# Using Constraints to Verify Properties of Rule Programs

Bruno Berstel[†][*]

[*]Software Engineering
Universität Freiburg
Freiburg, Germany
Email: berstel@informatik.uni-freiburg.de

Michel Leconte[†]

[†]IBM
Gentilly, France
Email: michel.leconte@fr.ibm.com
Email: bruno.berstel@fr.ibm.com

***Abstract*—Rule-based programming has been gaining interest in the industry for several years, through the growing use of Business Rules Management Systems. A demand for verification of semantic properties on rule programs has thus emerged. In this paper we present an approach to rule program verification, using constraints to model program executions and verification properties, and a Constraint-Based Programming Solver (CP Solver) to compute the answers to verification questions. We also study the use of constraint-based programming in rule program verification, and the consequences of this usage on the CP Solver compared to combinatorial optimization problems.**

***Keywords*-rule-based programming; program analysis; program verification; constraint-based programming; constraint satisfiability.**

## I. Introduction

Programs made of production (or condition-action) rules [1]–[5], which are at the basis of Business Rules Management Systems (BRMS) [6], [7], are gaining more and more interest in the industry, as a way to externalize the behaviours of applications, thus lowering the cost of frequent changes due for instance to regulation updates or to competitive pressure.

Program verification problems, that is, the question of whether a program satisfies a given property, can be modelled using constraints [8], [9], and expressed as implication problems [9], or as satisfiability or non-satisfiability problems. Using a Constraint-Based Programming Solver (CP Solver) to solve such problems [8], [10] has the advantage of being able to address a large class of formulas. This comes at the price of completeness, but practical experience shows that it is most of the time effective [11]. The solutions that are found correspond to answers (witnesses or counterexamples) to the program verification questions.

The purpose of this paper is to present an approach to the verification of rule programs, based on constraints and the use of a CP Solver. It also presents an extensive use of constraints, from modelling programs and properties to solving satisfiability problems to get verification results. In addition we study the challenges that rule program verification represents for a CP Solver. Section II gives an industrial motivation to the problem of rule program verification and introduces an example rule program that will be used throughout the paper.

*This work has been partially supported by the "CAVERN" ANR-07-SESUR-003 project.*

Unlike sequential programming languages, rule-based programming languages are declarative, meaning that the rules are enumerated independently from each other, and the program has no main entry. The semantics of a rule program is thus defined with an implicit rule execution procedure (the *rule engine*) in mind. Modelling the semantics of a rule program for verification thus requires that the implicit rule engine be explicited. Moreover, it so happens that several alternative rule engine semantics exist and are used in BRMSs. Section III presents a simple rule language, a general semantics for rule engines, and how they can be modelled using constraints.

The properties of interest to rule program verification include those of interest to general program verification, such as safety properties and termination. They also include more specific properties such as confluence of the rule program. Section IV reviews verification properties, and presents how they can be modelled as constraint problems.

Section V details the computation to perform to determine whether the example program is confluent or not. Then Section VI discusses the consequences that rule program verification has on a CP Solver, and gives directions to accomodate them.

## II. Motivation and Example

Business policies define how a company, small or large, should be run. Policies take various forms, from informal manuals, to methodologies, to standards. In some cases they can be automated, that is, implemented as software to be executed by a computer.

It is also often the case that companies have to implement changes in policy. These change may come from new regulations (such as laws addressing environmental concerns, or new accounting standards), or may be part of the strategy of the company, for instance to implement a new pricing policy.

When business policies are both automated and changing, their implementation as rules provides a decisive advantage in terms of agility. In particular, modern business rules management systems give the ability to the business, non-IT user to express rules in a user-friendly environment, using a natural-language-like rule language.

As a running example in this paper, we will consider a company running an e-commerce web site. Customers of this company have a registered profile on the site, which includes the customer's age and category (Silver, Gold, or Platinum). During a session on the site, a customer puts items in his/her shopping cart.

