Logics for Rule-Based Configuration Systems

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science of
Saarland University

Ching Hoo Tang

Saarbrücken
February 2017

Date of the colloquium:	16 August 2017
Dean:	Prof. Dr. Frank-Olaf Schreyer
Examiners:	Prof. Dr. Christoph Weidenbach
	Prof. Dr. Andreas Herzig
Chair:	Prof. Dr. Jörg Hoffmann
Academic assistant:	Dr. Andreas Nonnengart

Abstract

Rule-based configuration systems are being successfully used in industry, such as DOPLER at Siemens. Those systems make complex domain knowledge available to users and let them derive valid, customized products out of large sets of components. However, maintenance of such systems remains a challenge. Formal models are a prerequisite for the use of automated methods of analysis. This thesis deals with the formalization of rule-based configuration. We develop two logics whose transition semantics are suited for expressing the way systems like DOPLER operate. This is due to the existence of two types of transitions, namely user and rule transitions, and a fixpoint mechanism that determines their dynamic relationship. The first logic, PIDL, models propositional systems, while the second logic, PIDL+, additionally considers arithmetic constraints. They allow the formulation and automated verification of relevant properties of rule-based configuration systems.

Zusammenfassung

Regelbasierte Konfigurationssysteme werden erfolgreich in der Industrie benutzt, wie etwa DOPLER bei Siemens. Diese Systeme machen komplexes Domänewissen Anwendern verfügbar und erlaubt es ihnen, gültige, angepasste Produkte aus großen Mengen von Teilen zu erstellen. Allerdings bleibt die Wartung solcher Systeme eine Herausforderung. Formale Modelle sind Voraussetzung für den Einsatz automatisierter Analysemethoden. Diese Arbeit beschäftigt sich mit der Formalisierung von regelbasierter Konfiguration. Wir entwickeln zwei Logiken, dessen Transitionssemantiken dafür geeignet sind, die Art und Weise, auf der Systeme wie DOPLER operieren, abzubilden. Dies ist möglich durch die Existenz von zwei Typen von Transitionen, nämlich User- und Regeltransitionen, und eines Fixpunktmechanismus, der ihre dynamische Beziehung bestimmt. Die erste Logik, PIDL, modelliert propositionale Systeme, während die zweite Logik, PIDL+, zusätzlich arithmetische Constraints betrachtet. Sie erlauben die Formulierung und automatische Verifikation relevanter Eigenschaften regelbasierter Konfigurationssysteme.

Acknowledgments

I want to thank my supervisor Christoph Weidenbach. I am very grateful he gave me the opportunity to join this great and highly pleasant group, and for his guidance and support he has offered throughout my time. Moreover, I want to thank Patrick Wischnewski for his generous help and advice, especially in the beginning. This work was partly funded by Siemens, and I would like to express my gratitude to Siemens for the scholarship and the collaboration. I would like to thank Deepak Dhungana from Siemens for his work in the collaboration and for always being willing to help with matters concerning configuration. I thank Martin Suda, whose work on superposition of labeled clauses provided the right inspirations for PIDL. I want to thank the Max Planck Institute for Informatics and its staff for being my academic home and making my work possible. Thank you very much to Jennifer Müller, who has always been a very kind and helpful colleague. A big thank you to the current and former members of the Automation of Logic group for their help and the many nice discussions and conversations we had. And lastly, I want to thank my mother for raising me and supporting me my entire life.

Contents

Chapter 1

Introduction

Rule-based configuration systems have their roots in classical *expert systems* (Jackson, 1999), one of the most prominent manifestations of the work connected to artificial intelligence. The purpose of these systems is to represent knowledge about a certain domain and then do reasoning with this knowledge in order to help humans with decision making in that domain. The automation and precision delivered by these systems is supposed to lead to a substantial reduction in human effort and errors when performing tasks involving large and complex amounts of data. Rule-based systems derive decisions by executing rules of the form `if <condition> then <action>`. A famous, early example is the R1/XCON system from the 1970s by Bachant and McDermott (McDermott and Bachant, 1984), which was used for the configuration of computer systems at DEC, saving the company considerable amounts of money. *Product configuration* is about finding the right combination of a predefined set of parts to get a product that satisfy the constraints of the respective domain and at the same time the preferences stated by the user. Under the notion of *mass customization* (Davis, 1989; Pine, 1993), the goal is to allow variability of products to meet the individual needs of customers while having fast production cycles using fixed sets of product components. Today, many configuration systems are in use. The most visible examples include configurators for cars and computers on the consumer side, but systems based on the same principle are also successfully being employed in industry. Rule-based configuration allows the encoding of constraints as rules in an easy and intuitive way. Examples of industrial rule-based configuration systems can be found at Siemens, where the DOPLER system (Dhungana and Grünbacher, 2008; Dhungana et al., 2011) is used, in particular in the domain of steelworks configuration, and LEEGOO (Struck, 2012), in the context of gas-turbine production.

Configuration systems such as DOPLER are interactive in nature, and the configuration flow of a *product derivation* is typically composed of the following steps: First, a user chooses an option with respect to a variable of the system by assigning one of the allowed values to it. This assignment is then matched with the condition parts of the rules. If a condition of a certain rule is satisfied by the current assignment state of the variables, the rule is fired. A fired rule can set further variables as defined in its action part. This potentially causes other rules to be executed. If all the activated rules have been applied and no changes due to rules can happen any more, where we also say that a *fixpoint* is reached, the user can set the next variable.

The drawback of those systems is that, due to interdependences between rules, with increasing size they become more difficult to maintain, with possible effects on the consistency of the rule bases. For example, one would not typically want to have rules that contradict each other during execution, or that the resulting product depends on the order in which the rules are executed, that is, the system is not *confluent*. The complex-

ity is given by the size and the combinatorics that arise with it, making monitoring and correcting those systems manually infeasible.

The solution is to automate the verification process, where the first step is having a clear semantics of the configuration system one wants to examine. Configuration systems are typically restricted to technical domains, which makes them amenable to translating them into formal logics. Expressing the systems formally enables the verification of important properties such as consistency and confluence. However, the use of formal descriptions that cover the operational semantics is still something that is rarely found among rule-based configuration systems.

This thesis presents work aimed at providing a formal framework for rule-based configuration. Motivated by DOPLER and LEEGOO, we have developed logics to express and formalize such systems in a more comprehensive way, while preserving decidability in order to have practicability.

The first logic is PIDL (Propositional Interactive Dynamic Logic) (Dhungana et al., 2013) for configuration systems whose variables are of type Boolean. The design of PIDL is inspired by a superposition framework sketched by Martin Suda, which describes change of states represented by sets of propositional literals (Suda, 2011). The framework's calculus of *labeled clauses* built on superposition (Bachmair and Ganzinger, 1994, 2001) contains a rule that corresponds to the change of worlds by virtue of transitions. PIDL adapts and extends those concepts in a way so that the resulting system can be used for the representation of rule-based configuration systems.

We can represent the relevant elements of a particular instance of a configuration system with a PIDL *specification*. The possible-worlds semantics of the logic has state transition systems as interpretations of specifications. Starting with the initial state, all states are considered that can be reached from it through transitions. States in PIDL contain propositional literals, describing facts that hold in the states, and correspond to states in the configuration process. Transitions between PIDL states embody changes happening during a product derivation. They consist of a condition part and an update part. If the condition part is entailed by a state S, there is a transition from S to a state S', where S' is created by updating S with the update part of the transition. The update mechanism allows new facts to replace old facts completely, giving flexibility in the modeling of configuration instances. In PIDL, we distinguish between two types of transitions: *User transitions* represent actions by the user, *rule transitions* express actions caused by rules. The semantics only allow a user transition from a state S to another if the state in question is *rule terminal*, that is, if no more rule transitions whose actions change S can be applied to S. This unique feature of PIDL allows us to faithfully represent the usage dynamics of interactive configuration systems as described above, including the fixpoints of the product derivations.

Within PIDL we can then express properties relevant to the consistency of configuration systems. For example, properties specific to the domain are written as propositional formulas in a set of constraints, which must be satisfiable in any state. With decision procedures exploring the structure given by the interpretations, further properties such as confluence or cyclicity can be verified.

We also give a sound and complete calculus based on superposition that shows how partial interpretations can be computed out of a specification. To test our approach, we made a first implementation containing the interpretation construction and procedures to check graph-based properties like the presence of cycles. We ran the implementation on a small DOPLER instance and a number of randomly generated examples, which in principle showed that PIDL offers a possible useful way to formalize and verify configuration systems.

The second logic is PIDL+ (Tang and Weidenbach, 2016), which is an extension to PIDL by including arithmetic constraints. This makes it possible to also model configuration systems dealing with arithmetic variables, such as LEEGOO. The base logic for the formulation of states and transitions is propositional logic plus a fragment of the theory of reals, consisting of expressions of the form $x \circ t$, where x is a variable, $\circ$ is one of the usual comparison operators, and t is a term containing integers and the known mathematical operators. However, we consider *admissible specifications* of PIDL+ in which all variables that are relevant to the transition flow, that is, they appear in the transition conditions and updates, are interpreted as bounded integers. This enables the logic to have a finite number of possible states and thus maintain decidability. Bounded variables are also typical in real-world configuration systems. Variables that do not affect transitions may be unbounded and of type real, functioning as result holders for the states they are in.

The presence of numerical decisions necessitates a different semantics. A direct way would be a view in which each variable mentioned in a state has a value, and whether transitions can be applied to the state depends on these values and the propositional literals of the state. However, this *small-steps* semantics does not lend itself well to computations as the number of possible states becomes the product of all the variables ranges. Instead, we consider a *big-steps* semantics where bounds in states represent a range of variable instantiations. For example, a state having the bounds $x \geq 2$ and $x \leq 49$ stands for all the worlds in which the variable x has an integer value between 2 and 49. This kind of aggregation offers a more compact representation of the up to exponentially many states in the small-steps semantics.

PIDL+ also associates each state with a set of *user variables*, which is a set that contains the variables that have been set by the user. It can change during product derivation, because domain constraints might require previous user decisions to be overwritten under certain circumstances. The transition semantics is based on one quantifier alternation with respect to the variables: For a transition to be applied, there must exist one instance of the user variables so that for all instances of the other variables, which are set by the system's rules, the condition of the transition is satisfied. We partition the set of instances represented by the bounds of a state into *selections* so that a selection entail certain transition conditions, and those transitions are then executed with respect to the respective selections. In particular, the concept of rule termination is adjusted to selections: A user transition can only be applied to a state if the state has a rule-terminal selection with respect to that user transition. Selections always exists because the application of transitions only depend on bounded integers. We also provide a sound and complete algorithm to derive a partial interpretation of an admissible PIDL+ specification, and a procedure to determine the rule-terminal selections needed to apply user transitions.

The thesis is structured in the following way. Chapter 2 is a brief introduction of rule-based configuration in general and the DOPLER system in particular. Chapter 3 contains basic definitions and notions that we use in this thesis. The two logics PIDL and PIDL+ are then presented in Chapter 4 and 5 respectively. Chapter 6 forms the conclusions of the thesis.

Related Work

Configuration has been the subject of research in various ways, where different approaches and techniques have been used. There exist also logics that can be associated with the analysis of configuration systems. The methods can be roughly divided into

two classes. The first class has a static view on configuration. It focuses on describing the components and constraints in a declarative way, possibly stressing the hierarchical organization of products and their features. The usual question to ask then is that given a user preference, does it make a valid configuration possible? The second class considers the dynamic aspects of configuration. There, one is interested in how configuration states can change during the product derivation process. This is in particular true of rule-based configuration, where the focus is on how systems behave under the execution of rules.

One common static approach is *constraint satisfaction* (Tsang, 1993), where configuration tasks, that is, the goal of finding a valid configuration satisfying constraints and user choices, are seen as *constraint satisfaction problems* (CSP) (Mailharro, 1998; Fleischanderl et al., 1998). A CSP features a set V of variables, where each variable has a finite domain of possible values, and a set C of constraints over the variables that have to be satisfied by an assignment of the variables. Constraint satisfaction offers a way to describe product artifacts and their dependencies in an intuitive way and comes with a semantics readily available to represent the task of finding valid configurations. Then, in the basic form, variables of the configuration system to be modeled are typically contained in V, while user requirements and constraints are elements of C. Solutions to the CSP, for which many search algorithms exist (Russell and Norvig, 2010), are thus the wanted configurations. Verification of configuration models based on constraint satisfaction, and other static approaches mentioned below, can be done by incrementally computing inconsistent subsets of the constraint sets, of which Felfernig et al. give an overview (Felfernig et al., 2014). Many other solutions based on CSPs in particular check the current configuration after each input by the user, making sure the user is guided to a valid configuration (Amilhastre et al., 2002; Rosa et al., 2009; Behjati and Nejati, 2014).

Answer set programming (ASP) (Marek and Truszczyński, 1999; Niemelä et al., 1999) has also been used to express configuration data and constraints in a declarative way. There, knowledge is encoded in programs consisting of sets of rules of logic programming. Those programs are then first grounded, that is, substituting component constants for variables, and then given to ASP solvers, which are typically based on methods for the Boolean satisfiability problem (SAT) and search for configurations satisfying the defined constraints. Soininen and Niemelä identified configuration as one of the first applications of ASP (Soininen and Niemelä, 1999).

Another popular concept are *feature models* (Kang et al., 1990), whose use as representations of configuration systems have been analyzed, for example, by Czarnecki (Czarnecki and Wasowski, 2007; Czarnecki and Pietroszek, 2006). A feature model is a graphical model in which the variability of a product is expressed as features and organized in a tree that highlights the hierarchical relationships between the features of the product. Such relationships can be "mandatory" or "optional", and determine which child features a parent feature include as well. Additionally, there can be further relationship constraints, in which features can "require" or "exclude" other features without having a direct parental link in the feature tree. The semantics and verification of feature models can be based on CSPs (Benavides et al., 2010), SAT (Mendonça et al., 2009), binary decision diagrams (Zhang et al., 2008), and atomic sets (Segura, 2008).

Description logics (DL) are a class of logics used for knowledge representation in general, and in particular are well known for their application in the area of the Semantic Web. In DL, *individuals* are grouped into *concepts* and *roles* describe relationships between them. Felfernig et al. discussed using DL to express configuration (Felfernig et al., 2003). In this context, representation again concentrates on the hierarchical

structure of products. McGuinness considered applying DL for on-the-fly verification, where each user input is verified during the configuration process (McGuinness, 2003).

DL subsume the *Unified Modeling Language* (UML) (Booch et al., 2005), a standard to describe software systems in a visual way, whose widespread use has inspired its adaptation also in the area of configuration (Felfernig et al., 2000). UML diagrams display the components of a product in hierarchies, depicting generalizations and multiplicities. Constraints can be formulated as elements of the diagrams or as external requirements. Aspects of the semantics of UML diagrams can be formalized as a decidable subset of first-order logic (Felfernig, 2007).

While the above approaches manage to formalize product configuration to a certain degree, they take a rather static perspective. They can mainly express the structural configuration data and the requirements, but lack the inherent ability to describe things that happen during entire configuration processes, which is an essential goal of the work of this thesis. Thus, logics are of interest with which one can tackle change within systems.

Logics generally exist that deal with dynamic aspects of systems. They can be primarily found in the family of *modal logics* and its dialects. These logics extend classical propositional logic with modal operators to express notions of change. Temporal logics (Pnueli, 1977; Clarke and Emerson, 1982) are classic examples, upon which the technique of model checking (Clarke et al., 1999) is built. Model checking was used, for example, by Lauenroth et al. to verify whether combinations of product components fulfill certain properties (Lauenroth et al., 2009), and Classen et al. to check properties of product line families (Classen et al., 2010).

Another branch of modal logic that formalizes change and that is of particular interest is *Propositional Dynamic Logic* (PDL), with the initial purpose of describing the behavior of programs (Fischer and Ladner, 1979). The logic considers constructs that are based on programs and the usual logical statements. The semantics involves Kripke structures that express the evolution of states during program execution. Balbiani et al. discussed a variant in which the programs are only based on Boolean assignments (Balbiani et al., 2013). Sinz proposed the use this special kind of PDL to formalize the rules and variables of a configuration system (Sinz, 2004). The models obtained are then further translated into propositional logic, where SAT solving can be used to verify properties of the system, including the confluence of rules. The main difference to our approach is that user preferences come as a priori assignments, which are then processed by the formalism, instead of considering the configuration process from the beginning as an interplay of user input and rule effects. Additionally, the rules are not allowed to have a cyclic behavior, and rule actions are restricted to Boolean assignments.

In PIDL and PIDL+, the distinction between user and rules resembles a system of two entities or *agents* bringing about change over variables of a given system. Numerous modal logics deal with statements about agents and the effects of their behaviors. For example, *Coalition Logic of Propositional Control* (CL-PC) (van der Hoek and Wooldridge, 2005) allows to express that a coalition of agents has the ability to make a certain goal formula true by setting the propositional variables they control accordingly, provided the variables set by the other agents remain unchanged. Herzig et al. showed how PDL with Boolean assignments can embed CL-PC (Herzig et al., 2011). *Alternating-time Temporal Logic* (ATL) combines temporal logic with action modality of agents, where one can talk about agents being able to achieve a goal, expressed as a temporal-logic formula, no matter how the other agents act (Alur et al., 2002). Belardinelli and Herzig analyzed the relationship between CL-PC and ATL, showing that CL-PC resembles the semantic structure of ATL (Belardinelli and Herzig, 2016). Playing a prominent role in particular

in the analysis of multi-agent systems, these logics are rather less suitable for the type of operational semantics given by the configuration rule bases we consider.

A framework that does show a certain similarity to one aspect of PIDL+ are Boolean games (Harrenstein et al., 2001; Bonzon et al., 2006), which model contexts arising in game theory (Osborne and Rubinstein, 1994). Basically, in a Boolean game players control disjoint sets of propositional variables. Each player i has an associated formula ϕ_i over the entire set of variables. The goal of each player i is to set the variables they control so that their formula ϕ_i becomes true. A variable assignment of player i is called *winning strategy* if ϕ_i becomes true irrespective of the assignments of the other players. In PIDL+, selections are defined in Definition 5.8 as integer intervals whose values, contained in the substitution σ, fulfill the condition

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

where the existence of selections determines whether a transition with the condition $\Lambda \wedge F$ can be applied. The abstract form of the condition,

$$\exists\vec{x}_u\forall\vec{y}\phi$$

can be seen as a Boolean game with two players: The user controls the variables given by $\vec{x}_u$ and the system controls the variables $\vec{y}$, with the user trying to make ϕ true. This is possible because the relevant variables of admissible specifications are over bounded integers, which means we can consider the corresponding assignments as propositions. Then, the quantifier alternation in the condition implies that the existence of a transition is connected to the existence of a winning strategy for the user. Despite this similarity, the focus of Boolean games is different from that of the analysis of rule-based configuration, as they rather lay emphasis on game-theoretical concepts such as equilibria between players.

The update of propositional knowledge has been discussed in the literature (Herzig et al., 2013). Our logics define an update scheme of the transitions that essentially can be seen as a special case of the update used by Winslett (Winslett, 1990), with the involved original and action formulas just being conjunctions of literals.

All in all, the methods and logics mentioned above differ from PIDL and PIDL+ insofar as they lack direct language support for modeling configuration systems and their whole dynamics as discussed in this thesis. Rather than treating partial aspects of systems, our logics can comprehensively represent the user-rules interactivity by having user and rule transitions, and using the concept of rule terminal states. The update schemes of the transitions also allows modifying facts by replacing complementary literals. In PIDL+, arithmetic constraints are additionally incorporated in the system with the help of selections.

Main Contributions

In this thesis, we formulate the logics PIDL and PIDL+ for formalizing rule-based configuration systems. The main contributions connected with those logics can be described as follows.

1. The semantics of PIDL and PIDL+ represent the dynamic behavior of rule-based configuration systems: The interaction process of user and system is modeled on the basis of two different types of transitions, which are user transitions and rule transitions. A user transition can only happen at a state if the state is rule terminal, that is, all rule transitions that can be applied to the state do not change

the state, or, in other words, the state is a fixpoint with respect to rule transition applications (Definitions 4.11 and 5.28). This is motivated by rule-based configuration in practice, in particular as used by Siemens. PIDL models systems containing propositional constraints (Definition 4.1), while PIDL+ is an extension that incorporates arithmetic constraints (Definition 5.1).

2. PIDL+ has a compact representation of arithmetic decisions: Bounds on variables aggregate up to exponentially many instances of variable assignments by the user (Section 5.2.1). Selections group decisions according to transitions they entail and are an integral part of the transition updates, making sure previous decisions are correctly preserved (Definition 5.8).

3. Calculus and algorithms: We provide a sound and complete superposition-based calculus for basically constructing the state graph of a PIDL model (Section 4.3), and we give sound and complete algorithms for computing the state graph of a PIDL+ model and for computing the rule-terminal subselections needed for applying user transitions (Section 5.3).

4. Important properties of rule-based configuration systems are expressible: Properties concerning the consistency of modeled configuration systems can be formulated as properties of PIDL and PIDL+ specifications (Sections 4.4 and 5.4). The properties considered in the thesis are in particular

   - soundness, which is about satisfying the constraints of the domain of the corresponding configuration system,
   - completeness, which is about valid configurations being actually configurable within the system,
   - cyclicity, which is about cyclic behavior of the rule base during execution, and
   - confluence, which is about configuration results being independent of the order of decisions taken and rules applied.

5. The properties are decidable: All the mentioned properties can be decided because they can be based on the finite state graphs of the PIDL and PIDL+ models representing the systems (Theorems 4.40 and 5.44).

Chapter 2

Motivation

Given a set of predefined components, product configuration is concerned with identifying the combinations of the components to build a product that satisfies the preferences of the end user and a number of back-end constraints, which are determined by factors like technical restrictions or sales/marketing aspects. The use of configuration systems is a success story and has made it possible to reconcile individualization with competitive production and pricing, following the notion of mass customization. In the design process of a configuration system, the domain knowledge about the products are translated into a *configuration model*, or *variability model*. Based on that model, tools that make up the final configuration system are developed that are eventually made available to the user for individual configuration. The use of a configuration system to derive a product is called a *configuration process* or *product derivation*. We overload the term "configuration" by using it to refer to the final product as well as the process to get to that product.

In this thesis, we focus on rule-based configuration, that is, systems that use `if-then` rules to reflect the domain constraints. We first establish the general context of rule-based configuration systems that we work with in Section 2.1. Then, we look at an example of such systems, the DOPLER configuration system, in Section 2.2.

2.1 Rule-Based Configuration

One of the earliest examples of configuration systems are rule-based expert systems such as R1/XCON (McDermott and Bachant, 1984), whose development started in the late 1970s. Since then, different approaches to model configuration have been used, with feature models being the most prominent one. Rule-based systems such as DOPLER (Dhungana and Grünbacher, 2008; Dhungana et al., 2011), which is described in the next section, emphasize the perspective of the user and their interaction with the system. Figure 2.1 is a high-level illustration of the structure of a rule-based configuration system.

A system as shown in the figure accepts user input, which are assignments to variables of the system. Whenever a user sets the value of a variable, the condition parts of the rules are evaluated. If the condition of a rule evaluates to true due to the current variables assignments, the rule can be fired, where it can carry out further variable assignments as defined in its action part. The components that are included in the final product are determined by the final values of the variables.

Operational semantics of rules. There are different ways in which rule bases can operate. In particular, the question arises as to which rules to execute if more than one

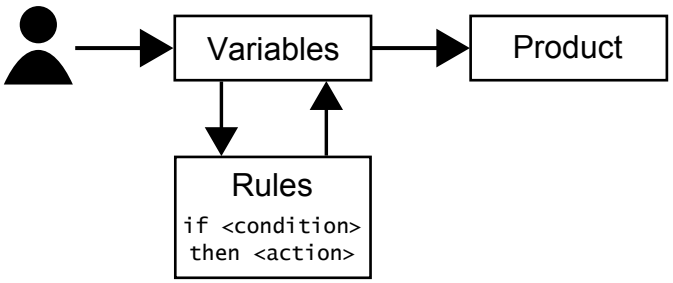

Figure 2.1: A rule-based configuration system in use.

is satisfied by the current variable state. In general, *conflict resolution* strategies deal with this issue by selecting rule executions according to certain heuristics. In this thesis, we consider systems that have the following operational rule semantics: After the user has entered their input, *all* entailed rules are executed. In addition, a rule's action can entail other executions of other rules. This loop of rule applications continues until all rules that are directly and indirectly implied by the user action have been fired. After that the user can assign the next variable. One consequence of this semantics is that it is usually wanted that the order of rule executions does not affect the result. This is called the *confluence* property and is discussed later in more detail.

2.2 The DOPLER Configuration System

Originating from the Johannes Kepler University Linz, DOPLER is a rule-based configuration system developed and used at Siemens. It is applied successfully in the steel-plant production domain, but has proven its versatility in various areas (Dhungana et al., 2011). DOPLER is also the system that we use as a starting point for our investigations. In this section, we give a compact description of the DOPLER system by using a small example of a DOPLER configuration instance. This is done on a reasonably abstract level, and we focus on the elements that are relevant to our work, which is the formulation of a logic for the configuration system, enabling automated analysis and verification. For example, we are more interested in aspects that are responsible for the functioning of the configuration process or for errors that might occur, and we are less interested in elements that have only informative character, such as messages to be displayed to the user. We refer to the original sources for more details (Dhungana and Grünbacher, 2008; Dhungana et al., 2011). At the end of this chapter, we highlight typical inconsistency problems that can occur in configuration models, again with the help of the DOPLER model example.

2.2.1 The Structure of DOPLER

Figure 2.2 shows a small example of a DOPLER configuration instance. The following core elements are used in the DOPLER modeling language:

- *Decisions*, which are the variables of the instance,
- *rules*, which describe the effects of variables assignments, and
- *assets*, which are the product components.

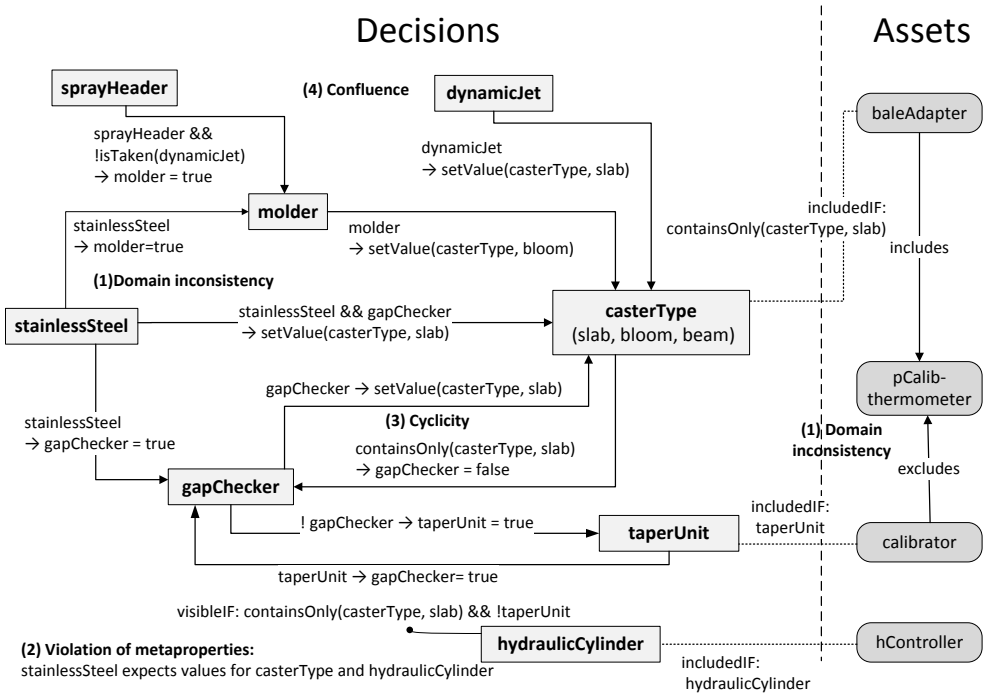

Figure 2.2: A DOPLER instance with decisions, assets and dependencies among them (rules, visibility conditions and asset inclusion conditions).

Configuration variables are called decisions in DOPLER, and the act of setting the value of a variable is also referred to as *taking a decision*, an expression that we also use to mean the assignment of configuration variables in general. There are different types of decisions, each of them indicating the range of possible values that can be assigned to the respective variable. In our example, we have two types of decisions:

- Boolean decisions are decisions that can have one of the values true or false. For example, `stainlessSteel` or `molder` are Boolean decisions in the configuration model of Figure 2.2.

- Enumeration decisions are decisions that have a fixed range of predefined values, offering a number of selectable options. For example, `casterType` is an enumeration decision that can take the values `slab`, `bloom`, or `beam`. Decisions of this type usually have a cardinality attached to it, which states the minimum and the maximum number of values that can be assigned to the decision at the same time. `casterType` has a cardinality of 1:1, that is, only one value can be selected.

Sometimes it is necessary to prevent a user from taking a decision, depending on the current assignments of the decisions taken so far. Each decision therefore has a *visibility* property, which is an indicator of whether a decision is visible to the user. A user can only take decisions that are visible. The visibility condition of a decision is an expression that evaluates to a Boolean value and that tells the system if the decision is visible or not. In the model, the decision `hydraulicCylinder` is visible if the condition

```
containsOnly(casterType, slab) && !taperUnit
```

is true, that is, if `casterType` has only the value `slab` and `taperUnit` is false. If that is not the case, it is not wanted that the user can decide on `hydraulicCylinder`, which determines the inclusion of the asset `hController`, hence `hydraulicCylinder` is not visible with the condition being false. The inclusion of assets, that is, product components, is described later below. A decision can also be visible by default. Then, its visibility condition is equivalent to true. This holds for the rest of the decisions in the variability model depicted in Figure 2.2.

DOPLER uses language constructs in a Java-style fashion to create Boolean expressions that form the elements important to the configuration process, such as the visibility condition mentioned above. `containsOnly` is an example of a function that can be part of those expressions. The set of functions include:

- `containsOnly(decision d, option o)`
  Boolean function that returns true if option `o`, and nothing else, is selected for enumeration decision `d`, else false is returned.
- `containsAny(Decision d, option` o_1`, option` o_2`, ..., option` o_n`)`
  Boolean function that returns true if at least one of the options o_1, o_2, ..., o_n is selected for enumeration decision `d`, else false is returned.
- `containsAll(Decision d, option` o_1`, option` o_2`, ..., option` o_n`)`
  Boolean function that returns true if all of the options o_1, o_2, ..., o_n are selected for enumeration decision `d`, else false is returned.
- `isTaken(Decision d)`
  Boolean function that returns true if decision d has been taken (that is, it has a value), else false is returned.
- `setValue(Decision d, option` o_1`, option` o_2`, ..., option` o_n`)`
  Void function that causes options o_1, o_2, ..., o_n to be selected for `d`.

The arrows between the decisions in Figure 2.2 represent rules in the variability model. They come in the form

`if <condition> then <action>`.

The condition part of a rule is a Boolean expression over the decisions, and the action part contains statements that assign values to decisions. If the condition part is satisfied by the current decisions, the rule is fired and the instructions contained in the action part are executed. For example, we look at two of the rules:

- `sprayHeader && !isTaken(dynamicJet)` → `molder = true`
  If `sprayHeader` has the value true and `dynamicJet` has not been set yet, assign true to `molder`.
- `stainlessSteel && gapChecker` → `setValue(casterType, slab)`
  If `stainlessSteel` and `gapChecker` are both true, select the value `slab` for `casterType`.

Assets represent the artifacts which products are composed of. In the example model, the components are `baleAdapter`, `pCalibthermometer`, `calibrator`, and `hController`. Whether an asset is part of the final product depends on its *inclusion condition*. Like in the case of rule conditions, it is a Boolean expression over the decisions, and the asset is included in the product if its inclusion condition evaluates to true. For example,

- the inclusion condition of `baleAdapter` is

  `containsOnly(casterType, slab)`,

  which means `baleAdapter` is included if only `slab` is selected for `casterType`, and

- the inclusion condition of `calibrator` is

  `taperUnit`,

  which means `calibrator` is included if `taperUnit` is true.

If an asset is supposed to be contained in the final product in any case, its inclusion condition is just true.

There are two kinds of dependencies between assets. An asset `a` can *include* another asset `b`. As a result, `b` is always included in the final product when `a` is included. The other relationship is *exclusion*. If `a` excludes `b`, `b` cannot be part of the final product whenever `a` is part of it. In the variability model example, `baleAdapter` includes `pCalibthermometer` while `calibrator` excludes `pCalibthermometer`.

DOPLER uses decisions to communicate the variability of the corresponding domains to the user. By taking decisions, the user makes their preferences known to the system, while the rules and visibility of decisions guide the user through the configuration process to make sure domain constraints are respected.

2.2.2 Types of Inconsistencies

Despite the means provided by DOPLER to encode domain knowledge and constraints to enable valid configuration processes, errors can be made during the the development of variability models. With increasing size of the models it gets difficult to ensure that rule executions do not actually break the constraints or lead to inconsistent behavior. The example model in Figure 2.2 contains some types of such inconsistencies for illustration, which we describe in the following.

1. *Domain inconsistency.* This error occurs when domain constraints are violated at some point in the configuration process. For example, the rules states that the action given by `stainlessSteel = true` leads to the assignments `molder = true` and `gapChecker = true`. Setting `molder` to true has the rule effect that `bloom` is selected for the enumeration decision `casterType`, while `gapChecker = true` results in `slab` being selected for `casterType`. However, the cardinality of `casterType` requires that only one value can be active for `casterType`.

   Another instance of domain inconsistency can be found among the assets. The asset `baleAdapter` includes the asset `pCalibthermometer`, but this can lead to a conflict once `calibrator` is included in the product as well, since `calibrator` excludes `pCalibthermometer`.

2. *Violation of metaproperties.* There can be a predefined set of properties that are not necessarily directly connected to the domain constraints but one nonetheless wants to see satisfied by a configuration system. In the example, a metaproperty is defined according to which it is expected that the variables `casterType` and `hydraulicCylinder` are set once the decision `stainlessSteel` is taken. This however is not the case in the model, because there are no rules that guarantee a value is assigned to `hydraulicCylinder` after `stainlessSteel` is taken.

3. *Cyclicity.* The rules in the rule base of the configuration system can induce a cyclic behavior. Considering the operational rule semantics as explained above, this means an endless loop in the rule executions. In the variability model, there is a cycle involving the decisions `casterType`, `gapChecker`, and `taperUnit`. Applying the rules attached to them in the model can result in the value of `gapChecker` alternating between true and false on a cyclic sequence of rule executions.

4. *Violation of confluence.* In our context, it is not wanted that the order in which decisions are taken matters with respect to the resulting configuration. This property is called confluence, and it is violated in the example because depending on whether `dynamicJet` or `sprayHeader` is taken first, the decision `casterType` will look differently.

