Original paper published in the proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) (DOI: 10.1109/ASE.2013.6693112).

Automated Verification of Interactive Rule-Based Configuration Systems

Deepak Dhungana*, Ching Hoo Tang†, Christoph Weidenbach†, Patrick Wischnewski ‡
*Siemens AG Österreich
Vienna, Austria
Email: deepak.dhungana@siemens.com
†Max Planck Institute for Informatics
Saarbrücken, Germany
Email: {chtang, weidenbach}@mpi-inf.mpg.de
‡Logic4Business GmbH
Saarbrücken, Germany
Email: patrick.wischnewski@logic4business.com

***Abstract*—Rule-based specifications of systems have again become common in the context of product line variability modeling and configuration systems. In this paper, we define a logical foundation for rule-based specifications that has enough expressivity and operational behavior to be practically useful and at the same time enables decidability of important overall properties such as consistency or cycle-freeness. Our logic supports rule-based interactive user transitions as well as the definition of a domain theory via rule transitions. As a running example, we model DOPLER, a rule-based configuration system currently in use at Siemens.**

I. Introduction

After their first successful application in the context of expert systems in the 1980's, rule-based specifications of systems have again become common in the context of product line variability modeling and configuration systems. Designing a rule-based language is always a compromise between expressivity, semantics, and decidability of overall properties. Expressivity starts at simple logics and ranges up to full programming language availability. Semantics, meaning what is the result of applying a set of rules in a particular context, ranges from a programming language style operational semantics to a model theoretic semantics. Finally, depending on the expressivity and underlying semantics, proving properties of a rule-based language ranges from polynomial decidability to undecidability.

In this paper, we investigate the role of rule-based languages in the context of interactive product configuration. Interactive configuration refers to the process of a user interactively assigning values to variables, under given restrictions specified using rules. Each step in the user-configurator interaction includes a user selecting a value from a domain and the configurator executing applicable rules to propagate the change. Our goal is to define a logical foundation that has enough expressivity and operational behavior to be practically useful and at the same time enables decidability of important overall properties such as consistency.

We will start from the available language DOPLER [1], which is a product line variability modeling tool-set currently in use at Siemens. A first attempt towards a formal semantics for DOPLER has been previously discussed in a workshop paper [2]. The initial workshop paper describes the key concepts of DOPLER, however a more comprehensive semantic framework that can eventually be subject to an automated analysis of existing knowledge bases is still missing. This paper provides a model-theoretic semantics for interactive rule-based systems, in particular DOPLER.

Our new logic PIDL (Propositional Interactive Dynamic Logic, see Section III) serves as a framework for the modeling, analysis and execution of rule-based configuration systems. It supports three fundamentally different types of formulas. The first formula type are constraints. Constraints describe necessary conditions of any configuration, e.g., that two components can never go together. The second type are rule transitions. Rule transitions describe necessary changes to the configuration typically as a result of a user decision, e.g., a user has selected two components but for technical reasons they have to be replaced by a third, different component. Finally, the third formula type are user transitions. They describe changes to the configuration done by a user in an interactive way, e.g., she selects a certain component. The semantics of these formula types is inherently different. Constraints must always be fulfilled while rule and user transitions must not lead to an inconsistent state including an appropriate notion of update. In PIDL, a user transition is only applicable if the exhaustive application of rule transitions reaches a unique consistent state, called rule-terminal state. The latter condition distinguishes PIDL from any other framework for describing rule-based systems, like guarded transition systems or temporal logics that lack language constructs supporting our semantics of rule and user transitions. It is in particular this semantics that enables a deep analysis of rule and user transitions including properties like confluence or cyclicity (see Section IV). When analyzing a PIDL specification user transitions are considered in a non-deterministic, exhaustive way.

In Section II, we present examples of PIDL properties which can be effectively analyzed and which are often highly indicative for errors in a rule-based configuration knowledge

base. They include inconsistency (conflicting rules, constraints), incompleteness (missing rules), redundancy (redundant rules), circularity (circularly depending rules), and confluence (result unique rule-based computations). The presented framework is field-tested, and has proved to be adequate to detect these errors in existing models. A summary of the results is presented in Section V.

Our main contribution is the new logic PIDL (see Section III) motivated by the semantics of DOPLER (see Section II). The logic is expressive enough to model DOPLER and at the same time it offers decidability of important properties of rule-based systems, such as inconsistency, incompleteness, redundancy, confluence, and cyclicity. This way it generalizes known approaches such as guarded transitions systems. At the same time it replaces the problem of undecidability of programming language verification applicable to rule-based systems written in a programming language by an expensive, but effective decision procedure for all the above properties. In particular, PIDL is expressive enough to support decision revision as expressed by rules of the form $A \wedge \phi \rightsquigarrow \{\neg A, \ldots\}$ and the concept of rule-terminal states. We show by a first prototypical implementation that PIDL can in fact be turned into a useful software system for the practical analysis of rule-based systems.

II. Illustrative Example: DOPLER

DOPLER is a rule-based tool suite for interactive product configuration. A DOPLER model describes the differences between products in a product line. The key modeling elements are *decisions* (representing configuration variables) and *assets* (representing artifacts being configured). Dependencies among decisions are modeled using rules of the form `if <condition> then <action>`. The assets are associated with decisions through boolean expressions called inclusion conditions. Assets may "include" or "exclude" other assets. Further details on the modeling approach have been described previously [1]. A DOPLER model serves as the running example in this paper. We present an example from the steel plant automation domain. Figure 1 depicts the key modeling elements and dependencies among them.

DOPLER models are used for interactive configuration. During configuration, a set of decisions is taken by the user. Some other decisions are assigned appropriate values through rules, which get executed after each user interaction. Each decision has an associated visibility condition to specify whether the variable is currently accessible to the user.

The operational semantics of DOPLER models can be informally described as follows. At runtime, decisions can either be visible or invisible to the user. All visible decisions (visibility condition evaluates to `true`) are presented to the user and the user assigns a value. Every user interaction triggers the rule engine, which evaluates all the rules and executes them if they are applicable. Rule execution can cause a variable binding, which leads to a recursive call of the rule engine. The user can also change the values of already taken decisions. Changing an already taken decision also causes a roll-back of the previous rule execution caused by the same decision. This ensures that the effect of the rules is undone when the condition of a rule no longer holds. An asset can either be included in or excluded from the desired final product (evaluation of the inclusion condition of the assets or the evaluation of an asset dependency). A state in DOPLER (the current assignment of values to decisions) can therefore be changed by user interactions and the subsequent execution of rules.

The running example (cf. Figure 1) shows decisions and assets as well as the relationships between them. Decisions are depicted by rectangle boxes on the left part of the image and assets by rounded corner boxes on the right part of the image. An arrow leading from a decision A to another decision B indicates that changing the value of A may also have an effect on B, depending on whether the rule, written as a label `<condition>` $\rightarrow$ `<action>` of the arrow, gets activated. If `<condition>` evaluates to `true`, the action is executed. The condition parts are written as usual Java-style Boolean expressions. The DOPLER framework also provides functions to manipulate the values of decisions and to query decision values, such as `setValue` and `isTaken`. For instance, `dynamicJet` evaluating to `true` makes `casterType` have the value `slab`. The former is a *Boolean* decision which can be assigned the value `true` or `false`, the latter is an *enumeration* decision whose range of possible finitely many values is predefined.