The pricing policy of the company, and hence the business rules implementing it, aim at computing a discount on each shopping cart, based on the customer profile and on the cart value. In our example we have three rules:

- The `gold-discount` rule implements a policy that increments the discount granted to Gold customers by 10 points, if their shopping cart is worth $2,000 or more.
- The `platinum-discount` rule implements a policy that increments the discount granted to Platinum customers by 15 points, if their shopping cart is worth $1,000 or more.
- The `upgrade` rule implements a policy that promotes Gold customers to the Platinum category, if they are aged 60 or more.

Using the rule language introduced in Section III the three rules informally exposed above would be implemented as shown in Fig. 1. In an industrial BRMS the business user would enter them in a structured natural language, which the system would then translate into a language close to the one used in Fig. 1, yielding a program ready for execution by a rule engine. Or for our purpose, ready for analysis by translation into constraints.

```
rule gold-discount:
 when:
   category == Gold,     // customer category
   value >= 2000         // shopping cart value
 then:
   discount := discount + 10; // cart discount
end

rule platinum-discount:
 when:
   category == Platinum,
   value >= 1000
 then:
   discount := discount + 15;
end

rule upgrade:
 when:
   category == Gold,
   age >= 60
 then:
   category := Platinum;
end
```

Fig. 1. The implementation of the three rules of the running example.

A number of properties can be checked on such a program; see Section IV for a survey of some of them. In particular, the reader may have noticed that there is an ambiguity between the upgrade and discount policies. If a Gold customer is eligible to both being granted the Gold discount and being upgraded to the Platinum category, then this customer may end up with either a 15 % or a 25 % discount, depending on whether the `upgrade` or the `gold-discount` is executed first.

Although rule engines are deterministic and will consistently compute the same discount for the same input data, such an ambiguity is a hazard for the business application and should be reported to the user. Section V details how the system will detect this ambiguity using the approach proposed in this paper.

Pragmatically, the issue would typically be fixed by setting priorities on the rules.

## III. From Rules to Constraints

In this section we define a simple rule language, together with the operational semantics of the programs written in this language. We then describe how we go from rules to constraints, with a word on the computation of execution traces.

### A. A Simple Rule Language

*1) Syntax:* The signature $\Sigma$ of symbols used to write our rule programs includes constant symbols for Booleans, integer and real numbers, as well as finite enumerations of symbolic values (such as the Silver/Gold/Platinum customer category). Il also includes variable symbols, as well as the arithmetic (including multiplication) and comparison operators, and the logical connectors $\wedge$, $\vee$ and $\neg$. Variable symbols are in finite number, their set is noted $\mathrm{Var}$.

A rule is made of a *guard* and an *action*. The guard of a rule is a formula built on constants and variable symbols using the operators and connectors. The action is a list of variable assignments. A rule program is an unordered collection of rules.

Although it supports the main features needed to illustrate our approach, the rule language considered here is purposedly simplistic. In particular, real BRMSs include the whole range of classical programming constructs in the actions of rules, from local variables to conditionals, to loops, etc. These are to be modeled using dedicated techniques (e.g. [10], [12], [13]).

The concrete syntax should be obvious from the example programs such as the one in Fig. 1. Of notice however, is the fact that conditions in a rule guard are implicitely connected by a conjunction.

*2) Semantics:* The domain $\mathcal{D}$ of the values handled by the rule programs include self-interpreted Booleans, numbers, and symbolic enumerations. The operators and connectors are interpreted in the classical way.

A program *state* is a valuation of the variables, that is, a function from $\mathrm{Var}$ to $\mathcal{D}$. A state allows us to interpret an expression, or a formula—that is, a rule guard or, as we shall see, a property to verify. We note $s(e)$ the value of an expression $e$ in a state $s$, resulting from the classical interpretation of operators applied on the valuation of variables provided by $s$. We say that a formula $\varphi$ holds in a state $s$, and we write $s \models \varphi$, if the valuation of variables provided by $s$ makes the formula valid.

In the context of rule-based programming, the current state is referred to as the *working memory*. When a rule guard holds in the current state, the rule is said to be *applicable* on the working memory.