In the next chapters, we show the logics we have developed to formalize configuration systems like DOPLER and to enable formal verification of such systems.

Chapter 3

Preliminaries

In the logic PIDL, which we present in the next chapter, we use propositional logic to express every basic part of a configuration-system specification and the states of the semantics. This logic is then later extended to PIDL+ by using arithmetic terms. As a base logic for these two frameworks, we consider a fragment of first-order logic, the theory of reals. The symbols of a first-order theory are determined by its *signature*. We specify the kinds of symbols we consider in PIDL+ by fixing the *order-sorted signature*

$$\Sigma := (\{\mathcal{R}, \mathcal{Z}\}, \mathbb{Z} \cup \{+, -, \cdot, <, \leq, >, \geq, \approx, \not\approx\})$$

with

- *sorts* $\mathcal{R}$ and $\mathcal{Z}$, where $\mathcal{Z}$ is a *subsort* of $\mathcal{R}$, $\mathcal{Z} \subset \mathcal{R}$,
- *constants* $\mathbb{Z}$, where the sort of each $c \in \mathbb{Z}$ is $\mathcal{Z}$, written as $sort(c) = \mathcal{Z}$,
- *function symbols* $+, -$, and $\cdot$, where it holds that

$$sort(\diamond) = \mathcal{R} \times \mathcal{R} \to \mathcal{R}, \diamond \in \{+, -, \cdot\},$$

- *predicate symbols* $<, \leq, >, \geq, \approx$, and $\not\approx$, where it holds that

$$sort(\circ) = \mathcal{R} \times \mathcal{R}, \circ \in \{<, \leq, >, \geq, \approx, \not\approx\}.$$

A set of variables X is Σ-*sorted* if the sort of every variable of X is one of the sorts of Σ: $sort(x) \in \{\mathcal{R}, \mathcal{Z}\}$ for all $x \in X$. We call the variables of X *arithmetic variables*. The sorts $\mathcal{R}$ and $\mathcal{Z}$ are to be interpreted as reals $\mathbb{R}$ and integers $\mathbb{Z}$, respectively. The explicit inclusion of $\mathcal{Z}$ in the signature is needed when we define a certain kind of PIDL+ specifications later. In those *admissible* specifications (Definition 5.4), we require variables that are relevant to the transition dynamics to be of sort $\mathcal{Z}$, which helps us to achieve decidability. The signature Σ induces the usual terms from the theory of reals using the known constants, functions and relational symbols.

A *term* t over the signature Σ and a Σ-sorted variable set X, called Σ-*term*, is constructed according to the following grammar:

$$\begin{array}{rcll} t & ::= & x & , x \in X \\ & & c & , c \in \mathbb{Z} \\ & & t_1 \diamond t_2 & , \diamond \in \{+, -, \cdot\} \end{array}$$

$T_\Sigma(X)$ denotes the set of all terms over the signature Σ and the variable set X.

Example 3.1. Let $X = \{x, y\}$. Valid Σ-terms include x, y, 83, -34, $x+900$ or $(9-x)\cdot y$.

We define arithmetic atoms with respect to Σ. Besides the usual definition of an atom, we also define certain classes of atoms that are heavily used in PIDL+, namely *simple atoms* and *simple bounds*. We also say atom for arithmetic atom for simplicity.

Definition 3.2. Let $t, t' \in T_\Sigma(X)$, $x \in X$, $c \in \mathbb{Z}$ and $\circ \in \{<, \leq, >, \geq, \approx, \not\approx\}$. An *arithmetic atom* over Σ or Σ-*atom* is a term of the form

$$t \circ t'.$$

A *simple atom* over Σ or *simple* Σ-*atom* is an atom of the form

$$x \circ t,$$

and a *simple bound* over Σ or *simple* Σ-*bound* is a simple atom of the form

$$x \circ c.$$

A *formula* over Σ, also called Σ-*formula*, over a Σ-sorted variable set X and a set of propositional variables Π is constructed according to the following grammar:

F	::=	$\bot$	(falsum)
	|	$\top$	(verum)
	|	P	$P \in \Pi$
	|	α	α is a Σ-atom
	|	$\neg F$	(negation)
	|	$F_1 \wedge F_2$	(conjunction)
	|	$F_1 \vee F_2$	(disjunction)
	|	$F_1 \to F_2$	(implication)
	|	$F_1 \leftrightarrow F_2$	(equivalence)
	|	$\forall x F$	(universal quantification)
	|	$\exists x F$	(existential quantification)

$F_\Sigma(X, \Pi)$ denotes the set of all formulas over Σ, X and Π. We can write $Qx_1x_2 \ldots x_n F$ for $Qx_1Qx_2 \ldots Qx_n F$, where $Q \in \{\forall, \exists\}$ and $F \in F_\Sigma(X, \Pi)$.

An *atom* is a propositional variable or an arithmetic atom. A *literal* is an atom A or its negation $\neg A$. The *complement* $\overline{L}$ of a literal L is is defined as

$$\overline{A} := \neg A \text{ and } \overline{\neg A} := A.$$

A *clause* is a disjunction of literals. A formula is in *conjunctive normal form* (cnf) if it is a conjunction of disjunction of literals.

As is usually done in the literature, we write $vars(F)$ to denote the set of variables that occur in a formula F, while $vars(N)$ denotes the set of variables that occur in a set N of formulas. In our framework, terms of the form $x \circ t$ play a prominent role. Therefore, we are also interested in directly identifying the left-hand-side variables x in those atoms:

Definition 3.3. Let $x \circ t$ be a simple atom over Σ. We write $varsl(x \circ t)$ to denote the variable that is the left operand of the atom $x \circ t$,

$$varsl(x \circ t) := x.$$

If N is a set containing simple atoms, then $varsl(N)$ denotes the set of variables that are the left operands of the simple atoms in N,

$$varsl(N) := \{x | x \circ t \in N\}.$$

F_Π denotes the set of all propositional formulas over Π:

$$F_\Pi = \{F \in F_\Sigma(X, \Pi) | vars(F) = \emptyset\}.$$

Substitutions in general are mappings on arithmetic variables, in which a variable is mapped to a term. In our context, we only consider substitutions

$$\sigma : X \to \mathbb{Z}$$

that map arithmetic variables to integers. We write substitutions as

$$\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\},$$

where the variables x_i are pairwise distinct, and the result of applying substitutions is then defined by

$$x\sigma = x\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} := \begin{cases} v_i & \text{, if } x = x_i \\ x & \text{, else} \end{cases}.$$

In this thesis, we do not consider substitutions applied to formulas that contain the quantifiers $\forall$ and $\exists$. Thus, the application of a substitution σ to terms and quantifier-free formulas is defined inductively in the usual way as follows:

$$\begin{aligned} c\sigma &:= c, \\ (t_1 \diamond t_2)\sigma &:= t_1\sigma \diamond t_2\sigma, \\ \bot\sigma &:= \bot, \\ \top\sigma &:= \top, \\ P\sigma &:= P, \\ (t_1 \circ t_2)\sigma &:= t_1\sigma \circ t_2\sigma, \\ \neg F\sigma &:= \neg(F\sigma), \\ (F_1 * F_2)\sigma &:= F_1\sigma * F_2\sigma, \end{aligned}$$

where

- $c \in \mathbb{Z}$,
- $\diamond \in \{+, -, \cdot\}$,
- $P \in \Pi$,
- $t_1, t_2 \in T_\Sigma(X)$,
- $\circ \in \{<, \leq, >, \geq, \approx, \not\approx\}$,
- $F, F_1, F_2 \in F_\Pi(X, \Pi)$, and
- $* \in \{\wedge, \vee, \to, \leftrightarrow\}$.

Often it is necessary to *restrict* sets that contain formulas from $F_\Sigma(X, \Pi)$. Note the focus on the left-hand-side variables of the arithmetic atoms.

Definition 3.4. The *restriction* $N|_M$ of a set N of formulas from $F_\Sigma(X, \Pi)$ to a set M of variables is defined as

$$N|_M := \{F \in N | varsl(F) \cap vars(M) \neq \emptyset\}.$$

Example 3.5. $\{x \geq w + 4, y \approx 23, z < y, C\}|_{\{x,z\}} = \{x \geq w + 4, z < y\}$

We can extend restrictions analogously to substitutions. For example,

$$\{x \mapsto 32, y \mapsto 1, z \mapsto 90\}|_{\{y,z\}} = \{y \mapsto 1, z \mapsto 90\}.$$

Facts about single states in the represented configuration process are expressed with formulas of $F_\Sigma(X, \Pi)$ in the semantics of PIDL+. The semantics with respect to $F_\Sigma(X, \Pi)$ is given by the Σ-*interpretation*, which interprets the elements of the signature Σ in the usual way of the theory of arithmetic over the reals, and valuations of propositional and arithmetic variables. We give a detailed definition in the following.

The Σ-interpretation I_Σ maps every object o of Σ to its meaning o_{I_Σ} as follows:

- The sorts $\mathcal{R}$ and $\mathcal{Z}$ are mapped to the sets of reals and integers, respectively,

$$\begin{aligned} \mathcal{R}_{I_\Sigma} &:= \mathbb{R}, \\ \mathcal{Z}_{I_\Sigma} &:= \mathbb{Z}, \end{aligned}$$

- the function symbols are mapped to the operations over the reals of the same symbols,

$$\begin{aligned} +_{I_\Sigma} &:= (v, w) \mapsto v +_\mathbb{R} w, \\ -_{I_\Sigma} &:= (v, w) \mapsto v -_\mathbb{R} w, \\ \cdot_{I_\Sigma} &:= (v, w) \mapsto v \cdot_\mathbb{R} w, \end{aligned}$$

- the constants $c \in \mathbb{Z}$ are mapped to integers of the same symbols,

$$c_{I_\Sigma} := c_\mathbb{Z},$$

- the predicates are mapped to the comparison operators over the reals of the same symbols,

$$\begin{aligned} \leq_{I_\Sigma} &:= \{(v, w) | v \text{ less than or equal to } w\}, \\ \geq_{I_\Sigma} &:= \{(v, w) | v \text{ greater than or equal to } w\}, \\ <_{I_\Sigma} &:= \{(v, w) | v \text{ less than } w\}, \\ >_{I_\Sigma} &:= \{(v, w) | v \text{ greater than } w\}, \\ \approx_{I_\Sigma} &:= \{(v, w) | v \text{ equal to } w\}, \\ \not\approx_{I_\Sigma} &:= \{(v, w) | v \text{ not equal to } w\}, \end{aligned}$$

An *arithmetic assignment* or *valuation* over the set X of arithmetic variables is a mapping β that maps variables of sort $\mathcal{R}$ to reals and variables of sort $\mathcal{Z}$ to integers,

$$\beta : X \to \mathbb{R}, \text{ where } \beta(x) \in \mathbb{Z} \text{ for all } x \in X \text{ with } \mathit{sort}(x) = \mathcal{Z}.$$

The *value of a term* $t \in T_\Sigma(X)$ with respect to a a valuation β is written as $\beta(t)$, and defined inductively as follows:

$$\beta(t_1 \diamond t_2) := \beta(t_1) \diamond_{I_\Sigma} \beta(t_2),$$

where $t_1, t_2 \in T_\Sigma(X), \diamond \in \{+, -, \cdot\}$.

Furthermore, we need *modified assignments* to define the interpretation of quantifiers later below. If β is an assignment, then the modified assignment $\beta[x \mapsto v]$ is defined as

$$\beta[x \mapsto v](y) = \begin{cases} v & \text{if } x = y \\ \beta(y) & \text{otherwise.} \end{cases}$$

A *propositional assignment* or *valuation* I_Π over the set Π of propositional variables is a mapping

$$I_\Pi : \Pi \to \{0, 1\}$$

which assigns the truth values 0 (false) or 1 (true) to the variables of Π.

Given an arithmetic assignment β and a propositional assignment I_Π, we write $I_\Pi(\beta)$ to denote the combination assignment consisting of β and I_Π. We simply call $I_\Pi(\beta)$ an *assignment*.

The truth value of a formula $F \in F_\Sigma(X, \Pi)$ under an assignment $I_\Pi(\beta)$ is defined inductively in the usual way as follows:

$$\begin{aligned}
I_\Pi(\beta)(\bot) &:= 0, \\
I_\Pi(\beta)(\top) &:= 1, \\
I_\Pi(\beta)(P) &:= I_\Pi(P), \\
I_\Pi(\beta)(t_1 \circ t_2) &:= (\beta(t_1), \beta(t_2)) \in \circ_{I_\Sigma}, \\
I_\Pi(\beta)(\neg F) &:= 1 - I_\Pi(\beta)(F), \\
I_\Pi(\beta)(F_1 \wedge F_2) &:= \min\{I_\Pi(\beta)(F_1), I_\Pi(\beta)(F_2)\}, \\
I_\Pi(\beta)(F_1 \vee F_2) &:= \max\{I_\Pi(\beta)(F_1), I_\Pi(\beta)(F_2)\}, \\
I_\Pi(\beta)(F_1 \to F_2) &:= \max\{1 - I_\Pi(\beta)(F_1), I_\Pi(\beta)(F_2)\}, \\
I_\Pi(\beta)(F_1 \leftrightarrow F_2) &:= \text{if } I_\Pi(\beta)(F_1) = I_\Pi(\beta)(F_2) \text{ then 1 else 0}, \\
I_\Pi(\beta)(\forall x F) &:= \min_{v \in sort(x)_{I_\Sigma}} \{I_\Pi(\beta[x \mapsto v])(F)\}, \\
I_\Pi(\beta)(\exists x F) &:= \max_{v \in sort(x)_{I_\Sigma}} \{I_\Pi(\beta[x \mapsto v])(F)\},
\end{aligned}$$

where

- $P \in \Pi$,
- $t_1, t_2 \in T_\Sigma(X)$,
- $\circ \in \{<, \leq, >, \geq, \approx, \not\approx\}$, and
- $F, F_1, F_2 \in F_\Pi(X, \Pi)$.

An assignment $I_\Pi(\beta)$ *satisfies* a Σ-formula F, written as $I_\Pi(\beta) \models F$, if $I_\Sigma(\beta)(F) = 1$. Furthermore, the following holds with respect to F:

- F is *satisfiable* if there is a an assignment $I_\Pi(\beta)$ with $I_\Pi(\beta) \models F$,
- F is *valid* if $I_\Pi(\beta) \models F$ for all assignments $I_\Pi(\beta)$,
- F *entails* a Σ-formula F', written as $F \models F'$, if the following holds: If $I_\Pi(\beta)$ is an assignment and $I_\Pi(\beta) \models F$, then also $I_\Pi(\beta) \models F'$, and
- a set N of Σ-formulas entails a Σ-formula F', written as $N \models F'$, if the following holds: If $I_\Pi(\beta)$ is an assignment and $I_\Pi(\beta) \models F$ for each $F \in N$, then also $I_\Pi(\beta) \models F'$.

If a formula F contains only propositional variables, that is, $vars(F) = \emptyset$, we can just write $I_\Pi \models F$ instead of $I_\Pi(\beta) \models F$ in the above since only the propositional assignment matters then.

We also consider tuples of intervals over $\mathbb{Z}$, which appear in the semantics of PIDL+. We are interested in the following operations involving two such tuples of the same length:

Definition 3.6. Let $t = (I_1, \ldots, I_n)$ and $t' = (I'_1, \ldots, I'_n)$ be tuples containing the same number of integer intervals.

- The *intersection* of t with t' is naturally defined component-wise:

  $$t \cap t' := (I_1 \cap I'_1, \ldots, I_n \cap I'_n).$$

  The intersection is *empty*, written as $t \cap t' = \emptyset$, if $I_i \cap I'_i = \emptyset$ for some i. In that case, we also say that t and t' are *disjoint* to each other. Moreover, we say that a tuple t is disjoint to a set N of tuples if t is disjoint to each $t' \in N$.

- The *union* of t and t' is defined as

  $$t \cup t' := (I_1 \cup I'_1, \ldots, I_n \cup I'_n).$$

- The *difference* of t and t' is defined as

  $$t \setminus t' := (I_1 \setminus I'_1, \ldots, I_n \setminus I'_n).$$

Furthermore, the *atomic representation* with respect to an integer interval and a variable is the set of simple bounds representing the interval:

Definition 3.7. Let $I = [v_1, v_2]$ be an integer interval and x a variable. The *atomic representation* $at(I, x)$ with respect to I and x is defined as

$$at(I, x) := \{x \geq v_1, x \leq v_2\}.$$

We also use an *extension operator* :: on general tuples, which adds a new element to a tuple.

Definition 3.8. Let $i_1, i_2, \ldots, i_n$ and i be some objects. Then

$$(i_1, i_2, \ldots, i_n) :: i \;:=\; (i_1, i_2, \ldots, i_n, i).$$

Example 3.9. Consider the following tuples of integers:

- $\tau_1 = (8, 3, 1, 7, 2)$, and
- $\tau_2 = ()$.

Then,

- $\tau_1 :: 19 = (8, 3, 1, 7, 2, 19)$, and
- $\tau_2 :: 4 = (4)$.

The *length* of a tuple τ, denoted by $|\tau|$, is the number of elements it contains.

Example 3.10. In example 3.9,

- $|\tau_1| = 5$,
- $|\tau_2| = 0$.

The tuple () has no elements and is called the *empty* tuple.

Chapter 4

PIDL

Chapter 2 showed what rule-based configurations systems are: Tools that help humans to configure valid products made up by a possibly very large number of components. Successful examples such as the DOPLER system at Siemens are testament to the wide applicability of those systems in the real world. However, problems pertaining to the consistency of those systems exist. Systems that are inconsistent may exhibit behavior that interferes with the functionality of the configuration system, such as rule effects that ultimately lead to the derivation of invalid products. The larger the system gets, the more difficult it becomes to keep the rule base in a state so that no undesired effects can appear. Formal methods of verification can do this in a systematic and automatic way. They are not widespread though, because the systems still often lack appropriate formalizations.

This chapter presents PIDL (Propositional Interactive Dynamic Logic), a logic to formalize rule-based configuration systems. PIDL's possible-worlds semantics is defined with the aim to capture every possible configuration process for such a system, therefore enabling formal verification of important consistency properties. In particular, the operational semantics of rule-based configuration systems is reflected by having two types of transitions in PIDL, user and rule transitions, interact with each other in a fixpoint fashion.

We present the syntax of PIDL in Section 4.1, the semantics is given in Section 4.2. A sound and complete calculus is provided in Section 4.3. In Section 4.4, we formulate properties of PIDL specifications, which can be linked to important properties of represented configuration systems. We show how DOPLER configuration systems can be translated to our framework in Section 4.5. Experiments indicate that our approach is in principle suited for the comprehensive formalization and verification of rule-based configuration systems, which is described in Section 4.6.

4.1 Syntax

Given an instance of a configuration system, PIDL represents it with a syntactic structure called *specification*. It is defined as follows:

Definition 4.1. A *PIDL specification* $\mathfrak{S}$ is a tuple

$$\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R),$$

where

- Π is a finite set of propositional variables,

- S_I is a satisfiable set of literals over Π, called the *initial state*,
- C is a finite set of propositional formulas over Π, called *constraints*,
- T_U is a finite set of pairs $\chi_i \rightsquigarrow E_i$, called *user transitions*, where χ_i is a propositional formula over Π, called the *condition* of the transition, and E_i is a satisfiable set of literals over Π, called the *update set* of the transition,
- T_R is a finite set of pairs $\chi_j \rightsquigarrow E_j$, called *rule transitions*, where χ_j is a propositional formula over Π, called the *condition* of the transition, and E_j is a satisfiable set of literals over Π, called the *update set* of the transition.

All the transitions in T_U and T_R have different indices i and j.

The transitions are indexed so that each transition can be identified in a compact way, in particular in the semantics (Section 4.2). However, we can occasionally relax the notation:

Convention 4.2. We may omit the index i when writing a transition $\chi_i \rightsquigarrow E_i \in T_U \cup T_R$ if the exact transition does not matter. In examples, we usually rather write $\chi \rightsquigarrow_i E$ instead of $\chi_i \rightsquigarrow E_i$ when concrete instances of χ and E are used. We may also write u_i and r_i instead of just i as an index in order to identify user and rule transitions, respectively.

The propositional variables Π are used for identifying variability decisions of the configuration system. For example, in a car configuration, a variable *manual* could mean that manual transmission is selected, and *automatic* could mean that automatic transmission is selected. *States* in PIDL reflect states in the configuration process of the modeled system, and the initial state S_I is part of the specification. It describes the starting state of the configuration system before any configuration step has taken place. The domain of the configuration system typically has specific requirements and constraints which has to be followed in any state of the configuration process. We formulate these constraints as propositional formulas in the set C. A possible example is the constraint $\neg(\mathit{manual} \wedge \mathit{automatic})$, stating that manual and automatic transmission cannot be selected at the same time in any configuration state. We use transitions $\chi \rightsquigarrow E$ to describe changes during a configuration process. Two types of transitions exist. User transitions represent actions by the user, rule transitions represent actions by the system's rules. This distinction of the transitions is a characteristic feature of PIDL and makes sure the dynamics of a configuration process is sufficiently modeled in the semantics of PIDL. A transition $\chi \rightsquigarrow E$ consists of a condition part χ and an update part E. If a condition of a transition is satisfied by the current state in some way, the transition can be applied to the current state. The result is a new state, which is the result of updating the current state with the set E. We define the concepts connected to the expressions "satisfied by the current state" and "updating the current state" later in the next section.

4.2 Semantics

Given a PIDL specification $\mathfrak{S}$ as a syntactic representation of a configuration system, a corresponding semantics tells us how to interpret the specification. From our perspective, the semantics should cover and express the relevant elements of what happens in an interaction with the configuration system encoded in $\mathfrak{S}$. More concretely, the semantics

should express the following sequence of events: A user takes a decision by setting the value of a variable, then certain rules are possibly triggered, which sets further variables and can activate further rules. If all the rules have been fired, the user can take another decision. We want our semantics to represent exactly this sequence of events and consider all interactions and rule actions that are possible according to the specification. We design it as a possible-worlds semantics, which sees a model of a specification as constraint-satisfying states reachable from the initial state via the transitions.

4.2.1 States and Transitions

We first define states in PIDL. They represent the states that appear during a product derivation with a configuration system.

Definition 4.3. A *PIDL state* is a finite set of propositional literals.

As already mentioned, a PIDL state S is a description of a state in the configuration flow. The propositional literals contained in S indicate how the current situation of the configuration looks like and may represent statements such as "manual transmission is selected" or "this variable is visible to the user". States change during a configuration process, either by user input or by rule effects. PIDL models this by user transitions and rule transitions. The effects of both transition types are stated in the update set E of a transition $\chi \rightsquigarrow E$. The update of a state is then determined by the *update operator*:

Definition 4.4. The *update operator* $\triangleleft$ takes a state S and an update set E, and returns a state S', defined by

$$S \triangleleft E := S',$$

where

$$\begin{aligned} S' := \ & \{L | L \text{ literal over } \Pi, L \in S, \overline{L} \notin E\} \cup \\ & \{L | L \text{ literal over } \Pi, L \in E\}. \end{aligned}$$

A literal L in the state S whose complement $\overline{L}$ does not appear in the update set E is carried over in the new set S'. Literals of E are contained in S', possibly replacing every literal in S that is of the same variable. This corresponds to the intuition that the literals in the update set E can represent new facts that are introduced to S, overwriting old and no longer valid facts. This also implicitly means a literal L in S whose variable does not occur in E at all is not affected by the update and is preserved: Facts in the current state that are not mentioned in the update set are untouched and therefore are also facts in the new state.

Example 4.5. Let $S = \{A, B, \neg C\}$ be a state and $E = \{\neg B, \neg C, D\}$ be an update set. Then

$$\begin{aligned} S \triangleleft E &= \{A, B, \neg C\} \triangleleft \{\neg B, \neg C, D\} \\ &= \{A, \neg B, \neg C, D\} \end{aligned}$$

We go through the literals of the updated set and explain why they are included. The first literal A is a literal of the state S, and since the update set E does not have it, it is just preserved by the update operation. Next comes variable B which appears in both sets S and E. The occurrence in E takes precedence, so the literal B of S is replaced by its complement $\overline{B} = \neg B$ of E, and the resulting state contains $\neg B$. The literal $\neg C$ occurs in S and E, so it is just preserved. The last variable to consider is D, which occurs in E but not in S. This means D is added to the new state.

We can now give the formal definition of rule transitions, which represent the rules of configuration systems.

Definition 4.6. A *rule transition* from a state S to a state S' with respect to a rule transition pair $\chi_i \rightsquigarrow E_i \in T_R$ is a triple

$$S \rightarrow_i S',$$

where

(i) $S \cup \mathsf{C} \not\models \bot$,

(ii) $S \cup \mathsf{C} \models \chi_i$, and

(iii) $S' = S \lhd E_i$.

We also say that the rule transition $\chi_i \rightsquigarrow E_i$ is *applied* to the state S.

The definition states criteria for applying a rule transition. Requirement (i) is our general *state consistency* criterion saying that the state S and the constraints C must be consistent in the classical propositional sense. Thus, no transition can be applied to a state if it is inconsistent by (i). Criterion (ii) is about *transition entailment* and intuitive to understand: The condition χ_i of the transition must be entailed by the current state S and the constraints C for the transition to exist. Finally, the third point is less of a "real criterion" and rather a description of how the new state S' looks like, which is the result of applying the update operator $\lhd$ with respect to S and E_i.

Convention 4.7. The index i in the triple $S \rightarrow_i S'$ is used to identify the corresponding transition with respect to a transition tuple $\chi_i \rightsquigarrow E_i$. In some contexts, we may drop the index if the exact transition does not matter.

Example 4.8. We assume the following state, constraints, and rule transitions pairs:

$S = \{A, \neg C, \neg G\}$

$\mathsf{C} = \{\neg C \rightarrow B\}$

$T_R = \{C \rightsquigarrow_1 \{F\},$

$\quad A \wedge B \rightsquigarrow_2 \{G, \neg H\}\}$

Note that we make use of Convention 4.2 to index the transitions in the example. We examine if we can apply the rule transitions given in T_R to the state S according to Definition 4.6. First, we observe that the state is consistent, that is, it holds that $S \cup \mathsf{C} \not\models \bot$, so criterion (i) is satisfied. The other criterion (ii) is transition entailment. It holds that

$$\begin{gathered} S \cup \mathsf{C} \not\models \chi_1 \\ \Leftrightarrow \{A, \neg C, \neg G\} \cup \{\neg C \rightarrow B\} \not\models C. \end{gathered}$$

Thus, transition 1 cannot be applied to S. The situation is different with transition 2:

$$\begin{gathered} S \cup \mathsf{C} \models \chi_2 \\ \Leftrightarrow \{A, \neg C, \neg G\} \cup \{\neg C \rightarrow B\} \models A \wedge B. \end{gathered}$$

The conditions are met with respect to rule transition 2. It follows that we have a transition $S \rightarrow_2 S'$ with

$$\begin{aligned} S' &= S \triangleleft E_2 \\ &= \{A, \neg C, \neg G\} \triangleleft \{G, \neg H\} \\ &= \{A, \neg C, G, \neg H\} \end{aligned}$$

as the result of the update using E_2 according to Definition 4.4.

The other type of transitions in PIDL is user transitions. They represent input by a user. Before we give the definition, we have to deal with an important prerequisite. As seen earlier, interaction with the configuration systems we consider is a sequence of alternations of user input and rule actions. Once a user has assigned a value to a variable of the system, they must wait until all the rules that are triggered have been executed before they can set the next variable. We need a notion in PIDL that covers this essential aspect of rule-based configuration in order to have a faithful and useful representation. We do this by introducing the concept of *rule-terminal* states. These are states that cannot be changed by any application of rule transitions anymore. Rule-terminal states are thus fixpoints with respect to rule transitions, and user transitions can only be applied to those states.

Definition 4.9. We say that a state S is *rule terminal* if there is no rule transition pair $\chi_i \rightsquigarrow E_i \in T_R$ such that there is a rule transition $S \rightarrow_i S'$ with $S' \neq S$.

Example 4.10. Assume the following constraints and rule transition pairs:

$$\begin{aligned} \mathsf{C} &= \{\neg(C \wedge G), \\ &\quad K \rightarrow H\} \\ T_R &= \{D \rightsquigarrow_1 \{A, \neg C\}, \\ &\quad \neg G \wedge H \rightsquigarrow_2 \{B\}, \\ &\quad \neg B \rightsquigarrow_3 \{A\}\} \end{aligned}$$

We consider four different states:

- $S_1 = \{\neg B, D, E\}$,
- $S_2 = \{C, K\}$,
- $S_3 = \{A, \neg C, D\}$, and
- $S_4 = \{A, \neg F\}$.

We check if these states are rule terminal, that is, we try to find rule transitions that can be applied to the states and whose updates yield states that differ from the original ones. All states are consistent with the constraints. For state S_1, we can see that rule transition 3 can be applied to it:

$$\begin{aligned} & S_1 \cup \mathsf{C} \models \chi_3 \\ \Leftrightarrow \{\neg B, D, E\} \cup \{\neg(C \wedge G), K \rightarrow H\} &\models \neg B. \end{aligned}$$

We apply the update operator to see how the new state looks like:

$$\begin{aligned} S_1' &= S_1 \triangleleft E_3 \\ &= \{\neg B, D, E\} \triangleleft \{A\} \\ &= \{\neg B, D, E, A\}. \end{aligned}$$

It follows that $S_1' \neq S_1$, so we conclude that S_1 is not rule terminal.

State S_2 entails the condition of transition 2:

$$\begin{aligned} & S_2 \cup \mathsf{C} \models \chi_2 \\ \Leftrightarrow & \{C, K\} \cup \{\neg(C \wedge G), K \to H\} \models \neg G \wedge H. \end{aligned}$$

We check the update:

$$\begin{aligned} S_2' &= S_2 \triangleleft E_2 \\ &= \{C, K\} \triangleleft \{B\} \\ &= \{C, K, B\}. \end{aligned}$$

Again, the update changes S_2, $S_2' \neq S_2$, and thus S_2 is not rule terminal.

State S_3 satisfies the condition of transition 1:

$$\begin{aligned} & S_3 \cup \mathsf{C} \models \chi_1 \\ \Leftrightarrow & \{A, \neg C, D\} \cup \{\neg(C \wedge G), K \to H\} \models D. \end{aligned}$$

We apply the update operator to S_3:

$$\begin{aligned} S_3' &= S_3 \triangleleft E_1 \\ &= \{A, \neg C, D\} \triangleleft \{A, \neg C\} \\ &= \{A, \neg C, D\}. \end{aligned}$$

The update results in the same state, $S_3' = S_3$. Since there are no other transitions entailed by S_3, this means S_3 is rule terminal.

Lastly, we consider state S_4. After inspection of the rule transition conditions, we can see that S_4 does not entail any of the rule transitions given in T_R. By Definition 4.9, S_4 is also rule terminal.

The definition of user transitions is very similar to that of rule transitions. It has the additional requirement that the original state must be rule terminal.

Definition 4.11. A *user transition* from a state S to a state S' with respect to a user transition pair $\chi_i \rightsquigarrow E_i \in T_U$ is a triple

$$S \to_i S',$$

where

(i) $S \cup \mathsf{C} \not\models \bot$,

(ii) S is rule terminal,

(iii) $S \cup \mathsf{C} \models \chi_i$, and

(iv) $S' := S \triangleleft E_i$.

We also say that the user transition $\chi_i \rightsquigarrow E_i$ is *applied* to the state S.

Example 4.12. Recall Example 4.10, where we have

$$\begin{aligned}\mathsf{C} &= \{\neg(C \wedge G),\\ &\quad K \to H\}\\ T_R &= \{D \rightsquigarrow_{r_1} \{A, \neg C\},\\ &\quad \neg G \wedge H \rightsquigarrow_{r_2} \{B\},\\ &\quad \neg B \rightsquigarrow_{r_3} \{A\}\}\end{aligned}$$

and the states

- $S_1 = \{\neg B, D, E\}$,
- $S_2 = \{C, K\}$,
- $S_3 = \{A, \neg C, D\}$, and
- $S_4 = \{A, \neg F\}$.

Now assume the following user transitions:

$$\begin{aligned}T_U &= \{K \rightsquigarrow_{u_1} \{\neg D\},\\ &\quad H \vee \neg F \rightsquigarrow_{u_2} \{D, F\}\}\end{aligned}$$

We check whether the user transitions can be applied to any of the above states. In Example 4.10, we established that S_1 and S_2 are not rule terminal, so no user transitions can be done with respect to these states. In particular, note that S_2 actually entails the condition of user transition u_1, but we cannot apply u_1 to S_2 since S_2 is not rule terminal. The remaining states are S_3 and S_4, which are rule terminal, as seen in Example 4.10. The state S_3 does not entail any of the user transitions, but S_4 does with respect to transition u_2:

$$\begin{aligned}&S_4 \cup \mathsf{C} \models \chi_{u_2}\\ \Leftrightarrow\ &\{A, \neg F\} \cup \{\neg(C \wedge G), K \to H\} \models H \vee \neg F.\end{aligned}$$

Thus, a user transition $S_4 \to_{u_2} S_4'$ is possible, with S_4' being

$$\begin{aligned}S_4' &= S_4 \triangleleft E_{u_2}\\ &= \{A, \neg F\} \triangleleft \{D, F\}\\ &= \{A, D, F\}.\end{aligned}$$

The notions defined so far are sufficient to essentially express rule-based configuration systems and their dynamics. A description of a configuration system is encoded in a PIDL specification. Its user transitions and rule transitions represent manual and automatic steps in the configuration process. The criteria formulated in the definitions of the transitions correspond to the way change happens in a configuration system. In particular, the notion of rule-terminal states makes sure that user transition applications in PIDL really correspond to user input in the configuration system. We can now use what we have developed for states and transitions to talk about *models* of PIDL specifications.

4.2.2 Interpretations and Models

A specification induces a set of states that are the results of applying user and rule transitions, starting with the initial state. We get a graph structure that represents

every product derivation that is possible with the modeled configuration system. This forms the backbone of what we consider to be *interpretations* and models of PIDL specifications.

Paths are a central notion to the semantics of PIDL. They allow us to see which user transitions and rule transitions are responsible for arriving at a state S', starting from a state S.

Definition 4.13. A *path* from a state S to a state S' is a tuple

$$(i_1, i_2, \ldots, i_{n-1}, i_n),$$

$n \geq 0$, such that $S_0 \to_{i_1} S_1 \to_{i_2} \cdots \to_{i_{n-1}} S_{n-1} \to_{i_n} S_n$ is a sequence of user and rule transitions, where $\chi_{i_j} \rightsquigarrow E_{i_j} \in (T_U \cup T_R)$ for all $j \in \{1, \ldots, n\}$ and $S_0 = S$ and $S_n = S'$. If $n = 0$, then the path is called *empty* and is denoted by $()$.