In this example, three decisions are visible to the user from the beginning: `sprayHeader`, `dynamicJet` and `stainlessSteel`. The decision `hydraulicCylinder` has a visibility condition, namely, it requires `casterType`'s only value to be `slab` and `taperUnit` to be `false`. If the visibility condition is evaluated to `true` then this decision is visible as well. The rest of the decisions are not visible and thus cannot be taken directly by the user.

The lines connecting assets and decisions represent the assets' inclusions conditions. Their evaluation depend on the decisions and determine whether assets get included in the product or not. The asset `hController` is included if its inclusion condition `hydraulicCylinder` evaluates to `true`. `baleAdapter` is included similarly and additionally requires the other asset `pCalibthermometer`. This is one example of the inclusion/exclusion relationships between assets.

The running example presents examples of different kinds of anomalies in a DOPLER Model.

- **Inconsistency** occurs when the execution of different rules in the rule base leads to conflicting values for a decision. For example, assignment of the decision `stainlessSteel = true` would result in `casterType = bloom` through the rules associated with `molder` and at the same time `casterType = slab` through the rules associated with `gapChecker`. This is an anomaly in the model, as the decision `casterType` can have only one value.
- **Incompleteness** occurs when an expected configuration cannot be reached due to the lack of transitions. For example, the modeler expects the value of

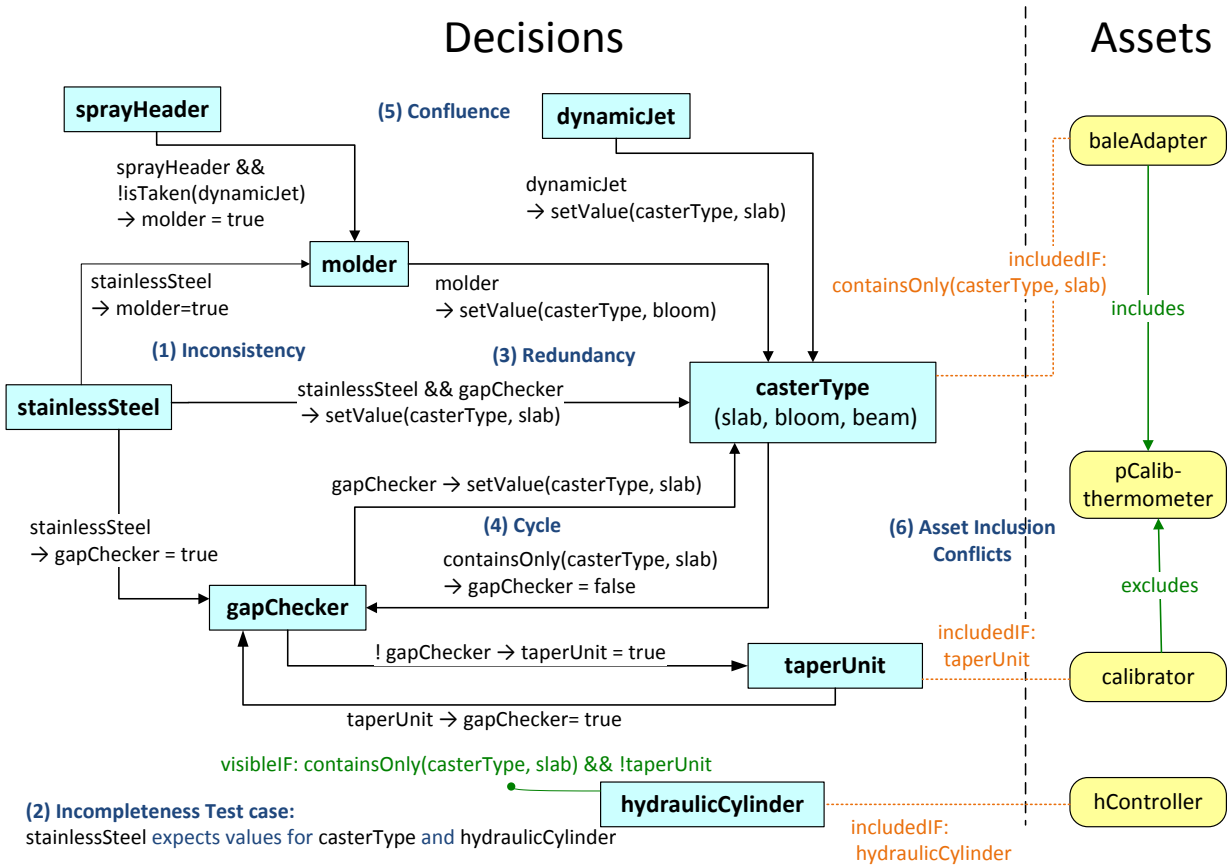

Fig. 1. Illustration of a DOPLER Model depicting decisions, assets and dependencies among them (rules, visibility conditions and asset inclusion conditions).

`hydraulicCylinder` to be set automatically, after `stainlessSteel` is assigned a value. However, there is no rule that would lead to this state.

- **Redundancy** occurs when more than one rule is modeled to achieve the same effect in the rule base. For example, after the value of `stainlessSteel` is set, we have `casterType = slab` through two different paths. This is an anomaly because it increases the maintenance effort of the rule base.
- **Cyclicity** occurs when the propagation of rules never stops because the involved rules change the variables such that there is always another rule that can be executed. For example, the three variables `gapChecker`, `taperUnit` and `casterType` form a cycle. The variable `gapChecker` is changed by `casterType` and `taperUnit`, making a different rule applicable after each execution.
- **Violation of Confluence** occurs when the order in which decisions are taken has an impact on the final configuration result. For example, depending on whether `sprayHeader` or `dynamicJet` is assigned a value first, the value of the variable `casterType` is either `bloom` or `slab`.
- **Asset Inclusion Conflicts** occur when the inclusion conditions of the assets are not consistent with the dependencies among the assets. For example, when `baleAdapter` and `calibrator` are both included in the configuration, they have a conflicting dependency to `pCalibthermometer` and it is not clear whether `pCalibthermometer` should be included or excluded.

III. PIDL: Propositional Interactive Dynamic Logic

PIDL is a new logic that provides detailed models for configuration systems. In particular, and in addition to all other temporal or dynamic proposition logics, it provides the notion of a rule-terminal state. Rule-terminal states are normal forms or fixed points with respect to a subset of the transition rules. They will later on be used to distinguish rules caused by user decisions from rules describing the domain. For the latter we expect uniqueness of the description, i.e., any user decision leads to a unique new state with respect to the domain rules. There is additional material available which includes the proofs for this section [3].

