A Dynamic Logic for Configuration

Ching Hoo Tang and Christoph Weidenbach

Max Planck Institute for Informatics, Saarbrücken, Germany
{chtang, weidenbach}@mpi-inf.mpg.de

Abstract

We define the new dynamic logic PIDL+ that extends our previously developed logic PIDL (Propositional Interactive Dynamic Logic) with arithmetic constraints. The language of PIDL+ is motivated by real world configuration systems, in particular, configuration systems for power plants. A PIDL+ specification consists of the description of an initial state, global constraints, and actions. Its semantics are the possible worlds starting from the initial state, spanned by the actions and restricted by the constraints. It distinguishes user actions from rule actions. Any user action is followed by a unique fixed point, called rule-terminal state, generated through exhaustive application of rule actions from the specification. The built in rule action fixpoint semantics and arithmetic constraints distinguish PIDL+ from known dynamic or action logics. Correctness of a PIDL+ specification as well as reachability of a particular state are decidable. We provide sound and complete algorithms.

1 Introduction

The dynamic logic PIDL+ is motivated by real-world configuration systems, e.g., configuration systems for steel or power plants as they are productive at Siemens [8]. A run of such a configuration system is a dialog between its user picking components or setting parameters inside certain bounds and the system reacting by adjusting the current configuration such that eventually a buildable product is the result. Today's implementations of such configuration systems are not based on formal specifications but are typically built on top of general-purpose programming languages. Hence, overall soundness or completeness of the configuration system cannot be automatically shown.

PIDL+ copes with the needed expressiveness of real-world configuration systems on one side and on the other side enables automatic verification of relevant system properties. The above form of dialog between the system and the user needs to distinguish variables controlled by the user and variables controlled by the system. Soundness is defined by one quantifier alternation: for all inputs to variables controlled by the user, the system can react by adjusting variables under its control such that eventually a buildable product is the result. If for any final product there is a sequence of user inputs reaching a state representing this product, the configuration system is complete.

From the user perspective, the system should terminate after each input and always produce the same result on the same inputs. In PIDL+ the user and the system are modeled by rules, called *transitions* (Definition 1). A user or system transition comes in a "precondition $\rightsquigarrow$ update" form: if the precondition of a transition is implied by the current state, its update is applied to the current state and yields the next state. After the application of a user transition, the system transitions, later called rules, are exhaustively applied until a unique state is reached that is a fixpoint with respect to the system transitions. We call such a state a rule-terminal state. The system is sound if whatever the user does inside the bounds specified eventually a rule-terminal state representing a product can be reached. Overall requirements to products are represented by constraint formulas in PIDL+. They have to be satisfied by any state and, in particular, by states representing a final product (Definition 17).

In real-world applications it is necessary that the set of variables controlled by the user dynamically changes. For example, if the user picks a certain component but subsequent user choices require an update to the component by the system, the user decision is overwritten and hence the component is no longer under the control of the user. In PIDL+ this is represented by a set of user controlled variables attached to each state (Definition 4). Eventually a state is represented by a set of literals and a set of user controlled variables.

Components are presented in PIDL+ by propositional variables and parameters by integer variables. For an admissible PIDL+ specification (Definition 3) all integer variables are finitely bounded. Therefore, the number of all reachable states is finite and properties of the specification become decidable. A priori bounded integer variables are typical for a real-world configuration system, e.g., out of a component box no car company is able to build an engine with arbitrarily high throughput. Still, result variables that are needed to represent the eventual product may be of type real in PIDL+ and computed by complex expressions. In order to preserve decidability, they must not occur in any transition (Definition 3).

Semantically, a system state is a possible world where all propositional and integer variables have fixed values. Then a user or system transition leads to the next world. We call this semantics the *small step* semantics. It is easy to understand but not well-suited for computing properties as the number of possible worlds becomes the product over the combination of all variable ranges. In this paper we present a *big step* semantics where a state of the big step semantics represents up to exponentially many states of the small step semantics. In any state the integer variables are bounded, but do not need to have a fixed single value. The semantics is existential for user and universal for all other variables: the precondition of a transition is satisfied in a state, if for some instance of the user variables satisfying the bounds of a state and all instances of all other variables satisfying the bounds of a state the precondition is implied. The bounds either come from the initial state or are updated by transition applications. With respect to the above semantics, a transition depending on a user variable might only be applicable to some of the potential values of the variable, but not to all. This makes it necessary to partition the values of variables in states with respect to a transition in order to compute the next state of the big step semantics. We call such partitions meeting the precondition of a transition with respect to a state a *selection* (Definition 5). Finite selections always exist because the applicability of transitions only depends on bounded integer variables.

PIDL+ is the extension of PIDL [9] with arithmetic constraints. Similar approaches that deal with logics formalizing change naturally arise from modal logics, originally introduced to describe the behavior of programs, prominently represented by Propositional Dynamic Logic [10] and its variants [2]. Boolean games [12] and variants thereof [17] are another mechanism to model a changing system, in this case by considering situations as they appear in game theory [14], where players play games by setting assignments of Boolean variables. These approaches are different from PIDL+ insofar that they do not have the inherent features to describe the characteristic user-rule relationship with rule-terminal states of the configuration system as mentioned earlier, or that they, in the case of Boolean games, focus on equilibria depending on certain strategies, as opposed to an exhaustive unfolding of user and rule actions in PIDL+. Also, our logic supports arithmetic constraints through selections. The same holds for approaches based on temporal logic [15, 4] and model checking [5], where solutions, for example, deal with the verification of configuration product variants [6] modeled as feature models [13], whose logics have been studied extensively [7]. Further work that explicitly centers on the analysis of configuration systems usually provides stage-wise verification, that is, it aims to analyze the configuration after each user input and guide the user through the configuration process, often through constraint satisfaction [1, 16, 3].

In Section 2 we introduce the above described syntax and semantics of PIDL+ in full detail. The section ends with an example specification illustrating the features of the logic (Example 20). Section 3 presents algorithms checking soundness and completeness and we end with a short conclusion, Section 4.

2 The Logic PIDL+

We give a detailed description of our logic PIDL+ in this section. Its design is geared towards modeling the dynamics of an interactive configuration system with the help of a states-and-transitions semantics. The states describes the possible worlds reachable from the initial state via defined sets of user and rule transitions. The base logic underlying the formulas used to express single states and transitions is propositional logic in combination with the theory of arithmetic over the reals. Relevant concepts connected with the base logic and general notions essential for the design of PIDL+ are given in the preliminaries subsection, which is followed by the definition of the actual syntax of PIDL+. Its semantics of states and transitions is contained in the final part of this section.

2.1 Preliminaries

The basic units of PIDL+ are terms and formulas derived from the fragment of first-order logic that is composed of propositional variables and the theory of reals. We provide the central notions needed in this paper.