A rule program $P$ is executed by a rule engine with the following semantics. An engine semantics has two parameters: a rule eligibility strategy $\mathcal{E}$, and a conflict set resolution strategy $\mathcal{C}$. Their roles are explained below.

1) Set the working memory $WM$ to the initial state.
2) Build the set $A = \{r = (g, a) \mid r \in \mathcal{E}(P) \wedge WM \models g\}$ of all applicable and eligible rules. This set is called the *agenda* of the rule engine.
3) If $A$ is empty, the execution ends.
4) Otherwise, choose a rule $r = \mathcal{C}(A)$ in the agenda.
5) Update the working memory by executing the action of $r$. That is, if the action of $r$ is the assignment $x := e$, replace the current state $WM$ with a new state $WM'$ such that

$$WM' : v \mapsto \begin{cases} WM(e) & \text{if } v \text{ is } x \\ WM(v) & \text{otherwise} \end{cases}$$

   If the rule action contains several assignments, execute them in sequence.
6) Go to step 2.

The purpose of the rule eligibility strategy $\mathcal{E}$ is to avoid trivial infinite loops caused by applying again and again the same rule. Indeed consider for instance rule `platinum-discount`: once this rule has been applied, its guard is still true; as a result, the rule could be applied indefinitely. The rule eligibility strategy defines what a "trivial" loop is, and avoids them by making some rules ineligible. Several eligibility strategies exist, the historical one being *refraction* [3]. In short, refraction forbids that the engine applies a rule a second time if the rule guard has not become false since the last time the rule was applied.

The purpose of the conflict set resolution strategy $\mathcal{C}$ is to pick the next rule to execute from the agenda. Again, several such strategies exist. Assigning a priority to each rule is a standard one. In practice, a rule engine always chooses the same rule in the same situation, even when the conflict set resolution strategy does not totally sort the rules (for instance, when several rules have the same priority). For this it resorts to secondary criteria with no business meaning, such as the name of the rule. In our program analyses, which are conducted from a business user's perspective, the rule engine semantics is modeled using only the primary, business-meaningful criteria of the conflict set resolution strategy.

*3) Program traces:* As described at step 5 of the semantics of a rule program, the execution of a rule causes a state transition. As a consequence, the execution of a rule program is made of a sequence of state transitions, characterized by the initial state and the sequence of executed rules.

A rule program execution may be finite or not, even in the presence of the rule eligibility strategy, which purpose is to avoid trivial loops and not all infinite executions. In the remainder of this paper, we shall consider rule programs with finite executions, which can be achieved for instance by using a rule eligibility strategy that allows at most one execution for each rule.

Given a rule program $P = \{R_1, \ldots, R_n\}$, we call $r_1 \circ \ldots \circ r_m$ an execution *trace* of $P$ if there exists at least one state from which an execution of $P$ can be triggerred, composed of the sequence of rules $r_1, \ldots, r_m$, with $r_j \in P$ for $j = 1, \ldots, m$.

An execution trace establishes a relation between states. The state pair $(s, s')$ is in the relation defined by the trace $r_1 \circ \ldots \circ r_m$ if the successive execution of $r_1$ from $s$, then of $r_2$ from the resulting state, etc., ends with the execution of $r_m$ leading to state $s'$. We note this $s \xrightarrow{r_1 \circ \ldots \circ r_m} s'$. A trace is called *valid* if it establishes a non-empty relation.

For a given initial state, a rule program may have several executions, if the conflict set resolution strategy used by the engine allows it. Even though these several executions will transition through different sequences of states, they may all end up in the same final state, or not. Computing the execution traces for a rule program depends on the strategies chosen for the engine; the authors have investigated trace computation algorithms for several strategies, yet they have no method generalizing across strategies.

### *B. State Constraints and Transition Constraints*

The *state constraints* (or simply *constraints*) we consider are first-order logic formulas over the signature $\Sigma$ defined in Section III-A. The translation of the guard of a rule is thus straightforward.