We first describe the syntax and semantics of PIDL and then provide a sound and complete calculus for it, based on the

ideas of superposition [4], [5], [6]. This calculus constitutes a decision procedure that will then be used in the rest of paper to actually analyze the properties of rule-based systems, in particular DOPLER.

Let F_Π denote the set of all propositional formulas over a finite set of propositional variables Π. A *state* is a consistent set of literals from Π.

A *PIDL specification* $\mathfrak{S}$ is a 5-tuple $(\Pi, S_I, \mathsf{C}, T_U, T_R)$, where

- Π is a finite set of propositional variables,
- S_I is the *initial* state,
- C is a finite set of propositional formulas over Π called *constraints*,
- T_U is a finite set of indexed tuples $\chi_i \rightsquigarrow E_i$ called *user transitions*, where $\chi_i \in F_\Pi$ and E_i is a state,
- T_R is a finite set of indexed tuples $\chi_j \rightsquigarrow E_j$ called *rule transitions*, where $\chi_j \in F_\Pi$ and E_j is a state,

and we assume that all user and rule transitions have different indexes. The set Π contains a dedicated variable *start* that is not used elsewhere in the specification.

Starting from the initial state S_I, the specification $\mathfrak{S}$ induces a number of states. An *update* of a state S by an E, written $S \triangleleft E$, is defined by $S \triangleleft E := \{L | (L \in S \text{ and } \overline{L} \notin E) \text{ or } L \in E\}$. Literals in S are replaced by the literals in E that are of the same variable but have a different sign, and literals of E previously not contained in S are added to S.

Example 1: $\{A, B, \neg C\} \triangleleft \{\neg B, \neg C, D\} = \{A, \neg B, \neg C, D\}$

A *rule transition application* using a rule transition $\chi_i \rightsquigarrow E_i \in T_R$ is a tuple $S \rightarrow_i S'$, where

- $S \cup \mathsf{C} \not\models \bot$,
- $S \cup \mathsf{C} \models \chi_i$, and
- $S' = S \triangleleft E_i$.

Example 2: The state $S = \{A, \neg B, C\}$ induces via rule transition $A \wedge C \rightsquigarrow \{B\}$ the state $S \triangleleft E_i = \{A, B, C\}$.

For convenience, we may use the term rule transition instead of rule transition application.

We say that a state S is *rule-terminal* if for all $\chi_i \rightsquigarrow E_i \in T_R : S \cup \mathsf{C} \models \chi_i$ implies $(S = S \triangleleft E_i)$. A state S is therefore rule-terminal if no rule transition leads to a state that is different from S.

Example 3: The state $S = \{A, B, \neg C, D, \neg E\}$ is rule-terminal with respect to the set of rule transitions T_R that consists of the following transitions:

- $A \rightsquigarrow \{B, \neg C\}$,
- $\neg C \rightsquigarrow \{D\}$, and
- $A \wedge \neg D \rightsquigarrow \{E\}$.

A *user transition application* using a user transition $\chi_i \rightsquigarrow E_i$ is a tuple $S \rightarrow_i S'$, where

- $S \cup \mathsf{C} \not\models \bot$,
- S is rule-terminal,
- $S \cup \mathsf{C} \models \chi_i$, and
- $S' = S \triangleleft E_i$.

The conditions are the same as for rule transitions expect we have an additional requirement that the state S must be rule-terminal. As in the case of rule transitions, we may use the term user transition for user transition application. Given a configuration system, all possible user interactions are modeled by user transitions in PIDL.

A *path* τ from state S_1 to S_n is a finite list of indexes $[i_1, i_2, \ldots, i_{n-1}]$, such that $S_j \rightarrow_{i_j} S_{j+1}$. The empty path is denoted by $[]$. In other words, a path τ consists of indexes that correspond to the user and rule transitions. We want to be able to construct paths incrementally in our calculus. To this end, we use the notation $[i_1, i_2, \ldots, i_n] :: i \ := \ [i_1, i_2, \ldots, i_n, i]$ to denote the extension of paths. Furthermore, the *length of a path* τ is denoted by $|\tau|$ and is the number of elements it contains.

The set of all states that are reachable from the initial state S_I is denoted by $\mathcal{S}_\mathfrak{S}$:

$$\mathcal{S}_\mathfrak{S} := \{S | \text{ there is a path from } S_I \text{ to } S\}.$$

Note that $\mathcal{S}_\mathfrak{S}$ is well-defined, i.e., all $S \in \mathcal{S}_\mathfrak{S}$ are consistent sets of literals, i.e., they do not contain any complementary literals. The reason for this is that the initial state S_I is consistent by definition, and the update operations that define the states of $\mathcal{S}_\mathfrak{S}$ preserve this property.

An *interpretation* $\mathcal{I}$ of a specification $\mathcal{S}_\mathfrak{S}$ is a function

$$\mathcal{I} : \mathcal{S}_\mathfrak{S} \rightarrow 2^\Pi$$

such that $\mathcal{I}(S) \models S$ and $start \in \mathcal{I}(S)$ for all $S \in \mathcal{S}_\mathfrak{S}$.

The interpretation of a single state yields a Herbrand interpretation, so $\mathcal{I}(S) \models A$ if $A \in \mathcal{I}(S)$ and $\mathcal{I}(S) \models \neg A$ if $A \notin \mathcal{I}(S)$.

An interpretation $\mathcal{I}$ is a *model* of a specification $\mathfrak{S}$ if $\mathcal{I}(S) \models \mathsf{C}$ for all $S \in \mathcal{S}_\mathfrak{S}$. A specification is called *inconsistent* if it has no model.

Example 4: Assume the following specification $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ defined by

- $\Pi = \{A, B, C, D\}$,
- $S_I = \{\neg A, \neg B\}$,
- $\mathsf{C} = \{B \rightarrow C\}$,
- $T_U = \{\neg A \rightsquigarrow \{A, B\}\}$ and
- $T_R = \{C \rightsquigarrow \{D\}\}$.

Then $\mathcal{S}_\mathfrak{S}$ consists of the following states:

- $S_I = \{\neg A, \neg B\}$,
- $S_1 = \{A, B\}$,
- $S_2 = \{A, B, D\}$.

One possible interpretation $\mathcal{I}$ is

- $\mathcal{I}(S_I) = \emptyset$,
- $\mathcal{I}(S_1) = \{A, B, C\}$, and
- $\mathcal{I}(S_2) = \{A, B, C, D\}$.

Another interpretation $\mathcal{I}'$ is

- $\mathcal{I}'(S_I) = \emptyset$,
- $\mathcal{I}'(S_1) = \{A, B\}$, and
- $\mathcal{I}'(S_2) = \{A, B, D\}$.

$\mathcal{I}$ is a model of $\mathfrak{S}$, whereas $\mathcal{I}'$ is not.

The calculus for PIDL operates on clauses annotated with labels which are representative of the states induced by the specification. We show how those clauses are generated and what inference steps can be applied to them.

A *labeled clause* has the form $(S, \tau, p \,||\, C)$, where S is a state, τ is a path, $p \in \mathbb{N} \cup \{*\}$, and C is a propositional clause over Π including the variable $start$.