We fix the *order-sorted signature* $\Sigma := (\{\mathcal{R}, \mathcal{Z}\}, \mathbb{R} \cup \{+, -, \cdot, <, \leq, >, \geq, \approx, \not\approx\})$. It has *sorts* $\mathcal{R}$ and $\mathcal{Z}$, where $\mathcal{Z}$ is a *subsort* of $\mathcal{R}$, $\mathcal{Z} \subset \mathcal{R}$, *constants* $\mathbb{R}$, where the sort of each $c \in \mathbb{R}$ is $\mathcal{R}$, written as $sort(c) = \mathcal{R}$, *function symbols* $+, -$, and $\cdot$, where it holds that $sort(f) = \mathcal{R} \times \mathcal{R} \to \mathcal{R}, f \in \{+, -, \cdot\}$, and *predicate symbols* $<, \leq, >, \geq, \approx$, and $\not\approx$, where it holds that $sort(\circ) = \mathcal{R} \times \mathcal{R}, \circ \in \{<, \leq, >, \geq, \approx, \not\approx\}$.

We say that a set of variables X is Σ-sorted if the sort of each variable $x \in X$ is one of the sorts specified in Σ: $sort(x) \in \{\mathcal{R}, \mathcal{Z}\}$. We choose to have two sorts $\mathcal{R}$ and $\mathcal{Z}$, to be interpreted as the sets $\mathbb{R}$ and $\mathbb{Z}$, because, as is delineated later in the section, we want to interpret numerical expressions that have an effect on the action dynamics as bounded integers, which is a crucial requirement for achieving decidability of the logic. Additionally, we allow those expressions which do not affect the action flow to be over the reals. They represent results that are purely restricted to the single worlds, holding information values of the configurations.

A *term* t over the signature Σ and a Σ-sorted variable X is called Σ-term and is defined by the usual rules known from first-order logic using variables, constants and function symbols. $T_\Sigma(X)$ denotes the set of all terms over the signature Σ and the variable set X. We use the customary infix notation when writing Σ-terms. For example, we write $x + 9$ instead of $+(x, 9)$.

An *atom* over Σ is an expression of the form $t \circ t'$, a *simple atom* over Σ is an atom of the form $x \circ t$, and a *simple bound* over Σ is a simple atom of the form $x \circ c$, where $x \in X, t, t' \in T_\Sigma(X)$, $c \in \mathbb{Z}$ and $\circ \in \{<, \leq, >, \geq, \approx, \not\approx\}$.

A *formula* over Σ, a Σ-sorted variable set X and a set of propositional variables Π, also called Σ*-formula*, is a first-order formula constructed in the usual way using the terms from $T_\Sigma(X)$, the variables from Π, the atoms over Σ, the usual Boolean connectives $\neg, \wedge, \vee, \to$ and $\leftrightarrow$, and the quantifier symbols $\forall$ and $\exists$. We write $F_\Sigma(X, \Pi)$ to denote the set of all formulas over Σ, X and Π.

Let $var(F)$ denote the set of variables that occur in a formula F. Analogously, $var(N)$ denotes the set of variables that occurs in a set N of formulas. Also, $varl(x \circ t)$ denotes the variable that is the left operand of the atom $x \circ t$, that is, $\mathit{varl}(x \circ t) = x$. If N is a set containing atoms over Σ, then $\mathit{varl}(N)$ denotes the set of variables that are the left operands of the atoms in N: $\mathit{varl}(N) = \{x | x \circ t \in N\}$. We use *varl* because the relevant terms in the semantics take the form of simple atoms $x \circ t$ in which the relevant variables are always on the left side of the operation by design. Likewise, we define the *intersection* $N|_M$ of a set N of formulas with a set M of variables as the set $N|_M := \{F \in N | \mathit{varl}(F) \cap \mathit{varl}(M) \neq \emptyset\}$. For example, $\{x \geq w + 4, y \approx 23, z < y, C\}|_{\{x,z\}} = \{x \geq w + 4, z < y\}$.

The objects of the signature Σ are interpreted in the usual way as done in the theory of reals: The sorts $\mathcal{R}$ and $\mathcal{Z}$ are interpreted as $\mathbb{R}$ and $\mathbb{Z}$, respectively, the symbols are interpreted as the usual operations and comparisons in $\mathbb{R}$. We use I_Σ, called Σ-interpretation, to denote both the valuation of a term from $T_\Sigma(X)$ and the truth value of a formula from $F_\Sigma(X, \Pi)$. Again, valuations and truth values are defined as one expects from the first-order logic theory of reals.

We say that a Σ-interpretation *satisfies* a Σ-formula F, written as $I_\Sigma \models F$, if $I_\Sigma(F) = 1$. A formula F is *satisfiable* if there is a Σ-interpretation I_Σ with $I_\Sigma \models F$, and F is *valid* if $I_\Sigma \models F$ for all Σ-interpretations I_Σ. Formula F *entails* formula F', written as $F \models F'$, if the following holds: If I_Σ is a Σ-interpretation and satisfies F, then I_Σ also satisfies F'. A set of N of Σ-formulas *entails* a Σ-formula F', written as $N \models F'$, if the following holds: If I_Σ is a Σ-interpretation and satisfies each $F \in N$, then I_Σ also satisfies F'.

We consider tuples of integer intervals, which appear in the semantics of PIDL+. As expected, intersection of a tuple $t = (I_1, \ldots, I_n)$ with another $t' = (I'_1, \ldots, I'_n)$ is defined componentwise: $t \cap t' := (I_1 \cap I'_1, \ldots, I_n \cap I'_n)$. An intersection is empty, written $t \cap t' = \emptyset$, if $I_i \cap I'_i = \emptyset$ for some i. For an interval $I = [v_1, v_2]$, the *atomic representation* $\mathit{at}(I, x)$ of I with respect to a variable x is the set of arithmetic atoms $\mathit{at}(I, x) := \{x \geq v_1, x \leq v_2\}$.

2.2 Syntax

We now define the syntax of PIDL+. It takes the form of a specification tuple whose components comprehensively describe a configuration system including its variables, rules and possible user actions.