We also consider *transition constraints*, which are first-order logic formulas over the signature $\Sigma'$, defined as $\Sigma$ augmented with new variable symbols obtained by adding a prime to each variable symbol in Var. A transition constraint is thus a formula containing both primed and unprimed variables.

Each assignment in the action of a rule is translated into a transition constraint stating the equality between the primed version of the assigned variable and the expression being assigned to it. To these we must add the equalities between the primed and unprimed versions of all the other, non-assigned variables. The resulting conjunction forms the translation of the rule action into a transition constraint.

Given a rule $r$, let us note $g$ the state constraint resulting from the translation of its guard, and $a$ the transition constraint describing its action. The relation $\xrightarrow{r}$ defined by the trace consisting of only one execution of $r$ can be described by the transition constraint $\rho(r) \equiv g \wedge a$. In this constraint, $a$ binds the values of variables in the initial and final states, and $g$ restricts this binding to the initial states where $r$ is applicable.

*Example.* The guard of the `gold-discount` rule will be translated into $\mathit{category} = \text{Gold} \wedge \mathit{value} \geq 2000$, while the assignment in the action of this rule will be translated into $\mathit{discount}' = \mathit{discount} + 10$. As noted above, the complete translation of the action also includes identity assignments for the other variables. Thus, the complete translation of the action of `gold-discount` is $\mathit{discount}' = \mathit{discount} + 10 \wedge \mathit{age}' = \mathit{age} \wedge \mathit{category}' = \mathit{category} \wedge \mathit{value}' = \mathit{value}$.

The transition constraint $\rho(r_1 \circ r_2)$ for the relation $\xrightarrow{r_1 \circ r_2}$, defined by the trace consisting of one execution of $r_1$ followed by one execution of $r_2$, is built from the transition constraints $g_1 \wedge a_1$ for $\xrightarrow{r_1}$ and $g_2 \wedge a_2$ for $\xrightarrow{r_2}$ as follows. We first introduce a syntactic transformation: given a formula $\varphi$, we define the formula $\mathsf{primed}(\varphi)$ by replacing all variable symbols in $\varphi$ with the same variable symbols equiped with a prime (an additional one for variables already primed). The transition constraint $\rho(r_1 \circ r_2)$ is then $g_1 \wedge a_1 \wedge \mathsf{primed}(g_2) \wedge \mathsf{primed}(a_2)$, expressing that the action of $r_2$ is executed from those states resulting from the execution of $r_1$ where it is applicable.

*Example.* The transition constraint for the successive executions of the rules `upgrade` and `platinum-discount` is:

$$\begin{aligned}
&\mathit{category} = \text{Gold} \wedge \mathit{age} \geq 60 \\
&\quad \wedge \quad \mathit{category}' = \text{Platinum} \wedge \bigwedge_{v \text{ is not } \mathit{category}} v' = v \\
&\quad \wedge \quad \mathit{category}' = \text{Platinum} \wedge \mathit{value}' \geq 1000 \\
&\quad \wedge \quad \mathit{discount}'' = \mathit{discount}' + 15 \wedge \bigwedge_{v \text{ is not } \mathit{discount}} v'' = v'
\end{aligned}$$

which after simplification amounts to:

$$\begin{aligned}
&\mathit{category} = \text{Gold} \wedge \mathit{age} \geq 60 \wedge \mathit{value} \geq 1000 \\
&\quad \wedge \quad \mathit{category}' = \text{Platinum} \wedge \mathit{discount}' = \mathit{discount} + 15 \\
&\quad \wedge \quad \mathit{age}' = \mathit{age} \wedge \mathit{value}' = \mathit{value}
\end{aligned}$$

The primed variable elimination in the example above allowed us to rewrite the transition constraint as a constraint on unprimed and single-primed variables. For the sake of simplicity, we shall consider in the remainder of this paper that the transition constraints of traces, no matter how long, bind unprimed variables and variables with only one prime.

An execution trace $t$ is valid, that is, the relation $\xrightarrow{r}$ is non-empty, if and only if the transition constraint $\rho(t)$ is satisfiable.