For the sake of convenience, we use the term "path" to also refer to the sequence of transitions $S_0 \to_{i_1} S_1 \to_{i_2} \cdots \to_{i_{n-1}} S_{n-1} \to_{i_n} S_n$ that is described by the corresponding tuple $(i_1, i_2, \ldots, i_{n-1}, i_n)$.

Definition 4.14. The *length* $|\tau|$ of a path $\tau = (i_1, i_2, \ldots, i_{n-1}, i_n)$ is the number of elements the tuple $(i_1, i_2, \ldots, i_{n-1}, i_n)$ contains: $|\tau| := n$.

A well-known term used in the context of verification of state transition systems is *reachability*. It deals with the question of whether two states can be connected through a series of transitions.

Definition 4.15. We say that a state S' is *reachable* from a state S if there is a path from S to S'. A state is reachable from itself via the empty path $()$. We also simply say that a state is reachable if it is reachable from the initial state.

If we combine the states that are reachable from the initial state, the transitions that can be applied to them and the propositional interpretations that satisfy each state, we get an interpretation of a specification. It represents all the possible interaction and rule-effect steps that can be done with the corresponding configuration system.

Definition 4.16. An *interpretation* of a specification $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ is a tuple

$$(\mathcal{V}_\mathfrak{S}, \mathcal{T}_\mathfrak{S}, \mathcal{I}_\mathfrak{S})$$

with

- the *state space* $\mathcal{V}_\mathfrak{S} := \{S | S \text{ reachable from } S_I\}$,
- the *transition space*

$$\mathcal{T}_\mathfrak{S} := \{(S, i, S') | S, S' \in \mathcal{V}_\mathfrak{S}, S \to_i S', \chi_i \rightsquigarrow E_i \in (T_U \cup T_R)\}, \text{ and}$$

- the *state interpretations* $\mathcal{I}_\mathfrak{S} := \{(S, I_\Pi) | S \in \mathcal{V}_\mathfrak{S}, I_\Pi \text{ assignment over } \Pi, I_\Pi \models S\}$.

We also refer to $\mathcal{V}_\mathfrak{S}$ and $\mathcal{T}_\mathfrak{S}$ as the *state graph* of the specification.

Definition 4.17. Let $\mathfrak{S}$ be a specification. The *state graph* $\mathcal{G}_\mathfrak{S}$ is a pair consisting of the state space $\mathcal{V}_\mathfrak{S}$ and transition space $\mathcal{T}_\mathfrak{S}$:

$$\mathcal{G}_\mathfrak{S} := (\mathcal{V}_\mathfrak{S}, \mathcal{T}_\mathfrak{S}).$$

As we see later in Theorem 4.33(i), each specification has at least one interpretation.

Models of specifications are interpretations whose state interpretations also satisfy the constraints of the specifications.

Definition 4.18. Let $\mathfrak{S}$ be a specification and $(\mathcal{V}_\mathfrak{S}, \mathcal{T}_\mathfrak{S}, \mathcal{I}_\mathfrak{S})$ be an interpretation of $\mathfrak{S}$. We say that $(\mathcal{V}_\mathfrak{S}, \mathcal{T}_\mathfrak{S}, \mathcal{I}_\mathfrak{S})$ is a *model* of $\mathfrak{S}$ if $I_\Pi \models C$ for each $(S, I_\Pi) \in \mathcal{I}_\mathfrak{S}$.

The semantics of PIDL is now completely defined. It is a representation of the possible configuration states in the form of interpretations. Models are interpretations that represent a configuration system in which the user cannot get to a state that is inconsistent with respect to the domain constraints. The next step is to describe how computations based on the semantics can be done.

4.3 The Inference System W

The semantics of PIDL tells us how to interpret PIDL specifications and how it reflects the operational dynamics of configuration systems that are modeled according to PIDL. In practice, we would also want to know *how* to get the semantical structure of a given specification, that is, the PIDL representation of runs of the configuration system at hand. We give the calculus system W, a set of rules on how to compute exactly that. It is is based on the well-known superposition calculus (Bachmair and Ganzinger, 1994, 2001) and works with objects called *labeled clauses* that carry the relevant semantical information: Which worlds can be reached, how can they be reached, and what properties hold in them? This section starts with a detailed introduction on those clauses.

4.3.1 Labeled Clauses

A propositional clause is a disjunction of propositional literals. Decision procedures for propositional logic are typically based on formulas expressed as sets of clauses. In the calculus we present, we use annotated propositional clauses which we call labeled clauses. Such a clause contains, in addition to the literals, a path and a "clause-type" symbol.

Definition 4.19. A *labeled clause* takes the form

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

where S is a state, τ is a path, $p \in (\{*\} \cup \mathbb{N})$, and C is

- a propositional clause over Π, that is, a disjunction of literals over Π, or
- a special *start* variable.

The part consisting of S, τ, p is called *label*, the part consisting of C is called *core* of the clause. Clauses with *start* as their cores are also called *start clauses*.

A labeled clause $(S, \tau, p \,||\, C)$ means that we have a clause C that is associated with the state S reachable from the initial state S_I of the specification via path τ. The variable p determines the type of the clause C: Either $p = *$, which means that C is a clause that can be derived — using the inference rules we are about to define — from the state S and the constraints C, or p is a natural number i, indicating the clause is derived from the condition of a specific transition $\chi_i \rightsquigarrow E_i$. We give special names to these two types of labeled clauses:

Definition 4.20. A labeled clause $(S, \tau, p \,||\, C)$ is called

- *state clause* if $p = *$, and
- *transition clause* if $p \in \mathbb{N}$.

If the context is clear, we may just use the term "clause" instead of "labeled clause".

The W calculus uses an ordering as is done in many calculi in automated reasoning. In superposition, an ordering on clauses is the basis for the notion of redundancy and for restricting the inference space, that is, the set of clauses that can be derived from the rules of the calculus. W uses an ordering on labeled clauses that is very similar. It takes the labels and the cores into account when determining which labeled clause is larger.

We first define a total ordering $\succ_{\mathfrak{p}}$ on paths as follows: Let τ and τ' be two paths. Then

$$\tau \succ_{\mathfrak{p}} \tau' \text{ if and only if}$$

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

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

Example 4.21. Let the following paths be given:

- $\tau_1 = (5, 21, 7, 2, 13)$,
- $\tau_2 = (2, 6, 17)$,
- $\tau_3 = (5, 21, 20, 1, 6)$.

Then we have the following relationships: $\tau_1 \succ_{\mathfrak{p}} \tau_2$, $\tau_3 \succ_{\mathfrak{p}} \tau_2$, and $\tau_3 \succ_{\mathfrak{p}} \tau_1$.

Given a specification $\mathfrak{S}$, we base our ordering on labeled clauses on a total ordering $\succ$ on the propositional variables Π of $\mathfrak{S}$. The total ordering $\succ_L$ on literals over Π is then defined by:

1. $\neg P \succ_L P$,
2. $Q \succ_L P :\Leftrightarrow Q \succ P$,
3. $Q \succ_L \neg P :\Leftrightarrow Q \succ P$,
4. $\neg Q \succ_L P :\Leftrightarrow Q \succ P$, and
5. $\neg Q \succ_L \neg P :\Leftrightarrow Q \succ P$

for all $P, Q \in \Pi$.

The total ordering $\succ_C$ on propositional clauses is the multiset extension $(\succ_L)_{mul}$ of $\succ_L$, which is defined as follows:

$$C \; (\succ_L)_{mul} \; C'$$

if and only if

1. $C \neq C'$ and
2. for each literal L' with $occ(C', L') > occ(C, L')$ there is a literal L with $L \succ_L L'$ and $occ(C, L) > occ(C', L)$,

where $occ(C, L)$ denotes the number of occurrences of the literal L in the clause C.

Then, the partial ordering $\succ_{\mathfrak{S}}$ on labeled clauses is defined as follows:

$$(S, \tau, p \,||\, C) \succ_{\mathfrak{S}} (S', \tau', p' \,||\, C')$$

if and only if

1. $\tau \succ_{\mathfrak{p}} \tau'$ or
2. $\tau = \tau'$, $p = *$ or $p = p'$, and $C \succ_C C'$.

Note that $\succ_{\mathfrak{S}}$ is well-founded. For convenience, we also write $\succ$ to refer to $\succ_{\mathfrak{p}}$, $\succ_L$, $\succ_C$ and $\succ_{\mathfrak{S}}$ if the context is clear.

Example 4.22. Let the following paths be given:

- $\tau = [4, 2, 9, 7]$ and
- $\tau' = [3, 1, 5]$.

Assume an ordering $\succ$ with $D \succ C \succ B \succ A$. Then

- $(S, \tau, * \,||\, A \vee B \vee \neg C) \succ (S, \tau', * \,||\, C \vee D)$ because $\tau \succ \tau'$, and
- $(S, \tau, * \,||\, C \vee D) \succ (S, \tau, * \,||\, A \vee B \vee \neg C)$ because the paths are the same and $(C \vee D) \succ (A \vee B \vee \neg C)$.

Redundancy of labeled clauses is defined as a labeled clause being entailed by smaller ones that are labeled with the same state and the same clause type.

Definition 4.23. A labeled clause $(S, \tau, p \,||\, C)$ is *redundant* with respect to a set N of labeled clauses if there are clauses $(S, \tau', p \,||\, C_1), (S, \tau', p \,||\, C_2), \ldots, (S, \tau', p \,||\, C_n) \in N$ with $(S, \tau, p \,||\, C) \succ (S, \tau', p \,||\, C_i)$ for $1 \leq i \leq n$ and $C_1, C_2, \ldots, C_n \models C$.

Example 4.24. We again assume the paths τ and τ', and the ordering $\succ$ given in Example 4.22. Then, $(S, \tau, p \,||\, B \vee D)$ is redundant with respect to $(S, \tau', p \,||\, B)$, because $(S, \tau, p \,||\, B \vee D) \succ (S, \tau', p \,||\, B)$ and $B \models B \vee D$.

4.3.2 Inference Rules

We define the inference system W with respect to a specification $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$. It consists of six rules on how to infer new labeled clauses from existing labeled clauses. Four rules of W form a subset called w and can be seen as rules for inferences within a state in $\mathcal{V}_{\mathfrak{S}}$. The two other rules of W concern inferences that represent the transitions between the states. We first introduce the inference rules that constitute the subsystem w.

- **Units Creation**

$$\mathcal{I}\frac{S, \tau, * \,||\, start}{S, \tau, * \,||\, L}$$

  where $L \in S$.

- **Constraints Creation**

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

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

A clause of the form $(S, \tau, * \,||\, start)$ has the role of the "first" clause with respect to a state S. The first clause with respect to the initial state S_I is given by $(S_I, \tau, * \,||\, start)$, the first clauses of the other states are generated by rules of W that we present later. Given such a clause as a premise, the conclusions of the Units Creation and Constraints Creation rules are clauses that represent the state S and the constraints C. Units Creation produces labeled clauses whose cores are unit clauses derived directly from the literals in S. The core of a clause generated by Constraints Creation is a clause from the conjunctive normal form of the set of constraints C.

- **User Transition Conditions Creation**

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

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

- **Rule Transition Conditions Creation**

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

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

The next two rules also have the start clause as premises and are about deriving labeled clauses that represent the conditions of the transitions. User Transition Condition Creation yields labeled clauses with respect to user transitions and Rule Transition Conditions Creation produces labeled clauses with respect to rule transitions. The cores of those clauses are taken from the conjunctive normal form of the negated condition formulas of the respective transitions. We consider the negated forms because the calculus makes use of prove by refutation, as is described further below. The index i takes the role of the p variable to indicate the transition pair $\chi_i \rightsquigarrow E_i$ the resulting clause corresponds to.

- **Factoring**

$$\mathcal{I}\frac{S, \tau, p \,||\, C \vee L \vee L}{S, \tau, p \,||\, C \vee L} \ ,$$

  where L 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

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

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

Factoring and Superposition are derived from the rules of classical superposition (Bachmair and Ganzinger, 1994, 2001), restricted to the non-equational case, and work in the same way, adjusted to the context of reasoning within a world in PIDL. We use Factoring to get simpler clauses in which duplicate literals are removed. Superposition contains resolution on the cores of two labeled clauses, with a maximality requirement to establish a redundancy criterion. The values of p and p' determine whether the inference can be done with two given clauses: Either at least one of the premise clauses is a state clause or both premise clauses are transition clauses with respect to the same transition. The conclusion is a transition clause if at least one transition clause is involved as a premise, or else it is a state clause. That way we can distinguish inferences restricted to the state and the constraints, and inferences that additionally involve transition conditions. This is important because we use refutational completeness of resolution, in which we look for inferred clauses of the form $(S, \tau, p||\bot)$: If $p = *$, then we can say that $S \cup \mathsf{C}$ is inconsistent, and if p is an index with respect to a transition, then the condition of the transition is entailed by $S \cup \mathsf{C}$. This works because from the transition creation rules we get clauses based on the negation $\neg\chi_i$ of the transition conditions, and then we make use of the fact that the statements $S \cup \mathsf{C} \models \chi_i$ and $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot$ are equivalent.

The inference subsystem w is composed of the rules Units Creation, Constraints Creation, User Transition Conditions Creation, Rule Transition Conditions Creation, Factoring, and Superposition.

The rest of the rules of W uses the notion of *closure* of clause sets of the form $\{(S, \tau, *||start)\}$ under the inference rules. In general, we get the closure by repeatedly applying inferences on clause sets. Given an abstract inference system A and a set of clauses N, the closure N^*_A of N under A is defined inductively:

$$
\begin{aligned}
N_\mathsf{A} &:= \{(S, \tau, p||C) | (S, \tau, p||C) \text{ is a conclusion} \\
&\qquad\qquad \text{of an } \mathsf{A} \text{ inference with premises in } N\} \\
N^0_\mathsf{A} &:= N \\
N^{i+1}_\mathsf{A} &:= N^i_\mathsf{A} \cup (N^i_\mathsf{A})_\mathsf{A} \\
N^*_\mathsf{A} &:= \bigcup_{n \geq 0} N^i_\mathsf{A}
\end{aligned}
$$

We say that a set N is *saturated* with respect to A if $N_\mathsf{A} \subseteq N$.

The remaining rules of W are then as follows:

- **Forward Rule Transition**

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

  where

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

- **Forward User Transition**

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

  where

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

Forward Rule Transition corresponds to applying a rule transition to a state. It takes a clause $(S, \tau, i \,||\, \bot)$ as premise, which means that through Superposition bottom has been derived from the set $S \cup \mathsf{C} \cup \{\neg\chi_i\}$, and thus the transition condition χ_i is entailed by the state and the constraints, as already mentioned above. A number of side conditions has to be satisfied in order for the inference to take place. These conditions directly correspond to the requirements stated in the definition of rule transitions (Definition 4.6): First, the state and the constraints must be consistent, that is, $(S, \tau, * \,||\, \bot)$ must not appear in the inferences with respect to state clauses. Second, the value i is an index with respect to a rule transition, and lastly, the new state S' is the result of updating S with E_i. The conclusion of the inference is a new clause $(S', \tau :: i, * \,||\, \mathit{start})$, where the current path τ is extended by adding i to it. The new clause serves as the starting point for further inferences corresponding to the state S' with w. Forward User Transition works very similarly, but has an additional condition that incorporates in the calculus the criterion of rule termination as stated in the definition of user transitions (Definition 4.11): For every rule transition entailed by the current state, updating the state does not change the state.

The rules of w together with Forward Rule Transition and Forward User Transition constitute the inference system W.

Intuitively, the first clause of each state $S \in \mathcal{V}_{\mathfrak{S}}$ is $(S, \tau, * \,||\, \mathit{start})$, and $(S_I, (), * \,||\, \mathit{start})$ is the start clause with respect to the initial state, and thus typically the first clause of the whole inference with W.

Remark 4.25. The rules of W dictate that each derivation of any clause that is labeled with S, τ, p starts with $(S, \tau, * \,||\, \mathit{start})$.

We have seen how the semantics of PIDL specifications can be represented as labeled clauses. We then have presented the rules of the inference system W, which contain instructions on how to derive those clauses. The closure $\{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ describes all the clauses that can be inferred with W, starting from the clause $(S_I, (), * \,||\, \mathit{start})$. A clause from this set has information on a corresponding state and the path that leads to that state, and therefore how to compute the state space $\mathcal{V}_{\mathfrak{S}}$ and transition space $\mathcal{T}_{\mathfrak{S}}$, the elemental parts of a PIDL interpretation. The focus of the next section is to prove W does this in a correct way.

4.3.3 Soundness and Completeness of W

The correctness of W comes, as usual, twofold: The inference system must be sound, meaning all clauses that can be inferred with it correspond to states and paths actually occurring in $\mathcal{V}_{\mathfrak{S}}$ and $\mathcal{T}_{\mathfrak{S}}$, and complete, meaning every state that is reachable from the initial state is represented by a clause derived in W. We approach the proof by observing important properties of the calculus. The first one can be described as *local soundness*.

Lemma 4.26. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a PIDL specification. Then

1. $(S, \tau, * \,||\, C) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}} \Rightarrow S \cup \mathsf{C} \models C$,

2. $(S, \tau, i \,||\, C) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}} \Rightarrow S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C$,
   where $C \neq \mathit{start}$ and $\chi_i \rightsquigarrow E_i \in T_U \cup T_R$.

Proof. 1. By induction on the length of the derivation of $(S, \tau, * \,||\, C)$ relative to its corresponding first clause $(S, \tau, * \,||\, \mathit{start})$.

- Base case: The base case is defined by Remark 4.25. According to it, the clause $(S, \tau, * \,||\, C)$ is the conclusion of a Units Creation or Constraints Creation inference, with $(S, \tau, * \,||\, \mathit{start})$ being the premise. Then $C = L$ with $L \in S$ or $C \in \mathit{cnf}(\mathsf{C})$ respectively. In both cases $S \cup \mathsf{C} \models C$.
- Induction step:
  - $(S, \tau, * \,||\, C)$ is the conclusion of a Factoring inference, with the premise being a clause of the form $(S, \tau, * \,||\, C' \vee A \vee A)$ and $C = C' \vee A$. By induction hypothesis, $S \cup \mathsf{C} \models C' \vee A \vee A$. Thus, $S \cup \mathsf{C} \models C' \vee A$.
  - $(S, \tau, * \,||\, C)$ is the conclusion of a Superposition inference with premises being two clauses $(S, \tau, * \,||\, C_1 \vee L)$ and $(S, \tau, * \,||\, C_2 \vee \overline{L})$ and $C = C_1 \vee C_2$. By induction hypothesis, $S \cup \mathsf{C} \models C_1 \vee L$ and $S \cup \mathsf{C} \models C_2 \vee \overline{L}$. By soundness of propositional resolution, it then holds that $S \cup \mathsf{C} \models C_1 \vee C_2$.

2. By induction on the derivation length of $(S, \tau, i \,||\, C)$ relative to $(S, \tau, * \,||\, \mathit{start})$.

- Base case: The base case is defined by Remark 4.25. Then $(S, \tau, i \,||\, C)$ is the conclusion of a User or a Rule Transition Conditions Creation, with $(S, \tau, * \,||\, \mathit{start})$ being the premise. In both cases, $C \in \mathit{cnf}(\neg\chi_i)$ and thus we have $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C$.
- Induction step:
  - $(S, \tau, i \,||\, C)$ is the conclusion of a Factoring inference, with a premise of the form $(S, \tau, i \,||\, C' \vee A \vee A)$ and $C = C' \vee A$. By induction hypothesis, we get $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C' \vee A \vee A$. Then, $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C' \vee A$.
  - $(S, \tau, i \,||\, C)$ is the conclusion of a Superposition inference with premises $(S, \tau, p_1 \,||\, C_1 \vee L)$ and $(S, \tau, p_2 \,||\, C_2 \vee \overline{L})$, and $C = C_1 \vee C_2$. It holds that $p_1 = i$ or $p_2 = i$. Without loss of generality, let $p_1 = i$. By induction hypothesis, $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C_1 \vee L$. Now we consider two possibilities for the value of p_2: If $p_2 = i$, then also by induction hypothesis it holds that $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C_2 \vee \overline{L}$. If $p_2 = *$, then we have $S \cup \mathsf{C} \models C_2 \vee \overline{L}$ as shown above, and therefore $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C_2 \vee \overline{L}$. In any case, $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models C_1 \vee C_2$ by soundness of propositional resolution.

□

The above lemma basically expresses the soundness of W with respect to the single worlds represented by the states occurring in the label of the inferred clauses. This makes sense since, as mentioned earlier, Factoring and Superposition resemble the resolution calculus for propositional logic. Thus, Lemma 4.26 builds on the soundness of propositional resolution.

After local soundness we also establish *local refutational completeness* of W. It means that whenever a state is inconsistent with the constraints, W can derive a state clause whose core is bottom. In a similar vein, inconsistence of the state, constraints, and the negation of a transition condition means we can infer a transition clause with respect to that transition whose core is bottom. Again, this can be justified by refutational completeness of the propositional resolution calculus.

Lemma 4.27. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a PIDL specification. Furthermore, let $S \in \mathcal{V}_{\mathfrak{S}}$ and $(S, \tau, * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$. Then

1. $S \cup \mathsf{C} \models \bot \Rightarrow (S, \tau, * \,||\, \bot) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$,
2. $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot \Rightarrow (S, \tau, i \,||\, \bot) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$.

Proof. 1. We can infer clauses whose cores are taken from the state S and the constraints C by the assumption $(S, \tau, * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$, and the Units Creation and Constraints Creation rules. Then, the assumption $S \cup \mathsf{C} \models \bot$ and refutational completeness of resolution imply that $(S, \tau, * \,||\, \bot)$ can be inferred with Factoring and Superposition.

2. User or Rule Transition Condition Creation yields clauses with cores that are derived from $\neg\chi_i$ because of the existence of $(S, \tau, * \,||\, start)$ by assumption. Analogously to the above case, the assumption $S \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot$ and refutational completeness of resolution means that from those clauses we can derive $(S, \tau, i \,||\, \bot)$ using the Factoring and Superposition rules.

□

With local soundness and local refutational completeness, we can now show general soundness and completeness of the W calculus.

Theorem 4.28. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a specification. There are a state S and a path τ such that $(S, \tau, * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$ if and only if S is reachable from the initial state S_I via path τ.

Proof. ($\Rightarrow$) By induction on the path τ.

Let $\tau = ()$. Then $(S_I, (), * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$, and indeed $S_I \in \mathcal{V}_{\mathfrak{S}}$ via the empty path $()$.

Let $\tau = \tau' :: i$. We have a labeled clause $(S, \tau' :: i, * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$. According to the rules of W, there are two cases in which this clause could have been derived:

1. By Forward Rule Transition.

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

where

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

From the premises we conclude $(S', \tau', i \,||\, \bot) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$. As observed in remark 4.25, $(S', \tau', * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$. By induction hypothesis, we have $S' \in \mathcal{V}_{\mathfrak{S}}$, reachable via path τ'. We show that to S' we can apply the rule transition $\chi_i \rightsquigarrow E_i$ according to Definition 4.6.

- The condition $(S', \tau', * \,||\, \bot) \notin \{(S', \tau', * \,||\, start)\}^*_{\mathsf{w}}$ first means that we have $(S', \tau', * \,||\, \bot) \notin \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$, since w is contained in W, and the rules not in w, which are Forward Rule Transition and Forward User Transitions, cannot produce a clause whose core is bottom. We already know that $(S', \tau', * \,||\, start) \in \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$ from above. It holds that $S' \cup \mathsf{C} \not\models \bot$ by Lemma 4.27.

- As observed above, $(S', \tau', i \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$. By Lemma 4.26, we get $S' \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot$ and thus $S' \cup \mathsf{C} \models \chi_i$.
- Finally, $S = S' \triangleleft E_i$ indeed as required by the inference rule.

The conditions of Definition 4.6 are fulfilled, so we can do a rule transition $S' \to_i S$ using $\chi_i \rightsquigarrow E_i$, and thus $S \in \mathcal{V}_{\mathfrak{S}}$ via path $\tau = \tau' :: i$.

2. By Forward User Transition.

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

where

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

The premise $(S', \tau', i \,||\, \bot)$ indicates that $(S', \tau', i \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$, and thus $(S', \tau', * \,||\, \mathit{start}) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ by Remark 4.25. By induction hypothesis, $S' \in \mathcal{V}_{\mathfrak{S}}$. We show that to S' we can apply the user transition $\chi_i \rightsquigarrow E_i$ according to Definition 4.11.

- The condition $(S', \tau', * \,||\, \bot) \notin \{(S', \tau', * \,||\, \mathit{start})\}^*_{\mathsf{w}}$ entails $S' \cup \mathsf{C} \not\models \bot$, which can be shown in the same way as in the analogous case of Forward Rule Transition.
- From the condition

  $$S' = S' \triangleleft E_j \text{ for each } (S', \tau', j \,||\, \bot) \in \{(S', \tau', * \,||\, \mathit{start})\}^*_{\mathsf{w}}, \chi_j \rightsquigarrow E_j \in T_R,$$

  we get that each time we have
  - $(S', \tau', j \,||\, \bot) \in \{(S', \tau', * \,||\, \mathit{start})\}^*_{\mathsf{w}}$, which means
  - $(S', \tau', j \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ analogously to the previous case, which means
  - $S' \cup \mathsf{C} \cup \{\neg\chi_j\} \models \bot$ by Lemma 4.26, which means
  - $S' \cup \mathsf{C} \models \chi_j$,

  it holds that $S' = S' \triangleleft E_j$. Therefore, for all rule transitions $\chi_j \rightsquigarrow E_j$ that can be applied to S', the update does not change S'. We have that S' is rule-terminal according to Definition 4.9.
- As in the previous case, the premise $(S', \tau', i \,||\, \bot)$ gives us $S' \cup \mathsf{C} \models \chi_i$.
- $S = S' \triangleleft E_i$ indeed as required by the inference rule.

The requirements in Definition 4.11 are satisfied, and we can have a user transition $S' \to_i S$ by $\chi_i \rightsquigarrow E_i$, and thus $S \in \mathcal{V}_{\mathfrak{S}}$ via path $\tau = \tau' :: i$.

($\Leftarrow$) By induction on the path τ.

$\tau = ()$: The path consists of just the initial state S_I, and it indeed holds that $(S_I, (), * \,||\, \mathit{start}) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$.

$\tau = \tau_n :: i$: The path has the form $S_I \to S_1 \to \cdots \to S_n \to_i S$. By induction hypothesis, $(S_n, \tau_n, * \,||\, \mathit{start}) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$. There are two cases:

1. $S_n \rightarrow_i S$ is a rule transition, that is, $\chi_i \rightsquigarrow E_i \in T_R$. By Definition 4.6,

   (a) $S_n \cup \mathsf{C} \not\models \bot$,

   (b) $S_n \cup \mathsf{C} \models \chi_i$, and

   (c) $S = S_n \lhd E_i$.

   Then

   (a) $\Rightarrow (S_n, \tau_n, * \,||\, \bot) \notin \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ by Lemma 4.26, and

   (b) $\Rightarrow S_n \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot$
   $\Rightarrow (S_n, \tau_n, i \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ (Lemma 4.27)
   $\Rightarrow (S_n, \tau_n, i \,||\, \bot) \in \{(S_n, \tau_n, * \,||\, \mathit{start})\}^*_{\mathsf{w}}$ (Remark 4.25).

   The conditions of Forward Rule Transition are satisfied and we can infer the clause $(S, \tau, * \,||\, \mathit{start})$ with $\tau = \tau_n :: i$.

2. $S_n \rightarrow_i S$ is a user transition, that is, $\chi_i \rightsquigarrow E_i \in T_U$. By Definition 4.11,

   (a) $S_n \cup \mathsf{C} \not\models \bot$,

   (b) S_n is rule terminal,

   (c) $S_n \cup \mathsf{C} \models \chi_i$, and

   (d) $S := S_n \lhd E_i$.

   Then

   (a) $\Rightarrow (S_n, \tau_n, * \,||\, \bot) \notin \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ by Lemma 4.26,

   (b) $\Leftrightarrow S_n \cup \mathsf{C} \models \chi_j \Rightarrow S_n = S_n \lhd E_j$ for each $\chi_j \rightsquigarrow E_j \in T_R$
   $\Rightarrow S_n \cup \mathsf{C} \cup \{\neg\chi_j\} \models \bot \Rightarrow S_n = S_n \lhd E_j$ for each $\chi_j \rightsquigarrow E_j \in T_R$
   $\Rightarrow (S_n, \tau_n, j \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}} \Rightarrow S_n = S_n \lhd E_j$ for each rule transition $\chi_j \rightsquigarrow E_j \in T_R$ (Lemma 4.27)
   $\Rightarrow (S_n, \tau_n, j \,||\, \bot) \in \{(S_n, \tau_n, * \,||\, \mathit{start})\}^*_{\mathsf{w}}) \Rightarrow S_n = S_n \lhd E_j$ for each rule transition $\chi_j \rightsquigarrow E_j \in T_R$ (Remark 4.25), and

   (c) $\Rightarrow S_n \cup \mathsf{C} \cup \{\neg\chi_i\} \models \bot$
   $\Rightarrow (S_n, \tau_n, i \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ (Lemma 4.27)
   $\Rightarrow (S_n, \tau_n, i \,||\, \bot) \in \{(S_n, \tau_n, * \,||\, \mathit{start})\}^*_{\mathsf{w}}$ (Remark 4.25),

so we can apply Forward User Transition and we get a clause $(S, \tau, * \,||\, \mathit{start})$ via path $\tau = \tau_n :: i$. □

From the previous lemmas and theorem we can conclude that the question of whether a specification has a model or does not can be directly linked to the derivation of a clause with a bottom core in W.

Corolloray 4.29. Let $\mathfrak{S}$ be a PIDL specification. Then $\mathfrak{S}$ has a model if and only if there are no S and τ such that $(S, \tau, * \,||\, \bot) \in \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$.

From the tight correspondence between classical superposition and W in the single worlds we can derive the following result, which states that the set of clauses produced with W is finite, with possibly redundant clauses.

Theorem 4.30. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a specification. $\{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$ is finite up to redundancy.

Proof. Let $N = \{(S_I, (), * \,||\, start)\}^*_{\mathsf{W}}$. We define a subset $K \subseteq N$ as follows:

$$K := \{(S, \tau, * \,||\, start) | (S, \tau, * \,||\, start) \in N, \text{there is no } \tau' \text{ such that } \tau \succ \tau' \text{ and } (S, \tau', * \,||\, start) \in N\}.$$

K is not empty since the initial start clause $(S_I, (), * \,||\, start)$ is contained in K. We note that K is finite since it consists of clauses $(S, \tau, * \,||\, start)$, where

(i) the possibilities for S are finite, because S is a finite set of literals over the finite set Π of propositional variables, and

(ii) the possibilities for τ are finite, because $\succ$ is a total order on paths and $\succ$ is well-founded.

Furthermore, let $M \subseteq N$ be another subset with

$$M := \bigcup_{(S,\tau,* \,||\, start) \in K} \{(S, \tau, * \,||\, start)\}^*_{\mathsf{w}},$$

that is, M is the set of w inferences on the start clauses of K. Note that M contains K. The w subcalculus essentially corresponds to propositional resolution, which is known to be finite up to redundancy. Hence, a set $\{(S, \tau, * \,||\, start)\}^*_{\mathsf{w}} \subseteq M$ is finite up to redundancy, and so M is also finite up to redundancy.

Now, assume a clause $(S, \tau, * \,||\, C) \in N$ that is not in M. That means first that, by Remark 4.25, $(S, \tau, * \,||\, start) \in N$ and $(S, \tau, * \,||\, start) \notin M$, and thus $(S, \tau, * \,||\, start) \notin K$. By assumption and well-foundedness of $\succ$, there is a minimal τ' with $\tau \succ \tau'$ such that $(S, \tau', * \,||\, start) \in N$. It follows from the minimality of τ' that $(S, \tau', * \,||\, start) \in K$, which leads to $(S, \tau', * \,||\, C) \in M$, since the rules of w infer the same cores when the state is the same. It follows that $(S, \tau, * \,||\, C)$ is redundant with respect to M. □

4.3.4 Standard Interpretation

We present one way to construct an interpretation for a given satisfiable specification $\mathfrak{S}$ that is called the *standard interpretation* of the specification. Bachmair and Ganzinger described a method to incrementally construct an interpretation for a given set of propositional clauses in their proof of refutational completeness of propositional resolution, using an ordering on those clauses (Bachmair and Ganzinger, 2001). From this we derive an analogous approach based on an ordering on labeled clauses. The resulting interpretation can then be seen as the state interpretations $\mathcal{I}_{\mathfrak{S}}$ of a specification interpretation $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_{\mathfrak{S}})$, where the other components $\mathcal{V}_{\mathfrak{S}}$ and $\mathcal{T}_{\mathfrak{S}}$ are determined by the specification $\mathfrak{S}$. This standard interpretation is then a model for the specification if no labeled clause with bottom as its core is inferred, as shown in Theorem 4.33.

Let N be a set of labeled clauses and let $N_{S,\tau}$ be a subset of N that contains all the clauses of N labeled with S, τ, and $*$: $N_{S,\tau} := \{(S, \tau, * \,||\, C) | (S, \tau, * \,||\, C) \in N\}$. Furthermore, let $\succ$ be an ordering on labeled clauses as defined in Section 4.3.1. The *state clause interpretation* $I_{N_{S,\tau}}$ for $N_{S,\tau}$ is constructed as follows:

- $I_{(S,\tau,* \,||\, C)} := \bigcup_{(S,\tau,* \,||\, C) \succ (S,\tau,* \,||\, D)} \delta_{(S,\tau,* \,||\, D)}$
- $\delta_{(S,\tau,* \,||\, C)} := \begin{cases} \{A\} & \text{, if } (S, \tau, * \,||\, C) \in N_{S,\tau}, C = C' \vee A, A \succ C', I_{(S,\tau,* \,||\, C)} \not\models C \\ \emptyset & \text{, else} \end{cases}$

- $I_{N_{S,\tau}} := \bigcup_{(S,\tau,* \,||\, C) \in N_{S,\tau}} \delta_{(S,\tau,* \,||\, C)}$

The method corresponds directly to the construction as described by Bachmair and Ganzinger if we consider the clauses of the same label as representing a single propositional world. That is why we only work with state clauses, that is, clauses that are labeled with the type $*$. Having fixed the label, the relevant part for the construction are only the cores of the clauses, resembling the purely propositional case. Iterating through the clauses by the ordering, literals are added to the *partial interpretation* to make the cores they are in true. The state clause interpretation we get is a Herbrand interpretation, where variables contained in the set are considered to be true and variables not contained are considered to be false. The properties of the propositional model construction and their details can be found in the aforementioned work by Bachmair and Ganzinger (Bachmair and Ganzinger, 2001). State clause interpretations have one important property that can be seen easily from the semantics of PIDL: Two interpretations that refer to the same state are identical if we consider the closure set of clauses resulting from inferences of W starting with $(S_I, (), * \,||\, \mathit{start})$.