Definition 1. A PIDL+ *specification* is a tuple $\mathfrak{S}_+ = (\Pi, X, S_I, U_I, \mathsf{C}, T_U, T_R)$, where the components are as follows: Π is a finite set of propositional variables, X is a finite set of Σ-sorted variables, (S_I, U_I) is called the *initial state*, C is a finite set of Σ-formulas in $F_\Sigma(X, \Pi)$, called the *constraints*, and T_U and T_R are finite sets of *user* and *rule transitions* $\Lambda \wedge F \rightsquigarrow E$, respectively, where (1) S_I is a finite set of simple bounds $x \circ c$ in $F_\Sigma(X, \Pi)$ and propositional literals over Π with

$$\forall \vec{x}_u \exists \vec{y}\; S_I|_{U_I} \rightarrow (S_I \setminus S_I|_{U_I}) \cup \mathsf{C}$$

being satisfiable, $\vec{x}_u = x_1 \ldots x_k$ being all the variables in U_I and $\vec{y} = y_1 \ldots y_l$ being all the variables in $X \setminus U_I$, (2) $U_I \subseteq X$, called the set of *initial user variables*, (3) Λ is a conjunction of simple atoms $x \circ t$ over Σ, called the *arithmetic condition*, (4) F is a conjunction of literals over Π, called the *propositional condition*, (5) E is called the *update set* and is a satisfiable set of simple bounds $x \circ c$ over Σ and propositional literals over Π for user transitions and is a satisfiable set of simple atoms $x \circ t$ over Σ and propositional literals over Π for rule transitions.

The set Π provides the propositional variables needed to describe all the Boolean statements about the worlds occurring in a configuration process, such as "part A is active", whereas the set X represents the numerical variables of the configuration, such as the weight of a component in kg. The initial state describes the starting world of the modeled system. The set S_I contains simple bounds and propositional literals that represent the initial configuration state, together with U_I which indicates which of the system's variables are set by the user. The models of the specification, as explained in the next subsection, consider further states that arise from this initial state. The requirement that $\forall \vec{x}_u \exists \vec{y}\ S_I|_{U_I} \rightarrow (S_I \setminus S_I|_{U_I}) \cup \mathsf{C}$ must be satisfiable basically says that the world is consistent: The simple bounds $x \circ c$ in $S_I|_{U_I}$ represents the range of user input with respect to that state, and they are consistent with the other simple bounds and the formulas in C, the constraints. This consistency requirement is used for all the other states induced by the system, as can be seen in the semantics section. The general and domain-specific constraints of the configuration system are encoded in C. For example, it typically contains formulas that express "option A and option B cannot be simultaneously active". Finally, we have two different sets of of transitions, namely user transitions T_U and rule transitions T_R, whose elements are constructs $\Lambda \wedge F \rightsquigarrow E$. The conjunction $\Lambda \wedge F$ form the condition part of a transition and express what must be met in the current state in order to have a transition from it to another state. There is a clear separation between the arithmetic conditions Λ and the propositional conditions F. The set E determines how the next state is defined by updating the current state with the atoms and literals in E if a transition is indeed possible. It expresses user actions in the case of T_U and rule actions in the case of T_R. User actions can set simple bounds while rules can update with simple atoms, which is a natural representation of a configuration system.

Definition 1 defines specifications in a broad sense. We work with a restricted class of specifications that have certain properties that, in particular, enables us to achieve finiteness of the system and thus decidability. First, we identify the set of all variables involved in transitions, which plays an important role in this.

Definition 2. The *transition variables* X_T of a specification $\mathfrak{S}_+$ are the set
$$X_T := \{x | x \in \mathit{var}(\Lambda \wedge F \rightsquigarrow E), \Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)\}.$$

Now the desired kind of specification, which we call *admissible specifications*, is defined as follows:

Definition 3. A specification $\mathfrak{S}_+$ is *admissible* if the following holds. (1) For each $x \in X_T$, $\mathit{sort}(x) = \mathcal{Z}$, (2) the constraints set C has the form
$$\left(\bigwedge_{x \in X_T} x \geq c_{x1} \wedge x \leq c_{x2}\right) \wedge \left(\bigwedge_{x \in X_r} x \approx t_x\right) \wedge \Phi,$$
where $c_{x1}, c_{x2} \in \mathbb{Z}$ for all $x \in X_T$, $X_r \subseteq X \setminus X_T$, $t_x \in T_\Sigma(X_T)$ for all $x \in X_r$, and Φ is a propositional formula over Π, and (3) for each $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$, it holds that if $x \circ t \in E$, then $x \notin \mathit{var}(t)$.

The composition of the constraints C in admissible specifications is fixed. There, variables occurring in transitions are set to be over the integers and bounded by integer constants. They do not appear anywhere else in the constraints. As is explained later in the semantics, the atoms $x \circ c$ in a state with $x \in U$ of a state stand for the possible instances of the state modulo user choices. By bounding the variables relevant to the transition flow we achieve decidability of the transition system of PIDL+. Variables not in X_T do not affect transitions and function only as holders of result values specific to the current state. Their valuations are derived from the values of terms over the transition variables. Moreover, admissible specifications may have constraints that are purely propositional, which are encoded in the formula Φ.

2.3 Semantics

Given a PIDL+ specification, the semantics is a possible-worlds semantics with states and transitions between them, starting from the initial state and adhering to the constraints and transition conditions of the specification. For the rest of the paper, we assume that we only work with admissible specifications. While the meaning of the initial state is briefly given in the syntax section because the initial state is part of the specification, we now define states generally.

Definition 4. A *state* is a pair (S, U) of a set of *user variables* $U \subseteq X$, and a set of literals S containing simple atoms over Σ and propositional literals over Π. The subset $S|_U$ is always a set of simple bounds over Σ.

A state in PIDL+ contains the description of a corresponding state in the configuration system. As mentioned in the syntax section, propositional literals and arithmetic atoms fulfill this task. The user variables U tell us what variable has been set by the user during the configuration process so far. The simple bounds $x \circ c$ with $x \in U$ describe the range of user choices in that state. For example, the state $(\{x > 4, x \leq 21, A\}, \{x\})$ implicitly corresponds to a set of worlds. It includes an instance in which A holds and the user has chosen $x \approx 5$, another instance in which A holds and the user has set $x \approx 6$, or one where A holds and $x \approx 18$ by a user decision.

We adopt the following convention. Let (S, U) be a state. We write $\vec{x}_u$ to mean all the variables of U and $\vec{y}$ to mean all the variables in $X \setminus U$. If we use this notation, it should be clear from the context that we refer to the U of a certain state.

The view that a state is an aggregation of all the possible instances modulo the user decisions, embodied by the simple bounds in $S|_U$, has far-reaching implications for our semantics. We define transitions later, but we already want to say at this point that different instances of the user variable valuations can imply different behaviors concerning transitions. Consequently, it is necessary to divide a state with respect to its user choices. We do this by considering *selections*.

Definition 5. Let (S, U) be a state. Furthermore, let $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$ be a transition. Then, a *selection* γ with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$ is defined as follows: (1) If U is a non-empty set and $U = \{x_1, \ldots, x_n\}$, then γ is a tuple of interval integers $(I_1, \ldots, I_n)$, where

$$\forall \vec{y}(S|_U \wedge (S \setminus S|_U \cup \mathsf{C} \rightarrow \Lambda \wedge F))\sigma$$

is valid for all $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ with $v_i \in I_i$ and $i = 1, \ldots, n$, and (2) if $U = \emptyset$, then γ is the empty tuple $()$ and $\forall \vec{y}\ S \cup \mathsf{C} \rightarrow \Lambda \wedge F$ is valid.