Note that a formula $\rho$ over $\Sigma'$ is to be interpreted by a valuation $\langle s, s' \rangle$ based on a pair of states $s$ and $s'$. The valuation $\langle s, s' \rangle$ is defined as giving the same value as $s$ to unprimed variables, and to each primed variable the value that $s'$ gives to the corresponding unprimed variable. When a valuation $\langle s, s' \rangle$ makes a formula $\rho$ over $\Sigma'$ valid, we note $\langle s, s' \rangle \models \rho$.

As noted before, a rule program can have multiple execution traces, of unbounded lengths. These traces are described by the collection of the corresponding transition constraints. Computing all of them can be tricky and/or costly with complex rule eligibility and conflict set resolution strategies. However, it is sometimes sufficient in practice to use approximations of the set of all traces, to find bugs in rule programs. Such approximations include simplifying the rule eligibility strategy in a way that introduces no spurious executions, or considering bounded length executions [9]. An extreme, yet sometimes useful approximation is to consider only traces consisting in one rule execution.

## IV. Verification of Properties as Constraint Problems

As mentioned in the introduction, rule program verification includes verifying safety properties on the program executions, as well as termination and confluence of the rule program.

In this section we study how the verification of safety and confluence properties can be formulated as the satisfiability or unsatisfiability of a constraint problem.

To express these properties, state and transition constraints are used. It sometimes happens that a property has to hold only for executions starting from a given set of initial states. These initial states are specified by an assertion, which is usually noted $\mathsf{init}$.

### *A. Safety*

A safety property is an assertion $\mathsf{safe}$ defining a set of states, which represent desirable termination states for the program. A program is considered safe if for all states $s$ and $s'$ and all execution traces $\mathsf{trace}$ of the program such that $s \models \mathsf{init}$ and $s \xrightarrow{\mathsf{trace}} s'$, then $s' \models \mathsf{safe}$.

Verifying a safety property is done by refutation on all program traces. That is, all program traces are considered and for each one, it is tested whether the transition constraint

$$\mathsf{init} \wedge \rho(\mathsf{trace}) \wedge \neg\, \mathsf{primed}(\mathsf{safe}) \tag{1}$$

is satisfiable, that is, two states $s$ and $s'$ can be found such that $\langle s, s' \rangle$ makes (1) valid. If the constraint is unsatisfiable then the program is safe; otherwise solutions can be provided to the user as examples of unsafe executions.

*Example.* On the running example program, one may want to verify that customer cannot be downgraded, that is, their category cannot be changed from Platinum to Gold or Silver. This could be expressed by $\mathsf{init} \equiv \mathsf{safe} \equiv \mathit{category} = \text{Platinum}$.

### *B. Confluence*

A rule program is considered confluent when from any initial state, all program executions will lead to the same final state. This property is also referred to as consistency, or semantic interference-freedom [14]. A non-confluent rule program is the sign of either a design or an implementation error.

Again, the verification of a rule program confluence is done by refutation. A rule program is non-confluent if there exist two traces $\mathsf{trace}_1$ and $\mathsf{trace}_2$ for which an initial state can be found from which both traces are applicable and lead to distinct states. That is, there exist two traces and three states $s, s'_1, s'_2$ such that

$$s \models \mathsf{init} \text{ and } s \xrightarrow{\mathsf{trace}_1} s'_1 \text{ and } s \xrightarrow{\mathsf{trace}_2} s'_2 \text{ and } \neg(s'_1 \simeq s'_2)$$

where $s_1 \simeq s_2$ means that for any variable symbol $v$, $s_1(v) = s_2(v)$.

Let us define, for any trace $t$, the transition constraints $\rho_1(t)$ and $\rho_2(t)$ by replacing in the transition constraint $\rho(t)$ all the primed variable symbols with the same variable symbols indexed by 1 (resp. 2). The non-confluence of a rule program can then be assessed by testing, for each pair $(\mathsf{trace}_1, \mathsf{trace}_2)$ of traces, whether the transition constraint (2) is satisfiable

$$\mathsf{init} \wedge \rho_1(\mathsf{trace}_1) \wedge \rho_2(\mathsf{trace}_2) \wedge \neg\,\mathsf{samefinal} \qquad (2)$$

with samefinal defined as the pairwise equality of all the primed variables in their indexed-by-1 and indexed-by-2 versions.