One important concept is the ordering of labeled clauses which plays a role for redundancy and in proving completeness of the calculus. The ordering on clauses is based on a total ordering on paths. We define $\tau \prec \tau'$ if

- $|\tau| < |\tau'|$, or
- $|\tau| = |\tau'|$ and $\tau <_{lex} \tau'$,

where $<_{lex}$ is the lexicographic extension of the $<$-ordering on natural numbers.

Example 5:

- Let $\tau_1 = [3, 5, 2]$ and $\tau_2 = [4, 9, 2, 1, 3]$. Then $\tau_1 \prec \tau_2$ because $|\tau_1| = 3 < 5 = |\tau_2|$.
- Let $\tau_3 = [2, 9, 4, 2]$ and $\tau_4 = [2, 9, 4, 5]$. Then $\tau_3 \prec \tau_4$ because $|\tau_3| = 4 = |\tau_4|$ and $\tau_3 <_{lex} \tau_4$.

Intuitively, we associate a propositional clause C with the state it is derived from. What derived means is made more precise by the calculus description below. In addition to the state itself, the label of a clause also contains the path, i.e., the sequence of rule applications which led to this state. Furthermore, the symbol p indicates whether C is a general clause of the state, in which case $p = *$, or a specific clause connected to a rule condition, in which case $p \in \mathbb{N}$. The special $start$ clause functions as the "first clause" of each state in the calculus.

An interpretation $\mathcal{I}$ for a specification $\mathfrak{S}$ is a *model* of a labeled clause $(S, \tau, * \,||\, C)$, written $\mathcal{I} \models (S, \tau, * \,||\, C)$, if $S \in \mathcal{S}_{\mathfrak{S}}$ and $\mathcal{I}(S) \models C$. Moreover, $\mathcal{I}$ is a model of a set of labeled clauses if $\mathcal{I}$ is a model of each clause of the set. In the rest of the paper, we may refer to labeled clauses simply as clauses if the context is clear.

As usual as for a superposition-based calculus, redundancy and model assumptions are defined with respect to a total ordering lifted from the propositional variables to clauses.

Let $\prec$ be a total ordering on Π. It can be lifted to literals by $P \prec \neg P \prec Q$ if $P \prec Q$. Then it is lifted to clauses by its multiset extension on literals and finally to a partial ordering on labeled clauses by

$$(S, \tau, p \,||\, C) \prec (S', \tau', p' \,||\, C')$$

if $[\tau \prec \tau']$ or $[\tau = \tau'$, $p = *$ or $p = p'$, and $C \prec C']$. Note that $\prec$ is well-founded on labeled clauses.

Example 6: Let S, S' be two states and $\tau = [3, 1, 6], \tau' = [8, 2, 6, 1]$ be two paths. Furthermore, let the following ordering on variables be given: $A \prec B$.

- $(S, \tau, * \,||\, A \vee B) \prec (S', \tau', * \,||\, \neg B)$ because $\tau \prec \tau'$.
- $(S, \tau, * \,||\, A \vee B \prec (S', \tau, * \,||\, \neg B))$ because $A \vee B \prec \neg B$.

A labeled clause $(S, \tau, p \,||\, C)$ is *redundant* with respect to a set N of labeled clauses if there are clauses $(S, \tau', p'_1 \,||\, C_1)$, $(S, \tau', p'_2 \,||\, C_2)$, $\ldots$, $(S, \tau', p'_n \,||\, C_n) \in N$ with $(S, \tau', p'_i \,||\, C_i) \prec (S, \tau, p \,||\, C)$ for $1 \leq i \leq n$ and $C_1, C_2, \ldots, C_n \models C$.

Example 7:

- $(S, [5, 6, 9], * \,||\, A \vee B)$ is redundant with respect to $\{(S, [2, 3], * \,||\, A)\}$ because

$$(S, [2, 3], * \,||\, A) \prec (S, [5, 6, 9], * \,||\, A \vee B)$$

and $A \models A \vee B$.

- $(S, [5, 6, 9], * \,||\, A \vee B)$ is redundant with respect to $\{(S, [5, 6, 9], * \,||\, B)\}$ because

$$(S, [5, 6, 9], * \,||\, B) \prec (S, [5, 6, 9], * \,||\, A \vee B)$$

and $B \models A \vee B$.

Our notion of redundancy prevents the duplication of clauses sharing the same state at all: in the presence of a clause $(S, \tau, * \,||\, start)$ any other clause $(S, \tau', * \,||\, start)$ with $\tau \prec \tau'$ is redundant.

We now describe the inference rules $SInf$ consisting of Units Creation, Constraints Creation, User Transition Condition Creation, Rule Transition Condition Creation, Factoring, and Superposition that serve as a calculus with respect to the specification $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ for reasoning in one particular state.

- **Units Creation:** $\mathcal{I} \dfrac{S, \tau, * \,||\, start}{S, \tau, * \,||\, L}$,

  where $L \in S$.
- **Constraints Creation:** $\mathcal{I} \dfrac{S, \tau, * \,||\, start}{S, \tau, * \,||\, C}$,

  where $C \in cnf(\mathsf{C})$.

Units and Constraints Creation take the start clause $(S, \tau, * \,||\, start)$ of the state and produce labeled clauses for the constraints C and unit clauses out the state literals. Each literal of the state and each constraint is represented as labeled clauses by virtue of the two rules. It will become apparent below where the start clause comes from. $cnf(N)$ is the set of clauses that is the result of transforming a set of propositional formulas N into conjunctive normal form.

- **User Transition Conditions Creation:**

  $\mathcal{I} \dfrac{S, \tau, * \,||\, start}{S, \tau, i \,||\, C}$,

  where $C \in cnf(\neg\chi_i), \chi_i \rightsquigarrow E_i \in T_U$.
- **Rule Transition Conditions Creation:**

  $\mathcal{I} \dfrac{S, \tau, * \,||\, start}{S, \tau, i \,||\, C}$,

  where $C \in cnf(\neg\chi_i), \chi_i \rightsquigarrow E_i \in T_R$.

Having the start clause as premise, these rules yield labeled clauses that represent the conditions χ_i of the user and rule transitions. The propositional clauses C come from the negated conditions χ_i because we want to work with refutations, which will be explained more precisely. The label of such a clause contains the index i of the transition it corresponds to.

- **Factoring:** $\mathcal{I} \dfrac{S, \tau, p \,||\, C \vee A \vee A}{S, \tau, p \,||\, C \vee A}$,

  where C is a propositional clause and A is a literal.
- **Superposition:**

$$\mathcal{I}\frac{S,\tau,p \,||\, C \vee L \qquad S,\tau,p' \,||\, D \vee \overline{L}}{S,\tau,p \oplus p' \,||\, C \vee D} \ ,$$

where

- L and $\overline{L}$ are maximal in their respective clauses with respect to $\prec$,
- $p = *$ or $p' = *$ or $p = p'$, and
- the value of $p \oplus p'$ is defined by

$$p \oplus p' = \begin{cases} p' & \text{, if } p = *, \\ p & \text{, if } p' = * \text{ or } p = p' \end{cases} .$$

The two rules largely resemble rules of the well-known resolution calculus [4]. Factoring produces clauses where duplicate literals are removed. Superposition is resolution on the labeled clauses where the p in the label of the conclusion clause indicates if the result is connected to the transitions ($p = i$) or not ($p = *$).