We write $\gamma(i)$ to denote the i-th component I_i of $\gamma = (I_1, \ldots, I_n)$. The intervals $I_1, I_2, \ldots, I_n$ of a selection contain values of the user variables $x_1, x_2, \ldots, x_n$ for which $\forall \vec{y}(S|_U \wedge (S \setminus S|_U \cup \mathsf{C} \rightarrow \Lambda \wedge F))$ is valid, expressed by all substitutions that map the variables from U to the values of the intervals. These intervals exist if there is at least one valuation of the user variables that makes the above statement true. If this is the case, we consider the condition of the corresponding transition $\Lambda \wedge F \rightsquigarrow E$ to be fulfilled. It can be read as "all assignments done by the user that are based on the values occurring in the selection and that are consistent with the simple atoms of $S|_U$ entail the transition condition $\Lambda \wedge F$, thus that transition is possible". As is defined later in this section, all transitions are with respect to some selections. If the set of user variables is empty, the criterion for allowing a transition reduces to essentially checking the validity of $S \cup \mathsf{C} \rightarrow \Lambda \wedge F$.

In our setting, considering subsets of user choices and their properties makes sense. We then talk about *subselections*.

Definition 6. Let γ and γ' be two selections of the same length. We say that γ' is a *subselection* of γ, written as $\gamma' \subseteq \gamma$, if $\gamma'(i) \subseteq \gamma(i)$ for all $i = 1, \ldots, n$. We say that γ' is a *proper subselection* of γ, written as $\gamma' \subset \gamma$, if $\gamma' \subseteq \gamma$ and $\gamma'(i) \subset \gamma(i)$ for an $i \in \{1, \ldots, n\}$.

In general, there can be more than one possible selection with respect to a transition. In our semantics, we restrict the possibilities to consider to *maximal* selections. Maximal selections are defined with the help of subselections.

Definition 7. A selection γ with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$ is *maximal* if either $\gamma = ()$ or there is no selection γ' with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$ such that $\gamma \subset \gamma'$.

Information about current selections may have to be included in the states. This means we have to write the intervals occurring the selections as simple bounds. *Atomic representations* denote those simple bounds that corresponds to the intervals of the respective selection. They are based on the atomic representations of integer intervals mentioned in the preliminaries of this section.

Definition 8. Let γ be a selection with respect to a state (S, U) and a transition. The atomic representation $at(\gamma)$ of γ is defined as follows: (1) If $U = \{x_1, \ldots, x_n\}$, then

$$at(\gamma) := \bigcup_{i=1,\ldots,n} at(\gamma(i), x_i), \text{ and}$$

(2) if $U = \emptyset$, then $at(\gamma) := \emptyset$.

In a transition from a state to a new state, the new state is the result of an *update* of the former state. The updates are based on the update sets E of the transitions $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$. The way how updates are done depends on the type of the transition: Updates with respect to rule transitions are different from updates with respect to user transitions. We first define updates of rule transitions.

Definition 9. The *rule update operator* $\triangleleft_R$ takes a state (S, U) and a pair (E, γ) as arguments, where (1) E is an update set and (2) γ is a selection with respect to (S, U) and a rule transition $\Lambda \wedge F \rightsquigarrow E \in T_R$. It is defined as $(S, U) \triangleleft_R (E, \gamma) := (S', U')$, where

- $U' := \quad U \setminus \mathit{varl}(E)$,
- $S' := \quad \{L | L \text{ literal over } \Pi, L \in S, \overline{L} \notin E\} \cup$
  $\{L | L \text{ literal over } \Pi, L \in E\} \cup$
  $\{x \circ t | x \circ t \in S, x \in X \setminus U, x \notin \mathit{varl}(E)\} \cup$
  $\{x \circ t | x \circ t \in at(\gamma), x \in U, x \notin \mathit{varl}(E)\} \cup$
  $\{x \circ t | x \circ t \in E\}$.

The propositional literals and simple atoms of the update set E replaces the propositional literals and simple atoms of the old state literal set S if those are of the same propositional variables and left-hand variables, respectively, to form the new state literal set S'. We fix the variables on the left-hand sides of simple atoms $x \circ t$ to be the relevant variables determining the updates of atoms. Expressions of E with variables new to the old state are added to S' too. Expressions in S whose propositional variables or left-hand variables do not occur as propositional variables or left-hand variables in E are preserved in S'. However, there is a distinction in what is preserved with respect to whether the left-hand variables x of the simple atoms $x \circ t$ are in U. If x is not a user variable, then the simple atoms $x \circ t$ from S are just carried over in the next state. If $x \in U$, then it is not the simple atoms from S that are preserved but the simple atoms in the atomic representation $at(\gamma)$ where x appears on the left sides of the atoms. The general rationale behind this is the following: The selection γ can be seen as a subset of the user choices defined by $S|_U$ that makes the transition possible, as described earlier with the definition of selections. It is crucial to keep track of what selection is responsible for the transition. Therefore, a simple atom $x \circ t$ of $at(\gamma)$ is contained in the new state by definition unless x occurs in $varl(E)$. In that case, the corresponding atom from E takes precedence and is contained in S' instead of the one from $at(\gamma)$. We consider the expressions appearing in the update set E of a rule transition to be "overwriting" existing user decisions. That is why, after the update, we do not see those variables of U occurring in $varl(E)$ as user variables anymore and we set $U' := U \setminus varl(E)$. This corresponds to the situation in the configuration when it is necessary to overwrite user choices due to the system constraints. In general, if $(S', U') \neq (S, U)$, we also say that the update or transition *changes* or *alters* the state (S, U).

We can now formally define rule transitions. If a state is consistent and there is an appropriate maximal selection with respect to a transition, then there is a transition from that state to a new state with respect to the corresponding transition, with the new state being the result of an update by the rule update operator $\triangleleft_R$.

Definition 10. A *rule transition* from a state (S, U) to a state (S', U') with respect to a rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$ and a selection γ is written as

$$(S, U) \rightarrow_{(i,\gamma)} (S', U'),$$

where (1) $\forall \vec{x}_u \exists \vec{y}\, S|_U \rightarrow (S \setminus S|_U) \cup \mathsf{C}$ is satisfiable, (2) γ is a maximal selection with respect to (S, U) and $\Lambda_i \wedge F_i \rightsquigarrow E_i$, and (3) $(S', U') = (S, U) \triangleleft_R (E_i, \gamma)$.