Lemma 4.31. Let $\mathfrak{S}$ be a specification and $N = \{(S_I, (), * \,||\, \mathit{start})\}^*_A$. If $(S, \tau, * \,||\, C)$ and $(S, \tau', * \,||\, D)$ are clauses in N, and $I_{N_{S,\tau}}$ and $I_{N_{S,\tau'}}$ are the respective state clause interpretations, then $I_{N_{S,\tau}} = I_{N_{S,\tau'}}$.

Proof. By assumption, there are clauses $(S, \tau, * \,||\, C)$ and $(S, \tau', * \,||\, D)$ in N. According to Remark 4.25, it holds that $(S, \tau, * \,||\, \mathit{start}) \in N$ and $(S, \tau', * \,||\, \mathit{start}) \in N$. All state clauses are the result of inferences starting with these two start clauses. Both refer to the same state S, which means the rules of W make the cores of the inferred clauses identical in both cases: $\{C | (S, \tau, * \,||\, C) \in N_{S,\tau}\} = \{D | (S, \tau', * \,||\, D) \in N_{S,\tau'}\}$. Since the interpretation construction actually works only with the cores of the clauses in $N_{S,\tau}$ and $N_{S,\tau'}$, we have that $I_{N_{S,\tau}} = I_{N_{S,\tau'}}$. □

As a consequence of Lemma 4.31, we drop the path in the index of state clause interpretations, and we write I_{N_S} to denote one of the interpretations $I_{N_{S,\tau}}$.

We now repeat the refutational-completeness result of resolution in our context of the W calculus and labeled clauses.

Lemma 4.32. Let $\mathfrak{S}$ be a specification and $N = \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$. If N does not contain a clause of the form $(S', \tau', * \,||\, \bot)$, then $I_{N_S} \models C$ for all $(S, \tau, * \,||\, C) \in N$.

Proof. Suppose the contrary, that is, there is a minimal clause $(S, \tau, * \,||\, C) \in N$ labeled with S such that $I_{N,S} \not\models C$. It holds that $C \neq \bot$ by assumption, so C has a maximal literal L. We can distinguish between the following cases:

- L is positive, $L = P$:
  - First assume $(S, \tau, * \,||\, C) = (S, \tau, * \,||\, C' \vee P \vee P)$. Then there must be an inference using Factoring, yielding the clause $(S, \tau, * \,||\, C' \vee P)$ whose core is still false under I_{N_S} and also smaller than $(S, \tau, * \,||\, C)$, a contradiction.
  - The other possibility is that $(S, \tau, * \,||\, C) = (S, \tau, * \,||\, C' \vee P)$ with P being strictly maximal. In addition, we have $I_{(S,\tau,* \,||\, C)} \not\models C' \vee P$ in the interpretation construction of I_{N_S}. Consequently, $\delta_{(S,\tau,* \,||\, C)} = \{P\}$ and $I_{N_S} \models C' \vee P$, making C actually true under I_{N_S}, again a contradiction.

- L is negative, so $L = \neg P$ and $(S, \tau, * \,||\, C) = (S, \tau, * \,||\, C' \vee \neg P)$. Since it holds that $I_{N,S} \not\models C' \vee \neg P$, there must be the case that $P \in I_{N_S}$. Therefore, there is a clause $(S, \tau, * \,||\, D \vee P)$ with $\delta_{(S,\tau,* \,||\, D \vee P)} = \{P\}$ and $(S, \tau, * \,||\, C' \vee \neg P) \succ (S, \tau, * \,||\, D \vee P)$. We can apply superposition with premises $(S, \tau, * \,||\, C' \vee \neg P)$ and $(S, \tau, * \,||\, D \vee P)$. This yields $(S, \tau, * \,||\, C' \vee D)$. We have that $I_{N_S} \not\models C' \vee D$. This and $(S, \tau, * \,||\, C' \vee \neg P) \succ (S, \tau, * \,||\, C' \vee D)$ contradict the assumption.

□

Lemma 4.32 says that for each clause that can be derived with W, we can construct a corresponding state clause interpretation that satisfies the propositional cores of the clauses, barring the existence of a clause with bottom as its core. We now define the standard interpretation of a specification, which consists of the state clause interpretations. Let $\mathfrak{S}$ be a specification and $N := \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$. The standard interpretation is $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ with $\mathcal{I}_N$ being defined as

$$\mathcal{I}_N := \{(S, I_{N_S}) | S \in \mathcal{V}_{\mathfrak{S}}\}.$$

The following theorem states that $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ is a correctly defined interpretation according to Definition 4.16 and possibly a model according to Definition 4.18.

Theorem 4.33. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a specification. Furthermore, we set $N := \{(S_I, (), * \,||\, \mathit{start})\}^*_{\mathsf{W}}$. Then the following holds:

(i) $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ is an interpretation of $\mathfrak{S}$.

(ii) If N does not contain a clause of the form $(S, \tau, * \,||\, \bot)$, then $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ is a model of $\mathfrak{S}$.

Proof. Let $S \in \mathcal{V}_{\mathfrak{S}}$. There is a clause $(S, \tau, * \,||\, \mathit{start}) \in N$ by Theorem 4.28. By the Units and Constraints Creation rules of W we have

- $(S, \tau, * \,||\, L) \in N$ for each $L \in S$, and
- $(S, \tau, * \,||\, C) \in N$ for each $C \in \mathit{cnf}(\mathsf{C})$.

By assumption and Lemma 4.32, it first holds that

$$I_{N_S} \models L \text{ for each } L \in S,$$

which means that $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ is indeed an interpretation of $\mathfrak{S}$. Moreover,

$$I_{N_S} \models C \text{ for each } C \in \mathit{cnf}(\mathsf{C}),$$

which means that $I_{N_S} \models \mathsf{C}$, and $(\mathcal{V}_{\mathfrak{S}}, \mathcal{T}_{\mathfrak{S}}, \mathcal{I}_N)$ is thus a model of $\mathfrak{S}$. □

4.4 Properties of PIDL Specifications

As we have seen in Chapter 2, the correctness of configuration systems is embodied by a number of properties, some of which are domain specific and some are domain agnostic. In the following, we show how such properties of configuration systems can be seen as properties of their corresponding PIDL specifications. Once expressed in the language of our logic, verification of these properties can be done to enable us to draw conclusions about the modeled configuration systems. We discuss how to formulate properties in PIDL, assuming a specification $\mathfrak{S}$ with respect to a configuration system. All of the following properties are decidable because of the finiteness of PIDL interpretations (Theorem 4.40).

Soundness. We call a specification PIDL $\mathfrak{S}$ *sound* if it has a model. According to Definition 4.18, this means that there are propositional assignments that satisfy the states in $\mathcal{V}_{\mathfrak{S}}$ as well as the constraints C, that is, each S of $\mathcal{V}_{\mathfrak{S}}$ is consistent with C. This corresponds to domain consistency of configuration systems, which is about consistency properties that are specific to the domain of the the systems, such as compatibility of parts or decision cardinalities. We can encode those properties into the constraints C as propositional formulas. The essential question to ask is if there is a state S that is reachable from the initial state S_I, and $S \cup \mathsf{C} \models \bot$, in which case the specification is not sound. If no such state can be reached, the specification is sound. This can be answered by constructing and inspecting the state graph $\mathcal{G}_{\mathfrak{S}}$, which is finite. From the point of view of the W calculus, this means whether we can derive a clause $(S, \tau, * \,||\, \bot)$ from the initial start clause $(S_I, (), * \,||\, \mathit{start})$ (Corollary 4.29).

Example 4.34. Assume the following parts of a specification $\mathfrak{S}$:

$$S_I = \{A, B\}$$
$$\mathsf{C} \ = \{A \to \neg E\}$$
$$T_R = \{A \rightsquigarrow_1 \{C\},$$
$$\quad C \wedge B \rightsquigarrow_2 \{E\}\}$$

We construct the state graph $\mathcal{G}_{\mathfrak{S}}$ out of the given information. First, it holds that rule transition 1 can be applied to the initial state S_I. We get the rule transition

$$S_I \to_1 S_1$$

with

$$S_1 = S_I \triangleleft E_1 = \{A, B\} \triangleleft \{C\} = \{A, B, C\}.$$

according to the definitions of rule transitions (Definition 4.6) and updates (Definition 4.4). From S_1, there is a rule transition using transition 2 with the following outcome:

$$S_1 \to_2 S_2$$

with

$$S_2 = S_1 \triangleleft E_2 = \{A, B, C\} \triangleleft \{E\} = \{A, B, C, E\}.$$

However, state S_2 is inconsistent with the constraints C, that is,

$$S_2 \cup \mathsf{C} = \{A, B, C, E\} \cup \{A \to \neg E\} \models \bot.$$

We can say that $\mathfrak{S}$ is not sound, because an inconsistent world, represented by S_2, can be reached from the initial state.

Specification Completeness. Given a configuration system, one would want to be able to configure all the products that satisfy the constraints with that system. Such a system is then called complete. We can establish a corresponding notion of completeness for PIDL specifications. A specification is *complete* with respect to a set $\Pi' \subseteq \Pi$ of propositional variables if for every set M of propositional literals with

- $\mathit{vars}(M) = \mathit{vars}(\Pi')$, and
- $M \cup \mathsf{C} \not\models \bot$,

there exists a state $S \in \mathcal{V}_{\mathfrak{S}}$ such that

- $M \subseteq S$,
- $S \cup \mathsf{C} \not\models \bot$, and
- S is rule terminal.

Thus, completeness with respect to a set Π' of propositional variables means that for each literal set formed from Π' and consistent with the constraints, a consistent state can be reached in which the literals are contained. In other words, a valid configuration of the variables can be found in the reachable states. Since we only consider completeness with regard to states that corresponds to states visible to the user in the configuration process, the states have to be rule terminal.

Example 4.35. Assume the following parts of a specification $\mathfrak{S}$:

$$\begin{aligned} S_I &= \{\neg A\} \\ \mathsf{C} &= \{B \leftrightarrow D\} \\ T_U &= \{\neg A \rightarrow_{u_1} \{A, B\}, \\ &\quad\ \neg A \rightarrow_{u_2} \{A, C\}\} \\ T_R &= \{B \rightsquigarrow_{r_1} \{D\}, \\ &\quad\ C \rightsquigarrow_{r_2} \{\neg B, \neg D\}\} \end{aligned}$$

The initial state S_I is rule terminal, and both user transitions u_1 and u_2 can be applied to it. With u_1, we get

$$S_I \rightarrow_{u_1} S_1$$

where

$$S_1 = S_I \triangleleft E_{u_1} = \{\neg A\} \triangleleft \{A, B\} = \{A, B\}.$$

From S_1, a rule transition using r_1 is possible, leading to

$$S_1 \rightarrow_{r_1} S_2$$

with

$$S_2 = S_1 \triangleleft E_{r_1} = \{A, B\} \triangleleft \{D\} = \{A, B, D\}.$$

Applying u_2 to S_I gives

$$S_I \rightarrow_{u_2} S_3$$

where

$$S_3 = S_I \triangleleft E_{u_2} = \{\neg A\} \triangleleft \{A, C\} = \{A, C\}.$$

Finally, rule transition r_2 can be applied to S_3:

$$S_3 \rightarrow_{r_2} S_4$$

with

$$S_4 = S_3 \triangleleft E_{r_2} = \{A, C\} \triangleleft \{\neg B, \neg D\} = \{A, \neg B, C, \neg D\}.$$

The corresponding paths are

$$S_I \rightarrow_{u_1} S_1 \rightarrow_{r_1} S_2$$

and

$$S_I \rightarrow_{u_2} S_3 \rightarrow_{r_2} S_4.$$

The rule-terminal states are S_1, S_2, and S_4. The specification is complete with respect to, for example, the set $\{B, D\}$ of propositional variables, because all the literal combinations of the two variables that are consistent with the constraint $B \leftrightarrow D$ can be reached at rule-terminal states: B and D in state S_2, and $\neg B$ and $\neg D$ in state S_4. On the other hand, the specification is not complete with respect to the set $\{B, C\}$. For example, B and C is a valid combination, that is, the set $\{B, C\}$ is consistent with the constraints, but does not occur in any rule-terminal state.

Metaproperties. A configuration system can have desired metaproperties. For example, one such property could state that if decision `d` is set then decision `d'` must be set as well. We can model scenarios like this with a special kind of reachability, where metaproperties are translated into appropriate propositional formulas ϕ, and then $S \cup \mathsf{C} \models \phi$ is checked for all $S \in \mathcal{V}_{\mathfrak{S}}$ with S being rule terminal. The state S has to be rule terminal because the states that are not correspond to the intermediate states between user decisions and within rule executions in the configuration systems, that is, states that are not visible to the user. Therefore, it is typically not meaningful to take those states into consideration when talking about metaproperties.

Example 4.36. Assume the following parts of a specification $\mathfrak{S}$:

$$\begin{aligned} S_I &= \{C\} \\ \mathsf{C} &= \{\neg(A \wedge E)\} \\ T_R &= \{C \rightsquigarrow_1 \{\neg B\}, \\ &\quad\ \neg B \rightsquigarrow_2 \{A, D\}\} \end{aligned}$$

Moreover, consider the following metaproperty:

$$\phi = A \rightarrow F.$$

There is a rule transition

$$S_I \rightarrow_1 S_1$$

with

$$S_1 = S_I \triangleleft E_1 = \{C\} \triangleleft \{\neg B\} = \{\neg B, C\}.$$

Then, rule transition 2 can be applied to S_1,

$$S_1 \rightarrow_2 S_2$$

with

$$S_2 = S_1 \triangleleft E_2 = \{\neg B, C\} \triangleleft \{A, D\} = \{A, \neg B, C, D\}.$$

We observe that S_2 is rule terminal, which means it is a state that is to be checked for the metaproperty ϕ. Since

$$S_2 \cup \mathsf{C} = \{A, \neg B, C, D\} \cup \{\neg(A \wedge E)\} \not\models A \rightarrow F = \phi,$$

the specification does not satisfy the metaproperty. Note that the constraints C are consistent with all the states appearing in this example, while the rule-terminal state S_2 fails to entail the metaproperty.

Cyclicity. The existence of a cyclic behavior within a configuration run corresponds to the existence of a simple cycle of length greater than one in the state graph $\mathcal{G}_{\mathfrak{S}}$. This means in $\mathcal{G}_{\mathfrak{S}}$ we have

(i) a sequence of transitions

$$S_0 \to S_1 \to \cdots \to S_n$$

with $n \geq 2, S_1, \ldots, S_n$ being distinct, and

(ii) $S_0 = S_n$.

We require the length of the cycle to be greater than one to filter out the the cycles containing just one state, which are not interesting in terms of the consistency of the configuration system modeled. Note that such cycles are dealt with when checking for rule-terminal states. The cyclicity property can be checked by running a cycle-detection algorithm on the state graph $\mathcal{G}_{\mathfrak{S}}$.

Example 4.37. Assume the following parts of a specification $\mathfrak{S}$:

$S_I = \{A, \neg B, \neg C, E\}$

$\mathsf{C} = \emptyset$

$T_U = \{A \rightsquigarrow_{u_1} \{B\}\}$

$T_R = \{B \rightsquigarrow_{r_1} \{C\},$

$C \wedge E \rightsquigarrow_{r_2} \{\neg B, \neg C\}\}$

The initial state is rule terminal, so we can apply user transition u_1 to it. We get

$$S_I \to_{u_1} S_1$$

with

$$S_1 = S_I \triangleleft E_{u_1} = \{A, \neg B, \neg C, E\} \triangleleft \{B\} = \{A, B, \neg C, E\}.$$

From state S_1, there is a rule transition using r_1.

$$S_1 \to_{r_1} S_2$$

where

$$S_2 = S_1 \triangleleft E_{r_1} = \{A, B, \neg C, E\} \triangleleft \{C\} = \{A, B, C, E\}.$$

S_2 entails the condition of rule transition r_2, hence

$$S_2 \to_{r_2} S_3$$

with

$$S_3 = S_2 \triangleleft E_{r_2} = \{A, B, C, E\} \triangleleft \{\neg B, \neg C\} = \{A, \neg B, \neg C, E\}.$$

As we can see, $S_3 = S_I$, which means we have a cycle consisting of the path

$$S_I \to_{u_1} S_1 \to_{r_1} S_2 \to_{r_2} S_I.$$

The example shows a cycle that involves user and rule transitions. The occurrence of user transitions means that in the corresponding configuration system the user can take a decision that eventually leads back to the state at which the user took the decision. It depends on the graph context which the cycle is in and the intentions of those who use the verification to draw further conclusions. We briefly discuss the possible cases and give examples of possible interpretations, considering a cycle of the above form, that is, a cycle containing user and rule transitions:

$$S_0 \to S_1 \to \cdots \to S_n, n \geq 2, S_1, \ldots, S_n \text{ distinct}, S_0 = S_n.$$

- There is a state S_i with $i \in \{1, \ldots, n\}$ such that S_i is rule terminal and there is a user transition $S_i \to S$ that is not part of the cycle.

  This represents the case in which the user can avoid the cycle by taking a different decision. The resulting PIDL transition would lead away from the cycle. One could then interpret the cycle as reflecting a wanted mechanism to let the user enter their input a second time after they have selected an option that is not considered appropriate or correct at that state of the configuration.

- There is a state S_i with $i \in \{1, \ldots, n\}$ such that S_i is not rule terminal and there is a rule transition $S_i \to S$ that is not part of the cycle.

  We can observe two possible subcases.

  (i) The rule transition leads to another rule-terminal state that is outside the cycle. This would violate the property of rule confluence, which is explained further below. Violation of rule confluence basically means that a different order of rule executions results in different rule-terminal states. This is usually seen as an inconsistency in the configuration system.

  (ii) The rule transition leads to another cycle consisting of rule transitions only if (i) is not the case. Because of the finiteness of the state graph of the specification (Theorem 4.40), the path starting from S_i with the rule transition must eventually lead to a cyclic path where we only encounter states that are not rule terminal if we do not reach a rule-terminal state at all. Cycles consisting of only rule transitions are typically seen as an inconsistency in the corresponding configuration system.

- There is no state as specified in the above two cases. This means there is no way to exit the cycle. As in all cases, it is left to the engineers of the configuration systems to interpret and evaluate such scenarios.

We give an example of a cycle in which only rule transitions are involved. Such cycles represent a cyclic behavior in the configuration system's rule base and are usually regarded as inconsistencies.

Example 4.38. Consider the following parts of a specification $\mathfrak{S}$:

$$S_I = \{\neg A, \neg B\}$$
$$\mathsf{C} = \emptyset$$
$$T_U = \{\neg A \rightsquigarrow_{u_1} \{A, C\}\}$$
$$T_R = \{C \rightsquigarrow_{r_1} \{B\},$$
$$A \wedge B \rightsquigarrow_{r_2} \{\neg C\}\}$$
$$A \wedge \neg C \rightsquigarrow_{r_3} \{\neg B, C\}$$

The initial state is rule terminal and implies the user transition u_1.

$$S_I \to_{u_1} S_1$$

where

$$S_1 = S_I \triangleleft E_{u_1} = \{\neg A, \neg B\} \triangleleft_{u_1} \{A, C\} = \{A, \neg B, C\}.$$

Rule transition r_1 can be applied to S_1:

$$S_1 \to_{r_1} S_2$$

with

$$S_2 = S_1 \triangleleft E_{r_1} = \{A, \neg B, C\} \triangleleft \{B\} = \{A, B, C\}.$$

There is a rule transition from S_2 to S_3 using r_2:

$$S_2 \rightarrow_{r_2} S_3$$

where

$$S_3 = S_2 \triangleleft E_{r_2} = \{A, B, C\} \triangleleft \{\neg C\} = \{A, B, \neg C\}.$$

Applying rule transition r_3 to S_3 gives us

$$S_3 \rightarrow_{r_3} S_4$$

with

$$S_4 = S_3 \triangleleft E_{r_3} = \{A, B, \neg C\} \triangleleft \{\neg B, C\} = \{A, \neg B, C\}.$$

With $S_4 = S_1$, we have a cycle consisting of the path

$$S_1 \rightarrow_{r_1} S_2 \rightarrow_{r_2} S_3 \rightarrow_{r_3} S_1.$$

Confluence. Again, confluence properties of the configuration system are expressed as properties of the state graph $\mathcal{G}_{\mathfrak{S}}$ of the corresponding specification $\mathfrak{S}$, which can be identified by graph-based algorithms. We distinguish between two kinds of confluence. The state graph $\mathcal{G}_{\mathfrak{S}}$ is

- *rule confluent* if for each state $S \in \mathcal{V}_{\mathfrak{S}}$ the next rule-terminal state $S' \in \mathcal{V}_{\mathfrak{S}}$ that can be reached via a sequence of rule transitions and that is not S itself is unique,
- *user confluent* if for for each pair of rule-terminal states $S, S' \in \mathcal{V}_{\mathfrak{S}}$ it holds that if S and S' are reachable from the initial state S_I using the same set of user transitions, then $S = S'$.

Example 4.39. We show small examples of specifications that are confluent and specifications that violate the confluence property.

- Consider the following parts of a rule-confluent specification $\mathfrak{S}_1$:

  $S_I = \{A, C\}$

  $\mathsf{C} = \emptyset$

  $T_R = \{C \rightsquigarrow_1 \{B, E\},$

  $\quad A \wedge C \rightsquigarrow_2 \{F\}\}$

  We begin with a rule transition using transition 1 on the initial state:

  $$S_I \rightarrow_1 S_1,$$

  where

  $$S_1 = S_I \triangleleft E_1 = \{A, C\} \triangleleft \{B, E\} = \{A, B, C, E\}.$$

  Then, there is a rule transition using transition 2,

  $$S_1 \rightarrow_2 S_2,$$

  with the resulting state

  $$S_2 = S_1 \triangleleft E_2 = \{A, B, C, E\} \triangleleft \{F\} = \{A, B, C, E, F\},$$

which is rule terminal.

In the initial state, there is also the possibility of applying rule transition 2, giving us

$$S_I \rightarrow_2 S_3,$$

with

$$S_3 = S_I \triangleleft E_2 = \{A, C\} \triangleleft \{F\} = \{A, C, F\}.$$

State S_3 entails transition 1, where the updated state is again S_2:

$$S_3 \rightarrow_1 S_2,$$

with

$$S_2 = S_3 \triangleleft E_1 = \{A, C, F\} \triangleleft \{B, E\} = \{A, B, C, E, F\}.$$

This means from the initial state S_I, all possible applications of rule transitions lead to the same rule-terminal state S_2:

$$S_I \rightarrow_1 S_1 \rightarrow_2 S_2, \text{ and}$$

$$S_I \rightarrow_2 S_3 \rightarrow_1 S_2.$$

Thus, the above specification $\mathfrak{S}_1$ is rule confluent.

- The following specification $\mathfrak{S}_2$ violates rule confluence and its relevant components are:

  $S_I = \{A, \neg B\}$

  $\mathsf{C} = \emptyset$

  $T_R = \{\neg B \rightsquigarrow_1 \{C, \neg F\},$

  $A \rightsquigarrow_2 \{B\}\}$

  There is a rule transition using transition 1 from the initial state:

  $$S_I \rightarrow_1 S_1$$

  with

  $$S_1 = S_I \triangleleft E_1 = \{A, \neg B\} \triangleleft \{C, \neg F\} = \{A, \neg B, C, \neg F\}.$$

  This is followed by applying transition 2 on S_1:

  $$S_1 \rightarrow_2 S_2$$

  where

  $$S_2 = S_1 \triangleleft E_2 = \{A, \neg B, C, \neg F\} \triangleleft \{B\} = \{A, B, C, \neg F\}.$$

  S_2 is rule terminal. At the same time, transition 2 can be applied to the initial state as well:

  $$S_I \rightarrow_2 S_3$$

  where

  $$S_3 = S_I \triangleleft E_2 = \{A, \neg B\} \triangleleft \{B\} = \{A, B\},$$

  also a rule-terminal state. Consequently, we have two possible and different rule-terminal states that can be reached from the initial state by an exhaustive application of the rule transitions:

  $$S_I \rightarrow_1 S_1 \rightarrow_2 S_2, \text{ and}$$

  $$S_I \rightarrow_2 S_3.$$

  This means the specification $\mathfrak{S}_2$ is not rule confluent.

- Consider the following specification $\mathfrak{S}_3$, which is user confluent:

  $S_I = \{\neg A, \neg C\}$

  $\mathsf{C} = \emptyset$

  $T_U = \{\neg A \rightsquigarrow_{u_1} \{A\},$
  $\neg C \rightsquigarrow_{u_2} \{C\}\}$

  $T_R = \{A \wedge C \rightsquigarrow_{r_1} \{B\}\}$

  There is a transition using u_1 to the initial state,

  $$S_I \rightarrow_{u_1} S_1$$

  with

  $$S_1 = S_I \triangleleft E_{u_1} = \{\neg A, \neg C\} \triangleleft \{A\} = \{A, \neg C\}.$$

  From there, we can apply user transition u_2:

  $$S_1 \rightarrow_{u_2} S_2$$

  with

  $$S_2 = S_1 \triangleleft E_{u_2} = \{A, \neg C\} \triangleleft \{C\} = \{A, C\}.$$

  S_2 entails rule transition r_1:

  $$S_2 \rightarrow_{r_1} S_3$$

  where

  $$S_3 = S_2 \triangleleft E_{r_1} = \{A, C\} \triangleleft \{B\} = \{A, B, C\}.$$

  If user transition u_2 is applied to the initial state, we get

  $$S_I \rightarrow_{u_2} S_4$$

  with

  $$S_4 = S_I \triangleleft E_{u_2} = \{\neg A, \neg C\} \triangleleft \{C\} = \{\neg A, C\}.$$

  S_4 is rule terminal, and user transition u_1 can be applied to it, which leads to the known state S_2:

  $$S_4 \rightarrow_{u_1} S_2$$

  with

  $$S_2 = S_4 \triangleleft E_{u_1} = \{\neg A, C\} \triangleleft \{A\} = \{A, C\}.$$

  As can be seen above, the rule-terminal state S_3 can be reached from S_2 with transition r_1.

  We observe that the rule-terminal state S_3 is reachable from the initial state S_I via two possible paths, both involving the user transitions u_1 and u_2. In one path, u_1 is executed before u_2, while the reverse takes place in the other path. The paths show that it does not matter in which order the user transitions are applied. In both cases, the same rule-terminal state S_3 is reached:

  $$S_I \rightarrow_{u_1} S_1 \rightarrow_{u_2} S_2 \rightarrow_{r_1} S_3, \text{ and}$$

  $$S_I \rightarrow_{u_2} S_4 \rightarrow_{u_1} S_2 \rightarrow_{r_1} S_3.$$

  It follows that specification $\mathfrak{S}_3$ is user confluent.

- We can make the previous example break user confluence by introducing a slight change. Assume the following specification $\mathfrak{S}_4$:

  $S_I = \{\neg A, \neg C\}$

  $\mathsf{C} = \emptyset$

  $T_U = \{\neg A \rightsquigarrow_{u_1} \{A\},$
  $\neg C \rightsquigarrow_{u_2} \{C\}\}$

  $T_R = \{A \wedge \neg C \rightsquigarrow_{r_1} \{B\}\}$

  The difference to $\mathfrak{S}_3$ is that rule transition r_1 has condition $A \wedge \neg C$.

  We apply user transition u_1 to S_I:

  $$S_I \rightarrow_{u_1} S_1,$$

  where

  $$S_1 = S_I \triangleleft E_{u_1} = \{\neg A, \neg C\} \triangleleft \{A\} = \{A, \neg C\}.$$

  Applying r_1 to S_1 yields

  $$S_1 \rightarrow_{r_1} S_2$$

  with

  $$S_2 = S_1 \triangleleft E_{r_1} = \{A, \neg C\} \triangleleft \{B\} = \{A, B, \neg C\}.$$

  S_2 is rule terminal. There is a user transition from S_2, using u_2:

  $$S_2 \rightarrow_{u_2} S_3$$

  with

  $$S_3 = S_2 \triangleleft E_{u_2} = \{A, B, \neg C\} \triangleleft \{C\} = \{A, B, C\}.$$

  This is the same rule-terminal state that is reached in the example of $\mathfrak{S}_3$. The following shows that this time we eventually get a different rule-terminal state when we apply user transition u_2 to the initial state:

  $$S_I \rightarrow_{u_2} S_4$$

  where

  $$S_4 = S_I \triangleleft E_{u_2} = \{\neg A, \neg C\} \triangleleft \{C\} = \{\neg A, C\}.$$

  S_4 is rule terminal, user transition u_1 can be applied:

  $$S_4 \rightarrow_{u_1} S_5$$

  with

  $$S_5 = S_4 \triangleleft E_{u_1} = \{\neg A, C\} \triangleleft \{A\} = \{A, C\}.$$

  State S_5 is rule terminal. We see that it is different from the other rule-terminal state S_3. The paths are the following:

  $$S_I \rightarrow_{u_1} S_1 \rightarrow_{r_1} S_2 \rightarrow_{u_2} S_3, \text{ and}$$

  $$S_I \rightarrow_{u_2} S_4 \rightarrow_{u_1} S_5.$$

  Both paths have the user transitions u_1 and u_2, but the order in which they are applied is different, which leads to two distinct rule-terminal states. Therefore, $\mathfrak{S}_4$ is not user confluent.

The properties mentioned in this section are decidable for PIDL, which follows from the finiteness of the semantics of a PIDL specification:

Theorem 4.40. The components $\mathcal{V}_{\mathfrak{S}}$ and $\mathcal{T}_{\mathfrak{S}}$ of a PIDL interpretation are finite.

Proof. A PIDL state S is a finite set of propositional literals by Definition 4.3. This is not changed by the updates in the transitions since the update sets are also finite sets of literals. The number of possible states is bounded because the set of propositional variables Π is finite. Hence, the state space $\mathcal{V}_{\mathfrak{S}}$ is a finite set. This and the finite number of transitions T_R and T_U mean that the transition space $\mathcal{T}_{\mathfrak{S}}$ is finite as well. □

Theorem 4.41. Let $\mathfrak{S} = (\Pi, S_I, \mathsf{C}, T_U, T_R)$ be a PIDL specification. The state space $\mathcal{V}_{\mathfrak{S}}$ has worst-case space complexity $O(3^{|\Pi|})$.

Proof. There are three possibilities for a propositional variable P of Π with respect to its presence in a state of $\mathcal{V}_{\mathfrak{S}}$: As a literal P, as a literal $\neg P$, and not present at all. Hence, there are $O(3^{|\Pi|})$ possibilities a state can be constructed. □

4.5 Translating DOPLER

Before automated verification of a configuration system with PIDL can happen, one needs to determine the relevant components of the system and then express them with the syntactical means we have presented earlier in this chapter. In this section, we demonstrate how we can use PIDL to represent a DOPLER configuration instance. We again work with the example of a DOPLER model shown in Figure 2.2 in Chapter 2 and go through its elements and the domain constraints, and explain how to formulate these in the language of PIDL, which means deriving a PIDL specification $\mathfrak{S}$ from the DOPLER model.

Considering what we learnt about the DOPLER system in Chapter 2, we identify the following aspects that should be the subject of the modeling with PIDL:

- Decisions,
- assets,
- DOPLER functions,
- visibility,
- rules,
- user interaction,
- the initial state, and
- properties.

Decisions in DOPLER take the form of variables and are the basic unit to influence how the products should look like. Assets are the parts that can be used to construct the final product. Their inclusion in the product usually depends on the decisions. DOPLER uses a number of functions such as `containsAny` in its expressions to formulate statements about decisions in the rule and visibility conditions. Decision variables can be set by the user, but only if the variables are visible to the user. Sometimes it is useful to hide variables, so DOPLER offers visibility as a means to control which decisions can be taken

by the user and which cannot. Rules of the system enforce domain-specific restrictions by automatically setting decisions depending on the current values of other decision variables. Besides rules, as already mentioned, users can take decisions according to their preferences. We also have to somehow express what the initial configuration state is, which is the state before any user interaction and rule execution. Each of the above item is considered in the modeling of a DOPLER instance in PIDL. In the following, we show how the DOPLER elements are translated into PIDL and give short examples referring to the DOPLER model described in Chapter 2.

Decisions. There are two types of decisions in DOPLER, Boolean decisions and enumeration decisions. For each Boolean decision `d`, we introduce two propositional variables in Π of the specification $\mathfrak{S}$.

DOPLER	PIDL
Boolean decision `d`	$d\_Yes, d\_No \in \Pi$
`stainlessSteel`	$stainlessSteel\_Yes, stainlessSteel\_No \in \Pi$

We create two propositional variables by adding the suffixes *Yes* and *No* to the decision variable name, where d_Yes means that the value of `d` is true, and d_No means that the value is false. One might argue that since `d` is a Boolean variable, one propositional variable might be sufficient to model it in PIDL. The reason we choose to have two variables is that we can express the fact that a certain decision has not been set yet that way. For example, some transition might require that certain decisions have no value, which can be expressed with $\neg d\_Yes \wedge \neg d\_No$. This is not possible by having only one variable representing the decision in PIDL.

With two variables explicitly expressing the two possible values of a DOPLER Boolean decision `d`, it is necessary to take the fact that `d` cannot be simultaneously true and false into account in PIDL. We do this by encoding this mutual-exclusivity property into the states and transitions, that is, we make sure that this property is not violated in the initial state and it is not violated by the transition updates. This is discussed in more detail when we explain the translation of user and rule actions. An alternative would be to include $\neg(d\_Yes \wedge d\_No)$ in the constraints C. However, if combined with relaxing the above requirements on the states and transitions, we effectively disallow the changing of Boolean decisions: If a state has $d\_Yes (d\_No)$, a transition with just $d\_No (d\_Yes)$ in its update will lead to an inconsistent state.

Note that, in general, any identifier can be used for the propositional variables. For our purposes, we choose to use names that are derived from the objects in the DOPLER model they represent and that indicate their purpose intuitively.

Enumeration decisions in DOPLER can have multiple values to choose from. For each option `o` of an enumeration decision `d`, we introduce a propositional variable d_o in PIDL. The variable d_o means option `o` of decision `d` is selected.

DOPLER	PIDL
Enumeration decision `d` with options $\mathtt{o}_1$, $\mathtt{o}_2$, ..., $\mathtt{o}_n$	$d\_o_1, d\_o_2, \ldots, d\_o_n \in \Pi$
`casterType` with options `slab`, `bloom`, `beam`	$casterType\_slab$, $casterType\_bloom$, $casterType\_beam \in \Pi$

The cardinalities of enumeration decisions, that is, the number of options that can be selected, can be expressed as constraints as shown later when we talk about translating inconsistencies of DOPLER models.