Thus, the rule program is confluent if for each pair of execution traces of the rule program, no states $s, s'_1, s'_2$ can be found that make (2) valid, where $s$ gives values to unprimed variables, and $s'_1$ and $s'_2$ give values to primed variables indexed by 1 and 2, respectively. If on the other hand such states can be found for some pair of traces, the initial state $s$ and the traces provide two conflicting executions demonstrating the non-confluence of the rule program.

## V. Non-confluence of the running example

In this section we describe the computations to follow in order to verify the confluence of the running example rule program. As just seen, the process consist in considering all pairs $(\mathsf{trace}_1, \mathsf{trace}_2)$ of execution traces for this program, and test whether the constraint (2) is satisfiable.

Let us assume no particular knowledge on the initial state, that is, $\mathsf{init} \equiv \mathsf{true}$. Also, in our case

$$\begin{aligned}\mathsf{samefinal} \quad \equiv \quad & \mathit{age}'_1 = \mathit{age}'_2 \wedge \mathit{category}'_1 = \mathit{category}'_2 \\ \wedge \quad & \mathit{value}'_1 = \mathit{value}'_2 \wedge \mathit{discount}'_1 = \mathit{discount}'_2\end{aligned}$$

The valid execution traces of the running example rule program can be enumerated as follows:

- `gold-discount`
- `platinum-discount`
- `upgrade`
- `upgrade` ∘ `platinum-discount`
- `gold-discount` ∘ `upgrade` ∘ `platinum-discount`

Among these traces, the only ones whose transition constraints are pairwise compatible (the conjunction of the guards of the first rules in other pairs of traces is unsatisfiable) are

$\mathsf{trace}_1 \equiv$ `upgrade` ∘ `platinum-discount`
$\mathsf{trace}_2 \equiv$ `gold-discount` ∘ `upgrade` ∘ `platinum-discount`

The transition constraints for these traces are computed as shown in the last example of Section III-B. After simplifications, the non-confluence constraint (2) expresses as follows for the example program:

$$\begin{aligned}&\mathit{category} = \text{Gold} \wedge \mathit{age} \geq 60 \wedge \mathit{value} \geq 2000 \\ \wedge \quad & \mathit{category}'_1 = \text{Platinum} \wedge \mathit{discount}'_1 = \mathit{discount} + 15 \\ \wedge \quad & \mathit{category}'_2 = \text{Platinum} \wedge \mathit{discount}'_2 = \mathit{discount} + 25 \\ \wedge \quad & \mathit{age}'_1 = \mathit{age}'_2 = \mathit{age} \wedge \mathit{value}'_1 = \mathit{value}'_2 = \mathit{value} \\ \wedge \quad & (\mathit{category}'_1 \neq \mathit{category}'_2 \vee \mathit{discount}'_1 \neq \mathit{discount}'_2)\end{aligned}$$

This constraint satisfiability problem is then submitted to the constraint solver, which will answer "the constraint has solutions". This indicates that the rule program is non-confluent. The constraint solver is then asked to provide a solution. Depending on the labelling strategy, the solution may for instance be:

$$\begin{array}{r|r|r} \mathit{age} = 60 & \mathit{age}'_1 = 60 & \mathit{age}'_2 = 60 \\ \mathit{category} = \text{Gold} & \mathit{category}'_1 = \text{Plat.} & \mathit{category}'_2 = \text{Plat.} \\ \mathit{value} = 2000 & \mathit{value}'_1 = 2000 & \mathit{value}'_2 = 2000 \\ \mathit{discount} = -128 & \mathit{discount}'_1 = -113 & \mathit{discount}'_2 = -103 \end{array}$$

The business rules management system then presents the solution to the user in an error message such as: *The rule program is ambiguous. For instance, starting with a customer age equal to 60, a customer category equal to Gold, and a cart value equal to $2,000, the execution of rules* `upgrade` *and* `platinum-discount` *will not yield the same result as the execution of rules* `gold-discount`, `upgrade`, *and* `platinum-discount`.