The criterion that $\forall \vec{x}_u \exists \vec{y}\, S|_U \rightarrow (S \setminus S|_U) \cup \mathsf{C}$ must be satisfiable is the same as the one used for the initial state (S_I, U_I) in the definition of specifications. It is the general criterion determining the consistency of a state.

In the configuration systems we consider, as mentioned earlier, all the rules are applied whenever possible until a fixed point is reached. Then, a user decision may happen after which another round of rule applications takes place. As long as rules can be applied that can change a state, no user action is allowed. To define user updates and user transitions, we therefore need the notion of *rule-terminal* states to denote states that are indeed "ready" for user transitions.

Definition 11. Let γ be a maximal selection with respect to a state (S, U) and a user transition $\Lambda \wedge F \rightsquigarrow E \in T_U$. (1) If $\gamma \neq ()$, then we call a selection γ' with $\gamma' \subseteq \gamma$ a *rule-terminal subselection of* γ if the following holds: (a) there is no selection γ^* with respect to (S, U) and a rule transition $\Lambda^* \wedge F^* \rightsquigarrow E^* \in T_R$, such that $\gamma^* \cap \gamma' \neq \emptyset$ and $(S, U) \neq (S, U) \triangleleft_R (E, \gamma^*)$, and (b) there is no other selection $\gamma^\#$ that fulfills (a) and $\gamma' \subset \gamma^\#$. (2) If $\gamma = ()$, then we call γ *rule-terminal* if $()$ is not a maximal selection with respect to (S, U) and a rule transition $\Lambda^* \wedge F^* \rightsquigarrow E^* \in T_R$ such that $(S, U) \neq (S, U) \triangleleft_R (E, ())$. In the respective cases, we say that (S, U) is *rule-terminal* with respect to γ' or $()$ and to $\Lambda \wedge F \rightsquigarrow E$.

A state is thus rule-terminal with respect to a selection γ' and a user transition if γ' is such a subselection of a maximal selection γ with respect to the state and the user transition that it does not have a non-empty intersection with another selection that can be used for a rule transition changing the state. A non-empty intersection would mean that the current selection making the user transition possible also contains instances of the user decisions that enable rule transitions that alter the state. Therefore, the state cannot be considered to be a fixed point yet with respect to that selection. What we are interested in is a subselection γ' for which the state becomes a fixed point. To have uniqueness, we only consider those γ' that are not subselections of selections having the same property, analogously to the definition of maximal selections.

The definition of the *user update operator* is very similar to that of the rule update operator. The difference is that rule updates may reduce the set of user variables, while user updates may increase it. This is because user transitions represent user actions, so the corresponding update set E can introduce new user variables.

Definition 12. The *user update operator* $\lhd_U$ takes a state (S, U) and a pair (E, γ) such that (1) E is an update set, (2) γ is a rule-terminal selection with respect to (S, U) and a user transition $\Lambda \wedge F \rightsquigarrow E \in T_U$, and is defined as $(S, U) \lhd_U (E, \gamma) := (S', U')$, where $U' := U \cup \mathit{varl}(E)$ and S' is defined in the exact same way as in the case of the rule update operator $\lhd_R$.

User transitions are then defined analogously to rule transitions, with respect to rule-terminal states.

Definition 13. A *user transition* from a state (S, U) to a state (S', U') with respect to a user transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U$ and a selection γ is written as

$$(S, U) \rightarrow_{(i,\gamma)} (S', U'),$$

where (1) $\forall \vec{x}_u \exists \vec{y}\, S|_U \rightarrow (S \setminus S|_U) \cup \mathsf{C}$ is satisfiable, and (2) (S, U) is rule-terminal with respect to γ and $\Lambda_i \wedge F_i \rightsquigarrow E_i$, and (3) $(S', U') = (S, U) \lhd_U (E_i, \gamma)$.

A classic notion connected to state transition systems is that of *reachability.* We give the corresponding concepts in our semantics involving *paths* and reachability. A path tells us, in particular, what user actions and rule actions were responsible for reaching a certain state, starting from the initial state (S_I, U_I) of a specification.

Definition 14. A *path* from a state (S, U) to a state (S', U') is a tuple $((i_1, \gamma_1), \ldots, (i_n, \gamma_n))$ such that $(S_0, U_0) \rightarrow_{(i_1,\gamma_1)} (S_1, U_1) \rightarrow_{(i_2,\gamma_2)} \cdots \rightarrow_{(i_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(i_n,\gamma_n)} (S_n, U_n)$ is a sequence of transitions, where for all $j = 1, \ldots, n$ it holds that γ_j is a maximal selection with respect to (S_{i-1}, U_{i-1}) and a $\Lambda_{i_j} \wedge F_{i_j} \rightsquigarrow E_{i_j} \in T_R$ or (S_{i-1}, U_{i-1}) is rule-terminal with respect to γ_j and a $\Lambda_{i_j} \wedge F_{i_j} \rightsquigarrow E_{i_j} \in T_U$, and moreover $(S_0, U_0) = (S, U)$ and $(S_n, U_n) = (S', U')$.

Definition 15. A state (S', U') is *reachable* from a state (S, U) if there is a path from (S, U) to (S', U'). A state (S, U) is always reachable from itself via the empty path.

The question whether a specification is *consistent* is determined by whether it has a *model.* A model is basically the set of all states and transitions between them when starting from the initial state, where no inconsistent world according to the definition of Σ-interpretations as described in the preliminaries subsection is reachable.

Definition 16. An *interpretation* of an admissible specification $\mathfrak{S}_+$ is a tuple $(\mathcal{V}_{\mathfrak{S}+}, \mathcal{T}_{\mathfrak{S}+}, \mathcal{I}_{\mathfrak{S}+})$ with (1) the *state space* $\mathcal{V}_{\mathfrak{S}+} := \{(S, U) | (S, U) \text{ reachable from } (S_I, U_I)\}$, (2) the *transition space*

$$\mathcal{T}_{\mathfrak{S}+} := \{((S, U), i, \gamma, (S', U')) | (S, U), (S', U') \in \mathcal{V}_{\mathfrak{S}+},$$
$$(S, U) \rightarrow_{(i,\gamma)} (S', U'), \Lambda_i \wedge F_i \rightsquigarrow E_i \in (T_U \cup T_R)\},$$ and

(3) the *state interpretations* $\mathcal{I}_{\mathfrak{S}+} := \{((S, U), I_\Sigma) | (S, U) \in \mathcal{V}_{\mathfrak{S}+}, I_\Sigma \models S\}$.

Note that indeed $(S_I, U_I) \in \mathcal{V}_{\mathfrak{S}+}$ according to our definition of reachability. An interpretation is a model if the constraints C are satisfied in each world.