Assets. Each asset `a` in DOPLER is represented by a propositional variable a in PIDL. The presence of such a variable a then means that asset `a` is included in the final product. An asset inclusion condition $\mathcal{A}$ of an asset `a` is translated to a logical formula ϕ, and the formula $\phi \rightarrow a$ is added to the constraints C. This way, it becomes clear what components are part of the current configuration at each state. Finally, inclusion and exclusion relationships between assets are modeled with implications, also contained in the constraints C: A formula of the kind $a \rightarrow b$ is to be interpreted as asset `a` including asset `b`, and a formula $a \rightarrow \neg b$ means `a` excludes `b`.

DOPLER	**PIDL**
Asset `a`	$a \in \Pi$
`baleAdapter`	$baleAdapter \in \Pi$
Inclusion condition $\mathcal{A}$ of `a`	$\phi \rightarrow a \in \mathsf{C}$ $\phi \in F_{\Pi}$, logical translation of $\mathcal{A}$
`containsOnly(casterType, slab)`	$(casterType\_slab \wedge$ $\neg casterType\_bloom \wedge$ $\neg casterType\_beam \rightarrow$ $baleAdapter) \in \mathsf{C}$
`a` includes `b` `a` excludes `b`	$a \rightarrow b \in \mathsf{C}$ $a \rightarrow \neg b \in \mathsf{C}$
`baleAdapter` includes `pCalibthermometer` `calibrator` excludes `pCalibthermometer`	$baleAdapter \rightarrow pCalibthermometer \in \mathsf{C}$ $calibrator \rightarrow \neg pCalibthermometer \in \mathsf{C}$

DOPLER functions. The DOPLER functions are Boolean functions that check which values are currently assigned to certain decisions. We translate each such function into its corresponding logical formula. The resulting formulas are usually contained in the condition part of transitions, and in the constraints to express asset inclusion and relationships.

DOPLER	PIDL
`containsOnly(d, o`$_i$`)` enumeration decision `d` with options `o`$_1$`, o`$_2$`, ..., o`$_n$	$\neg d\_o_1 \wedge \neg d\_o_2 \wedge \cdots \wedge d\_o_i \wedge \cdots \wedge \neg d\_o_n$
`containsOnly(casterType, beam)`	$\neg casterType\_slab \wedge$ $\neg casterType\_bloom \wedge$ $casterType\_beam$
`containsAny(d, o`$_{i_1}$`, o`$_{i_2}$`, ..., o`$_{i_m}$`)`	$d\_o_{i_1} \vee d\_o_{i_2} \vee \cdots \vee d\_o_{i_m}$
`containsAny(casterType, slab, beam)`	$casterType\_slab \vee casterType\_beam$
`containsAll(d, o`$_{i_1}$`, o`$_{i_2}$`, ..., o`$_{i_m}$`)`	$d\_o_{i_1} \wedge d\_o_{i_2} \wedge \cdots \wedge d\_o_{i_m}$
`containsAll(casterType, slab, beam)`	$casterType\_slab \wedge casterType\_beam$
`isTaken(d)` Boolean decision `d`	$d\_Yes \vee d\_No$
`isTaken(stainlessSteel)`	$stainlessSteel\_Yes \vee stainlessSteel\_No$
`isTaken(d)` enumeration decision `d` with options `o`$_1$`, o`$_2$`, ..., o`$_n$	$d\_o_1 \vee d\_o_2 \vee \cdots \vee d\_o_n$
`isTaken(casterType)`	$casterType\_slab \vee$ $casterType\_bloom \vee$ $casterType\_beam$

There is one more function that is used in DOPLER models, namely the `setValue` function. It assigns values to decision variables in the action part of the system rules. We provide the translation of this function further below when we talk about how to deal with rule actions in PIDL.

Visibility. We express the visibility of decisions by introducing a propositional visibility variable $visible\_d$ for each decision `d`, for the fact that `d` is visible. Moreover, we introduce a formula $\phi \rightarrow visible\_d$ in the constraints C to model the visibility mechanism of DOPLER, where the subformula ϕ is a propositional formula and, possibly using the logical translations of DOPLER functions as shown above, represents the visibility condition of decisions: If the condition ϕ is true, then the decision is visible.

DOPLER	PIDL
Decision `d`	$visible\_d \in \Pi$
`hydraulicCylinder`	$visible\_hydraulicCylinder \in \Pi$
Visibility condition $\mathcal{C}$ of `d`	$\phi \rightarrow visible\_d \in \mathsf{C}$ $\phi \in F_\Pi$, logical translation of $\mathcal{C}$
`containsOnly(casterType,` `slab) && !taperUnit`	$(casterType\_slab \wedge$ $\neg casterType\_bloom \wedge$ $\neg casterType\_beam \wedge$ $taperUnit\_No \rightarrow$ $visible\_hydraulicCylinder) \in \mathsf{C}$

Rules. We transform a DOPLER rule `if <condition> then <action>` into a rule transition in T_R. The `<condition>` part of the rule is translated into a formula over

Π, which is the condition χ of the transition. Occurrences of DOPLER functions are translated according to the above list.

The `<action>` part assigns values to decisions. These assignments are mapped to propositional literals in the update set E of the transition, where a distinction has to be made in terms of the type of the decisions involved. If the decision `d` is Boolean, then the action expression is `d = v`, where `v` is true or false. As for the translation,this means that either d_Yes and $\neg d\_No$, or $\neg d\_Yes$ and d_No are added to E. The former case says that `d` has been assigned the value true, while the latter means `d` has value false. In both cases, we need two literals to express the assignments, which is necessary because of the mutual exclusivity of Boolean values, as mentioned earlier. If the decision `d` is an enumeration decision, the action is of the form `setValue(d, o`$_{i_1}$`, ..., o`$_{i_n}$`)`, where `o`$_{i_1}$`, ..., o`$_{i_n}$ are valid options of the decision. Here, we include the corresponding literals $d\_o_{i_1}, \ldots, d\_o_{i_n}$ in E. The resulting transition $\chi \rightsquigarrow E$ then becomes an element of the rule transitions T_R.

DOPLER	PIDL
`<condition>`	$\chi \in F_{\Pi}$, logical translation of `<condition>`

`d = true` in `<action>`
`d = false` in `<action>`
`setValue(d, o`$_{i_1}$`, ..., o`$_{i_n}$`)`
	$d\_Yes, \neg d\_No \in E$
	$\neg d\_Yes, d\_No \in E$
	$d\_o_{i_1}, \ldots, d\_o_{i_n} \in E$

`if <condition> then <action>`	$\chi \rightsquigarrow E \in T_R$

`!gapChecker then`
`taperUnit = true`
 	
`if molder then`
`setValue(casterType, bloom)`
	$gapChecker\_No \rightsquigarrow$
	$\{taperUnit\_Yes, \neg taperUnit\_No\} \in T_R$
	
	$molder\_Yes \rightsquigarrow \{casterType\_bloom\} \in T_R$

User interaction. Decision making that is caused by the user is represented by user transitions $\chi \rightsquigarrow E$ in PIDL. The χ part is a formula expressing what condition must be fulfilled so that the user can set a variable. In our framework, χ is always the following statement: The decision is visible and it has not been taken yet. The update set E then contains the literals that represent a possible variable assignment, as done in the rule transitions. That means that, depending on the type of the decision, from a decision we derive several user transitions to cover each possible case in which the user assigns certain values to the decision.

DOPLER	PIDL
Boolean decision `d`	$\chi = visible\_d \wedge \neg d\_Yes \wedge \neg d\_No$ $E = \{d\_Yes, \neg d\_No\}$ $E' = \{\neg d\_Yes, d\_No\}$ $\chi \rightsquigarrow E \in T_U, \chi \rightsquigarrow E' \in T_U$
`stainlessSteel`	$\chi = visible\_stainlessSteel \wedge$ $\neg stainlessSteel\_Yes \wedge$ $\neg stainlessSteel\_No$ $E = \{stainlessSteel\_Yes, \neg stainlessSteel\_No\}$ $E' = \{\neg stainlessSteel\_Yes, stainlessSteel\_No\}$ $\chi \rightsquigarrow E \in T_U, \chi \rightsquigarrow E' \in T_U$

In the case of a Boolean decision `d`, we get two user transitions. One transition represents the user setting `d = true` by d_Yes, the other transition represents the user decision `d = false` by d_No in the update sets. Note that by also including $\neg d\_No$ and $\neg d\_Yes$ in the respective update sets we make sure the invariance that Boolean decisions can only have one value at the same time is preserved after an update.

Translations of enumeration decisions work in the same way except that the update sets have to correspond to the cardinality of the decisions.

The initial state. The initial state S_I of literals represents the initial configuration state. Any combination of literals is possible as the initial state as long as the mutual exclusivity of Boolean decisions is not violated. One obvious possibility is to include the negations of all variables representing the decisions in the initial state, to express a situation in which nothing has been selected yet.

Inconsistencies. We showed some types of inconsistencies with the help of an example DOPLER model in Section 2.2.2. We now discuss how they can be expressed and dealt with in the context of PIDL.

- *Domain consistency*: We consider two prominent cases of domain consistency of DOPLER models. First, there are cardinality restrictions on enumeration decisions, stating how many options can be selected. We would typically translate these restrictions as propositional formulas over Π involving the corresponding variables representing the decisions and include the formulas in the set of constraints C. For example, with

  $$\neg(casterType\_slab \wedge casterType\_bloom \wedge casterType\_beam) \in \mathsf{C}$$

  we express that not all three options of the decision `casterType` can be selected at the same time. In fact, the cardinality of that decision, which states that only one option may be selected, necessitates three more such formulas, to forbid the selection of two options:

  $$\neg(casterType\_slab \wedge casterType\_bloom) \in \mathsf{C},$$

  $$\neg(casterType\_slab \wedge casterType\_beam) \in \mathsf{C}, \text{ and}$$

  $$\neg(casterType\_bloom \wedge casterType\_beam) \in \mathsf{C}.$$

  The second type of domain-specific constraints we mention here are asset compatibilities. There can be assets whose inclusion imply further inclusion of other assets, and there can also be exclusions of assets by the inclusion of some assets. We have already shown above how to translate these relationship properties of assets as formulas in the set of constraints C.

- *Metaproperties*: As an example of a metaproperty, the DOPLER instance from Chapter 2 requires that if the decision `stainlessSteel` has been taken, it is expected that the values of both of the decisions `casterType` and `hydraulicCylinder` are assigned as a consequence. We write this as the following propositional formula:

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

Table 4.1: Properties in the DOPLER model example.

	Number of States
Total	99
Decision Cardinalities	12
Asset Inclusion Conflicts	12
Violation of Metaproperties	7
Cycle	*detected*

Then it can be checked for each rule-terminal state in the state graph if the above formula is entailed by the state.

- *Cyclicity*: The state graph obtained from the specification of the DOPLER model allows the treatment of the cyclicity property just as described in Section 4.4.

- *Confluence*: Likewise, we can check confluence of a DOPLER instance by examining confluence of the state graph of its corresponding specification as described in Section 4.4.

4.6 Experiments

The previous sections expound the foundations of PIDL as a logic for representing rule-based configuration systems. As a first step in gaining practical experience, we have implemented a prototype program that, with the exception of confluence, verifies DOPLER models with respect to the mentioned properties, and that is based on the theoretical framework provided by PIDL. Given a DOPLER configuration instance, the program derives a PIDL specification $\mathfrak{S}$ and constructs the major part of the corresponding interpretation, that is, the state graph $\mathcal{G}_{\mathfrak{S}}$. Its reasoning procedure to compute the states is based on the W calculus. We then did experiments on the DOPLER example model described in Chapter 2 and a set of randomly generated configuration problems, the results of which we present in this section.

The example DOPLER instance as depicted in Chapter 2 contains a representative range of properties that are generally interesting with respect to the correctness of configuration systems. In the instance, there are certain errors that one would typically like to identify by verification and that are explained in Section 2.2.2: Violation of decision cardinalities, conflicting asset inclusions, cycles in the rule base, and metaproperties. We translated the configuration instance into a format readable for our implementation and then let the tool create the corresponding specification and its state graph. The construction of the states effectively resembles a SAT problem for each state, based on the rules of W. Domain-specific inconsistencies, that is, states with incorrect cardinalities, asset inclusions, and violation of metaproperties, are found during the computation of the states. Inconsistency concerning the presence of a cycle is dealt with by a standard approach based on depth-first search that examines the state graph for cycles. We ran the instance on a machine with an Intel Xeon E5-4640 at 2.4 GHz and 512 GB of RAM, which took 0.037 seconds. The results are recorded in Table 4.1. Out of the 99 generated states, decision cardinality and asset inclusion conflicts can both be found in 12 states each. Violation of metaproperties occurs in 7 states. The built-in cycle in the DOPLER model is detected by the program.

Running the prototype on the DOPLER instance illustrates how the PIDL approach can verify important properties of the system automatically. Because the size of this

example problem with 8 decisions and 10 rules is rather limited, we additionally carried out experiments on problems that are generated randomly. Those problems are divided into three groups in which problems consist of 20, 60, and 100 Boolean decisions respectively. Each group contains 20 single randomly generated problems. The properties that are checked here are domain consistency and cycles. In each problem instance, we have rules of the following format:

```
if (d || [!]e && [!]f) then g = [true/false],
```

where `d`, `e`, `f`, and `g` are pairwise distinct Boolean decisions. The expression in the condition part contains a disjunction and a conjunction. There can be negation signs before `e` and `f`, and in the action part `g` is assigned one of the Boolean values. We set a 1:1.5 ratio of variables to rules. At most half of the variables are visible, and visible variables do not appear in the action part of the rules in order to have a minimum degree of involvement of rule transitions in the generation of states. This means the number of visible variables, that is, variables that can be set by user transitions, vary from instance to instance. Moreover, we simulated domain-consistency properties by including random formulas of the type

```
([!]d || [!]e || [!]f)
```

in the constraints C of the respective specifications. The ratio of variables to constraints is 1:1.

The results of the experiments on random DOPLER models are shown in Table 4.2. A run is aborted with the result "inconsistent" if it detects a state in which the constraints are broken. If an instance turns out to have no domain inconsistency, it is given how many states the state graph has and whether a cycle is in the model. For example, the result 1079/Y of the instance rnd_2 means that there were 1079 states in total and a cycle was found in the instance. Only the group of problems with 20 variables had instances that are domain consistent. The runs of those problems mostly took less than 1 second. If we look at the category of 60 variables, the experiments noticeably started to have longer runtimes between a couple of seconds and as much as over 7 minutes. Every instance in that group is inconsistent. The group of 100 variables shows a similar picture, except for 4 runs that did not finish before a timeout of 12 minutes.

We see that some randomly generated problems can be difficult for the prototype to solve. More refined techniques on the implementation side, in particular those which identify and exploit the structure of the problems might help to reduce the search space, which potentially has 3^{v+a} states, where v is the number of visible decisions and a is the number of decisions in the action part of the rules. However, the experiments indicate that PIDL can in principle be a useful basis for the automated verification of rule-based configuration systems.

Table 4.2: Generated random DOPLER models.

20 variables, 30 rules			
Name	Visible Variables	Time	Results
rnd_1	5	0m0.05s	inconsistent
rnd_2	3	0m1.00s	1079/Y
rnd_3	6	0m1.45s	inconsistent
rnd_4	4	0m0.07s	inconsistent
rnd_5	4	0m0.05s	inconsistent
rnd_6	6	0m0.04s	inconsistent
rnd_7	3	0m0.03s	inconsistent
rnd_8	2	0m0.05s	inconsistent
rnd_9	3	0m0.09s	inconsistent
rnd_10	2	0m0.26s	211/N
rnd_11	3	0m0.04s	inconsistent
rnd_12	2	0m0.04s	9/N
rnd_13	5	0m0.20s	inconsistent
rnd_14	7	0m0.06s	inconsistent
rnd_15	3	0m0.02s	inconsistent
rnd_16	5	0m1.11s	inconsistent
rnd_17	3	0m0.11s	inconsistent
rnd_18	4	0m0.24s	inconsistent
rnd_19	3	0m0.61s	558/Y
rnd_20	4	0m0.43s	inconsistent
60 variables, 90 rules			
Name	Visible Variables	Time	Results
rnd_21	16	0m0.47s	inconsistent
rnd_22	10	0m0.61s	inconsistent
rnd_23	11	0m4.38s	inconsistent
rnd_24	12	0m2.84s	inconsistent
rnd_25	13	7m44.81s	inconsistent
rnd_26	14	4m51.23s	inconsistent
rnd_27	14	0m0.38s	inconsistent
rnd_28	13	0m0.39s	inconsistent
rnd_29	15	0m0.51s	inconsistent
rnd_30	15	0m0.77s	inconsistent
rnd_31	15	0m0.70s	inconsistent
rnd_32	14	0m0.30s	inconsistent
rnd_33	11	0m1.01s	inconsistent
rnd_34	15	0m0.65s	inconsistent
rnd_35	12	0m36.00s	inconsistent
rnd_36	16	0m0.50s	inconsistent
rnd_37	9	0m2.00s	inconsistent
rnd_38	14	0m0.40s	inconsistent
rnd_39	11	2m13.94s	inconsistent
rnd_40	10	0m44.69s	inconsistent
100 variables, 150 rules			
Name	Visible Variables	Time	Results
rnd_41	24	0m1.55s	inconsistent
rnd_42	24	>12m	-
rnd_43	25	0m2.42s	inconsistent
rnd_44	22	>12m	-
rnd_45	18	5m59.85s	inconsistent
rnd_46	29	0m0.81s	inconsistent
rnd_47	20	0m1.51s	inconsistent
rnd_48	22	0m2.84s	inconsistent
rnd_49	23	7m15.73s	inconsistent
rnd_50	19	0m42.68s	inconsistent
rnd_51	26	>12m	-
rnd_52	28	0m16.12s	inconsistent
rnd_53	21	0m1.28s	inconsistent
rnd_54	17	0m0.73s	inconsistent
rnd_55	18	0m1.48s	inconsistent
rnd_56	25	0m2.18s	inconsistent
rnd_57	20	0m1.34s	inconsistent
rnd_58	21	>12m	-
rnd_59	21	0m1.13s	inconsistent
rnd_60	23	0m1.56s	inconsistent

Chapter 5

PIDL+

PIDL formalizes interactive rule-based configuration systems that feature Boolean decisions. In practice, there are also configuration systems that do not only handle Boolean variables but also work with numbers, depending on the domain, such as the amount of a certain liquid in liters, the length of a required steel beam, or the number of expected users of the product. We call variables that are in arithmetic expressions and that take numerical values arithmetic variables, corresponding to the set X mentioned in Chapter 3.

In this chapter, we present an extended version of PIDL, called PIDL+, which still features propositional variables but also additionally contains an arithmetic component. More precisely, we allow expressions that come from the first-order theory of the reals. Increasing expressiveness has serious implications for the semantics of the logic. In particular, we have to define how decisions with respect to numeric variables are represented while preserving decidability. We do this by a priori bounding the variables relevant to the configuration flow and considering intervals for arithmetic variables instead of single values. With PIDL+, configuration systems containing arithmetic constraints can be represented as admissible specifications. Selections, expressed as simple bounds in the states, allows to split up the range of user decisions into more compact representations. Admissible PIDL+ specifications have finite models that, as in PIDL, cover all the configuration steps that are possible with the given configuration systems, featuring the same transition principle with user transitions, rule transitions, and rule termination as seen in PIDL. We discuss the syntax in Section 5.1 and semantics in Section 5.2. Then, we give a sound and complete algorithm that computes partial models of consistent PIDL+ specifications in Section 5.3, serving as the basis for further verification procedures with respect to the modeled configuration systems. Properties of admissible PIDL+ specifications are listed in Section 5.4.

5.1 Syntax

Analogously to PIDL, we encode the relevant elements of a configuration system in a PIDL+ specification. We first give the formal definition and then provide motivations to each part of a specification. More detailed explanations are given in the semantics section, as the meaning of the specification design is best understood in connection with the semantics of PIDL+. The logic is in particular based on elements from the theory with respect to the signature Σ as defined in the preliminaries in Chapter 3, which contains the definitions of the corresponding concepts used in the following definition.

Definition 5.1. A *PIDL+ specification* is a seven-tuple $\mathfrak{S}_+ = (\Pi, X, S_I, U_I, \mathsf{C}, T_U, T_R)$, where

- 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*, where
  - $U_I \subseteq X$,
  - S_I is a finite set of
    - simple bounds $x \circ c$ in $F_\Sigma(X, \Pi)$ and
    - propositional literals over Π,
- C is a finite set of quantifier-free Σ-formulas in $F_\Sigma(X, \Pi)$, called the *constraints*,
- T_U is a finite set of tuples $\Lambda_i \wedge F_i \rightsquigarrow E_i$ called *user transitions*, where
  - Λ_i is a conjunction of simple atoms $x \circ t$ over Σ, called the *arithmetic condition* of the transition,
  - F_i is a conjunction of literals over Π, called the *propositional condition* of the transition, and
  - E_i is a finite, satisfiable set of simple bounds $x \circ c$ over Σ and propositional literals over Π, called the *update set* of the transition,
- T_R is a finite set of tuples $\Lambda_j \wedge F_j \rightsquigarrow E_j$ called *rule transitions*, where
  - Λ_j is a conjunction of simple atoms $x \circ t$ over Σ, called the *arithmetic condition* of the transition,
  - F_j is a conjunction of literals over Π, called the *propositional condition* of the transition, and
  - E_j is a finite, satisfiable set of simple atoms $x \circ t$ over Σ and propositional literals over Π, called the *update set* of the transition.

All the transitions in T_U and T_R have different indices i and j.

Analogously to PIDL, we allow certain relaxations of the way transitions are written:

Convention 5.2. We may omit the index i when writing a transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U \cup T_R$ if the exact transition does not matter. In examples, we usually rather write $\Lambda \wedge F \rightsquigarrow_i E$ instead of $\Lambda_i \wedge F_i \rightsquigarrow E_i$ when concrete instances of Λ, F, and E are used. We may also write u_i and r_i instead of just i as an index in order to identify user and rule transitions, respectively.

Boolean statements are made with propositional variables from the set Π as previously done in PIDL. We now additionally have a set X of variables to cover numerical decisions, which can be over reals and integers. However, further restrictions on the variables' ranges are made in the kind of specifications we eventually work with, as shown later in Definition 5.4. Like PIDL, the PIDL+ semantics has states to represent the states in the configuration process. The difference is that PIDL+ states also contain arithmetic atoms and a set of variables that is a subset of X. This set contains the variables that the user has decided on so far. The initial state (S_I, U_I) is part of the specification. The set C holds the constraints as usual, which must not be violated by

any state for the specification to be consistent. We again have two types of transitions, the user transitions T_U and rule transitions T_R. If a state satisfies the condition part $\Lambda \wedge F$ of a transition tuple, a transition to a new state is done by updating the old state with the update set E. We have a clear separation between an arithmetic part Λ and a propositional F part in the conditions. Notice the difference between T_U and T_R: In user transitions, the update sets are only allowed to have simple bounds $x \circ c$ as occurring atoms, whereas in rule transitions the update sets can have simple atoms $x \circ t$, which are more general. This correspondents to the situation we have in real-world configuration systems: Users can set individual values of variables, but rules can also assign more complex expressions to variables as their actions.

PIDL+ specifications as defined above describe configuration systems with arithmetic. However, to achieve the important property of decidability, it is necessary to restrict ourselves to a certain class of specifications. We define this class, which we call *admissible specifications*, in the following. For this we need to first identify those variables of the specification that occur in the transitions and can thus have an effect on the transition dynamics.

Definition 5.3. The *transition variables* X_T of a specification $\mathfrak{S}_+$ are the set

$$X_T := \{x | x \in \mathit{vars}(\Lambda \wedge F \rightsquigarrow E), \Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)\}.$$

Definition 5.4. A specification $\mathfrak{S}_+$ is *admissible* if the following holds.

(i) For each $x \in X_T$, $\mathit{sort}(x) = \mathcal{Z}$.

(ii) The constraints set C has the following form:

$$\mathsf{C} = \left\{ \bigwedge_{x_i \in X_T} x_i \geq c_i \wedge x_i \leq d_i \right\} \cup \left\{ \bigwedge_{x_j \in X'} x_j \approx t_j \right\} \cup \{\Phi\},$$

where

- $c_i, d_i \in \mathbb{Z}$, $c_i \leq d_i$, for all $x_i \in X_T$,
- $t_j \in T_\Sigma(X_T)$ for all $x_j \in X'$, where $X' \subseteq X \setminus X_T$, and
- Φ is a propositional formula over Π.

(iii) For each $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$, it holds that if $x \circ t \in E$, then $x \notin \mathit{vars}(t)$.

Admissible specifications require that variables appearing in transitions are over integers and bounded by integer constants. The purpose of this is to get finiteness and thus decidability of the system. We do not allow arithmetic expressions in the constraints other than the bounds except for equalities $x_j \approx t_j$, where x_j is a variable not occurring in the transitions and t_j is a term over the transition variables X_T. Each x_j has at most one such equation. These equalities represent calculations of output parameters in the configuration system that otherwise do not affect the configuration flow. In fact, the constants occurring in the terms t_j may be numbers that go beyond integers, without influencing the transitions. Propositional constraints are expressed in the formula Φ as done in the case of PIDL.

5.2 Semantics

The PIDL+ semantics is similar to that of PIDL in the sense that it is a possible-worlds semantics with models consisting of states and transitions, with the states being reachable from the initial state of the specification at hand. The inclusion of arithmetic makes certain changes necessary. One concerns the structure of states. On top of the propositional literals known in PIDL, we add simple atoms $x \circ t$ to express arithmetic decisions in the configuration system. We understand certain simple bounds $x \circ c$ as a compact representation of a number of actual assignments of the variables by the user. We need to know for each state which variables have been decided by the user. This is why each state in PIDL+ also has an associated set of variables, called the *user variables*. Because of the new structure of states, new definitions of transitions between states are required. Essentially, the question is what part of the user choices represented by the current state entails which transitions. This leads to the notion of *selections*.

We first motivate and then give the definition of states in PIDL+. While doing so, we lay emphasis on giving intuition rather than being formally strict before we present the exact definitions. After that we explain and define transitions between states in PIDL+ in the same manner, followed by the definition of interpretations and models of PIDL+ specifications.

5.2.1 States and Transitions

In PIDL, we use propositional literals to make statements about a single state. It can then be updated by the literals of an update set of a transition. However, the involvement of arithmetic variables makes some distinct changes to the previous framework necessary. We start by reviewing a typical situation in the purely Boolean context of PIDL, shown in Figure 5.1. In a configuration system, a user can assign two values to a Boolean variable `d`, which are true or false. As seen in Chapter 4, we model this by using a propositional variable for each of the two possible decisions, d_Yes and d_No. From the perspective of the original state, there are two possible successor states with either d_Yes or d_No in them. This way we cover the full range of possible user input with respect to the Boolean decision variable `d`.

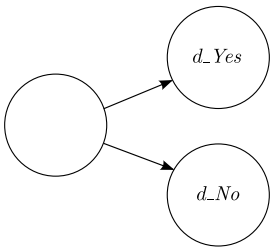

Figure 5.1: User decision of a Boolean variable.

What is the situation when dealing with arithmetic user decisions? For a given arithmetic variable, a user can, in principle, input any integer within the given bounds of the system. Figure 5.2 depicts the next example, in which x is an arithmetic variable whose value has been set by a user, that is, we talk about how the successor states of a user transition in PIDL+ should look like. Assume the user can choose a value for x that is between 0 and 100. One could represent all the possible user input with respect to x by assignment atoms $x \approx 0$, $x \approx 1$, $x \approx 2$, ..., $x \approx 100$, one for each successor state.

This semantics of the user decisions is what we would call a *small-steps semantics*, in which each instance $x \approx c$ of the user decisions manifests itself in a state.

Small-Steps Semantics

The small-steps semantics uses assignments $x \approx t$ instead of bounds $x \circ t$, where $\circ \in \{<, \leq, >, \geq\}$, to represent arithmetic decisions. We sketch the central notions concerning states and transitions, and show what the adjustments are.

States. In addition to propositional variables, states consist of simple atoms of the form $x \approx t$, where x is an arithmetic variable and t is a Σ-term.

Transitions. The criteria for applying a transition $\Lambda \wedge S \rightsquigarrow E$ to a state S are as usual: The state has to be consistent, that is,

$$S \cup \mathsf{C} \not\models \bot,$$

and the state has to entail the transition's condition, that is,

$$S \cup \mathsf{C} \models \Lambda \wedge F.$$

The additional requirement of rule termination for user transitions is defined analogously to the PIDL case, as shown below.

Updates. Assuming a state S and an update set E, the update operator is defined as

$$S' := S \triangleleft E,$$

with

$$\begin{aligned} S' := \quad & \{L | L \text{ literal over } \Pi, L \in S, \overline{L} \notin E\} \cup \\ & \{L | L \text{ literal over } \Pi, L \in E\} \cup \\ & \{x \approx t | x \approx t \in S, x \notin \mathit{varsl}(E)\} \cup \\ & \{x \approx t | x \approx t \in E\}. \end{aligned}$$

This means that the update follows the same spirit of Definition 4.4: Propositional literals are updated exactly as in the PIDL case, and simple atoms from E replace atoms in S with the same left-hand-side variable x, while atoms from E with fresh left-hand-side variables are added to the resulting set.

Rule termination. Again, this follows Definition 4.9 of PIDL. A state S is rule terminal if there is no rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i$ such that $S \rightarrow_i S'$ with $S' \neq S$.

The small-steps semantics is a straightforward continuation of the PIDL semantics. However, the explicit enumeration of user decisions with respect to the arithmetic variables is not very practical with regard to decision procedures in particular. This is why we use the *big-steps semantics* for PIDL+, which offers a more compact representation. In it, we make use of simple bounds $x \circ c$ rather than explicit assignment statements. In the above example, user input with respect to a variable x is expressed with assignments $x \approx 0$, $x \approx 1$, $x \approx 2$, ..., $x \approx 100$. Instead of dealing with 101 successor states, we define bounds like $x \geq 0, x < 40$ and $x \geq 40, x \leq 100$ as the results of the user decision, yielding

two successor states (Figure 5.2). We would have two corresponding user transitions, one having an update set containing $x \geq 0, x < 40$ and one having an update set containing $x \geq 40, x \leq 100$. This informal view is made precise later in Definition 5.28. These two

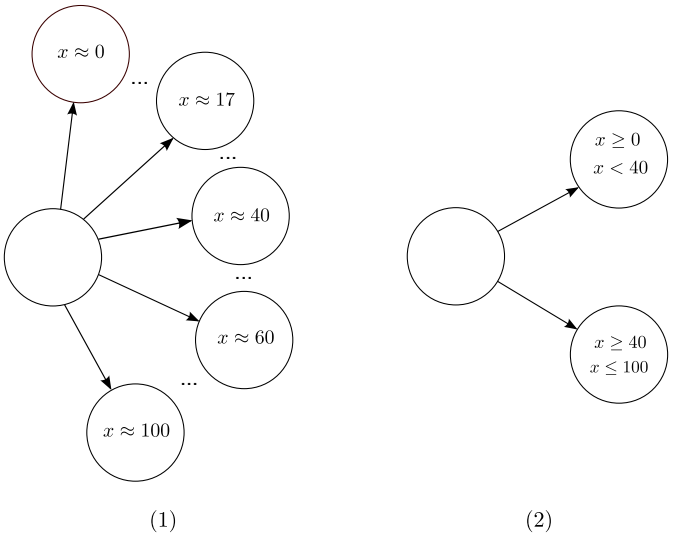

Figure 5.2: Modeling arithmetic user decisions. (1) Explicit representation as assignments. (2) Symbolic representation as bounds.

states then symbolically represent all the instances of x that would occur if assignment statements were employed. They express the range of all possible decisions that a user could make in that situation. Figure 5.3 illustrates the difference in representing states in the two kinds of semantics.

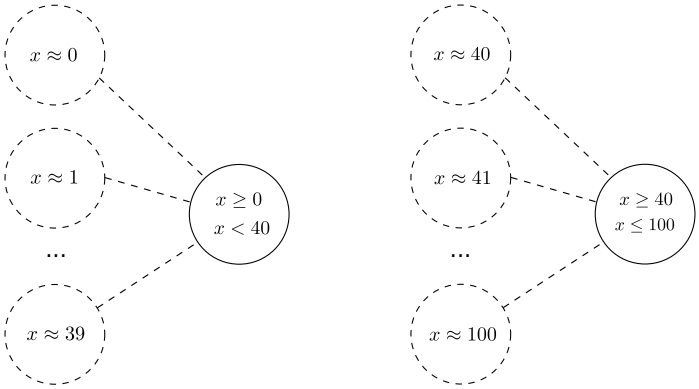

Figure 5.3: Representation of decisions in the small-steps semantics (dashed) and in the big-steps semantics (solid).

As before, the transition effects are written in the update set of a transition tuple. Translating the above example into our framework, the update set E of one user transition would contain the atoms $x \geq 0, x < 40$ and the update set E' of another user transition would contain the atoms $x \geq 40, x \leq 100$. On the other hand, the inclusion of assignment atoms $x \approx c$ in the update set of a user transition is also possible, as this is subsumed by the more general case $x \circ c$. The resulting updated state would then represent just one state in the configuration, namely the one in which $x = c$.

In PIDL+, the additional information on which variables occurring in a state have been set by a user and which ones have been set by a rule is of central importance, since the transition criteria (Definition 5.22 and 5.28) depend on such a distinction: First, for all assignments of the user variables there exists an assignment of the rest of the variables such that the state and the constraints are consistent. Second, there is an assignment of the user variables such that for all assignments of the rest of the variables the condition of some transition is entailed by the current state and the constraints, as stated in the definition of selections (Definition 5.8).

A state is thus not a single set of literals any more but a tuple (S, U) with the first component S being a set of propositional literals and simple atoms, and the second component being a subset U of the variable set X, indicating the set of variables in S that have been decided by the user.

Definition 5.5. A *PIDL+ state* is a pair (S, U) consisting of

- a finite set S of simple atoms $x \circ t$ over Σ and propositional literals over Π, and
- a set $U \subseteq X$ of *user variables*.

We also use the term "state" to just refer to the set S of a PIDL+ state (S, U). States are the result of transition applications using the updates of the user transitions T_U and rule transitions T_R. By definition, atoms occurring in the update sets of T_U are simple bounds $x \circ c$, as are the atoms contained in the initial state (S_I, U_I). As a consequence, the subset $S|_U$ is always a set of simple bounds over Σ.

We use the following convention. We write $\vec{x}_u$ to refer to the variables of U and $\vec{y}$ to refer to the variables in $X \setminus U$. The set of user variables U always refers to the current state being considered.

The above examples already hint at how we imagine transitions and updates in PIDL+. The next subsections give a more detailed view on these elements of the semantics.

