} + u_{e_2} \\
\underset{\alpha}{\mathbb{U}}[e_1 - e_2] & = \underset{\alpha}{\mathbb{U}}[e_1] + \underset{\alpha}{\mathbb{U}}[e_2] & u_{e_1 - e_2} & \leftarrow u_{e_1} + u_{e_2} \\
\underset{\alpha}{\mathbb{U}}[e_1 \times e_2] & = \underset{\alpha}{\mathbb{U}}[e_1] + \underset{\alpha}{\mathbb{U}}[e_2] & u_{e_1 \times e_2} & \leftarrow u_{e_1} + u_{e_2} \\
\underset{\alpha}{\mathbb{U}}[c] & = 0 & u_c & \leftarrow 0
\end{array}
$$

Fig. 10. Violations of Constraints over Conditional Variables and their Invariants.

Second, the search always selects the variable with the steepest gradient and always assigns to it the value producing the steepest increase in objective O. Third, tabu-search is used as the meta-heuristic. Fourth, the search always includes a restarting strategy. The probabilistic constraint system is handled in an initial step. For each probabilistic constraint, the search flips a coin and imposes the appropriate constraint based on the provided distribution.

This core can be enhanced with a strategic oscillation strategy which adjusts the weights to balance the time spent in the feasible and infeasible region [9]. However, the core procedure achieves the best quality/efficiency tradeoff over the benchmarks presented in the next section. The strategic oscillation offers benefits in solution quality at the expense of an increase in computation times. Since efficiency is a critical factor in ATGPs, this strategy was not retained.

A critical aspect of the search procedure is also its handling of large domains. As mentioned earlier, domains in ATGPs can vary in size considerably. A domain may be small (e.g., the available registers) or very large (e.g., the set of all 32-bit addresses or the set of all 16-bit constants). It is not practical to find the value that decreases the objective the most by differentiation in these cases: It would take too much time to enumerate all the values. Our search handles large domains differently by performing a random sampling of the domain.

It is important to emphasize that ATGPs have many local minima and it is not easy to escape them. This is the main justification for restarts which are critical to achieve a reasonable tradeoff between solution quality and efficiency. This situation is partly due to our modeling which is geared toward feasibility: Violations of the conditions from conditional variables have a significant penalty (the ϕ value in the violation definition). However, without this penalty, the search has difficulty finding feasible solutions and is heavily biased toward the Hamming distance.

```
Test Scenario 2:
   Hard: OpTypeSp1 = imm32 && OpValue1 > 0x12345
   Soft: OpValue1 < 0x22222 || (OpValue1 > 0x33333 && OpValue1 < 0x77777)
Test Scenario 3:
   Prob: InstructionGroups = {arithm%70, logic%20, cmp%10}
Test Scenario 7:
   Hard: OpType1 = immediate || OpType2 = immediate
   Hard: if (OpType1 = immediate) then (OpValue1 > 0xff)
   Hard: if (OpType2 = immediate) then (OpSize2 > 8)
   Soft: OpRole0 = dest
Test Scenario 8:
   Hard: InstructionClass != GP && InstructionGroups = logic
   Soft: OpValue0 = OpValue1
```

Fig. 11. The ATG Test Scenarios

5 Experimental Results

This section presents the experimental results on some benchmarks provided by Intel. The benchmarks consist of an instruction set consisting of 80 instruction types with up to 20 fields and 8 test scenarios. For space reasons, it is not possible to present all the results but Figure 11 depicts some interesting scenarios. Scenario 2 features both hard and soft constraints, including some over large domains. Scenario 3 features a probabilistic constraint. Scenario 7 features hard and soft constraints with implications and disjunctions. Scenario 8 features a soft constraint which restricts the feasible region significantly and creates a significant tension with the Hamming distance. Incremental-CBLS is implemented on top of the Comet system [4] (Significant improvement in speed would result from a native support of conditional variables) and the experiments were run on Intel Xeon CPU 2.80GHz machines running 64-bit Linux Debian.

Figures 12 and 13 depict the experimental results on the above scenarios. Other results are consistent but cannot be included for space reasons. Once again, the graphs on the left give the average Hamming distance between two solutions as a function of the number of solutions and each point corresponds to an average of 10 runs. In general, it is difficult to compute a tight upper bound on ATGPs because of the conditional variables, some of which are not active. The figure reports an upper bound on scenario 8, since the set of feasible instructions is more restricted in this case and the upper bound can exploit that information (a similar result holds for scenario 5 whose results are not shown for space reasons). The results show a significant benefit in solution quality for Incremental-CBLS compared to Control-CBLS. Interestingly, the shape of the results closely follows the queen and allinterval results presented earlier. The curves for the average Hamming distance do not converge, Incremental-CBLS is close to the upper bound on scenarios 5 and 8, and Incremental-CBLS has significantly smaller standard deviations on ATGPs.