Definition 17. An interpretation $(\mathcal{V}_{\mathfrak{S}+}, \mathcal{T}_{\mathfrak{S}+}, \mathcal{I}_{\mathfrak{S}+})$ is a *model* of a an admissible specification $\mathfrak{S}_+$ if $I_\Sigma \models \mathsf{C}$ for each $((S, U), I_\Sigma) \in \mathcal{I}_{\mathfrak{S}+}$.

Definition 18. An admissible interpretation $\mathfrak{S}_+$ is *sound* if it has a model. Moreover, $\mathfrak{S}_+$ is *complete* with respect to a set X_U of user variables if for every $\sigma : X_U \rightarrow \mathbb{Z}$ with $\mathsf{C}\sigma$ being satisfiable, there is $(S, U) \in \mathcal{V}_{\mathfrak{S}+}$ such that $(S \cup \mathsf{C})\sigma$ is satisfiable and for each $x \in X_U$ there is a user transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U$ with $x \in \mathit{varl}(E_i)$, $((S_{j-1}, U_{j-1}), i, \gamma, (S_j, U_j)) \in \mathcal{T}_{\mathfrak{S}+}$.

Soundness and completeness can be effectively checked, once a finite representation of the transition graph is computed. Such a graph is the result of the algorithm buildInterpretation, Algorithm 1.

From a user's perspective, soundness means that he/she cannot navigate the system into an inconsistent state. Completeness means that within the initial bounds with respect to the overall constraints, any product can be configured by respective user actions for a given set of user controllable variables.

The following theorem follows from the finiteness of the components of admissible specifications and the fact that the transition variables in them are over bounded integers.

Theorem 19. The components $\mathcal{V}_{\mathfrak{S}+}$ and $\mathcal{T}_{\mathfrak{S}+}$ of each interpretation of an admissible specification $\mathfrak{S}_+$ are finite. Soundness and completeness are decidable.

We conclude this section with an example.

Example 20. Assume the following admissible specification:

$$\begin{aligned}
\Pi &= \{A, B, C, D, E\},\\
X &= \{x_1, x_2, y_1\},\\
S_I &= \{\neg A\},\\
U_I &= \emptyset,\\
\mathsf{C} &= \{x_1 \geq 0, x_1 \leq 100,\\
&\quad x_2 \geq 0, x_2 \leq 100,\\
&\quad y_1 \geq 0, y_1 \leq 1000,\\
&\quad C \rightarrow D\},\\
T_U &= \{\neg A \rightsquigarrow_{u_1} \{A, B, x_1 \geq 20, x_1 \leq 90\},\\
&\quad x_1 < 74 \wedge B \rightsquigarrow_{u_2} \{x_2 > 5, x_2 < 20\}\},\\
T_R &= \{x_1 \leq 30 \rightsquigarrow_{r_1} \{\neg C\},\\
&\quad x_2 \geq 10 \wedge D \rightsquigarrow_{r_2} \{y_1 \approx x_1 \cdot x_2, C\},\\
&\quad y_1 \geq 600 \wedge y_1 \leq 800 \rightsquigarrow_{r_3} \{\neg E\}\}.
\end{aligned}$$

In the following, we describe how the set of states that are reachable from the initial state can be derived. We refer to transitions $\Lambda_i \wedge F_i \rightsquigarrow E_i$ by their indices i. From the initial state $(S_I, U_I) = (\{\neg A\}, \emptyset)$, we see that user transition u_1 is possible using the empty selection $()$, since U_I is empty. With that, the relevant criterion for $()$ to be the right selection is that $\forall \vec{y}\{\neg A\} \cup \mathsf{C} \rightarrow \neg A$ is valid, which is obviously the case. We get a transition

$(S_I, U_I) \rightarrow_{(u_1,())} (S_1, U_1)$, where $S_1 = \{A, B, x_1 \geq 20, x_1 \leq 90\}$ and $U_1 = \{x_1\}$,

according to the update $(S_I, U_I) \triangleleft_U (\{A, B, x_1 \geq 20, x_1 \leq 90\})$ as defined in Definition 12. Note that the update overwrites the literal $\neg A$ from the initial state and replaces it with the literal A. The new state (S_1, U_1) symbolizes user choices of the variable x_1 with integer values between 20 and 90. From (S_1, U_1), there are two possible transitions, one of which is a rule transition and the other is a user transition. First, rule transition r_1 is applicable for the maximal selection $([20, 30])$. Note that indeed $\forall \vec{y}(S_1|_{U_1} \wedge (S_1 \setminus S_1|_{U_1} \cup \mathsf{C} \rightarrow x_1 \leq 30))\sigma$ is valid for all $\sigma = \{x_1 \mapsto v\}, v \in [20, 30]$ as required in Definition 5. We register a transition

$(S_1, U_1) \rightarrow_{(r_1,([20,30]))} (S_2, U_2)$,where $S_2 = \{A, B, \neg C, x_1 \geq 20, x_1 \leq 30\}$ and $U_2 = \{x_1\}$,

according to $\triangleleft_R$ as defined in Definition 9. On the other hand, there is a rule-terminal selection with respect to (S_1, U_1) and user transition u_2. The maximal selection with respect to (S_1, U_1) and user transition u_2 is $[20, 73]$. The subselection that is rule-terminal with respect to that is $[31, 73]$. No other rule transition that changes the state is possible in this subselection. Thus, we have

$$(S_1, U_1) \rightarrow_{(u_2,([31,73]))} (S_3, U_3), \text{ where}$$
$$S_3 = \{A, B, x_1 \geq 31, x_1 \leq 73, x_2 > 5, x_2 < 20\} \text{ and } U_3 = \{x_1, x_2\},$$
which represents that the user has now additionally set x_2. State (S_3, U_3) warrants rule transition r_2 with the selection $([31, 73], [10, 19])$. We get
$$(S_3, U_3) \rightarrow_{(r_2,([31,73],[10,19]))} (S_4, U_4), \text{ where}$$
$$S_4 = \{A, B, C, x_1 \geq 31, x_1 \leq 73, x_2 \geq 10, x_2 \leq 19, y_1 = x_1 \cdot x_2\} \text{ and } U_4 \approx \{x_1, x_2\}.$$
Finally, there are more than one possible selections to enable rule transition r_3. If we choose the selection $([50, 61], [12, 13])$, we get
$$(S_4, U_4) \rightarrow_{(r_3,([50,61],[12,13]))} (S_5, U_5), \text{ where}$$
$$S_5 = \{A, B, C, \neg E, x_1 \geq 50, x_1 \leq 61, x_2 \geq 12, x_2 \leq 13, y_1 \approx x_1 \cdot x_2\} \text{ and } U_5 = \{x_1, x_2\}.$$
Again, note that all selections given in this example are maximal in the sense that they are not subselections of any other selections with the same properties demanded in the definition of transitions and rule-terminal states.