Given a set of inference rules, such as *SInf*, we define $N^0_{SInf} = N$, $N^{i+1}_{SInf} = N^i \cup \{(S,\tau,p \,||\, C) | (S,\tau,p \,||\, C)$ is a conclusion of an *SInf* inference with premises in $N^i\}$, and $N^*_{SInf} := \bigcup_{i \geq 0} N^i_{SInf}$.

Now the inference rules *SRInf* include the rules *SInf* plus the rules Forward Rule Transition and Forward User Transition defined below.

- **Forward Rule Transition:**

$$\mathcal{I}\frac{S,\tau,i \,||\, \bot}{S',\tau :: i,* \,||\, start} \ ,$$

where

- $(S,\tau,* \,||\, \bot) \notin \{(S,\tau,* \,||\, start)\}^*_{SInf}$
- $\chi_i \rightsquigarrow E_i \in T_R$,
- $S' = S \lhd E_i$.

- **Forward User Transition:**

$$\mathcal{I}\frac{(S,\tau,i \,||\, \bot)}{S',\tau :: i,* \,||\, start} \ ,$$

where

- $S = S \lhd E_j$ for each $(S,\tau,j \,||\, \bot) \in \{(S,\tau,* \,||\, start)\}^*_{SInf}$
- $(S,\tau,* \,||\, \bot) \notin \{(S,\tau,* \,||\, start)\}^*_{SInf}$
- $\chi_i \rightsquigarrow E_i \in T_U$,
- $\chi_j \rightsquigarrow E_j \in T_R$, and
- $S' = S \lhd E_i$.

Forward Rule Transition says that whenever there is a clause $(S,\tau,i \,||\, \bot)$ corresponding to a rule transition $\chi_i \rightsquigarrow E_i$, and the inferences $SInf_{\mathfrak{S}}$ starting with $(S,\tau,* \,||\, start)$ have not yielded $(S,\tau,* \,||\, \bot)$, we can derive a new start clause $(S',\tau :: i,* \,||\, start)$ with S' being the state S updated by the rule transition and the transition being stored in the path τ. Forward User Transition works analogously with the additional premise that for all $(S,\tau,j \,||\, \bot)$ corresponding to rule transitions derived , the updates of the current state S by the rule transitions does not change the state.

We observe that each derivation of a clause labeled with S and τ must start with $(S,\tau,* \,||\, start)$ except for the starting clause itself which is derived through the transition rules.

Theorem 1 (Soundness and Completeness of SRInf): Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a PIDL specification. Then $\mathfrak{S}$ is inconsistent iff there is a labeled clause $(S,\tau,* \,||\, \bot) \in \{(S_I, [], * \,||\, start)\}^*_{SRInf}$.

Theorem 2 (Decidability of PIDL): Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a PIDL specification. Then $\{(S_I, [], * \,||\, start)\}^*_{SRInf}$ is finite up to redundancy.

It is well-known that *SInf* terminates on propositional logic with respect to redundancy, corresponding here to reasoning on clauses sharing the same path and state label.

Exploring Theorem 2, given the saturation $N^* = \{(S_I, [], * \,||\, start)\}^*_{SRInf}$ of a PIDL specification $\mathfrak{S}$, the *state graph* $G_{\mathfrak{S}}$ of $\mathfrak{S}$ consists of the vertices $V = \{S \mid (S,\tau,* \,||\, start) \in N^*\}$ and labeled edges $E = \{(S,i,T) \mid (S,\tau,i \,||\, \bot) \in N^*$ and $(T,\tau :: i,* \,||\, start) \in N^*\}$.

The state graph of some PIDL specification $\mathfrak{S}$ corresponds to the semantics of PIDL, i.e., $V = \mathcal{S}_{\mathfrak{S}}$ and if state T is reachable from state S in $G_{\mathfrak{S}}$, then there is a path from S to T. This justifies confusion of $G_{\mathfrak{S}}$ and the semantics for $\mathfrak{S}$.

IV. Properties

In this section, we define properties of rule-based configuration systems in terms of PIDL and show how they can be verified with our calculus. In the next section, we present how to use these properties in order to detect anomalies in a DOPLER model. We assume a given PIDL specification $\mathfrak{S}$ and its state graph $G_{\mathfrak{S}}$ with $V = \{S \mid (S,\tau,* \,||\, start) \in N^*\}$ and $E = \{(S,i,T) \mid (S,\tau,i \,||\, \bot) \in N^*$ and $(T,\tau :: i,* \,||\, start) \in N^*\}$.

- *Inconsistency*: $\mathfrak{S}$ is inconsistent iff there is a labeled clause $(S,\tau,* \,||\, \bot) \in \{(S_I,\epsilon,* \,||\, start)\}^*_{SRInf}$.
- *Incompleteness*: Let ϕ be a formula over Π. $\mathfrak{S}$ is incomplete with respect to ϕ iff there is a S in the state graph $G_{\mathfrak{S}}$ such that S is rule-terminal and $S \cup \mathsf{C} \not\models \phi$.
- *Redundancy*: Two rule transitions $\chi_i \rightsquigarrow E_i$ and $\chi_j \rightsquigarrow E_j \in T_R$ are redundant with respect to a state $T \in V$ iff there are edges (S,i,T) and $(S,j,T) \in E$.
- *Cycle*: A cycle in $\mathfrak{S}$ is a simple cycle of length greater than one in the state graph $G_{\mathfrak{S}}$, i.e., a path of the form $S_1, S_2, \ldots, S_n$ with $n \geq 3$, $S_1 = S_n$, $(S_i, j_i, S_{i+1}) \in E$ and the vertices $S_2, \ldots, S_{n-1}$ are all different from each other.
- *Confluence*: We distinguish between two types of confluences:
  - $\mathfrak{S}$ is *rule-confluent* iff for each state $S \in V$ the next rule-terminal state $T \in V$ that can be reached from S is unique.
  - $\mathfrak{S}$ is *user-confluent* iff for states $S, S' \in V$ that can be reached from $S_I \in V$ via paths τ and τ' respectively, where τ and τ' contain the same set of indexes that represent user transitions, the next rule-terminal state $T \in V$ that can be reached from S and S' is unique.

Note that these properties are decidable in PIDL because of the decidability theorem.

V. A Model of DOPLER

In this section, we show how PIDL encodes DOPLER models. We first describe the translation of DOPLER models into the logic. Then, we describe how anomalies in a DOPLER model can be detected using our PIDL framework from Section III. In the last part of the section, we evaluate our first prototypical implementation of the PIDL calculus.