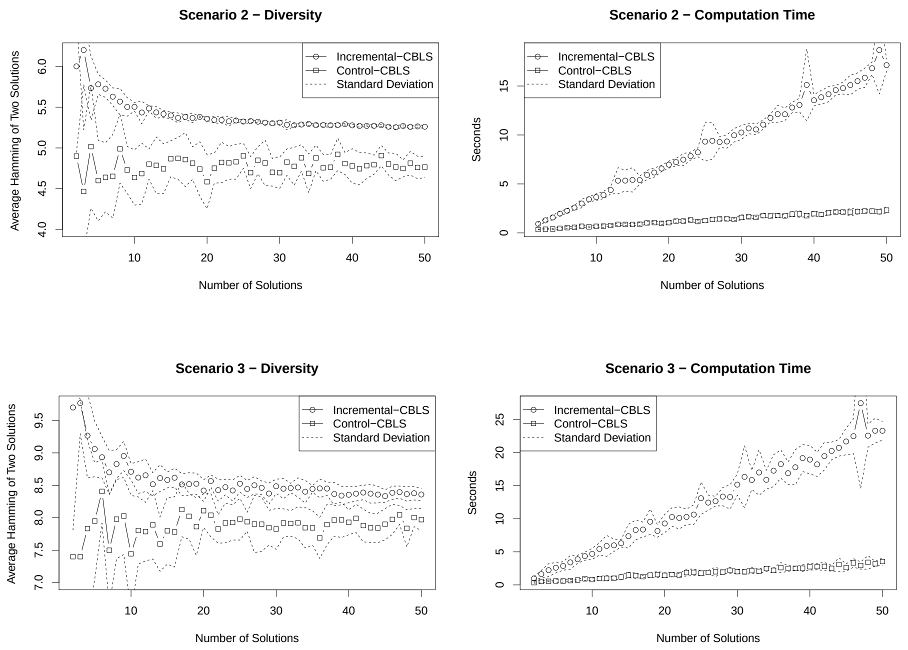

Fig. 12. Diversity Results and Computation Times for the ATGPs.

The computation times remain reasonable for Incremental-CBLS but obviously the computation times increase compared to Control-CBLS which only searches for random feasible solutions. Scenario 8 is more demanding, since the soft constraint is in direct contradiction with the Hamming distance and restricts the search space significantly when enforced. Only assignments were considered as local moves. An improvement could be gained from considering value swaps as well.

The tradeoffs between solution quality and efficiency was also investigated since both diversity and efficiency are important in ATGPs. Figure 14 shows the results on scenario 2. The number of restarts is reduced from 8 to 4 without significant degradation in quality but with a 50% reduction in computation time. This indicates that Incremental-CBLS can likely be tuned to meet strong timing constraints while still bringing significant benefits.

Finally, Figure 15 shows the benefits of parallelism in ATGPs problems. It transforms the search into a multistart procedure which are executed on 4 processors. The figure shows that the computation times are decreased by 50% again for 8 and 4 multistarts, closing further the gap with Control-CBLS. The generation of 50 diverse solutions now takes less than 6 seconds.