## VI. Consequences for the CP Solver

The constraint problems that derive from program verification questions carry specificities that challenge the efficiency of a "plain" CP Solver. The challenge essentially results from the combination of *slow convergence* of propagation and *large domains* of variables.

The large domains phenomenon relates to the fact that input variables are commonly ranging over all the possible values that can be represented in the computer. For integer variables, there are thus typically $2^{32}$ or $2^{64}$ such values. This comes in contrast with using a CP Solver to solve combinatorial problems, where the domains of the variables is usually not so unreasonably large.

Slow convergence during propagation (also known as *slow propagation*) appears when the time to reach the fix point of domain reduction is proportional to the size of the domains of the variables. This phenomenon is never desirable, but has a major impact when variables have extremely large domains.

As noted in [15], the slow propagation phenomenon is somehow unavoidable in general. In some constraint problems (such as the linear ones of Examples 2 and 3), if there is a solution, then there is one inside some bounds which can be computed [16]. This is not always true, as Example 1 demonstrates.

*Example 1.* Consider the constraints $x > ux$ for integer variables $x$ and $u$. Suppose also that $x$ can take any 32-bit values and that $u \in \{0, 1\}$. Here $x$ and $u$ satisfy the constraint if and only if $u$ is zero and $x$ is positive.

The CP Solver implements the usual interval reduction to propagate constraints. Here, the propagation will remove one by one all negative or null values from the domain of $x$. The fix point is $x \in [1, 2^{31}]$ and $u \in \{0, 1\}$. Reaching this fix point requires $2^{31}$ steps.

Note that, as the propagation is incomplete on this example, the value 1 is still in the domain of $u$ when the fix point is reached. If at some point $u$ is instanciated to 1, then the constraint $x > ux$ becomes unsatisfiable. To prove this, propagation will reduce the domain of $x$ by removing only the minimum and the maximum values from its domain at each step, taking $2^{30}$ additional steps.

Special procedures have however been developed to address some cases of slow propagation.

*Example 2.* Consider the constraints $x < y$ and $y < x$. In this example each constraint propagation step removes only one value from the domains of variables.

The procedure described in [17] allows a CP Solver to prevent the case of slow propagation illustrated in Example 2. This procedure detects the infeasibility of such constraints by looking for cycles in a graph built from the constraint problem.

*Example 3.* Consider the constraint $2x + 2y = 1$. Here also the domain reduction removes only the bounds of the domains at each constraint propagation.

The kind of slow propagation illustrated in Example 3 appears with integer linear constraints, and can be addressed as proposed in [18] by enriching the domains of variables with a congruence domain. On the constraint from Example 3 the CP Solver will recognize that the variables should be both odd and even, and conclude to the infeasibility of the constraints.

It must be noted that slow propagation is not always the sign of an unsatisfiable constraint problem, as Example 1 demonstrates. On the other hand, an unsatisfiable problem involves more propagation cycles, thus raising the impact of slow propagation. And since, as exposed in Section IV, properties are frequently proven by refutation, the verification of a bug-free program will produce an unsatisfiable constraint problem. As a result, verifying a bug-free program (and the major portions of programs are bug-free, after all) is very sensitive to slow propagation.

We use a home-made finite domain CP Solver, which is an Object-Oriented Library in Java along the lines of [19].

To address the challenges linked with slow propagation and large domains in our CP Solver, we use the congruence domains [18] together with usual interval propagation. The BRMS user can also specify bounds on variables, indicating for example that the age of a customer is between 0 and 100. Such a specification is however totally optional.

In addition, we adopt the pragmatic approach of stopping the reduction of a domain after a given number of steps in the same cycle of propagation. For instance, in Example 2 we may stop the propagation before the fixpoint is reached, that is, before the domains of $x$ and $y$ become empty. These domains would then look like $[-a, a]$ for some constant $a$. However we refrain from concluding from such an interruption that the problem is unsatisfiable, as justified by Example 1.