A. Translation

We consider each relevant element of a DOPLER model and explain how it is represented in a PIDL specification $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$. We give examples that refer to the DOPLER model illustrated in Figure 1.

Decisions

DOPLER decisions are modeled as propositional variables in Π. In DOPLER, there are two types of decisions: namely Boolean and enumeration decisions.

For each Boolean decision `d`, we introduce two propositional variables d_Yes and d_No . This allows us to distinguish taken from open decisions. If d_Yes is true then the decision `d` is assigned to `true`. If `d` is assigned to `false` then d_No is true. The following formula represents the fact that `d` has not been assigned to a value $\neg d\_Yes \wedge \neg d\_No$.

Example 8: In the example from Section II, the decision `stainlessSteel` is represented by the variables $stainlessSteel\_Yes$ and $stainlessSteel\_No$ in PIDL.

In a DOPLER state, a Boolean decision cannot be true and false at the same time, which has to be considered in the corresponding PIDL specification as well. One could do this by adding formulas $\neg(d\_Yes \wedge d\_No)$ to the constraints C. An alternative way is ensuring that this property holds in the initial state and formulating the transitions so that it is preserved in the induced states, which is what we did as described below in the explanations of how we model DOPLER rules, user decisions and the initial state.

For each enumeration decision and each of its options, we introduce a variable denoting that the respective option is selected.

Example 9: `casterType` leads to the variables $casterType\_slab$, $casterType\_bloom$ and $casterType\_beam$.

Assets

For each asset we introduce a propositional variable. If the variable is set to true in the PIDL model, this corresponds to the inclusion of the asset in the DOPLER model.

Example 10: For example, $baleAdapter$ means the asset `baleAdapter` is included in the product.

Visibility condition

A visibility condition of a decision is modeled as a propositional formula over Π.

Example 11: In our DOPLER example (Figure 1), the visibility condition of `hydraulicCylinder`, `containsOnly(casterType, slab) && !taperUnit`, is encoded by the following formula:

$$\begin{aligned} &casterType\_slab \wedge \\ &\neg casterType\_bloom \wedge \\ &\neg casterType\_beam \wedge \\ &taperUnit\_No. \end{aligned}$$

Furthermore, we represent the fact that a decision is visible as a variable in Π. For each decision `d`, a variable $Visible\_d$ is introduced.

Example 12: The variable $Visible\_stainlessSteel$ states that `stainlesSteel` is visible to the user.

Lastly, for each decision d, the formula

$$\phi \rightarrow Visible\_d$$

is contained in the constraints C, where ϕ is the formula derived from the visibility condition of `d`. This embodies the fact that whenever the visibility condition of `d` is fulfilled, `d` is visible.

Example 13: From the DOPLER example (Figure 1), we create the following formula denoting if the decision `hydraulicCylinder` is visible to the user:

$$\begin{aligned} &casterType\_slab \wedge \\ &\neg casterType\_bloom \wedge \\ &\neg casterType\_beam \wedge \\ &taperUnit\_No \\ &\rightarrow Visible\_hydraulicCylinder. \end{aligned}$$

Asset inclusion condition

An asset has an inclusion condition indicating if it is part of the final product. It can be translated into a propositional formula over Π.

Example 14: For example, the inclusion condition `containsOnly(casterType, slab)` of the asset `baleAdapter` gives the formula:

$$\begin{aligned} &casterType\_slab \wedge \\ &\neg casterType\_bloom \wedge \\ &\neg casterType\_beam. \end{aligned}$$

For each asset `a`, we derive a formula $\phi \rightarrow a$, where the inclusion condition of `a` is translated into a formula ϕ over Π that is added to the constraints C.

Example 15: To continue the last example, the following formula denotes the respective inclusion condition:

$$\begin{aligned} &casterType\_slab \wedge \\ &\neg casterType\_bloom \wedge \\ &\neg casterType\_beam \\ &\rightarrow baleAdapter. \end{aligned}$$

Moreover, for each asset `a` that includes another asset `b`, the formula $a \rightarrow b$ is contained in C, where a and b are the propositional variables expressing the inclusions of assets `a` and `b` respectively.

Analogously, for each asset a that excludes an asset b, the formula $a \rightarrow \neg b$ is contained in C.

Example 16: In our example, these are the formulas:

$$baleAdapter \rightarrow pCalibthermometer$$

$$calibrator \rightarrow \neg pCalibthermometer.$$

Rules

For each DOPLER rule which has the form

```
if <condition> then <action>,
```

we add a rule transition $\chi_i \rightsquigarrow E_i$ to T_R as follows: We translate the <condition> part of a rule into a formula χ_i over Π. The set E_i then contains the literals that reflect the assignment of values to decisions caused by the rule's <action> part. If true (false) is assigned to a Boolean decision d in <action>, then E_i contains d_Yes ($\neg d\_Yes$) and $\neg d\_No$ (d_No). This is to ensure the consistency of the representation of the Boolean decisions as mentioned above.

Example 17: For example, the rule

```
if !gapChecker then taperUnit = true
```

becomes the rule transition

$$gapChecker\_No \rightsquigarrow \{taperUnit\_Yes, \neg taperUnit\_No\}.$$

If the decision d is an enumeration decision that is assigned an option o, then E_i contains d_o.

Example 18: The rule

```
if molder then setValue(casterType, bloom)
```

becomes the rule transition

$$molder\_Yes \rightsquigarrow \{casterType\_bloom\}.$$

Decisions taken by the user

The user transitions $\chi_i \rightsquigarrow E_i \in T_U$ model decision taking by the users in a DOPLER model. In a user transition $\chi_i \rightsquigarrow E_i$, χ_i states the conditions that must be fulfilled in order to carry out the user decision, E_i contains the changes in the set of decisions after the user taking the decision. For each Boolean decision d, T_U contains two user transitions $\chi_i \rightsquigarrow E_i$ and $\chi_{i+1} \rightsquigarrow E_{i+1}$. χ_i and χ_{i+1} are the same formula

$$Visible\_d \wedge \neg d\_Yes \wedge \neg d\_No,$$

stating that d is visible and has not been taken yet.

Example 19: Consider the user decision stainlessSteel from the example in Figure 1 that is represented by the following user transitions $\chi_i \rightsquigarrow E_i$ and $\chi_{i+1} \rightsquigarrow E_{i+1}$ as follows:

$$\begin{aligned}\chi_i = \chi_{i+1} = &Visible\_stainlessSteel \wedge \\ &\neg stainlessSteel\_Yes \wedge \\ &\neg stainlessSteel\_No.\end{aligned}$$

E_i and E_{i+1} are sets of literals that denote the update of the variables after the transition:

$$\begin{aligned}E_i &= \{stainlessSteel\_Yes, \neg stainlessSteel\_No\} \\ E_{i+1} &= \{stainlessSteel\_No, \neg stainlessSteel\_Yes\}.\end{aligned}$$

User transitions for enumeration decisions are analogously obtained.