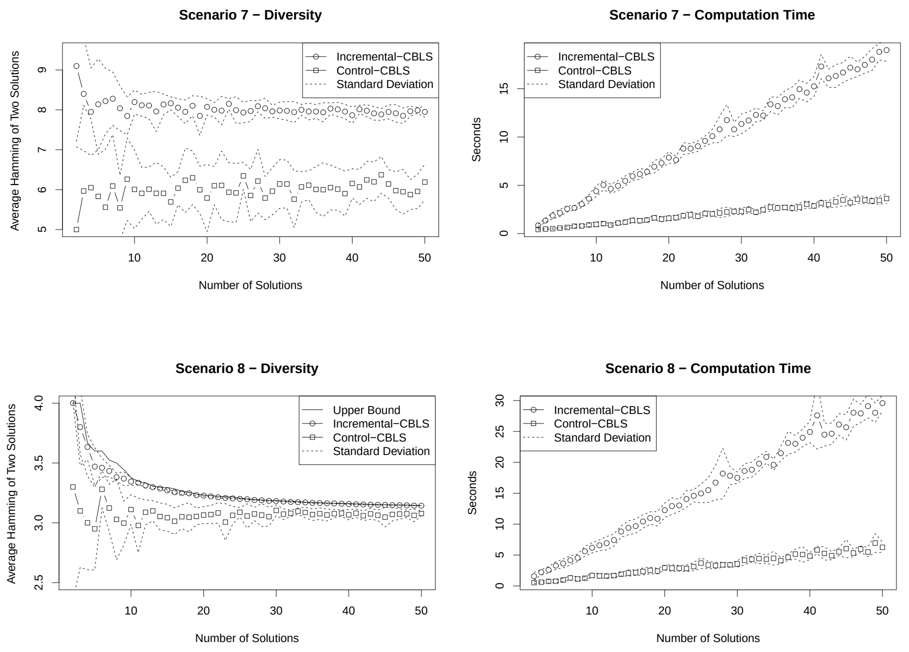

Fig. 13. Diversity Results and Computation Times for the ATGPs.

6 Conclusion

This paper reconsidered the automatic generation of architectural tests (ATGP), a fundamental problem in processor validation. It proposed to view ATGPs as MaxDiverse$k$Set problems to produce more diverse solutions than the random exploration traditionally used. The paper showed that constraint-based local search over conditional variables can provide significant benefits in solution quality, while retaining reasonable efficiency. The paper described a semantics and implementation of constraint-based local search over conditional variables, which is particularly appropriate for ATGPs. It also showed that constraint-based local search brings significant computational benefits over existing techniques as an implementation technique for approximating MaxDiverse$k$Set problems.

There are many directions for future work. The treatment of large domains should be enhanced and the tabu search should be complemented by constraint-based repair techniques to suggest moves that can efficiently reduce the violations of constraints involving large domains. Our prototype implementation should be embedded in the Comet kernel and tested on large scale instances modeling IA-32 and IA-64 processors.

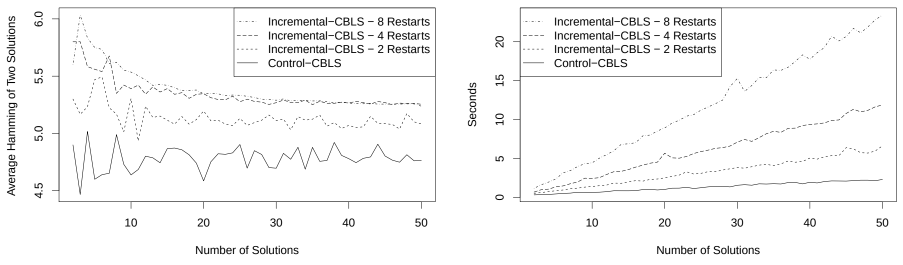

Fig. 14. Tradeoff Between Solution Quality and Efficiency on Scenario 2.

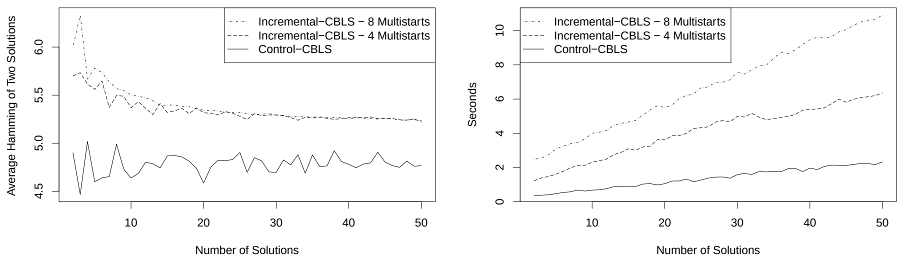

Fig. 15. The Benefits of Parallelism on Scenario 2.

References

1. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proceedings of the Eighth National Conference on Artificial Intelligence. (July 1990)
2. Moss, A.: Constraint patterns and search procedures for cp-based random test generation. In: Hardware and Software: Verification and Testing. (February 2008)
3. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In: Proceedings of the Twentieth National Conference on Artificial Intelligence. (May 2005)
4. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
5. Bin, E., Emek, R., Shurek, G., Ziv, A.: Using a constraint satisfaction formulation and solution techniques for random test program generation. In: IBM Systems Journal 41(3). (2002)
6. Geller, F., Veksler, M.: Assumption-based pruning in conditional csp. In: Principle and Practice of Constraint Programming (CP'05). (2005)
7. Sabin, D., Freuder, E.C.: Configuration as composite constraint satisfaction. In: Proceedings of Artificial Intelligence and Manufacturing Research Planning Workshop. (1996)
8. Van Hentenryck, P., Michel, L.: Differentiable invariants. In: 12th International Conference on Principles and Practice of Constraint Programming. (CP'06). Lecture Notes in Computer Science, Nantes, France (September 2006)
9. Glover, F., Laguna, M.: Tabu Search. Kluwer (1997)