We now discuss transition in PIDL+. Before a transition can be applied to a state in PIDL, the state has to meet certain criteria. First, the state has to be consistent. Second, the state must entail the condition of the transition. If these points are satisfied, a transition to a new state can be done by updating the old state with the update set of the transition. This principle is used in the new logic as well. However, the presence of arithmetic atoms in conjunction with the nature of the difference between user and rule decisions requires different transition criteria in PIDL+. We look at PIDL's criteria of state consistency and transition entailment again, and develop the respective versions for PIDL+.

State consistency. In PIDL, we demand that no transition is applied to a state S if S and the constraints C together are inconsistent. This is written as the criterion

$$S \cup \mathsf{C} \not\models \bot.$$

We cannot use the same criterion in the new logic, which is illustrated by Figure 5.4.

In this example, we assume a state (S, U) with the atom $x > 0$ that comes from a user decision. We have a constraint set consisting of the only formula $x > 5$. $S \cup \mathsf{C} \not\models \bot$ clearly holds in this case. However, the states represented by S include those in which x is assigned numbers less than 5, and these states are inconsistent with respect to the constraint $x > 5$. Put more generally, we want to deem a state inconsistent when the bounds given by user decisions represent instances s such that $s \cup \mathsf{C} \models \bot$: The user

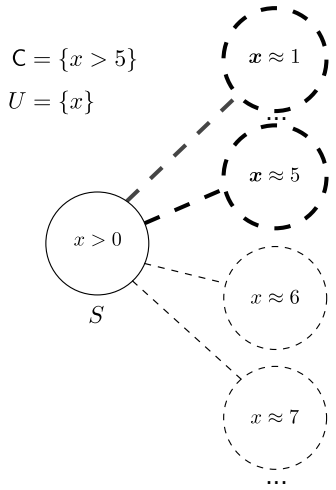

Figure 5.4: The state S and a selection of states represented by it. Among them are states violating the constraint $x > 5$ (bold).

can take a decision that conflicts with the constraints. In the following, we list some apparently possible formulations of consistency criteria together with counterexamples illustrating why we do not use them. Recall that C does not contain any quantifiers.

(i) $\forall \vec{x}_u \exists \vec{y}\, (S \cup \mathsf{C})$ *is satisfiable.*

Satisfiability of $S \cup \mathsf{C}$ alone proves to be not enough, because for us, bounds in S set by a user can represent inconsistent states while being logically consistent with the rest of S and C, as seen in the previous example. To identify these inconsistent instances we could add the above requirement using universal quantification of the user variables to make sure *all* user input is considered. The other variables are existentially quantified because we follow the intuitive idea of consistency: In each instance of the user variables, S and C are satisfiable.

Counterexample: $S = \{x \geq 0\}$, $\mathsf{C} = \emptyset$, $U = \{x\}$. Then, $\forall \vec{x}_u \exists \vec{y}\, (S \cup \mathsf{C})$ is obviously not satisfiable, since there are assignments that do not satisfy the set of formulas $S \cup \mathsf{C}$. One example is $\beta(x) = -1$, which gives us

$$I_\Pi(\beta)(x \geq 0) = -1 \geq 0 = 0.$$

However, we want to see (S, U) as perfectly consistent, since there are no constraints in C that could be violated, and S itself is a consistent set.

(ii) $S \models \mathsf{C}$.

(i) involves all possible instances over the user variables. As it turns out, this range is too broad since it can also contain valuations of the user variables that violate bounds in the state literal set S. This results in a contradiction to our intuitive notion of consistency. We therefore try restricting the range of user instances to those that satisfy S.

Counterexample: $S = \{A\}$, $\mathsf{C} = \{B\}$. This state does not produce any logical conflicts, yet it fails to meet the criterion as clearly $\{A\} \not\models \{B\}$. That is, the criterion does not work properly with respect to the purely propositional part of the logic.

(iii) $\forall \vec{x}_u \exists \vec{y}\, (S \to \mathsf{C})$ *is satisfiable.*

This statement removes the problem encountered with (ii). It does not require the propositional literals of the state to imply the propositional formulas of the constraints, while the requirement for arithmetic variables is taken account of by the quantifiers: Following the same rationale of (i), for all instances of the user input there must be a consistent world.

Counterexample: $S = \{y > 0\}$, $\mathsf{C} = \{y < 0\}$, $U = \emptyset$. We get the instance $\forall \vec{x}_u \exists y\ (y > 0 \to y < 0)$ which is satisfiable but we obviously do not want this state to be called consistent. Another flaw in this criterion becomes apparent if there are no arithmetic expressions in S and C, and thus no arithmetic variables. The criterion then reduces to the satisfiability of $S \to \mathsf{C}$, which is trivially the case when assuming the satisfiability of C.

The problem of (iii) is that the atoms with variables not decided by the user are contained in the left-hand side of the implication $S \to \mathsf{C}$. There can be cases where one can always find assignments for variables of $\vec{y}$ that make the implication and thus the whole statement true due to the existential quantification. From this observation we deduce that we need a further refinement. The final step is to move the atoms $y \circ t$ with $y \notin U$ to the right-hand side of the implication. The criterion we want to use now is therefore

$$\forall \vec{x}_u \exists \vec{y}\, S|_U \to (S \setminus S|_U) \cup \mathsf{C} \text{ is satisfiable.} \tag{5.1}$$

This statement corresponds to the semantics that we want: Consistency of a state is considered as the usual satisfiability of the propositional formulas and the atoms not set by the user with respect to all the *reasonable* user input within the bounds given in $S|_U$.

Transition entailment. In PIDL, a transition can only be applied to a consistent state S if the condition χ of a transition tuple $\chi \rightsquigarrow E$ is entailed by S and the constraints C: The criterion is

$$S \cup \mathsf{C} \models \chi.$$

The transitions we consider now are of the form $\Lambda \wedge F \rightsquigarrow E$ as specified in Definition 5.1. If we translate the PIDL case in a naive way, the criterion could look like

$$S \cup \mathsf{C} \models \Lambda \wedge F.$$

Unfortunately, this does not fit the intended meanings behind the new interval representation by arithmetic atoms. The entailment $\models$ demands essentially that all instances of the user variables satisfying the bounds in S, together with the constraints C, imply the transition condition $\Lambda \wedge F$. For example, consider $S = \{x < 10\}$, $U = \{x\}$, $\mathsf{C} = \emptyset$, and $\Lambda \wedge F = x < 5$. It clearly holds that $S \cup \mathsf{C} \not\models \Lambda \wedge F$ because there are user instances that satisfy the bound $x < 10$ of the state but not $x < 5$ of the condition, for example with and $\beta(x) = 8$. However, $x < 10$ also includes a range of possible variable assignments that actually imply the condition, namely those in which x is assigned a value less than 5, as shown in Figure 5.5. According to our intuition, this should indeed support a transition application.

We see that it makes sense to not rule a transition out when there are suitable variable assignments represented by the bounds in the state, resulting from user decisions. An improvement would be to use quantifiers and say that the transition can be done if there exists user input that satisfies the transition condition, as expressed in the following statement:

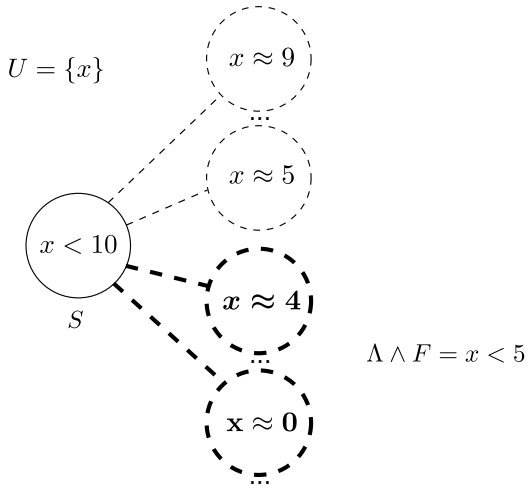

Figure 5.5: Instances of S in which $x < 5$ (bold) satisfy the transition condition.

$$\exists \vec{x}_u \forall \vec{y}\, (S \cup \mathsf{C}) \rightarrow (\Lambda \wedge F) \textit{ is valid.}$$

However, there is a counterexample to this. Take $S = \{x > 0\}$, $U = \{x\}$, $\mathsf{C} = \emptyset$ and $\Lambda \wedge F = y > 7$. Then, the above statement is true with an assignment of the form $\beta(x) = -1$, but according to our understanding, the state with S containing only $x > 0$ should obviously not entail a transition with the condition $\Lambda \wedge F$ containing only $y > 7$.

It follows from the example that we have to limit the range of user instances that are considered to a meaningful subset. We do this in a fashion similar to the state consistency criterion. The only instances of interest are those which satisfy the state atoms. We get the criterion

$$\exists \vec{x}_u \forall \vec{y}\, S|_U \wedge (S \setminus S|_U \cup \mathsf{C} \rightarrow \Lambda \wedge F) \text{ is valid.} \tag{5.2}$$

If there is an instance from all the arithmetic user decisions represented by the bounds of S that makes the condition of a transition be implied by the arithmetic decisions taken by the system and propositional decisions, then the transition can be applied to the state.

Criterion 5.2 is sufficient to determine when a transition is applicable to a state. However, we need one more refinement step due to another peculiarity of PIDL+. This peculiarity consists in the way state updates are done. Basically, it is important that the representation of updated states considers under what user instances transitions happen.

First, we look at an example where we assume an update operator $\triangleleft_P$ for PIDL+ that is "naively" derived from PIDL: The update replaces every occurrence of atoms $x \circ t$ in S with atoms $x \circ t'$ of E that have the same left-hand-side variable, and adds atoms $x' \circ t'$ of E whose left-hand-side variables are not contained in S. Propositional literals are updated in the usual way.

Example 5.6. Let

$$(S, U) = (\{x > 3, y < 92, C\}, \{x\})$$

and

$$E = \{y \approx 32, z \geq 10, \neg D\}.$$

Then,

$$(S', U') = (S, U) \triangleleft_P E$$

with

$$(S', U') = (\{x > 3, y \approx 32, z \geq 10, C, \neg D\}, \{x\}).$$

The update set E contains $y \approx 32$ which replaces $y < 92$ of S. The variable z does not occur in S, so the atom $z \geq 10$ is included in the new set S'. The variable x is not mentioned in E and $x > 3$ thus stays in the set after the update.

The deficiencies of this adopted update method become apparent in the next example, where we have a sequence of transitions.

Example 5.7. Consider the following specification.

$$\begin{aligned} S &= \{x \geq 40, x \leq 90\} \\ U &= \{x\} \\ \mathsf{C} &= \{x \geq 0, x \leq 100\} \\ T_R &= \{x \leq 50 \rightsquigarrow_{r_1} \{B\}, \\ &\qquad x > 70 \rightsquigarrow_{r_2} \{C\}\} \end{aligned}$$

The state (S, U) fulfills the transition condition of r_1 according to Criterion 5.2, that is,

$$\exists x \, \{x \geq 40, x \leq 90\} \wedge (\{x \geq 0, x \leq 100\} \rightarrow x \leq 50) \text{ is valid.}$$

We update to state (S', U'):

$$\begin{aligned} (S', U') &= (S, U) \triangleleft_P E_{r_1} \\ &= (x \geq 40, x \leq 90, \{x\}) \triangleleft_P \{B\} \\ &= (\{x \geq 40, x \leq 90, B\}, \{x\}). \end{aligned}$$

From this new state, it is perfectly fine to see that another rule transition can be done with r_2, that is,

$$\exists x \, \{x \geq 40, x \leq 90\} \wedge (\{B\} \cup \{x \geq 0, x \leq 100\} \rightarrow x > 70) \text{ is valid.}$$

We get the state

$$\begin{aligned} (S'', U'') &= (S', U') \triangleleft_P E_{r_2} \\ &= (x \geq 40, x \leq 90, B, \{x\}) \triangleleft_P \{C\} \\ &= (\{x \geq 40, x \leq 90, B, C\}, \{x\}). \end{aligned}$$

Recall that (S, U) represents all instances of x in which x has a value between 40 and 90. Transition r_1 is possible because among these instances there are those that fulfill Criterion 5.2, namely all instances of x between 40 and 50. They satisfy the condition $x \leq 50$ of r_1 (Figure 5.6). This information is lost in the new state (S', U'), which is the result of the naive update operation. From the perspective of (S', U'), it represents all the user instances of x between 40 and 90, just like the original state (S, U). Considering the meaning of states discussed at the beginning of Section 5.2.1, it would make much more sense if (S', U') represented the instances of x between 40 and 50 because (S', U') is the result of a transition that is based on these instances. This would also make the application of transition r_2 to (S', U') impossible, since r_2 requires user instances in which $x > 70$. Thus, the transition from (S', U') to (S'', U'') using r_2 is actually unwanted in our intended context.

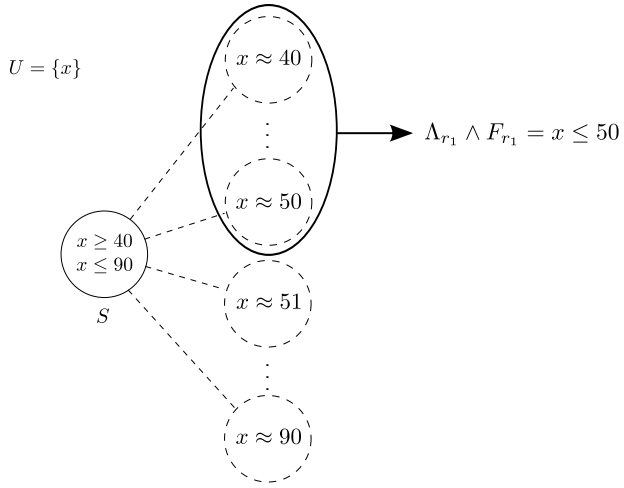

Figure 5.6: In Example 5.7, the state (S, U) entails transition r_1 for instances where $x \geq 40$ and $x \leq 50$.

States representing the "wrong" user instances can lead to unwanted transitions that are at odds with the semantics of the configuration systems we want to model. To solve this issue, we have to incorporate information about user instances into the update process so that a new state represents the correct instances with respect to transitions. We do this by partitioning the user instances of a state into appropriate *selections*. A selection consists of integer intervals for each of the user variables. These intervals contain the values of the variables under which a transition can be applied. The selections are taken into account by the updates, making sure that user decisions are not "forgotten" during several transitions but accumulated in the simple bounds representing the selections. Thus, each state holds information about which user instances of the previous state are responsible for the transition that leads to it. We first give a formal definition.

Definition 5.8. 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:

(i) 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

(ii) if $U = \emptyset$, then γ is the empty tuple $()$, and $\forall \vec{y}\ S \cup \mathsf{C} \rightarrow \Lambda \wedge F$ is valid.

Given a state and a transition tuple, a selection basically describes the subset of user instances of the state that entail the transition. Its intervals contain the values that, by substituting them for the user variables, satisfy Criterion 5.2. The above definition also explicitly considers the case when a state has no user variables, that is, $U = \emptyset$. The corresponding selection is then always the empty tuple $()$ and the criterion, since $S|_U$ becomes empty, reduces to the validity of $\forall \vec{y}\ S \cup \mathsf{C} \rightarrow \Lambda \wedge F$.

Example 5.9. Looking at Example 5.7, selections with respect to the state (S, U) and the transition r_1 include ([40, 50]), ([40, 44]), ([43, 43]), ([48, 50]), ... In general, all tuples with intervals that are non-empty subsets of the interval [40, 50] are selections according to Definition 5.8.

The next remark follows directly from Definition 5.8 because a selection is a collection of user instances under which the corresponding transition is entailed.

Remark 5.10. Let (S, U) be a state and $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$ be a transition. A selection γ with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$ always exists if

$$\exists \vec{x}_u \forall \vec{y}\, S|_U \wedge (S \setminus S|_U \cup \mathsf{C} \rightarrow \Lambda \wedge F) \text{ is valid.}$$

Selections give us the ability to keep track of and express the correct user instances of states in a sequence of transitions. The idea is to make selections an essential part of PIDL+ transition applications. A transition from one state to another should always be with respect to a selection, and that selection is carried over in the next state. Note that each tuple of intervals having the properties of Definition 5.8 is a valid selection corresponding to a state and a transition. Therefore, there can be more than one possible selection with respect to a state and a transition, as seen in Example 5.9. For our semantics, we restrict the range of possible selections to be considered in transitions to those which are "maximal" in some sense. We express this property with the help of the *subselection* relationship between selections.

Definition 5.11. Let γ and γ' be two selections of the same length n. 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 at least one $i \in \{1, \ldots, n\}$.

Example 5.12. Let

- $\gamma_1 = ([32, 89])$,
- $\gamma_2 = ([0, 100])$,
- $\gamma_3 = ([23, 50], [3, 50], [11, 43])$,
- $\gamma_4 = ([20, 60], [3, 50], [5, 63])$ and
- $\gamma_5 = ([17, 99], [37, 62], [0, 80])$

be selections. Then $\gamma_1 \subset \gamma_2$, $\gamma_3 \subset \gamma_4$, but $\gamma_3 \not\subset \gamma_5$ and $\gamma_4 \not\subset \gamma_5$.

Selections are *maximal* if they are not proper subselections of other selections with respect to the same states and transitions. Later, we use maximal selections to define transition between states in PIDL+.

Definition 5.13. A selection γ with respect to a state (S, U) and a transition $\Lambda \wedge F \rightsquigarrow E$ is *maximal* if either $\gamma = ()$ or there is no selection γ' with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$ such that $\gamma \subset \gamma'$.

Example 5.14. We look at two example cases of maximal selections.

- In Example 5.9, the selection $\gamma = ([40, 50])$ with respect to (S, U) and r_1 is maximal.
- Assume the following state and constraints:

$$
\begin{aligned}
S = \{&x_1 \geq -100, x_1 \leq 100,\\
&x_2 > 5, x_2 \leq 98,\\
&y_1 = x_1^2 + 2\},\\
U = \{&x_1, x_2\},\\
\mathsf{C} = \{&x_1 \geq -100, x_1 \leq 100,\\
&x_2 \geq 0, x_2 \leq 100,\\
&y_1 \geq 0, y_1 \leq 100\}.
\end{aligned}
$$

Also, assume a rule transition $y_1 > 8 \rightsquigarrow_r \{A\}$. Note that the presence of the atom $y_1 = x_1^2 + 2$ is consistent with the definition of admissible specifications (Definition 5.4), since it is the state that the atom is contained in.

The set U tells us that the atoms $x_1 \geq -100, x_1 \leq 100$ and $x_2 > 5, x_2 \leq 98$ can be considered as having been set by user transitions, while the atom $y_1 = x_1^2 + 2$ is the product of a rule transition. Corresponding selections with respect to state (S, U) and transition r contain user instances of the variables in U such that

$$\forall y_1 \; S|_U \wedge (S \setminus S|_U \rightarrow y_1 > 8)$$

is valid under those instances. There are two maximal selections that have this property:

$$\gamma_1 = (\underbrace{[3, 100]}_{x_1}, \underbrace{[6, 98]}_{x_2})$$

and

$$\gamma_2 = (\underbrace{[-100, -3]}_{x_1}, \underbrace{[6, 98]}_{x_2}).$$

In other words, the state entails the transition if either the user variable x_1 is assigned a value between 3 and 100 and the user variable x_2 is assigned a value between 6 and 98, or x_1 is assigned a value between -100 and -3 and x_2 is assigned a value between 6 and 98.

If there is a transition from a state S to a state S' with respect to a selection, we have to express the selection within the state S', otherwise this important piece of information is lost. To do this, we define the *atomic representation* $at(\gamma)$ of a selection γ with respect to the original state and the transition. It is the union of the atomic representations $at(\gamma(i), x_i)$, as defined in Definition 3.7 of Chapter 3, of the intervals in γ. The simple bounds in $at(\gamma)$ are then contained in the new state S'.

Definition 5.15. 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:

(i) If $U = \{x_1, \ldots, x_n\}$, then

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

and

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

Example 5.16. We look at the atomic representations of the previous examples. The atomic representation of the selection γ mentioned in Example 5.14 is

$$at(\gamma) = at([40, 50], x) = \{x \geq 40, x \leq 50\}.$$

The atomic representation of γ_1 is

$$\begin{aligned} at(\gamma_1) &= at(([3, 100], [6, 98])) \\ &= at([3, 100], x_1) \cup at([6, 98], x_2) \\ &= \{x_1 \geq 3, x_1 \leq 100, x_2 \geq 6, x_2 \leq 98\}, \end{aligned}$$

and in the case of γ_2 we get

$$\begin{aligned} at(\gamma_2) &= at(([-100, -3], [6, 98])) \\ &= at([-100, -3], x_1) \cup at([6, 98], x_2) \\ &= \{x_1 \geq -100, x_1 \leq -3, x_2 \geq 6, x_2 \leq 98\}. \end{aligned}$$

It is also useful to consider *base selections* of a state (S, U), which represent exactly the instances expressed by the user bounds in the state. Those selections are not associated with any transitions. Before we define them, we need a syntactical tool:

Definition 5.17. Let N and N' be sets of simple bounds. Then $N \cong N'$ if and only if N has the exact same bounds as N' modulo strict inequalities.

Definition 5.17 means that two sets containing simple bounds are actually the same if they can be made equal by turning strict inequalities occurring in the bounds to weak inequalities.

Example 5.18. It holds that

$$\{x > 3, y \leq 5\} \cong \{x \geq 4, y \leq 5\}$$

and

$$\{x > 9, x < 21\} \cong \{x \geq 10, x \leq 20\}$$

Definition 5.19. Let (S, U) be a state with $U = \{x_1, \ldots, x_n\}$. The *base selection* γ_S of (S, U) is a tuple of integer intervals of length n such that $at(\gamma_S(i), x_i) \cong S|_{x_i}$ for all $i = 1, \ldots, n$ If $U = \emptyset$, then $\gamma_S = ()$.

Example 5.20. Let

$$(S_1, U_1) = (\{x_1 \geq 23, x_1 \leq 110, y_1 > 0, C\}, \{x_1\})$$

and

$$(S_2, U_2) = (\{x_1 > 1, x_1 < 90, x_2 \geq 40, x_2 < 78, y_1 > 50, y_1 \leq 200\}, \{x_1, x_2\})$$

be states. Then the base selections γ_{S_1} of (S_1, U_1) and γ_{S_2} of (S_2, U_2) are

$$\gamma_{S_1} = ([23, 110])$$

and

$$\gamma_{S_2} = ([2, 89], [40, 77]).$$

Atomic representations enable us to integrate selections in PIDL+ updates, which we are now ready to define. Unlike PIDL, where we only have one type of update, we define two update operators in PIDL+, *rule updates* and *user updates*. The reason for this is the presence of user variables, which play a crucial role for PIDL+ states (Definition 5.5) and selections (Definition 5.8). In PIDL+, we allow a rule transition to replace atoms in the old state with atoms occurring in its update set E, analogously to the update of propositional literals in PIDL. We understand this as an overwriting of user decisions, hence the state's set U of user variables has to be updated accordingly too. In the case of rule transitions, U can be decreased, while in the case of user transitions, U can be increased, corresponding to the user taking additional decisions with respect to additional variables. We first define rule updates in PIDL+. User updates are defined later in this section in Definition 5.27.

Definition 5.21. The *rule update operator* $\lhd_R$ takes a state (S, U), and a pair (E, γ) as arguments, where

(i) E is an update set, and

(ii) 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 $(S, U) \lhd_R (E, \gamma) := (S', U')$, where

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

The rule update operator is a relatively natural extension of the PIDL update operator. The purely propositional part of a state is updated exactly as before. In terms of arithmetic atoms, we have three different cases. First, an atom $x \circ t$ where x is not a user variable is preserved in the updated state if x does not occur as a left-hand-side variable of an atom in the update set E. Second, an atom $x \circ t$ of the atomic representation $\mathit{at}(\gamma)$ of the selection γ, where x is a user variable, is contained in the updated state if x is not a left-hand-side variable of an atom in E. Third, atoms $x \circ t$ occurring in E are included in the updated state, effectively replacing all atoms $x \circ t'$ that have the same variable x on the left sides in the old state S. With $\lhd_R$ being embedded in the definition of rule transitions below, we in particular preserve information about the selection that makes the transition possible by including the atoms corresponding to the selection in the new state. However, these are not included if they are overwritten by E as described above. In that case, the update set takes higher precedence. This also means that in the new state, the set of user variables U' can be smaller than the previous U because user decisions are overwritten by rule decisions, hence $U' = U \setminus \mathit{varsl}(E)$. It is necessary to update the set of user variables in this way since transition applications depend on instances of a state's user variables, so we must know for the updated state which user variables to consider. The rule update operator is the last component needed to define rule transitions in PIDL+.

Definition 5.22. A *rule transition* from a state (S, U) to a state (S', U') with respect to a rule transition tuple $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$ and a selection γ is written as

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

where

(i) $\forall \vec{x}_u \exists \vec{y}\, S|_U \to (S \setminus S|_U) \cup \mathsf{C}$ is satisfiable,

(ii) γ is a maximal selection with respect to (S, U) and $\Lambda_i \wedge F_i \rightsquigarrow E_i$, and

(iii) $(S', U') = (S, U) \lhd_R (E_i, \gamma)$.

The definition is structured in the same way as the one of rule transitions in PIDL: state consistency, transition entailment and update. A rule transition can be applied to a state only if the state is consistent according to (5.1), and there is a maximal selection with respect to the state and the transition. The new state is then the old state updated by $\lhd_R$.

Given a state (S, U), rule updates can change the set of user variables by reducing it. Analogously, we expect updates from user transitions, expressed by a *user update operator* $\lhd_U$, to be able to change the set U of user variables by increasing it. This is because a user can set new variables, introducing them to the current U. Otherwise $\lhd_U$ should update the set S of literals and atoms just like $\lhd_R$.

Before we can give the definitions of the user update operator $\lhd_U$ and user transitions, we need to address one issue that is of central importance: rule termination. The concept of rule-terminal states known from PIDL again plays a major role in PIDL+. We first give an example highlighting what the situation is like in PIDL+ and how we derive a suitable notion of rule termination for it. For the purposes of the example, we assume a user update operator $\lhd_U$ which just behaves like the rule update operator $\lhd_R$ without taking rule termination into account.

Example 5.23. Consider the following specification.

$$
\begin{aligned}
S &= \{x \geq 0, x \leq 100\} \\
U &= \{x\} \\
\mathsf{C} &= \{x \geq 0, x \leq 100\} \\
T_R &= \{x < 30 \rightsquigarrow_{r_1} \{A\}, \\
&\qquad x \leq 20 \wedge B \rightsquigarrow_{r_2} \{C\}\} \\
T_U &= \{x \leq 50 \rightsquigarrow_{u_1} \{B\}\}
\end{aligned}
$$

We do not care about rule termination and assume that user transitions work in the same way as rule transitions. We see that user transition u_1 can be applied to (S, U) since there is a maximal selection $\gamma_1 = ([0, 50])$ with respect to the state (S, U) and user transition u_1. Thus,

$$(S, U) \to_{(u_1, \gamma_1)} (S', U')$$

with

$$
\begin{aligned}
(S', U') &= (S, U) \lhd_U (E_{u_1}, \gamma_1) \\
&= (\{x \geq 0, x \leq 100\}, \{x\}) \lhd_U (\{B\}, ([0, 50])) \\
&= (\{x \geq 0, x \leq 50, B\}, \{x\}).
\end{aligned}
$$

The new state (S', U') warrants a rule transition using r_2 with the corresponding selection $\gamma_2 = ([0, 20])$. We get

$$(S', U') \to_{(r_2, \gamma_2)} (S'', U'')$$

with

$$\begin{aligned}(S'', U'') &= (S', U') \lhd_R (E_{r_2}, \gamma_2) \\ &= (\{x \geq 0, x \leq 50, B\}, \{x\}) \lhd_R (\{C\}, ([0, 20])) \\ &= (\{x \geq 0, x \leq 20, B, C\}, \{x\}).\end{aligned}$$

The last transition reveals the problematic nature of the semantics used in the example. According to our understanding so far, it is the user instances of x between 0 and 20 for the state (S', U') that make the application of transition r_2 possible. This conflicts with the user instances involved in the transition $(S, U) \rightarrow_{(u_1, \gamma_1)} (S', U')$. Since the instances in which $x \geq 0$ and $x < 30$ can indeed be changed by transition r_1, it is the instances $x \geq 30$ and $x \leq 50$ which are rule terminal in the small-steps semantics. This means (S', U') should actually represent those instances rather than the ones expressed by γ_1. Then, transition $(S', U') \rightarrow_{(u_1, \gamma_2)} (S'', U'')$ should not be possible. It is a situation similar to Example 5.7, where we have an unwanted transition contradicting the intended semantics. Figure 5.7 illustrates the situation.

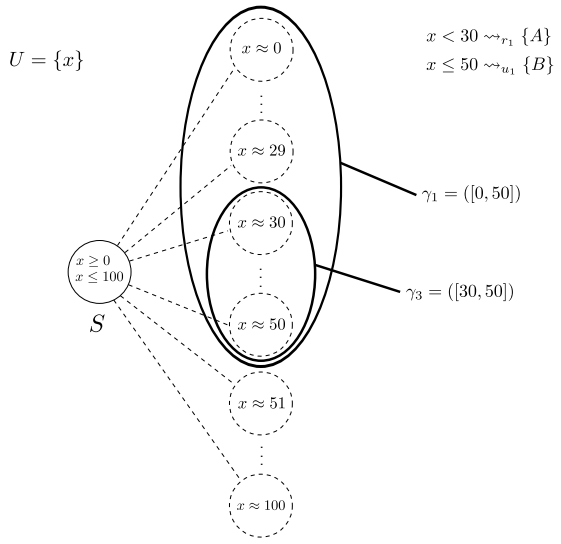

Figure 5.7: Given user transition u and rule transition r_1, the instances of (S, U) represented by selection γ_1 entail user transition u_1, but only the instances represented by subselection γ_3 are rule terminal.

Example 5.23 shows that the application of user transitions in PIDL+ again has to rely on rule termination. We now transfer the notion of rule-terminal states known from PIDL to PIDL+. In PIDL, user transitions are allowed to be applied only to states that are rule terminal, that is, there are no rule transitions whose updates would change those states. The same principle applies here. However, this time selections have to be taken into account to determine whether a state is rule terminal. We approach the new situation by again considering the fact that states (S, U) in PIDL+ are representations of instances with respect to the arithmetic atoms in $S|_U$. Each of those instances is a state in the small-steps semantics as depicted at the beginning of Section 5.2.1. If we look at those states, we observe that we can declare such states as rule terminal if they are rule terminal in the sense of PIDL. This is reasonable because each assignment statement $x \approx c$ in the small-steps semantics can be seen as a propositional statement. The application of PIDL's concept of rule-terminal states is thus natural. Given a state

(S,U), a user transition u and a maximal selection γ with respect to (S,U) and u, some of the instances represented by γ can be rule terminal and some can be not. We are then interested in those which are rule terminal in the PIDL fashion. Those instances form a subselection γ' of γ under which (S,U) is rule terminal in PIDL+. As a result, the update of the user transition u should not be based on the selection γ but on γ'. Before we define *rule-terminal subselections* in general, it is useful to introduce a term that describes selections that lead to rule transitions changing a state.

Definition 5.24. Let (S,U) be a state. A selection γ is a *change-admitting* selection with respect to (S,U) if

(i) γ is a maximal selection with respect to (S,U) and a rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$, and

(ii) $(S,U) \neq (S,U) \lhd_R (E_i,\gamma)$.

Definition 5.25. Let γ be a maximal selection with respect to a state (S,U) and a user transition $\Lambda \wedge F \rightsquigarrow E \in T_U$.

(i) If $\gamma \neq ()$, then we call a selection γ' with $\gamma' \subseteq \gamma$ a *rule-terminal subselection of* γ, or a *rule-terminal selection* with respect to (S,U) and $\Lambda \wedge F \rightsquigarrow E$, if the following holds:

   (a) There is no change-admitting selection γ^* with respect to (S,U) such that $\gamma^* \cap \gamma' \neq \emptyset$, and

   (b) there is no other selection $\gamma^\#$ that fulfills (a) and $\gamma' \subset \gamma^\#$.

(ii) If $\gamma = ()$, then we call γ a *rule-terminal selection* with respect to (S,U) and $\Lambda \wedge F \rightsquigarrow E$, if $()$ is not a change-admitting selection with respect to (S,U).

In the respective cases, we also say that (S,U) is *rule terminal* with respect to γ' or $()$ and to $\Lambda \wedge F \rightsquigarrow E$.

According to the definition, we first look at a maximal selection γ with respect to a state and a user transition. As we know already, the selection represents the user instances of the state that entail the user transition. If there is a subselection $\gamma' \subseteq \gamma$ such that γ' is disjoint to any other change-admitting selection with respect to the state, it means that γ' represents only instances that are rule terminal in the small-steps semantics. This is a reasonable requirement for calling a selection rule terminal in PIDL+. It also needs to be maximal, that is, the selection is not a proper subselection of another one having the same property. If the selection is $()$, then the set of user variables is empty by definition and the notion of rule termination carries over analogously from PIDL.

From the base selection of a state, we can derive a subselection thereof that can be considered as rule terminal in general, that is, its rule termination is not restricted to a certain user transition.

Definition 5.26. Let (S,U) be a state and γ_S its base selection. A selection $\gamma \subseteq \gamma_S$ is the *rule-terminal base* of (S,U) if

(a) there is no change-admitting selection γ^* with respect to (S,U) such that $\gamma \cap \gamma^* \neq \emptyset$, and

(b) there is no other selection $\gamma^\#$ with $\gamma \subset \gamma^\#$ and that fulfills (a).

The user update operator $\triangleleft_U$ is defined almost like the rule update operator $\triangleleft_R$ of Definition 5.21 except that, as mentioned earlier, the set of user variables of a state can be increased after an update, corresponding to the user taking additional decisions with respect to variables not in the user variables U of the current state. The updates of the propositional literals and arithmetic atoms stay the same.

Definition 5.27. The *user update operator* $\triangleleft_U$ takes a state (S, U), and a pair (E, γ) such that

(i) E is an update set,

(ii) 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 $(S, U) \triangleleft_U (E, \gamma) := (S', U')$, where $U' := U \cup \mathit{varsl}(E)$ and S' is defined in the exact same way as in the case for the rule update operator $\triangleleft_R$.

User transitions differ from rule transitions as follows: To apply a user transition to a state, the state must be rule terminal with respect to an appropriate selection according to Definition 5.25 and the user transition, and the state is updated using the user update operator.

Definition 5.28. 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

(i) $\forall \vec{x}_u \exists \vec{y}\, S|_U \rightarrow (S \setminus S|_U) \cup \mathsf{C}$ is satisfiable,

(ii) (S, U) is rule terminal with respect to γ and $\Lambda_i \wedge F_i \rightsquigarrow E_i$, and

(iii) $(S', U') = (S, U) \triangleleft_U (E_i, \gamma)$.