3 Algorithms

The algorithm BUILDINTERPRETATION, Algorithm 1, computes an abstract partial interpretation according to Definition 16 of an admissible specification if it exists. It takes an admissible specification $\mathfrak{S}_+$ and computes the components $\mathcal{V}_{\mathfrak{S}+}$ and $\mathcal{T}_{\mathfrak{S}+}$ of a possible interpretation of $\mathfrak{S}_+$, named V and T in the algorithm. It aborts and returns "inconsistent" if it encounters an inconsistent state, which means the specification is inconsistent. Otherwise, it returns the resulting *state graph* $G = (V, T)$ that can serve as the basis for further analyses of the properties of the specification and thus of the modeled configuration system.

A central element is the computation of selections, that is, integer intervals as defined in the semantics (Definition 5). Since the intervals represent valuations of bounded user variables, as explained in the previous section, this task is decidable and can be carried out in the most naive way by enumerating the finitely many possibilities. A more efficient alternative to this is an appropriate use of interval arithmetic [11]. In this paper, we restrict ourselves to assuming that selections relevant to transitions as defined in the semantics (Definitions 10, 11 and 13) are available in the form of $\mathit{maxSelections}_{\mathfrak{S}_+}$, given a state and a transition.

Definition 21. Let (S, U) be a state, and $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$ be a transition. Then $\mathit{maxSelections}_{\mathfrak{S}_+}(S, U, \Lambda \wedge F \rightsquigarrow E)$ is the set of all maximal selections γ with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$.

BUILDINTERPRETATION starts with the initial state of the input specification, as shown by the initialization of N, which is the set of states not yet processed by the algorithm. The set H stores all the states that the algorithm has computed so far. The algorithm runs as long as there are states to be processed (line 5). In each iteration, consistency of the current state is checked (line 7) first, using an external solver, according to the same consistency criterion as stated in the transition semantics (Definitions 10 and 13). The algorithm stops if the state is not consistent, otherwise it continues. Then, for each suitable maximal selection with respect to the current state and a rule transition, a new state is computed according to $\lhd_R$ (line 11). The new transition is registered in line 12. If the new state differs from the current one, the current selection is one of the selections under which the state is truly changed, and these are collected in a set Γ (line 14). Also, if the newly generated state has not appeared before, it is added to V, H and N (lines 15-18). This makes sure that we do not process states more than once.

After that, the user transitions with respect to the current state are investigated. If $\Gamma = \{()\}$ (check in line 19), it means that there has been a selection, namely $()$, that changes the state with the set of user variables being empty (Definition 5). This means the state is not rule-terminal in any case by Definition 11. If this is not the case, it is checked if the current state is rule-terminal with respect to any selections. Given the current maximal selection γ with respect to the current state and current user transition, and the set Γ of all selections where rule transitions alter the state, REDUCESELECTION (Algorithm 2) is used to find all rule-terminal subselections of γ (line 22). The subselections γ' returned by REDUCESELECTION that are sound and complete (Theorem 22), have the following property, as explained further below: If γ' does not include an empty interval or γ' is the empty selection $()$, then the current state is rule-terminal with respect to γ' and the current user transition (line 23). BUILDINTERPRETATION then creates a new state with γ' and does the same steps as in the case of rule transitions. At the end of the while loop, we remove the current state from the set N of states to be processed (line 33). Provided that the algorithm has not encountered an inconsistent state, it returns V and T, which are the components of every interpretation and, in particular, every model of the consistent specification.

We give an overview on the recursive algorithm REDUCESELECTION in the following. It takes as arguments a selection γ and a set Γ of selections of the same length, and it returns a set of selections Δ such that all elements of Δ are (1) subselections of γ and (2) have only empty intersections with the selections in Γ. In addition, each selection $\delta \in \Delta$ is maximal in the sense that there is no other selection δ' with $\delta \subset \delta'$ and that fulfills (1) and (2). The basic idea is that, in each recursive step, γ is reduced to a subselection so that it just about has an empty intersection with one selection γ' from Γ. That selection γ' is removed from Γ for the next recursion step, and once Γ has become empty, we get a reduced γ that has the above properties. This is why the current γ in line 2 is returned as a singleton. If Γ is not empty, we keep removing selections γ' from Γ until we have removed everything or we have found a selection γ' that has a non-empty intersection with the current γ (lines 3 to 7). Again, REDUCESELECTION returns $\{\gamma\}$ if it turns out that it has no non-empty intersections with the rest of Γ (line 9). If we now assume we have found a γ' with non-empty intersection with γ, the next step is to reduce γ so that there is an empty intersection. There is potentially more than one possibility to do the reduction. At each position i of the $|\gamma|$ positions, we consider the difference $\gamma(i) \setminus \gamma'(i)$ (line 12) and replace $\gamma(i)$ with that difference, thus making γ having an empty intersection with γ'. Another branching action can be required at this point: If the set difference consists of two intervals that are not connected (line 13), which is the case if $min(\gamma) < min(\gamma') \leq max(\gamma') < max(\gamma)$, then we consider two separate recursive calls of REDUCESELECTION with $\gamma(i)$ being replaced by each one of the intervals respectively (line 14). Otherwise, we simply replace $\gamma(i)$ with the difference (line 16) in one recursive call. We collect all the possible solutions into a result set R.

Theorem 22 says that the use of REDUCESELECTION(γ, Γ) in line 22 of BUILDINTERPRETATION gives us the correct rule-terminal subselections as defined in the semantics in a complete way. Note that the results of REDUCESELECTION can include "extra" subselections that have been so much reduced from γ that they contain empty components. These selections are discarded by BUILDINTERPRETATION because they are no longer selections with respect to the current state and user transition (line 23). The case $\gamma' = ()$ means that there are no user variables (Definition 5) and that $\Gamma = \emptyset$, since otherwise $\Gamma = \{()\}$, which is not the case (line 19). Consequently, REDUCESELECTION immediately returns $\{()\}$, a singleton containing a rule-terminal selection.