In each user transition, we ensure in χ_i that the corresponding decision has not been taken yet. As a consequence, we do not consider user changing decisions. This does not affect the functionality of the semantics being discussed because retracting decisions just means reverting to the state before the decision was taken.

We use rule transitions for DOPLER rule execution and user transitions for user-decision taking. This is reasonable if we look at how the rule engine of DOPLER works as described in Section II: Once the user has taken a decision, it is checked which rules can be triggered. Then the action of the rules whose conditions are satisfied are executed, possibly leading to new checks and executions of rules. When this procedure is over, the user can take the next decision. The user cannot take a decision while the rule engine is operating. We take this into account by considering user transitions that additionally require a state to be rule-terminal as defined in Section III in order to apply the transition to the state.

Initial state

The initial state S_I of the PIDL specification of a DOPLER model consists of all the variables of Π representing the decisions as negative literals.

The reason why we have only negative literals here is that we reflect the fact that in the beginning of the execution of a DOPLER model, nothing is selected yet, i.e., no value is set for any decision and each decision has been taken neither by the user nor by any rule. Also, note that this initial state satisfies the consistency of Boolean decisions, which is then preserved by the transitions.

B. Detecting DOPLER Anomalies

By translating a DOPLER model into PIDL we can use our new calculus to analyze a DOPLER model. We consider the anomalies listed in Section II and explain how the calculus detects them. In the following, we assume a DOPLER model and its corresponding PIDL specification $\mathfrak{S}$ with its state graph $S_{\mathfrak{S}}$ as defined in Section III.

- **Inconsistency:** Consistency properties of the DOPLER model are modeled as formulas in C. Then inconsistency of the DOPLER model corresponds to inconsistency of $\mathfrak{S}$. As one example of such a property, an enumeration decision d has a minimum number and a maximum number of options that can be selected. With the variables of Π, propositional formulas ϕ stating these values restrictions can be derived. These ϕ are then contained in C.

Example 20: The formula

$$\neg(casterType\_slab \wedge casterType\_bloom \wedge casterType\_beam)$$

says that `casterType` cannot have all three values selected at the same time.

- **Incompleteness:** The DOPLER incompleteness test case is expressed as a formula ϕ over Π as the following example depicts.
  Example 21: Consider the incompleteness test case from the DOPLER example in Figure 1: The modeler expects the value of `hydraulicCylinder` to be set automatically, after `stainlessSteel` is assigned a value.
  This is expressed as the following formula

$$\begin{aligned}\phi = & stainlessSteel\_Yes \vee stainlessSteel\_No \\ & \rightarrow \\ & (casterType\_slab \vee casterType\_bloom \vee casterType\_beam) \wedge \\ & (hydraulicCylinder\_Yes \vee hydraulicCylinder\_No).\end{aligned}$$

  Then it is checked if $\mathfrak{S}$ is incomplete with respect to ϕ.
- **Redundancy:** Two DOPLER rules are redundant iff there is a state $S \in V$ such that the two rule transitions that represent these rules are redundant with respect to S.
- **Cyclicity:** A cycle in the DOPLER model is detected by checking if $\mathfrak{S}$ has a cycle.
- **Confluence:** We have confluence in the DOPLER model iff $\mathfrak{S}$ is rule-confluent and user-confluent.
- **Asset Inclusion Conflicts:** As mentioned before, one way to model the inclusion of assets in products by inclusion conditions and includes- and excludes-relationships between assets is to translate them into formulas over Π that are contained in the constraints C. Assets conflicts can then be identified by checking inconsistency of $\mathfrak{S}$.

C. Implementation

We made a first implementation of the PIDL framework to see how it could be used in practice. The tool takes DOPLER models as inputs and checks them for anomalies. It translates a DOPLER model to elements of PIDL, creating a specification as described in the previous subsection. A state is the current truth assignment of the variables corresponding to the DOPLER decisions. Inconsistency of the states and transitions to new states are then determined by superposition-based SAT solving, following the calculus in Section III. The state graph of the specification is produced, which is used to detect graph-based properties such as cyclicity by standard graph algorithms. Our first prototypical implementation does currently not contain the confluence check.

Table I shows the results of running our implementation on the example in Figure 1, displaying what kind of anomalies were found. All in all, 99 states were created during the run, which took 0.037 seconds on an Intel Xeon E5-4640 running at 2.4 GHz and 512 GB of RAM. The program detected 12 inconsistent states. Incompleteness was found for 7 states. Out of the 99 states, 10 states showed rule redundancy and in 12 cases there were conflicts in the asset inclusions. The mentioned cycle in the DOPLER model example was identified.

Additionally, we ran the implementation on a set of randomly generated DOPLER models. We used models with 20, 60 and 100 Boolean decision variables respectively. Each model contains a set of random rules according to the predefined fixed form `if (d||[!]e&&[!]f) then g = [true/false]`, where `d`, `e` `f` and `g` are pairwise distinct Boolean decision variables, with `e` and `f` possibly being negated. The number of the rules are such that we have a ratio of 1:1.5 between variables and rules. The visibility of decisions is organized such that at most one half of the variables are visible, but visible variables are not allowed to appear on the action sides of the rules. This is to ensure that the rule's contribution to the states generation is not diminished. Consequently, each generated model may differ in the number of visible decisions. We furthermore added random constraint clauses of the form `([!]d||[!]e||![f])` to get more realistic examples. Without any constraints, the models would amount to a mere enumeration of reachable states. We used a ratio of 1:1 between variables and constraints. For each of the three model sizes, we generated 20 instances.

A representative part of the results is shown in Table II. The full table with all the experiments is contained in the additional material [3]. If no inconsistency with respect to the random constraints can be found, a triple is shown indicating how many states were generated, if a cycle was detected (Y) or not (N) and the number of redundant rules applications. For example, the result 211/N/8 of the model rnd_10 means that a state graph with 211 states was created, there was no cycle and there were 8 cases of redundant rules applications. Consistent models only occurred with 20 variables. Although most models are still solved in a relatively short time, one can see that problems get harder with rising numbers of variables. In the group of models with 60 variables instances that required up to several minutes run-time can be found, whereas most of the examples with 20 variables stayed under one second. Finally, when dealing with 100 variables, we see two cases in the table where the run was aborted by the system after approximately 12 minutes (two more cases not shown in this selection).

The potential search space has 3^{v+a} states, where v is the number of visible decisions and a is the number of decisions occurring on the action sides of the rules (in our experiments mostly $a = n - v$ with n being the number of variables). Improvements to this first implementation can reduce the search space. This could be done by taking invariants among the states into account and by considering similarities and dependencies between them. Nevertheless, it can be seen that PIDL can in

TABLE I
ANOMALIES IN THE DOPLER MODEL EXAMPLE.

	Number of States
Total	99
Inconsistency	12
Incompleteness	7
Redundancy	10
Cycle	*detected*
Asset Inclusion Conflicts	12

TABLE II
GENERATED RANDOM DOPLER MODELS.