5.2.2 Interpretations and Models

With states and transitions defined in the previous subsections, we can talk about interpretations and models of PIDL+ specifications. Starting from the initial state, we consider successive applications of transitions, corresponding to the stages of a configuration process. This induces a set of states that are *reachable* from the initial state through *paths* identifying which user and rule transitions lead to a certain state.

Definition 5.29. A *path* from a state (S, U) to a state (S', U') is a tuple

$$((i_1, \gamma_1), (i_2, \gamma_2), \ldots, (i_{n-1}, \gamma_{n-1}), (i_n, \gamma_n)),$$

$n \geq 0$, 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 $(S_0, U_0) = (S, U)$ and $(S_n, U_n) = (S', U')$. If $n = 0$, then the path is called *empty* and is denoted by ().

For the sake of convenience, we use the term "path" to also refer to the sequence of transitions

$$(S_0, U_0) \rightarrow_{(i_1,\gamma_1)} (S_1, U_1) \rightarrow_{(i_2,\gamma_2)} \cdots \rightarrow_{(i_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(i_n,\gamma_n)} (S_n, U_n)$$

that is described by the corresponding tuple

$$((i_1, \gamma_1), (i_2, \gamma_2), \ldots, (i_{n-1}, \gamma_{n-1}), (i_n, \gamma_n)).$$

Definition 5.30. The *length* $|\tau|$ of a path $\tau = ((i_1, \gamma_1), (i_2, \gamma_2), \ldots, (i_{n-1}, \gamma_{n-1}), (i_n, \gamma_n))$ is the number of elements the tuple $((i_1, \gamma_1), (i_2, \gamma_2), \ldots, (i_{n-1}, \gamma_{n-1}), (i_n, \gamma_n))$ contains: $|\tau| := n$.

Definition 5.31. 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 (). We also simply say that a state is reachable if it is reachable from the initial state.

The states that are reachable from the initial state, the transitions that can be applied to the states and a set of assignments for each of the states form an interpretation of an admissible PIDL+ specification $\mathfrak{S}_+$.

Definition 5.32. An *interpretation* of an admissible specification $\mathfrak{S}_+$ is a tuple

$$(\mathcal{V}_{\mathfrak{S}+}, \mathcal{T}_{\mathfrak{S}+}, \mathcal{I}_{\mathfrak{S}+})$$

with

- the *state space* $\mathcal{V}_{\mathfrak{S}+} := \{(S, U) | (S, U) \text{ reachable from } (S_I, U_I)\}$,
- the *transition space*
  $\mathcal{T}_{\mathfrak{S}+} := \{((S, U), i, \gamma, (S', U')) | (S, U), (S', U') \in \mathcal{V}_{\mathfrak{S}+},$
  $(S, U) \rightarrow_{(i,\gamma)} (S', U'), \Lambda_i \wedge F_i \rightsquigarrow E_i \in (T_U \cup T_R)\}$,
- the *state interpretations*
  $\mathcal{I}_{\mathfrak{S}+} := \{((S, U), I_\Pi(\beta)) | (S, U) \in \mathcal{V}_{\mathfrak{S}+}, I_\Pi(\beta) \text{ assignment}, I_\Pi(\beta) \models S\}$.

The components $\mathcal{V}_{\mathfrak{S}+}$ and $\mathcal{T}_{\mathfrak{S}+}$ also called the *state graph* of the specification.

Definition 5.33. Let $\mathfrak{S}_+$ be an admissible specification. The *state graph* $\mathcal{G}_{\mathfrak{S}+}$ is a pair consisting of the state space $\mathcal{V}_{\mathfrak{S}+}$ and the transition space $\mathcal{T}_{\mathfrak{S}+}$:

$$\mathcal{G}_{\mathfrak{S}+} := (\mathcal{V}_{\mathfrak{S}+}, \mathcal{T}_{\mathfrak{S}+}).$$

An interpretation of an admissible specification is a model if its assignments satisfy the constraints. Because of the structure of admissible specifications, this in particular means that the propositional constraints and the bounds on the transition variables X_T are satisfied.

Definition 5.34. 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 $I_\Pi(\beta) \models \mathsf{C}$ for each $((S, U), I_\Pi(\beta)) \in \mathcal{I}_{\mathfrak{S}+}$.

5.3 Algorithms

In this section, we present the algorithm buildInterpretation (Algorithm 1), which constructs a partial interpretation of an admissible specification $\mathfrak{S}_+$ by computing the state space $\mathcal{V}_{\mathfrak{S}+}$ and transition space $\mathcal{T}_{\mathfrak{S}+}$, that is, the state graph. Starting from the initial state of the specification, the algorithm successively applies transitions to states and generates states according to the updates of the transitions. The output can then be the basis for other decision procedures to analyze properties of the state graph and thus of the configuration system modeled by the specification. To determine rule termination and thus user transitions, buildInterpretation makes use of the algorithm reduceSelection (Algorithm 2) which identifies the rule-terminal subselections of maximal selections. Both buildInterpretation and reduceSelection are sound and complete.

A central element are the maximal selections, which are needed for the transition applications that occur during the computation of the state graph. Since the relevant variables are over bounded integers in admissible specifications, finding those selections is a decidable subtask. This can happen by naive enumeration or by more efficient techniques such as interval arithmetic (Fränzle et al., 2007). For our purposes, we just assume the existence of appropriate selections that we can work with and denote those as $maxSelections_{\mathfrak{S}_+}$:

Definition 5.35. Let (S, U) be a state, and $\Lambda \wedge F \rightsquigarrow E \in (T_U \cup T_R)$ be a transition. Then $maxSelections_{\mathfrak{S}_+}(S, U, \Lambda \wedge F \rightsquigarrow E)$ is the set of all maximal selections γ with respect to (S, U) and $\Lambda \wedge F \rightsquigarrow E$.

The input of buildInterpretation is an admissible specification $\mathfrak{S}_+$. The algorithm outputs the state space $\mathcal{V}_{\mathfrak{S}+}$ and transition space $\mathcal{T}_{\mathfrak{S}+}$ of a model of the specification if it exists.

The variables V and T correspond to $\mathcal{V}_{\mathfrak{S}+}$ and $\mathcal{T}_{\mathfrak{S}+}$ respectively, while N is a set containing the states to be processed and H is a set containing the states that have occurred so far. The first state that is processed is the initial state (S_I, U_I) of the specification. Computations last as long as there are states to be processed (line 5). In each iteration of the while loop, a state is taken from N and it is checked whether it is consistent (line 7). This can be done by using a solver that can handle nonlinear integer arithmetic as they may appear in our formulas, for example Z3 (de Moura and Bjørner, 2008). If a state is not consistent, it is just removed from N. If it is consistent, we consider all rule transitions (line 9). For each maximal selection γ with respect to the current state and the current rule transition tuple (line 10), a new state is computed according to the update operator $\triangleleft_R$ (line 11) and the transition application is added to T (line 12). If the new state differs from the current one (line 13), we add the current selection γ to the set Y that contains all the change-admitting selections with respect to the current state (line 14). Y is later used for determining rule-terminal subselections when checking user transition applications. The new state is also registered in V, H and N if we have not seen it before (lines 15–18). After iterating through the rule transitions, buildInterpretation continues if Y is not a singleton containing the empty selection $()$. Otherwise, $()$ is a maximal selection under which there is a rule transition with an update that changes the current state, which means the state is not rule terminal in any way by Definition 5.25, and there is thus no need to look at user transitions. If $\mathsf{Y} \neq \{()\}$, we check the user transitions almost in the same way as done for the rule transitions. The difference is that each selection γ of the current $maxSelections_{\mathfrak{S}_+}$ is reduced to a set of subselections of γ with reduceSelection (line 22). These subselections have the

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

```
1  N := {(S_I, U_I)}
2  H := {(S_I, U_I)}
3  V := {(S_I, U_I)}
4  T := ∅
5  while N ≠ ∅ do
6      let (S, U) ∈ N
7      if ∀x⃗_u ∃y⃗ S|_U → (S \ S|_U) ∪ C is satisfiable then
8          Y := ∅
9          for each Λ_i ∧ F_i ⇝ E_i ∈ T_R do
10             for each γ ∈ maxSelections_{𝔖+}(S, U, Λ_i ∧ F_i ⇝ E_i) do
11                 (S', U') := (S, U) ◁_R (E_i, γ)
12                 T := T ∪ {((S, U), i, γ, (S', U'))}
13                 if (S', U') ≠ (S, U) then
14                     Y := Y ∪ {γ}
15                     if (S', U') ∉ H then
16                         V := V ∪ {(S', U')}
17                         H := H ∪ {(S', U')}
18                         N := N ∪ {(S', U')}
19         if Y ≠ {()} then
20             for each Λ_i ∧ F_i ⇝ E_i ∈ T_U do
21                 for each γ ∈ maxSelections_{𝔖+}(S, U, Λ_i ∧ F_i ⇝ E_i) do
22                     for each δ ∈ reduceSelection(γ, Y) do
23                         if δ(j) ≠ ∅ for all j or δ = () then
24                             (S', U') := (S, U) ◁_U (E_i, δ)
25                             T := T ∪ {((S, U), i, δ, (S', U'))}
26                             if (S', U') ≠ (S, U) then
27                                 if (S', U') ∉ H then
28                                     V := V ∪ {(S', U')}
29                                     H := H ∪ {(S', U')}
30                                     N := N ∪ {(S', U')}
31     N := N \ {(S, U)}
32 return (V, T)
```

property that they can be rule-terminal subselections, which is important for applying the current user transition to the current state. If such a subselection δ has at least one empty component, then it is not a selection with respect to the current state and the user transition by Definition 5.8 — the user instances with respect to at least one variable have been reduced to "nothing". If this is not the case, or δ is just the empty selection () (line 23), which means that no user variables are involved and the selection is rule terminal, BUILDINTERPRETATION proceeds exactly as in the case of the loop for rule transitions by creating a new state and register it accordingly (lines 26–30). At the end of each iteration, the current state is removed from the set of states to be processed (line 31). When this set is finally empty, the state graph with respect to $\mathfrak{S}_+$ is returned in the form of (V, T) (line 32).

Algorithm 2: REDUCESELECTION(γ, Y)

```
1  if Y = ∅ then
2  |  return {γ}
3  let ρ ∈ Y
4  Y′ := Y \ {ρ}
5  while Y′ ≠ ∅ and γ ∩ ρ = ∅ do
6  |  let ρ ∈ Y′
7  |  Y′ := Y′ \ {ρ}
8  if γ ∩ ρ = ∅ then
9  |  return {γ}
10 R := ∅
11 for i = 1 to |γ| do
12 |  s := γ(i) \ ρ(i)
13 |  if s = I1 ∪ I2, I1, I2 intervals and I1 ∩ I2 = ∅ then
14 |  |  R := R ∪ reduceSelection(γ[i/I1], Y′) ∪ reduceSelection(γ[i/I2], Y′)
15 |  else
16 |  |  R := R ∪ reduceSelection(γ[i/s], Y′)
17 return R
```

BUILDINTERPRETATION uses REDUCESELECTION to determine the rule-terminal subselections needed for user transitions. REDUCESELECTION takes as arguments a selection γ and a set Y of selections. It returns a set of subselections of γ that have empty intersections with all the selections in Y. Since BUILDINTERPRETATION calls REDUCESELECTION with γ being a maximal selection with respect to the current state and the current user transition, and Y being the set containing all the change-admitting selections with respect to the current state, the output is a set of candidates for rule-terminal subselections of γ because the maximality property stated in Definition 5.25 is preserved by BUILDINTERPRETATION.

The first thing REDUCESELECTION does is to check if Y is empty. There is nothing to be done if this is the case, and a singleton containing γ is returned (lines 1–2). Next, we iterate through Y by removing its elements until we have found a selection ρ that is not disjoint to γ or until Y has become empty (lines 3–7). This means we discard the selections of Y having an empty intersection with γ up to the point of line 8 because those selections are not critical to the rule termination of γ: They represent instances of the current state that entail rule transitions changing the state, but γ does not share any instances with them. After the while loop, Y can be empty and the last selection ρ visited can thus have an empty intersection with γ. In this case, we again just return

$\{\gamma\}$ (lines 8 and 9). Beyond this point, we have a selection ρ from Y that is not disjoint to γ, that is, for each position i, $\gamma(i) \cap \rho(i) \neq \emptyset$. This means that γ also represents instances of the current state, namely some of those contained in ρ, that entail rule transitions whose updates change the state, so γ cannot be seen as rule terminal, given the relationship to ρ alone. The next steps of the algorithm are therefore supposed to eliminate this relationship by reducing γ to subselections that do not share any instances with ρ anymore. The variable R holds the results of those reductions (line 10). The reduction can be done by choosing a position i of γ and then doing a set difference $\gamma(i) \setminus \rho(i)$ at this position. This is because for a selection to have an empty selection with another one, it suffices that they do not share any instances at one position. Thus, if we subtract ρ from γ at one position, the resulting selection does not share any instances with ρ and it is still a selection with respect to the current state and user transition. To cover all the possibilities, we do the difference operation (line 12) for each position of γ (line 11). Note that γ and ρ have the same length because all selections appearing in this algorithm are with respect to the same state (S, U) in BUILDINTERPRETATION and therefore the same set of user variables U, which determines the length of the corresponding selections. When doing the set difference, there is a case distinction. If the result of the difference are two intervals I_1 and I_2 that are not connected (line 13), which is the case when $\min(\gamma(i)) < \min(\rho(i)) \leq \max(\rho(i)) < \max(\gamma(i))$, we have to have two recursive calls of REDUCESELECTION covering these two possibilities: One with an argument γ whose i-th component is replaced by the reduced I_1 and one with an argument γ whose i-th component is replaced by the reduced I_2 (line 14). If the result of the difference is just a single interval, we have one recursive call of REDUCESELECTION with an argument γ whose i-th element is replaced by the set difference s (line 16). The results are collected in the variable R which is returned in the end (line 17). After that, R contains all the possible subselections of the input γ that only have empty intersections with the selections in Y and are therefore candidates for rule-terminal subselections. Elements of R are not rule-terminal subselections if they have an empty component, as mentioned earlier: They represent no instances at all that can entail user transitions. This is checked in line 23 of BUILDINTERPRETATION. Those selections of R which have no empty components are indeed rule-terminal subselections of γ, as they are also maximal because REDUCESELECTION only reduces the subselections of γ in a way that they just about have empty intersections with the selections of Y by applying set difference.

We formalize what we have said above and prove the correctness of the algorithms. The following theorem prepares the correctness of REDUCESELECTION.

Theorem 5.36. The set returned by REDUCESELECTION(γ, Y) contains all the selections $\delta \subseteq \gamma$ with the following properties:

(i) δ is disjoint to Y, and

(ii) there is no selection $\omega \subseteq \gamma$ such that ω is disjoint to Y and $\delta \subset \omega$.

Proof. By induction on the size of Y.

Let $|\mathsf{Y}| = 0$. It holds that $\mathsf{Y} = \emptyset$. Then, $\{\gamma\}$ is returned by REDUCESELECTION. We see that γ is trivially disjoint to Y, and there is no other selection $\omega \subseteq \gamma$ that is disjoint to Y and $\gamma \subset \omega$. The result $\{\gamma\}$ also contains the only selection that satisfies the points (i) and (ii) of the theorem.

Let $|\mathsf{Y}| = n$. There are two cases.

(a) After the while loop, $\mathsf{Y} = \emptyset$ and γ is disjoint to the "last" element ρ of Y (line 8). The algorithm returns $\{\gamma\}$, and the loop shows that it is disjoint to Y. Like in the

base case, there is no selection $\omega \subseteq \gamma$ that is disjoint to Y and $\gamma \subset \omega$, and, again, $\{\gamma\}$ is the only possible result in this case.

(b) After the while loop, we have a selection $\rho \in \mathsf{Y}$ with $\gamma \cap \rho \neq \emptyset$. The for loop iterates through the positions of γ. We look at the recursive calls of REDUCESELECTION inside the for loop with respect to a position i. Consider the case where the set difference $s = \gamma(i) \setminus \rho(i)$ is not connected (line 13). We then have two recursive calls of REDUCESELECTION with the selections $\gamma[i/I_1]$ and $\gamma[i/I_2]$, and the reduced set Y' as the respective arguments. We observe the following with respect to $\gamma[i/I_1]$:

$$\begin{gathered}\gamma[i/I_1] \text{ is disjoint to } \rho, \text{ and} \\ \text{there is no other selection } \omega \subseteq \gamma \text{ that is disjoint to } \rho \text{ and } \gamma[i/I_1] \subset \omega. \quad (5.3)\end{gathered}$$

This is because the set difference removes just enough elements from γ to make it disjoint to ρ. The same holds for the other reduced selection $\gamma[i/I_2]$ and the case $\gamma[i/s]$, when s is a single connected interval. By iterating through all positions of γ as is done in the for loop, and taking connectedness of the intervals into account when doing the set difference, we exhaust all the possibilities of subselections of γ that are disjoint to ρ and that are not proper subselections of any other selections $\omega \subseteq \gamma$ with the same property.

By induction hypothesis, REDUCESELECTION$(\gamma[i/I_1], \mathsf{Y}')$ returns the set of all selections $\delta \subseteq \gamma[i/I_1]$, where δ is disjoint to Y', and there is no selection $\omega \subseteq \gamma[i/I_1]$ with ω disjoint to Y' and $\delta \subset \omega$.

Consider such a selection $\delta \in$ REDUCESELECTION$(\gamma[i/I_1], \mathsf{Y}')$. Since $\delta \subseteq \gamma[i/I_1]$ and $\gamma[i/I_1]$ is disjoint to ρ, it holds that δ is also disjoint to ρ. Note that ρ is the first selection of Y found in the while loop that is not disjoint to γ. The other selections removed from Y in the while loop are disjoint to γ. Therefore, we can also say that δ is disjoint to Y.

Now the question is whether there is a selection $\omega \subseteq \gamma$ such that ω is disjoint to Y and $\delta \subset \omega$. Assume that there is such an ω. With $\delta \subset \omega$ and $\omega \subseteq \gamma$, it holds that $\omega \subseteq \gamma[i/I_1]$, because else $\gamma[i/I_1] \subset \omega$ with ω being disjoint to ρ, which is a contradiction to (5.3). Since ω is disjoint to Y, it is also disjoint to Y'. This means that ω contradicts the role of δ given by the induction hypothesis.

We get that, with respect to $\gamma[i/I_1]$, the call REDUCESELECTION$(\gamma[i/I_1], \mathsf{Y}')$ returns the set of all selections $\delta \subseteq \gamma$ that are disjoint to Y, and there is no selection $\omega \subseteq \gamma$ such that ω is disjoint to Y and $\delta \subset \omega$. Since the for loop determines the recursive calls of REDUCESELECTION with respect to the subselections of γ and the selection ρ exhaustively, as described above, the union R of all the recursive calls indeed is by induction hypothesis the set of all selections δ satisfying the points (i) and (ii) of the theorem.

□

Corolloray 5.37. Let γ be a maximal selection with respect to a state (S, U) and a user transition $\Lambda \wedge F \rightsquigarrow E \in T_U$. A selection δ with $\delta \subseteq \gamma$ and $\delta(i) \neq \emptyset$ for all $i \in \{1, \ldots, |\delta|\}$ is a rule-terminal subselection of γ if and only if $\delta \in$ REDUCESELECTION(γ, Y), where Y is the set of all change-admitting selections with respect to (S, U).

Theorem 5.38. Let $\mathfrak{S}_+ = (\Pi, X, S_I, U_I, \mathsf{C}, T_U, T_R)$ be an admissible specification and BUILDINTERPRETATION$(\mathfrak{S}_+) = (V, T)$. 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

$$((S_0, U_0), i_1, \gamma_1, (S_1, U_1)), \ldots, ((S_{n-1}, U_{n-1}), i_n, \gamma_n, (S_n, U_n)),$$

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 $j = 1, \ldots, n$, $|\tau| = n$, $(S_0, U_0) = (S_I, U_I)$, $(S_n, U_n) = (S, U)$.

Proof. ($\Rightarrow$) By induction on the path τ.

Let $\tau = ()$. We only have to consider the initial state (S_I, U_I), which is reachable from itself with the empty path $()$. In BUILDINTERPRETATION, V is initialized with $\{(S_I, U_I)\}$, and because τ is empty, we do not have to look at T. Since V is never reduced in the algorithm, $(S_I, U_I) \in V$ indeed after a run of the algorithm.

Let $\tau = \tau' :: (i, \gamma)$. The last transition represented by the path has the form

$$(S_n, U_n) \rightarrow_{(i,\gamma)} (S, U). \tag{5.4}$$

We have two cases:

- The transition is done with γ being a maximal selection with respect to (S_n, U_n) and a rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$. By induction hypothesis, $(S_n, U_n) \in V$ and there is a sequence

  $$((S_I, U_I), i_1, \gamma_1, (S_1, U_1)), \ldots, ((S_{n-1}, U_{n-1}), i_n, \gamma_n, (S_n, U_n)) \in T$$

  with $\tau'(j) = (i_j, \gamma_j)$ for all $j = 1, \ldots, n$. We have to show that the last transition using (i, γ) in τ is also present in the algorithm's output.

  Since the state (S_n, U_n) is in V, it must be in N at some point. Consider the point when (S_n, U_n) is the current state as chosen in line 6 in the algorithm. The existence of the last transition (i, γ) above (5.4) implies that the state (S_n, U_n) is consistent, that is,

  $$\forall \vec{x}_u \exists \vec{y}\ S_n|_{U_n} \rightarrow (S_n \setminus S_n|_{U_n}) \cup \mathsf{C} \text{ is satisfiable}$$

  by Definition 5.22. This means the consistency check in line 7 is positive and BUILDINTERPRETATION advances to the for loops iterating through the rule transitions and respective selections (lines 9 and 10). By the definition of $maxSelections_{\mathfrak{S}_+}$ (Definition 5.35), the rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i$ and the corresponding selection γ are eventually identified by the loops. We get the state (S, U) as usual by virtue of the rule update operator (line 11). The state-transition-state tuple $((S_n, U_n), i, \gamma, (S, U))$ is added to T (line 12) and the state (S, U) is added to V (line 16) if (S, U) differs from (S_n, U_n). If (S, U) is equal to (S_n, U_n), then (S, U) is not added to V, but is already in V because it must have been added to V at some point before the current iteration, since it is taken from N. In any case, since nothing is ever removed from S and T, we have that $(S, U) \in V$ and $((S_n, U_n), i, \gamma, (S, U)) \in T$.

- The transition is done with γ being a maximal selection with respect to (S_n, U_n) and a user transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U$. By induction hypothesis, $(S_n, U_n) \in V$ and there is a sequence

  $$((S_I, U_I), i_1, \gamma_1, (S_1, U_1)), \ldots, ((S_{n-1}, U_{n-1}), i_n, \gamma_n, (S_n, U_n)) \in T$$

  with $\tau'(j) = (i_j, \gamma_j)$ for all $j = 1, \ldots, n$. Analogously to the first case, we have to show that the last transition using (i, γ) in τ is has corresponding elements in V and T produced by the algorithm.

The proof is very similar to that of the rule transition case. We therefore mainly describe the differences that are due to the inherent properties of user transitions.

The state (S_n, U_n) must have been in N at a certain point. We look at the iteration in which it is the current state taken from N. After the loop examining the rule transitions, it is checked if $\mathsf{Y} = \{()\}$ (line 19). This cannot be the case, or else $()$ would be a maximal selection with respect to (S_n, U_n) and a rule transition, which would contradict this case's assumption that in (5.4) a user transition can be applied to (S_n, U_n). So indeed $\mathsf{Y} \neq \{()\}$ and we enter the loops iterating through the user transitions and their respective selections. At one point the user transition $\Lambda_i \wedge F_i \rightsquigarrow E_i$ is found. Because $maxSelections_{\mathfrak{S}_+}$ contains all the maximal selections with respect to (S_n, U_n) and $\Lambda_i \wedge F_i \rightsquigarrow E_i$, and REDUCESELECTION contains all the rule-terminal subselections of those selections by Corollary 5.37, we eventually arrive at the selection γ_i responsible for the user transition in (5.4). The steps afterward correspond exactly to the first case of a rule transition in this proof. Thus, we get $(S, U) \in V$ and $((S_n, U_n), i, \gamma, (S, U)) \in T$.

($\Leftarrow$) The state-transition-state tuples in T describe paths τ as defined in Definition 5.29. We prove this direction by induction on those paths.

Let $\tau = ()$. This corresponds to the case when the state in question is $(S_I, U_I) \in V$. This state is obviously reachable from itself by the empty path.

Let $\tau = \tau' :: (i, \gamma)$. We have $(S, U) \in V$ and a sequence

$$((S_I, U_I), i_1, \gamma_1, (S_1, U_1)), \ldots, ((S_{n-1}, U_{n-1}), i_n, \gamma_n, (S_n, U_n)), ((S_n, U_n), i, \gamma, (S, U)) \in T,$$

with $\tau'(j) = (i_j, \gamma_j)$ for all $j = 1, \ldots, n$.

There are two cases:

- The last tuple $((S_n, U_n), i, \gamma, (S, U))$ in the sequence is with respect to a rule transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_R$.

  By induction hypothesis, (S_n, U_n) is reachable from the initial state via path τ'. We have to show that there is a transition $(S_n, U_n) \rightarrow_{(i,\gamma)} (S, U)$. The state (S_n, U_n) is contained in N and thus processed at some point. There, the consistency check in line 7 is positive, meaning that

  $$\forall \vec{x}_u \exists \vec{y}\ S_n|_{U_n} \rightarrow (S_n \setminus S_n|_{U_n}) \cup \mathsf{C} \text{ is satisfiable.}$$

  The completeness of $maxSelections_{\mathfrak{S}_+}$ gives us the correct selection γ, and the following update as usual by Definition 5.21 yields the correct state (S, U). All in all, we indeed get a transition $(S_n, U_n) \rightarrow_{(i,\gamma)} (S, U)$ as required by Definition 5.22.

- The last tuple $((S_n, U_n), i, \gamma, (S, U))$ in the sequence is with respect to a user transition $\Lambda_i \wedge F_i \rightsquigarrow E_i \in T_U$.

  By induction hypothesis, (S_n, U_n) is reachable from the initial state via path τ'. Again, we need to show that there is a transition $(S_n, U_n) \rightarrow_{(i,\gamma)} (S, U)$. The steps and reasoning needed are mostly the same as in the proof for rule transitions. The only thing to consider is that by the completeness of $maxSelections_{\mathfrak{S}_+}$ and REDUCESELECTION (Corollary 5.37) we get the correct rule-terminal selection γ. Then, it also holds in this case that $(S_n, U_n) \rightarrow_{(i,\gamma)} (S, U)$ according to Definition 5.28.

□

5.4 Properties of Admissible PIDL+ Specifications

Analogously to PIDL, we show how certain properties of configuration systems can be expressed within the PIDL+ framework. Some concepts can be carried over straightforwardly, some other have to be adapted because of the big-steps semantics of the states in PIDL+. As with the case of PIDL, all properties can be decided and computed by analyzing the finite state graph of the corresponding specification (Theorem 5.44).

Soundness. An admissible specification $\mathfrak{S}_+$ is *sound* if it has a model according to Definition 5.34. As in PIDL, consistency properties given by the corresponding domain of the configuration system modeled are expressed as formulas in the set C of constraints as usual. However, these formulas must follow the restrictions stated in the definition of admissible specifications (Definition 5.4), that is, they must be purely propositional formulas over Π. Then, the specification is sound if there is no state (S, U) reachable from the initial state (S_I, U_I) with

$$\forall \vec{x}_u \exists \vec{y}\, S|_U \rightarrow (S \setminus S|_U) \cup \mathsf{C} \text{ not satisfiable.}$$

The property can be checked while constructing the state graph $\mathcal{G}_{\mathfrak{S}+}$.

Example 5.39. Consider the following admissible specification $\mathfrak{S}_+$:

$$\begin{aligned}
S_I &= \{A\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x \geq 0, x \leq 100, \\
&\qquad y \geq 0, y \leq 10\} \\
T_U &= \{A \rightsquigarrow_{u_1} \{x > 0, x < 20\}\} \\
T_R &= \{x > 0 \wedge x \leq 10 \rightsquigarrow_{r_1} \{y \approx 19\}\}
\end{aligned}$$

The initial state (S_I, U_I) does not have any user variables and is rule terminal. There is a user transition with u_1 and selection $\gamma_1 = ()$,

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

where

$$\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \lhd_R (E_{u_1}, \gamma_1) \\
&= (\{A\}, \emptyset) \lhd_R (\{x > 0, x < 20\}, ()) \\
&= (\{A, x > 0, x < 20\}, \{x\}).
\end{aligned}$$

From (S_1, U_1), we can apply rule transition r_1 with selection $\gamma_2 = ([1, 10])$:

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

with

$$\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\
&= (\{A, x > 0, x < 20\}, \{x\}) \lhd_R (\{y \approx 19\}, [(1, 10)]) \\
&= (\{A, x \geq 1, x \leq 10, y \approx 19\}, \{x\}).
\end{aligned}$$

(S_2, U_2) is inconsistent because the rule-set value of y violates the bounds $y \geq 0, y \leq 10$ defined in C, that is,

$$\forall \vec{x}_u \exists \vec{y}\, S_2|_{U_2} \rightarrow (S_2 \setminus S_2|_{U_2}) \cup \mathsf{C} \text{ is not satisfiable.}$$

$\mathfrak{S}_+$ is therefore not sound.

Specification Completeness. Completeness of an admissible specification with respect to a set X' of arithmetic variables means the following: Every valuation of the variables in X' meeting the bounds in the constraints can be found in one of the rule-terminal states that is reachable from the initial state of the specification. This notion of completeness aims at expressing that all products represented by a valid valuation of variables according to the constraint bounds in C are buildable. The propositional part of the completeness definition corresponds to the case in PIDL (Section 4.4). Only states that are fixpoints with respect to rule transitions can represent products, hence the rule-termination requirement.

An admissible PIDL+ specification $\mathfrak{S}_+$ is *complete* with respect to a set $\Pi' \subseteq \Pi$ of propositional variables and a set $X' \subseteq X$ of arithmetic variables if for every pair (M, σ), where

- M is a set of propositional literals with $vars(M) = vars(\Pi')$ and $M \cup \mathsf{C} \not\models \bot$, and
- σ is a substitution $\sigma : X' \to \mathbb{Z}$ with $\mathsf{C}\sigma$ being satisfiable,

there is a state $(S, U) \in \mathcal{V}_{\mathfrak{S}+}$ such that the following holds:

(i) $M \subseteq S$.

(ii) $vars(X') \subseteq varsl(S)$.

(iii) $(S \cup \mathsf{C})\sigma$ is satisfiable.

(iv) If $U \neq \emptyset$ and

  (a) $X' \cap U \neq \emptyset$, then there is no change-admitting selection γ with respect to (S, U) such that
  $$v_j \in \gamma(j) \text{ for all } x_j \mapsto v_j \in \sigma|_U.$$

  (b) $X' \cap U = \emptyset$, then there is a a subselection γ'_S of the base selection γ_S of (S, U) such that
  - $\gamma'_S(k) \neq \emptyset$ for all $k = 1, \ldots, |U|$, and
  - there is no change-admitting selection γ with respect to (S, U) with $\gamma'_S \cap \gamma \neq \emptyset$.

(v) If $U = \emptyset$, then () is not a change-admitting selection with respect to (S, U).

Given a specification $\mathfrak{S}_+$, a literal set M, and a substitution σ that maps variables in X' to integers, completeness implies that there is a reachable state (S, U) that is rule terminal in a certain sense, contains the literals M and represents the instances in σ. Here, rule termination can occur in three different possibilities. In (iv)(a), the set of variables X' has a non-empty intersection with the user variables U. This means some of the variables in X' are also user variables. The valuation of those variables represented by σ is not contained in any selection that causes a rule transition that changes the state. In other words, the user assignments contained in σ are not part of a selection that is not rule terminal with respect to the state. In requirement (iv)(b), X' does not contain any user variables. Therefore, rule termination is decided by whether there exists a subselection of the base selection that does not share any instances with selections that break rule termination. Finally, (v) considers the scenario where the state has no user variables, which means that we simply check rule termination with respect to the empty selection ().

Example 5.40. In the following admissible specification $\mathfrak{A}$, we consider completeness with respect to criterion (iv)(a):

$$
\begin{aligned}
S_I &= \{A\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x \geq 0, x \leq 30, \\
&\qquad y \geq 0, y \leq 5\} \\
T_U &= \{A \rightsquigarrow_{u_1} \{B, x \geq 0, x \leq 20\}, \\
&\qquad A \rightsquigarrow_{u_2} \{\neg B, x \geq 21, x \leq 30\}\} \\
T_R &= \{x \geq 10 \wedge x \leq 15 \rightsquigarrow_{r_1} \{C\}, \\
&\qquad \neg B \rightsquigarrow_{r_2} \{y \approx 4\}\}
\end{aligned}
$$

From S_I there is a user transition using u_1 and $\gamma_1 = ()$, which gives

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

with

$$
\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\
&= (\{A\}, \emptyset) \lhd_U (\{B, x \geq 0, x \leq 20\}, ()) \\
&= (\{A, B, x \geq 0, x \leq 20\}, \{x\}).
\end{aligned}
$$

Then, rule transition r_1 with $\gamma_2 = ([10, 15])$ can be applied to (S_1, U_1),

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

where

$$
\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\
&= (\{A, B, x \geq 0, x \leq 20\}, \{x\}) \lhd_R (\{C\}, ([10, 15])) \\
&= (\{A, B, C, x \geq 10, x \leq 15\}, \{x\}).
\end{aligned}
$$

Going back, applying user transition u_2 with $\gamma_3 = ()$ to (S_I, U_I) yields

$$(S_I, U_I) \rightarrow_{(u_2,\gamma_3)} (S_3, U_3)$$

where

$$
\begin{aligned}
(S_3, U_3) &= (S_I, U_I) \lhd_U (E_{u_2}, \gamma_3) \\
&= (\{A\}, \emptyset) \lhd_U (\{\neg B, x \geq 21, x \leq 30\}, ()) \\
&= (\{A, \neg B, x \geq 21, x \leq 30\}, \{x\}).
\end{aligned}
$$

We apply rule transition r_2 with $\gamma_4 = ([21, 30])$ to (S_3, U_3) and get

$$(S_3, U_3) \rightarrow_{(r_2,\gamma_4)} (S_4, U_4)$$

with

$$
\begin{aligned}
(S_4, U_4) &= (S_3, U_3) \lhd_R (E_{r_2}, \gamma_4) \\
&= (\{A, \neg B, x \geq 21, x \leq 30\}, \{x\}) \lhd_R (\{y \approx 4\}, ([21, 30])) \\
&= (\{A, \neg B, x \geq 21, x \leq 30, y \approx 4\}, \{x\}).
\end{aligned}
$$

The paths are

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

and

$$(S_I, U_I) \rightarrow_{(u_2,\gamma_3)} (S_3, U_3) \rightarrow_{(r_2,\gamma_4)} (S_4, U_4).$$

For example, the specification is complete with respect to $\{x\}$, where rule termination is determined according to requirement (iv)(a):

- In (S_1, U_1), (iv)(a) is fulfilled for substitutions $x \mapsto 0, \ldots, x \mapsto 9$ and for substitutions $x \mapsto 16, \ldots, x \mapsto 20$,
- in (S_2, U_2), (iv)(a) is fulfilled for substitutions $x \mapsto 10, \ldots, x \mapsto 15$, and
- in (S_4, U_4), (iv)(a) is fulfilled for substitutions $x \mapsto 21, \ldots, x \mapsto 30$.