Theorem 22. Let γ be a maximal selection with respect to a state (S, U) and a user transition

Algorithm 1: buildInterpretation($\mathfrak{S}_+$)

```
$N := \{(S_I, U_I)\}$
$H := \{(S_I, U_I)\}$
$V := \{(S_I, U_I)\}$
$T := \emptyset$
while $N \neq \emptyset$ do
    let $(S, U) \in N$
    if $\forall \vec{x}_u \exists \vec{y} S|_U \to (S \setminus S|_U) \cup \mathsf{C}$ is satisfiable then
        $\Gamma := \emptyset$
        for each $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$ do
            for each $\gamma \in maxSelections_{\mathfrak{S}_+}(S, U, \Lambda_i \wedge F_i \rightsquigarrow E_i)$ do
                $(S', U') := (S, U) \lhd_R (E_i, \gamma)$
                $T := T \cup \{((S, U), i, \gamma, (S', U'))\}$
                if $(S', U') \neq (S, U)$ then
                    $\Gamma := \Gamma \cup \{\gamma\}$
                    if $(S', U') \notin H$ then
                        $V := V \cup \{(S', U')\}$
                        $H := H \cup \{(S', U')\}$
                        $N := N \cup \{(S', U')\}$
        if $\Gamma \neq \{()\}$ then
            for each $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U$ do
                for each $\gamma \in maxSelections_{\mathfrak{S}_+}(S, U, \Lambda_i \wedge F_i \rightsquigarrow E_i)$ do
                    for each $\gamma' \in$ reduceSelection$(\gamma, \Gamma)$ do
                        if $\gamma'(j) \neq \emptyset$ for all $j$ or $\gamma' = ()$ then
                            $(S', U') := (S, U) \lhd_U (E_i, \gamma')$
                            $T := T \cup \{((S, U), i, \gamma', (S', U'))\}$
                            if $(S', U') \neq (S, U)$ then
                                if $(S', U') \notin H$ then
                                    $V := V \cup \{(S', U')\}$
                                    $H := H \cup \{(S', U')\}$
                                    $N := N \cup \{(S', U')\}$
    else
        return inconsistent
    $N := N \setminus \{(S, U)\}$
return $(V, T)$
```

$\Lambda \wedge F \rightsquigarrow E \in T_U$. A selection γ^* with $\gamma^* \subseteq \gamma$ and $\gamma^*(i) \neq \emptyset$ for all i is a rule-terminal subselection of γ if and only if $\gamma^* \in$ reduceSelection(γ, Γ), where Γ is the set of all selections γ' such that γ' is a maximal selection with respect to (S, U) and a rule transition $\Lambda' \wedge F' \rightsquigarrow E' \in T_R$ with $(S, U) \neq (S, U) \lhd_R (E', \gamma')$.

Proof. (Idea) Using associativity of set differences, the fact that all selections occurring as the first argument in reduceSelection are subselections of the input selection γ and induction over the recursion depth. □

Apparently, each state computed by buildInterpretation implies a path of transitions as indicated by the set T in the algorithm. It is easy to see they are analogous to the paths defined

Algorithm 2: REDUCESELECTION(γ, Γ)

```
if $\Gamma = \emptyset$ then
    return $\{\gamma\}$
let $\gamma' \in \Gamma$
$\Gamma := \Gamma \setminus \{\gamma'\}$
while $\Gamma \neq \emptyset$ and $\gamma \cap \gamma' = \emptyset$ do
    let $\gamma' \in \Gamma$
    $\Gamma := \Gamma \setminus \{\gamma'\}$
if $\gamma \cap \gamma' = \emptyset$ then
    return $\{\gamma\}$
$R := \emptyset$
for $i = 1$ to $|\gamma|$ do
    $s := \gamma(i) \setminus \gamma'(i)$
    if $s = I_1 \cup I_2$, $I_1, I_2$ intervals and $I_1 \cap I_2 = \emptyset$ then
        $R := R \cup$ REDUCESELECTION$(\gamma[i/I_1], \Gamma) \cup$ REDUCESELECTION$(\gamma[i/I_2], \Gamma)$
    else
        $R := R \cup$ REDUCESELECTION$(\gamma[i/s], \Gamma)$
return $R$
```

in the semantics in Definition 14. Theorem 23 then says BUILDINTERPRETATION is sound and complete with respect to the semantics of PIDL+.

Theorem 23. Let $\mathfrak{S}_+ = (\Pi, X, S_I, U_I, \mathsf{C}, T_U, T_R)$ be an admissible specification. A state (S, U) is reachable from the initial state (S_I, U_I) via a path τ if and only if $(S, U) \in V$ and we have a sequence $((S_0, U_0), i_1, \gamma_1, (S_1, U_1))$, ..., $((S_{n-1}, U_{n-1}), i_n, \gamma_n, (S_n, U_n))$, where $((S_{j-1}, U_{j-1}), i_j, \gamma_j, (S_j, U_j)) \in T$ and $\tau(j) = (i_j, \gamma_j)$ for all $j = 1, \dots, n$, $|\tau| = n$, $(S_0, U_0) = (S_I, U_I)$, $(S_n, U_n) = (S, U)$ and BUILDINTERPRETATION$(\mathfrak{S}_+) = (V, T)$.

Proof. (Idea) By induction over the length of the path τ. □

4 Conclusions

We have presented PIDL+, a logic suited for formalizing configuration systems whose dynamics stem from the effects of user interaction and rule actions. Boolean variables and arithmetic expressions of the configurations are considered. The choice of polynomials is motivated by the real-world systems we have already investigated for the design of PIDL+. Of course, PIDL+ can be extended to other suitable choices of closed arithmetic expressions. What distinguishes PIDL+ from similar logics is that it takes the characteristic features of such configuration systems into account by incorporating two types of transitions and the notion of rule-terminal states. This, in principle, forms the basis for a systematic analysis of the whole configuration process, making an a priori verification of those systems possible. Natural next steps include the implementation of the shown decision procedures and conducting corresponding experiments on real-world data. Related to that is the development and implementation of further decision procedures to investigate properties of interests of configuration systems, as we have done in the case of PIDL [9].

References

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002.

[2] Philippe Balbiani, Andreas Herzig, and Nicolas Troquard. Dynamic logic of propositional assignments: A well-behaved variant of PDL. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 143–152, 2013.

[3] Razieh Behjati and Shiva Nejati. Interactive configuration verification using constraint programming. Lyon, France, 2014.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 335–344, 2010.

[7] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 23–34. IEEE Computer Society, 2007.

[8] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Autom. Softw. Eng.*, 18(1):77–114, 2011.

[9] Deepak Dhungana, Ching Hoo Tang, Christoph Weidenbach, and Patrick Wischnewski. Automated verification of interactive rule-based configuration systems. In *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, pages 551–561. IEEE Digital Library, 2013.

[10] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[11] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling, and Computation*, 1:209–236, 2007.

[12] Paul Harrenstein, Wiebe van der Hoek, John-Jules Ch. Meyer, and Cees Witteveen. Boolean games. In *Proceedings of the Eight Conference on Theoretical Aspects of Rationality and Knowledge*, pages 287–298, 2001.

[13] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[14] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. The MIT Press, Cambridge, USA, 1994. electronic edition.

[15] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[16] Marcello La Rosa, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and System Modeling*, 8(2):251–274, 2009.

[17] Paolo Turrini. Endogenous boolean games. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.