20 variables, 30 rules			
Name	Visible Variables	Time	Results
rnd_1	5	0m0.05s	inconsistent
rnd_6	6	0m0.04s	inconsistent
rnd_7	3	0m0.03s	inconsistent
rnd_9	3	0m0.09s	inconsistent
rnd_10	2	0m0.26s	211/N/8
rnd_11	3	0m0.04s	inconsistent
rnd_13	5	0m0.20s	inconsistent
rnd_15	3	0m0.02s	inconsistent
rnd_19	3	0m0.61s	558/Y/240
rnd_20	4	0m0.43s	inconsistent
60 variables, 90 rules			
Name	Visible Variables	Time	Results
rnd_23	11	0m4.38s	inconsistent
rnd_24	12	0m2.84s	inconsistent
rnd_25	13	7m44.81s	inconsistent
rnd_27	14	0m0.38s	inconsistent
rnd_29	15	0m0.51s	inconsistent
rnd_31	15	0m0.70s	inconsistent
rnd_33	11	0m1.01s	inconsistent
rnd_35	12	0m36.00s	inconsistent
rnd_38	14	0m0.40s	inconsistent
rnd_39	11	2m13.94s	inconsistent
100 variables, 150 rules			
Name	Visible Variables	Time	Results
rnd_42	24	>12m	-
rnd_45	18	5m59.85s	inconsistent
rnd_47	20	0m1.51s	inconsistent
rnd_48	22	0m2.84s	inconsistent
rnd_50	19	0m42.68s	inconsistent
rnd_51	26	>12m	-
rnd_52	28	0m16.12s	inconsistent
rnd_53	21	0m1.28s	inconsistent
rnd_55	18	0m1.48s	inconsistent
rnd_59	21	0m1.13s	inconsistent

fact be turned into a useful software system for the practical analysis of rule-based systems.

VI. RELATED WORK

Verification of configuration knowledge bases has been tackled by many researchers on varying levels of details and granularity. Yang et al. [7] present an approach based on petri nets, where all rules are first normalized into Horn clauses and transformed to petri nets.

Verification of models for product line engineering (typically feature models) have also been intensively studied in literature. Some approaches verify development artifacts [8] and some others verify that the variability specified by a feature model is correctly implemented in code [9]. Verification of the models themselves has been studied by Post and Sinz [10], where the authors describe the variants of the product line using a meta-program. All these approaches follow a constraint-based approach. Our approach deals with rules, which are easy to specify for the modelers but rather complex to verify and maintain.

Logical representation of feature models has been previously discussed by Czarnecki et al. [11]. Other analysis techniques available for product line models include approaches based on SAT solvers [12], atomic sets [13], BDDs [14] and CSPs [15] etc. The primary difference between all these contributions and our work is that these approaches do not consider the interactive nature of the configuration process and the rule-based specification (as opposed to constraints) of restrictions on the models.

In the context of propositional logic there are various extensions to propositional logic with time [16] or dynamic propositional logic [17] also based on superposition [18], [19]. For these logics there exists a variety of modern proof calculi. However, they do not directly support our transition semantics via language constructs. Nevertheless, the implementation techniques used for these logics have also potential for improving our current prototypical implementation.

Specific to our approach is the support of rules of the from $A \wedge \phi \rightsquigarrow \{\neg A, \ldots\}$ enabling revision of a decision. Such rules cannot be modeled in many of the aforementioned approaches or would lead to an inconsistency. Unique to PIDL is the concept of rule-terminal states that are a prerequisite for some rules (in case of DOPLER user decisions) to be applied.

VII. CONCLUSIONS

In this paper, we have defined the new logic PIDL that provides detailed models for rule-based configuration systems. In particular, it supports decision revision as expressed by rules of the form $A \wedge \phi \rightsquigarrow \{\neg A, \ldots\}$ and the concept of rule-terminal states. In addition, we provide a sound and complete calculus for PIDL that is based on the ideas of superposition. This calculus constitutes a decision procedure that analyzes the following properties of rule-based systems: inconsistency, incompleteness, redundancy, absence of cycles, confluence and conflicts of asset inclusion.

We have presented the automatic translation of DOPLER models to PIDL. DOPLER is a rule-based configuration system currently in use at Siemens. Furthermore, we showed by a first prototypical implementation that PIDL can in fact be turned into a useful software system for the practical analysis of rule-based systems.

ACKNOWLEDGMENT

This work was partly supported by Siemens. We would like to thank Martin Suda for his helpful advice and fruitful discussions.

REFERENCES

[1] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study," *Autom. Softw. Eng.*, vol. 18, no. 1, pp. 77–114, 2011.

[2] D. Dhungana, P. Heymans, and R. Rabiser, "A formal semantics for decision-oriented variability modeling with DOPLER," in *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, ser. ICB-Research Report, D. Benavides, D. S. Batory, and P. Grünbacher, Eds., vol. 37. Universität Duisburg-Essen, 2010, pp. 29–35.

[3] D. Dhungana, C. H. Tang, C. Weidenbach, and P. Wischneswki, "Automated verification of interactive rule-based configuration systems (additional material)," *CoRR*, vol. abs/1309.0065, 2013.

[4] L. Bachmair and H. Ganzinger, "Resolution theorem proving," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Elsevier, 2001, vol. I, ch. 2, pp. 19–99.

[5] R. Nieuwenhuis and A. Rubio, "Paramodulation-based theorem proving," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Elsevier, 2001, vol. I, ch. 7, pp. 371–443.

[6] C. Weidenbach, "Combining superposition, sorts and splitting," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Elsevier, 2001, vol. 2, ch. 27, pp. 1965–2012.

[7] S. J. H. Yang, J. J. P. Tsai, and C.-C. Chen, "Fuzzy rule base systems verification using high-level petri nets," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 2, pp. 457–473, Feb. 2003.

[8] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 269–280.

[9] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, Eds. ACM, 2006, pp. 211–220.

[10] H. Post and C. Sinz, "Configuration lifting: Verification meets software configuration," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE, 2008, pp. 347–350.

[11] 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*. IEEE Computer Society, 2007, pp. 23–34.

[12] M. Mendonça, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 231–240.

[13] S. Segura, "Automated analysis of feature models using atomic sets," in *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, S. Thiel and K. Pohl, Eds. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 201–207.

[14] W. Zhang, H. Yan, H. Zhao, and Z. Jin, "A BDD-based approach to verifying clone-enabled feature models' constraints and customization," in *High Confidence Software Reuse in Large Systems, 10th International Conference on Software Reuse, ICSR 2008, Beijing, China, May 25-29, 2008, Proceedings*, ser. Lecture Notes in Computer Science, H. Mei, Ed., vol. 5030. Springer, 2008, pp. 186–199.

[15] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615 – 636, 2010.

[16] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[17] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 194–211, 1979.

[18] M. Suda and C. Weidenbach, "A PLTL-prover based on labelled superposition with partial model guidance," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 537–543.

[19] ——, "Labelled superposition for PLTL," in *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, ser. Lecture Notes in Computer Science, N. Bjørner and A. Voronkov, Eds., vol. 7180. Springer, 2012, pp. 391–405.