## VII. Conclusion

In this paper we have described a constraint-based approach to the verification of rule programs, and we have studied the consequences of rule program verification specificities for a CP Solver.

To this end, we have first presented a simple rule language and explicited the semantics of rule engines. We have then described how to translate rule programs and verification properties into constraint satisfiability problems, and detailed the computation by which a verification system proves an example program to be non-confluent.

Then we have studied some specific challenges that verification of rule programs represents for a CP Solver. These challenges come from the fact that the domains of the input variables are commonly very large, combined with the (generally unavoidable) phenomenon of slow convergence in propagation. We have presented the approach used in our CP Solver to adress these challenges.

## References

[1] H. A. S. Allen Newell, *Human problem solving*. Englewood Cliffs, NJ, USA: Prentice Hall, 1972.

[2] R. Davis, B. G. Buchanan, and E. H. Shortliffe, "Production rules as a representation for a knowledge-based consultation program." *Artif. Intell.*, vol. 8, no. 1, pp. 15–45, 1977.

[3] C. Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem." *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[4] E. Baralis and J. Widom, "An algebraic approach to static analysis of active database rules." *ACM Trans. Database Syst.*, vol. 25, no. 3, pp. 269–332, 2000.

[5] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Patranjan, "Reactive rules on the web," in *Reasoning Web*, ser. LNCS, G. Antoniou, U. Aßmann, C. Baroglio, S. Decker, N. Henze, P.-L. Patranjan, and R. Tolksdorf, Eds., vol. 4636. Springer, 2007, pp. 183–239.

[6] ILOG JRules, ILOG, 2006, `http://www.ilog.com`.

[7] Drools, JBoss, 2006, `http://www.drools.org`.

[8] A. Gotlieb, "Euclide: A constraint-based testing framework for critical C programs," in *ICST*. IEEE Computer Society, 2009, pp. 151–160.

[9] H. Collavizza, M. Rueher, and P. V. Hentenryck, "CPBPV: A constraint-programming framework for bounded program verification," in *CP*, ser. LNCS, P. J. Stuckey, Ed., vol. 5202. Springer, 2008, pp. 327–341.

[10] H. Collavizza and M. Rueher, "Exploring different constraint-based modelings for program verification," in *CP*, ser. LNCS, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 49–63.

[11] ——, "Exploration of the capabilities of constraint programming for software verification." in *TACAS*, ser. LNCS, H. Hermanns and J. Palsberg, Eds., vol. 3920. Springer, 2006, pp. 182–196.

[12] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1684–1698, 1994.

[13] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 181–210, 1991.

[14] D. Shao, S. Khurshid, and D. E. Perry, "SCA: a semantic conflict analyzer for parallel changes," in *ESEC/SIGSOFT FSE*, H. van Vliet and V. Issarny, Eds. ACM, 2009, pp. 291–292.

[15] L. Bordeaux, Y. Hamadi, and M. Y. Vardi, "An analysis of slow convergence in interval propagation," in *CP*, ser. LNCS, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 790–797.

[16] S. A. Seshia and R. E. Bryant, "Deciding quantifier-free presburger formulas using parameterized solution bounds," *CoRR*, vol. abs/cs/0508044, 2005.

[17] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap, "Beyond finite domains," in *PPCP*, ser. LNCS, A. Borning, Ed., vol. 874. Springer, 1994, pp. 86–94.

[18] M. Leconte and B. Berstel, "Extending a CP solver with congruences as domains for program verification," in *CSTVA '06*, B. Blanc, A. Gotlieb, and C. Michel, Eds. IEEE Computer Society Press, 2006, pp. 22–33.

[19] J.-F. Puget and M. Leconte, "Beyond the glass box: Constraints as objects," in *ILPS*, 1995, pp. 513–527.

[20] C. Bessiere, Ed., *Principles and Practice of Constraint Programming - CP 2007, Providence, RI, USA, Proceedings*, ser. LNCS, vol. 4741. Springer, 2007.