As a result, all valid instances of x can be reached from (S_I, U_I).

If we look at $\{y\}$, we see that the specification is not complete with respect to it: There are no states in which instances of y different from 4 can be reached.

The second admissible specification $\mathfrak{B}$ in this example deals with completeness with respect to criterion (iv)(b) and (v):

$$\begin{aligned}
S_I &= \{A\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x \geq 0, x \leq 20, \\
&\quad\ y \geq 20, y \leq 90\} \\
T_U &= \{A \rightsquigarrow_{u_1} \{\neg A, x \geq 0, x \leq 10\}\} \\
T_R &= \{x \geq 0 \wedge x \leq 5 \wedge A \rightsquigarrow_{r_1} \{B, y \geq 20, y \leq 90\}, \\
&\quad\ B \rightsquigarrow_{r_2} \{C, x \approx 0\}, \\
&\quad\ C \rightsquigarrow_{r_3} \{D\}\}
\end{aligned}$$

We can apply user transition u_1 with $\gamma_1 = ()$ to (S_I, U_I),

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

where

$$\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\
&= (\{A\}, \emptyset) \lhd_U (\{\neg A, x \geq 0, x \leq 10\}, ()) \\
&= (\{\neg A, x \geq 0, x \leq 10\}, \{x\}).
\end{aligned}$$

Rule transition r_1 with $\gamma_2 = ([0, 5])$ applied to (S_1, U_1) gives

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

with

$$\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\
&= (\{\neg A, x \geq 0, x \leq 10\}, \{x\}) \lhd_R (\{B, y \geq 20, y \leq 90\}, ([0, 5])) \\
&= (\{\neg A, B, x \geq 0, x \leq 5, y \geq 20, y \leq 90\}, \{x\}).
\end{aligned}$$

Then, we apply rule transition r_2 with $\gamma_3 = ([0,5])$ to (S_2, U_2), which leads to

$$(S_2, U_2) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3)$$

where

$$\begin{aligned}(S_3, U_3) &= (S_2, U_2) \lhd_R (E_{r_2}, \gamma_3) \\ &= (\{\neg A, B, x \geq 0, x \leq 5, y \geq 20, y \leq 90\}, \{x\}) \lhd_R (\{C, x \approx 0\}, ([0,5])) \\ &= (\{\neg A, B, C, x \approx 0, y \geq 20, y \leq 90\}, \emptyset).\end{aligned}$$

Finally, there is a rule transition from (S_3, U_3) using r_3 and $\gamma_4 = ()$:

$$(S_3, U_3) \rightarrow_{(r_3,\gamma_4)} (S_4, U_4)$$

where

$$\begin{aligned}(S_4, U_4) &= (S_3, U_3) \lhd_R (E_{r_3}, \gamma_4) \\ &= (\{\neg A, B, C, x \approx 0, y \geq 20, y \leq 90\}) \lhd_R (\{D\}, ()) \\ &= (\{\neg A, B, C, D, x \approx 0, y \geq 20, y \leq 90\}, \emptyset).\end{aligned}$$

The corresponding path is

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3) \rightarrow_{(r_3,\gamma_4)} (S_4, U_4).$$

The specification is complete with respect to $\{y\}$. The relevant reachable state in this case is (S_4, U_4), where $U_4 = \emptyset$, since it contains all the instances of y between 20 and 90 and is rule terminal according to (v). The other states in which y occurs cannot be used as justification for completeness, because

- in (S_2, U_2), where $\{y\} \cap U_2 = \emptyset$, there is no subselection γ'_{S_2} of the base selection γ_{S_2} as specified in (iv)(b), as γ_{S_2} itself is change-admitting with respect to (S_2, U_2), and
- in (S_3, U_3), where $U_3 = \emptyset$, $()$ is actually a change-admitting selection with respect to (S_3, U_3).

Metaproperties. This case is handled in a similar way as done for PIDL. First, we express the metaproperty as a quantifier-free formula ϕ. We then consider states $(S, U) \in \mathcal{V}_{\mathfrak{S}+}$ and their user instances that do not contain any instances that entail rule transitions altering the state by their updates. If such a state entails ϕ modulo those rule-terminal instances, the metaproperty is satisfied. To state it formally, for each state $(S, U) \in \mathcal{V}_{\mathfrak{S}+}$ the following properties holds:

(i) If $U \neq \emptyset$ and there is a subselection γ of the base selection of (S, U), where there is no change-admitting selection γ' with respect to (S, U) and $\gamma \cap \gamma' \neq \emptyset$, then there is a substitution $\sigma : U \to \mathbb{Z}$ with

$$x_i\sigma \in \gamma(i) \text{ for all } x_1, \ldots, x_n \in U, \text{ and}$$

$$(S \cup \mathsf{C})\sigma \text{ satisfiable and } (S \cup \mathsf{C})\sigma \models \phi\sigma.$$

(ii) If $U = \emptyset$ and $()$ is not a change-admitting selection with respect to (S, U), then

$$S \cup \mathsf{C} \models \phi.$$

Example 5.41. Consider the following admissible specification $\mathfrak{S}_+$:

$S_I = \{A, \neg B\}$

$U_I = \emptyset$

$\mathsf{C} = \{x \geq 0, x \leq 100\}$

$T_U = \{A \rightsquigarrow_{u_1} \{x \geq 30, x \leq 80\}\}$

$T_R = \{x \leq 50 \rightsquigarrow_{r_1} \{\neg A, B\}\}$

Moreover, let the metaproperty

$$\phi = B \to x \leq 20$$

be given.

The initial state (S_I, U_I) does not have user variables, and () is not a change-admitting selection with respect to it. It holds that

$$S \cup \mathsf{C} = \{A, \neg B\} \models B \to x \leq 20 = \phi,$$

so the metaproperty is satisfied by this state. (S_I, U_I) is rule terminal with respect to $\gamma_1 = ()$ and user transition u_1:

$$(S_I, U_I) \to_{(u_1, \gamma_1)} (S_1, U_1)$$

where

$$\begin{aligned}(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\ &= (\{A, \neg B\}, \emptyset) \lhd_U (\{A, \neg B, x \geq 30, x \leq 80\}, ()) \\ &= (\{A, \neg B, x \geq 30, x \leq 80\}, \{x\}).\end{aligned}$$

We look at the rule-terminal instances of (S_1, U_1). They are represented by the selection $\gamma_2 = ([51, 80])$. We can see that

$$(S \cup \mathsf{C})\sigma = \{A, \neg B, x \geq 30, x \leq 80\}\sigma \text{ is satisfiable}$$

for all substitutions σ with

$$x\sigma \in \gamma_2,$$

and

$$(S \cup \mathsf{C})\sigma = \{A, \neg B, x \geq 30, x \leq 80\})\sigma \models (B \to x \leq 20)\sigma = \phi\sigma.$$

Hence, the metaproperty also holds in (S_1, U_1) with respect to γ_2. We apply the rule transition r_1 to (S_1, U_1) using the selection $\gamma_3 = ([30, 50])$ to get

$$(S_1, U_1) \to_{(r_1, \gamma_3)} (S_2, U_2)$$

with

$$\begin{aligned}(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_3) \\ &= (\{A, \neg B, x \geq 30, x \leq 80\}, \{x\}) \lhd_R (\{\neg A, B\}, ([30, 50])) \\ &= (\{\neg A, B, x \geq 30, x \leq 50\}, \{x\}).\end{aligned}$$

The base selection $\gamma_{S_2} = ([30, 50])$ of (S_2, U_2) already fulfills the requirements of the above definition, that is, there are no change-admitting selections with respect to (S_2, U_2) that are not disjoint to γ_{S_2}. Moreover, there is no substitution σ for which

$$(S \cup \mathsf{C})\sigma = \{\neg A, B, x \geq 30, x \leq 50\}\sigma \text{ is satisfiable}$$

and

$$(S \cup \mathsf{C})\sigma = \{\neg A, B, x \geq 30, x \leq 50\}\sigma \models (B \rightarrow x \leq 20)\sigma.$$

We conclude that (S_2, U_2) does not satisfy the metaproperty ϕ. Therefore, the specification does not satisfy the metaproperty ϕ.

Cyclicity. The formulation of cyclicity in PIDL+ has to take account of the fact that states in PIDL+ are aggregations of instances of the user variables appearing in the states. Therefore, cycles in PIDL+ are formulated modulo the instances represented by the states' bounds. What we are looking for are instantiations of paths that corresponds to simple cycles as defined for the case of PIDL. In a PIDL+ specification, there is a cycle if

(i) there is a sequence of transitions

$$(S_0, U_0) \rightarrow_{(t_1,\gamma_1)} (S_1, U_1) \rightarrow_{(t_2,\gamma_2)} \cdots \rightarrow_{(t_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(t_n,\gamma_n)} (S_n, U_n)$$

with $n \geq 2$ and $\Lambda_{t_i} \wedge F_{t_i} \rightsquigarrow E_{t_i} \in (T_U \cup T_R)$ for $i = 1, \ldots, n$, and

(ii) there is a sequence of substitutions $\sigma_0, \ldots, \sigma_n$ with $\sigma_i : U_i \rightarrow \mathbb{Z}$ for $i = 0, \ldots, n$, and

(a) $(S_i \cup \mathsf{C})\sigma_i$ is satisfiable for $i = 0, \ldots, n$,

(b) for all $i = 0, \ldots, n-1$ it holds that if $U_i = \{x_1, \ldots, x_m\}$, then $x_j\sigma_i \in \gamma_{i+1}(j)$ for all $j = 1, \ldots, m$,

(c) $x\sigma_i = x\sigma_{i+1}$ if $x \notin \mathit{varsl}(E_{t_{i+1}})$ for all $i = 0, \ldots, n-1$,

(d) $S_i\sigma_i$ are distinct for $i = 1, \ldots, n$, and

(e) $S_0\sigma_0 = S_n\sigma_n$.

As explained in the PIDL case, we only consider paths whose lengths are at least greater than one, that is, we ignore self-loop cycles. Condition (b) requires the substitutions to represent exactly those user instances that entail the transitions, that is, they must come from the respective selections annotating the transitions. The substitutions must also satisfy condition (c), which becomes clear from the perspective of the small-steps semantics (Section 5.2.1): As long as an instance of a user variable assignment that entails a transition is not replaced by the transition's update, it stays exactly the same in the following updated state. This is to preserve correct instances in the small-steps semantics. For example, assume the following state and transition, where we leave the parts out that are not relevant to illustrating the concept:

$$(S, U) = (\{A, x \geq 0, x \leq 20\}, \{x\})$$
$$t = A \rightsquigarrow \{y > 6\}$$

The state entails the transition given by t, and we have

$$(S, U) \rightarrow_{(t,\gamma)} (S', U')$$

with

$$\begin{aligned}(S', U') &= (S, U) \triangleleft (E_t, \gamma) \\ &= (\{A, x \geq 0, x \leq 20\}, \{x\}) \triangleleft (\{y > 6\}, [(0, 20)]) \\ &= (\{A, x \geq 0, x \leq 20, y > 6\}, \{x\})\end{aligned}$$

where the types of the transition t and update operator $\triangleleft$ do not matter for our purposes and are therefore left unspecified. In particular, the way the literal set S is updated is equal for user and rule transitions by Definition 5.27, so we only need to show the updated S here.

We look at an instance of x with respect to the state S and that entails the transition t, for example $x \approx 5$. The update set E_t of t does not mention the variable x, hence, if we look at what the transition means with respect to the instance, $x \approx 5$ should be present in the updated state S'.

We now give a full example of a cyclic behavior of a PIDL+ specification.

Example 5.42. Consider the following admissible specification $\mathfrak{S}_+$:

$$
\begin{aligned}
S_I &= \{A\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x \geq 0, x \leq 100, \\
&\qquad y \geq 0, x \leq 100\} \\
T_U &= \{A \rightsquigarrow_{u_1} \{x \geq 0, x \leq 100\}\} \\
T_R &= \{x \geq 0 \wedge x \leq 60 \rightsquigarrow_{r_1} \{B, y > 30, y < 50\}, \\
&\qquad x \geq 20 \wedge x \leq 50 \wedge B \rightsquigarrow_{r_2} \{\neg A\} \\
&\qquad x \geq 30 \rightsquigarrow_{r_3} \{A\}\}
\end{aligned}
$$

The initial state (S_I, U_I) is rule terminal, so we can apply user transition u_1 with $\gamma_1 = ()$.

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

where

$$
\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \triangleleft_U (E_{u_1}, \gamma_1) \\
&= (\{A\}, \emptyset) \triangleleft_U (\{x \geq 0, x \leq 100\}, ()) \\
&= (\{A, x \geq 0, x \leq 100\}, \{x\}).
\end{aligned}
$$

The rule transitions given in the specification can be applied to state (S_1, U_1). We show the transitions resulting from applying r_1, r_2, and r_3 in that order. First, rule transition r_1 with $\gamma_1 = ([0, 60])$ gives us

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

where

$$
\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \triangleleft_R (E_{r_1}, \gamma_2) \\
&= (\{A, x \geq 0, x \leq 100\}, \{x\}) \triangleleft_R (\{B, y > 30, y < 50\}, ([0, 60])) \\
&= (\{A, B, x \geq 0, x \leq 60, y > 30, y < 50\}, \{x\}).
\end{aligned}
$$

Applying r_2 with $\gamma_3 = ([20, 50])$ to (S_2, U_2) yields

$$(S_2, U_2) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3)$$

with

$$
\begin{aligned}
(S_3, U_3) &= (S_2, U_2) \triangleleft_R (E_{r_2}, \gamma_3) \\
&= (\{A, B, x \geq 0, x \leq 60, y > 30, y < 50\}, \{x\}) \triangleleft_R (\{\neg A\}, ([20, 50])) \\
&= (\{\neg A, B, x \geq 20, x \leq 50, y > 30, y < 50\}, \{x\}).
\end{aligned}
$$

Finally, rule transition r_3 with $\gamma_4 = ([30, 50])$ can be applied to (S_3, U_4):

$$(S_3, U_3) \rightarrow_{(r_3,\gamma_4)} (S_4, U_4)$$

where

$$\begin{aligned}(S_4, U_4) &= (S_3, U_3) \lhd_R (E_{r_3}, \gamma_4)\\ &= (\{\neg A, B, x \geq 20, x \leq 50, y > 30, y < 50\}, \{x\}) \lhd_R (\{A\}, ([30, 50]))\\ &= (\{A, B, x \geq 30, x \leq 50, y > 30, y < 50\}, \{x\}).\end{aligned}$$

The path resulting from the transitions represents the sequence

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3) \rightarrow_{(r_3,\gamma_4)} (S_4, U_4).$$

We can find an instantiation of the states so that a cycle is established. One example is:

$$\sigma_2 = \sigma_3 = \sigma_4 = \{x \mapsto 36\},$$

with respect to the states S_2, S_3, and S_4. Indeed, applying the substitution to the states gives us the cyclic segment

$$S_2\sigma_2 \rightarrow S_3\sigma_3 \rightarrow S_4\sigma_4 =$$

$$\{A, B, x \approx 36, y > 30, y < 50\} \rightarrow \{\neg A, B, x \approx 36, y > 30, y < 50\} \rightarrow$$

$$\{A, B, x \approx 36, y > 30, y < 50\},$$

satisfying the requirements for cyclicity stated above.

The observations made about possible types of cycles for PIDL in Section 4.4 can be carried over in the case of PIDL+: A cycle can consist of user and rule transitions. A user "reset scheme" may possibly be in place if there is a user transition from the cycle leading to a path away from the cycle, and a rule transition exiting the cycle may lead to another rule-terminal selection or to another cycle. Cycles that only have rule transitions usually indicate inconsistency in the corresponding configuration systems.

Confluence. Like cycles, the notion of confluence in PIDL+ needs a more involved adaptation of what has been defined as confluence in PIDL. Confluence of PIDL+ specifications is determined by the user instances of the affected states. Again, we consider two kinds of confluence.

An admissible PIDL+ specification $\mathfrak{S}_+$ is *rule confluent* if for each state $(S, U) \in \mathcal{V}_{\mathfrak{S}+}$ and for each substitution $\sigma : U \rightarrow \mathbb{Z}$ with $(S \cup \mathsf{C})\sigma$ being satisfiable the following holds: If there is a sequence of rule transitions

$$(S_0, U_0) \rightarrow_{(r_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_2,\gamma_2)} \cdots \rightarrow_{(r_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(r_n,\gamma_n)} (S_n, U_n)$$

and a sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n$$

of substitutions $\sigma_i : U_i \rightarrow \mathbb{Z}$ for $i = 0, \ldots, n$ where

(i) $(S_0, U_0) = (S, U)$,

(ii) $\sigma_0 = \sigma$,

(iii) $(S_i \cup \mathsf{C})\sigma_i$ is satisfiable for $i = 0, \ldots, n$,

(iv) for all $i = 0, \ldots, n-1$ it holds that if $U_i = \{x_1, \ldots, x_m\}$, then $x_j\sigma_i \in \gamma_{i+1}(j)$ for all $j = 1, \ldots, m$,

(v) $x\sigma_i = x\sigma_{i+1}$ if $x \notin \mathit{varsl}(E_{r_{i+1}})$ for all $i = 0, \ldots, n-1$, and

(vi) if $U_n = \{x_1, \ldots, x_l\}$, then $x_j\sigma_n \in \gamma(j)$ for all $j = 1, \ldots, l$, where γ is the rule-terminal base of (S_n, U_n), or if $U_n = \emptyset$, then () is the rule-terminal base of (S_n, U_n),

then the state (S_n, U_n) is unique.

Intuitively, we have rule confluence if from any user instance of any state we can reach only reach one rule-terminal user instance by using only rule transitions. We go through the points of the definition. The first two, (i) and (ii), simply state that the sequence of rule transitions starts with the state (S, U) and the substitution sequence starts with σ. Point (iii) makes sure the substitutions of the sequence are allowed ones that do not break the bounds given by the state and the constraints. Requirements (iv) and (v) are known from the definition of cyclicity in PIDL+. They say that instances must be in the corresponding selections entailing the transitions, and instances do not change if not overwritten by rule transition updates. Point (vi) finally states that the path mentioned above must end in a state with a rule-terminal base. There must not be another state with such properties that can be reached from the instance represented by σ and (S, U), otherwise the specification is not rule confluent.

In order to define user confluence for PIDL+ specifications, we need the notion of *user update instances*. Let a sequence of transitions

$$(S_0, U_0) \rightarrow_{(t_1,\gamma_1)} (S_1, U_1) \rightarrow_{(t_2,\gamma_2)} \cdots \rightarrow_{(t_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(t_n,\gamma_n)} (S_n, U_n)$$

and associated with it a sequence of substitutions

$$\sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n$$

with $(S_i \cup \mathsf{C})\sigma_i$ being satisfiable for $i = 0, \ldots, n$ be given. The user update instances M_U is a set of pairs

$$(u_i, \sigma_i|_{\mathit{varsl}(E_{t_i})})$$

where such a pair is contained in M_U whenever $(S_{i-1}, U_{i-1}) \rightarrow_{(u_i,\gamma_i)} (S_i, U_i)$ and σ_i are a user transition and a substitution contained in the above sequences. The substitution $\sigma_i|_{\mathit{varsl}(E_{t_i})}$ tells us which instance of the user variables in the user transition's update is contained in the substitution σ_i. This is needed to identify the instances of user transitions out of the aggregating states, and thus express user confluence in the context of PIDL+.

An admissible PIDL+ specification $\mathfrak{S}_+$ is *user confluent* if for each substitution $\sigma : U_I \rightarrow \mathbb{Z}$ with $(S_I \cup \mathsf{C})\sigma$ satisfiable and for each sequence of transitions

$$(S_0, U_0) \rightarrow_{(t_1,\gamma_1)} (S_1, U_1) \rightarrow_{(t_2,\gamma_2)} \cdots \rightarrow_{(t_{n-1},\gamma_{n-1})} (S_{n-1}, U_{n-1}) \rightarrow_{(t_n,\gamma_n)} (S_n, U_n)$$

and each sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n$$

of substitutions $\sigma_i : U_i \rightarrow \mathbb{Z}$ for $i = 0, \ldots, n$, where

(i) $(S_0, U_0) = (S_I, U_I)$,

(ii) $\sigma_0 = \sigma$,

(iii) $(S_i \cup \mathsf{C})\sigma_i$ is satisfiable for $i = 0, \ldots, n$,

(iv) for all $i = 0, \ldots, n-1$ it holds that if $U_i = \{x_1, \ldots, x_m\}$, then $x_j\sigma_i \in \gamma_{i+1}(j)$ for all $j = 1, \ldots, m$,

(v) $x\sigma_i = x\sigma_{i+1}$ if $x \notin \mathit{varsl}(E_{t_{i+1}})$ for all $i = 0, \ldots, n-1$, and

(vi) if $U_n = \{x_1, \ldots, x_l\}$, then $x_j\sigma_n \in \gamma(j)$ for all $j = 1, \ldots, l$, where γ is the rule-terminal base of (S_n, U_n), or if $U_n = \emptyset$, then $()$ is the rule-terminal base of (S_n, U_n),

the following holds: If there is another state (S', U') with a rule-terminal base reachable from (S_I, U_I) and σ in the same way as above using the same set of user transitions and the same set of user update instances occurring in the above sequence, then $(S', U') = (S, U)$.

Points (i) to (vi) are analogous to rule confluence, except that we now consider rule and user transitions. The core statement that defines user confluence is that if the same user transitions are applied on different paths starting from the initial state and a certain instance of it, we end up with the same state that has a rule-terminal base.

Example 5.43. We show small examples illustrating specifications that satisfy and violate confluence.

Consider the following admissible specification $\mathfrak{A}$, which is rule-confluent:

$$
\begin{aligned}
S_I &= \{A\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x \geq 0, x \leq 100\} \\
T_U &= \{A \rightsquigarrow_{u_1} \{x \geq 0, x \leq 100\}\} \\
T_R &= \{x \geq 0 \wedge x \leq 50 \wedge A \rightsquigarrow_{r_1} \{B\}, \\
&\qquad x \geq 51 \wedge x \leq 100 \wedge A \rightsquigarrow_{r_2} \{C\}\}
\end{aligned}
$$

We observe the following user transition from the initial state, with u_1 and $\gamma_1 = ()$:

$$(S_I, U_I) \rightarrow_{(u_1, \gamma_1)} (S_1, U_1)$$

where

$$
\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\
&= (\{A\}, \emptyset) \lhd_U (\{x \geq 0, x \leq 100\}, ()) \\
&= (\{A, x \geq 0, x \leq 100\}, \{x\}).
\end{aligned}
$$

Rule transition r_1 can be applied to (S_1, U_1) with $\gamma_2 = ([0, 50])$:

$$(S_1, U_1) \rightarrow_{(r_1, \gamma_2)} (S_2, U_2)$$

with

$$
\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\
&= (\{A, x \geq 0, x \leq 100\}, \{x\}) \lhd_R (\{B\}, ([0, 50])) \\
&= (\{A, B, x \geq 0, x \leq 50\}, \{x\}).
\end{aligned}
$$

It is also possible to apply rule transition r_2 to (S_1, U_1) with $\gamma_3 = ([51, 100])$:

$$(S_1, U_1) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3)$$

where

$$\begin{aligned}(S_3, U_3) &= (S_1, U_1) \lhd_R (E_{r_2}, \gamma_3) \\ &= (\{A, x \geq 0, x \leq 100\}, \{x\}) \lhd_R (\{C\}, ([51, 100])) \\ &= (\{A, C, x \geq 51, x \leq 100\}, \{x\}).\end{aligned}$$

There are no more transitions. The paths so far are

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

and

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3).$$

Thus, from the initial state paths lead to two different states with rule-terminal bases. However, from each user instance represented by (S_1, U_1), the first state in which user variables appear, there is only one rule-terminal state reachable by applying rule transitions. Each instance of x that is less than or equal to 50 leads to one instance in S_2, and each instance of x greater than or equal to 51 leads to exactly one instance in S_3, making $\mathfrak{A}$ rule confluent.

We continue with the admissible specification $\mathfrak{B}$:

$$\begin{aligned}S_I &= \{A, \neg B\} \\ U_I &= \emptyset \\ \mathsf{C} &= \{x \geq 0, x \leq 100\} \\ T_U &= \{A \rightsquigarrow_{u_1} \{x \geq 20, x \leq 80\}\} \\ T_R &= \{x \geq 20 \wedge x \leq 30 \wedge A \rightsquigarrow_{r_1} \{B\}, \\ &\quad\; x \geq 25 \wedge \neg B \rightsquigarrow_{r_2} \{C, y \approx 75\}\}\end{aligned}$$

The state (S_I, U_I) is rule terminal with respect to the selection $\gamma_1 = ()$ and user transition u_1,

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

with

$$\begin{aligned}(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\ &= (\{A, \neg B\}, \emptyset) \lhd_U (\{x \geq 20, x \leq 80\}, ()) \\ &= (\{A, \neg B, x \geq 20, x \leq 80\}, \{x\}).\end{aligned}$$

There is a rule transition from (S_1, U_1) using r_1 and $\gamma_2 = ([20, 30])$,

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

where

$$\begin{aligned}(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\ &= (\{A, \neg B, x \geq 20, x \leq 80\}, \{x\}) \lhd_R (\{B\}, ([20, 30])) \\ &= (\{A, B, x \geq 20, x \leq 30\}, \{x\}).\end{aligned}$$

We can also apply rule transition r_2 to (S_1, U_1) with $\gamma_3 = ([25, 80])$,

$$(S_1, U_1) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3)$$

with

$$\begin{aligned}(S_3, U_3) &= (S_1, U_1) \triangleleft_R (E_{r_2}, \gamma_3) \\ &= (\{A, \neg B, x \geq 20, x \leq 80\}, \{x\}) \triangleleft_R (\{C, y \approx 75\}, ([25, 80])) \\ &= (\{A, \neg B, C, x \geq 25, x \leq 80, y \approx 75\}, \{x\}).\end{aligned}$$

The resulting state (S_3, U_3) is followed by a rule transition to (S_4, U_4) by using r_1 and $\gamma_4 = ([25, 30])$,

$$(S_2, U_2) \rightarrow_{(r_1,\gamma_4)} (S_3, U_3)$$

with

$$\begin{aligned}(S_4, U_4) &= (S_3, U_3) \triangleleft_R (E_{r_1}, \gamma_4) \\ &= (\{A, \neg B, x \geq 20, x \leq 80\}, \{x\}) \triangleleft_R (\{B\}, ([25, 30])) \\ &= (\{A, B, C, x \geq 25, x \leq 30, y \approx 75\}, \{x\}).\end{aligned}$$

The paths from (S_I, U_I) so far are

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

and

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(r_2,\gamma_3)} (S_3, U_3) \rightarrow_{(r_1,\gamma_4)} (S_4, U_4),$$

with two states with rule-terminal bases emerging, (S_2, U_2) and (S_4, U_4). Consider the substitution

$$\sigma_1 = \{x \mapsto 25\}$$

with respect to (S_1, U_1). With more substitutions $\sigma_2, \sigma_3, \sigma_4$, where

$$\sigma_2 = \sigma_3 = \sigma_4 = \sigma_1,$$

we can construct two paths starting from $S_1\sigma_1$ and ending in two different rule-terminal instances $S_2\sigma_2$ and $S_4\sigma_4$ according to the above definition:

$$S_1\sigma_1 \rightarrow S_2\sigma_2$$

and

$$S_1\sigma_1 \rightarrow S_3\sigma_3 \rightarrow S_4\sigma_4,$$

where

$$S_2 \neq S_4.$$

Hence, $\mathfrak{B}$ is not rule-confluent.

We now demonstrate the concept of user confluence. Assume the admissible specification $\mathfrak{C}$:

$$\begin{aligned}S_I &= \{\neg A, \neg B\} \\ U_I &= \emptyset \\ \mathsf{C} &= \{x_1 \geq 0, x_1 \leq 100, \\ &\qquad x_2 \geq 0, x_2 \leq 100\} \\ T_U &= \{\neg B \rightsquigarrow_{u_1} \{x_1 \geq 0, x_1 \leq 30\}, \\ &\qquad \neg A \rightsquigarrow_{u_2} \{x_2 \geq 40, x_2 \leq 60\}\} \\ T_R &= \{x_1 \geq 0 \wedge x_2 \leq 70 \rightsquigarrow_{r_1} \{A, B\}\}\end{aligned}$$

From (S_I, U_I), user transition u_1 to (S_1, U_1) is possible with $\gamma_1 = ()$,

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

where

$$\begin{aligned}(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\ &= (\{\neg A, \neg B\}, \emptyset) \lhd_U (\{x_1 \geq 0, x_1 \leq 30\}, ()) \\ &= (\{\neg A, \neg B, x_1 \geq 0, x_1 \leq 30\}, \{x_1\}).\end{aligned}$$

There is another user transition from (S_1, U_1) to (S_2, U_2) with u_2 and $\gamma_2 = ([0, 30])$,

$$(S_1, U_1) \rightarrow_{(u_2,\gamma_2)} (S_2, U_2)$$

where

$$\begin{aligned}(S_2, U_2) &= (S_1, U_1) \lhd_U (E_{u_2}, \gamma_2) \\ &= (\{\neg A, \neg B, x_1 \geq 0, x_1 \leq 30\}, \{x_1\}) \lhd_U (\{x_2 \geq 40, x_2 \leq 60\}, ([0, 30])) \\ &= (\{\neg A, \neg B, x_1 \geq 0, x_1 \leq 30, x_2 \geq 40, x_2 \leq 60\}, \{x_1, x_2\}).\end{aligned}$$

We can apply rule transition r_1 to (S_2, U_2) with $\gamma_3 = ([0, 30], [40, 60])$,

$$(S_2, U_2) \rightarrow_{(r_1,\gamma_3)} (S_3, U_3)$$

with

$$\begin{aligned}(S_3, U_3) &= (S_2, U_2) \lhd_R (E_{r_1}, \gamma_3) \\ &= (\{\neg A, \neg B, x_1 \geq 0, x_1 \leq 30, x_2 \geq 40, x_2 \leq 60\}, \{x_1, x_2\}) \lhd_R \\ &\quad (\{A, B\}, ([0, 30], [40, 60])) \\ &= (\{A, B, x_1 \geq 0, x_1 \leq 30, x_2 \geq 40, x_2 \leq 60\}, \{x_1, x_2\}).\end{aligned}$$

Going back to the initial state, we see that user transition u_2 with $\gamma_4 = ()$ is possible from this state, yielding

$$(S_I, U_I) \rightarrow_{(u_2,\gamma_4)} (S_4, U_4)$$

with

$$\begin{aligned}(S_4, U_4) &= (S_I, U_I) \lhd_U (E_{u_2}, \gamma_4) \\ &= (\{\neg A, \neg B\}, \emptyset) \lhd_U (\{x_2 \geq 40, x_2 \leq 60\}, ()) \\ &= (\{\neg A, \neg B, x_2 \geq 40, x_2 \leq 60\}, \{x_2\}).\end{aligned}$$

If we apply user transition u_1 with $\gamma_5 = ([40, 60])$ to (S_4, U_4), the successor state is the existing (S_3, U_3):

$$(S_4, U_4) \rightarrow_{(u_1,\gamma_5)} (S_3, U_3)$$

with

$$\begin{aligned}(S_3, U_3) &= (S_4, U_4) \lhd_U (E_{u_1}, \gamma_5) \\ &= (\{\neg A, \neg B, x_2 \geq 40, x_2 \leq 60\}, \{x_2\}) \lhd_U (\{x_1 \geq 0, x_1 \leq 30\}, ([40, 60])) \\ &= (\{\neg A, \neg B, x_1 \geq 0, x_1 \leq 30, x_2 \geq 40, x_2 \leq 60\}, \{x_1, x_2\}).\end{aligned}$$

The resulting picture from the transitions is given by the paths

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1) \rightarrow_{(u_2,\gamma_2)} (S_2, U_2) \rightarrow_{(r_1,\gamma_3)} (S_3, U_3),$$

and

$$(S_I, U_I) \rightarrow_{(u_2,\gamma_4)} (S_4, U_4) \rightarrow_{(u_1,\gamma_5)} (S_3, U_3).$$

Inspecting the paths makes it clear that every path of user instances containing the same transitions u_1 and u_2 and the same user update instances leads to a unique state that has a rule-terminal base. The specification $\mathfrak{C}$ is therefore user confluent.

The last example specification $\mathfrak{D}$ is given by

$$\begin{aligned}
S_I &= \{\neg A, \neg B\} \\
U_I &= \emptyset \\
\mathsf{C} &= \{x_1 \geq 0, x_1 \leq 100, \\
&\qquad x_2 \geq 0, x_2 \leq 100\} \\
T_U &= \{\neg A \rightsquigarrow_{u_1} \{A, x_1 \geq 0, x_1 \leq 45\}, \\
&\qquad \neg B \rightsquigarrow_{u_2} \{B, x_2 \geq 58, x_2 \leq 90\}\} \\
T_R &= \{x_1 \leq 30 \wedge \neg B \rightsquigarrow_{r_1} \{C\}\}
\end{aligned}$$

There is a user transition from the initial state using u_1 and $\gamma_1 = ()$,

$$(S_I, U_I) \rightarrow_{(u_1,\gamma_1)} (S_1, U_1)$$

with

$$\begin{aligned}
(S_1, U_1) &= (S_I, U_I) \lhd_U (E_{u_1}, \gamma_1) \\
&= (\{\neg A, \neg B\}, \emptyset) \lhd_U (\{A, x_1 \geq 0, x_1 \leq 45\}, ()) \\
&= (\{A, \neg B, x_1 \geq 0, x_1 \leq 45\}, \{x_1\}).
\end{aligned}$$

We can apply rule transition r_1 to (S_1, U_1) with $\gamma_2 = ([0, 30])$,

$$(S_1, U_1) \rightarrow_{(r_1,\gamma_2)} (S_2, U_2)$$

where

$$\begin{aligned}
(S_2, U_2) &= (S_1, U_1) \lhd_R (E_{r_1}, \gamma_2) \\
&= (\{A, \neg B, x_1 \geq 0, x_1 \leq 45\}, \{x_1\}) \lhd_R (\{C\}, ([0, 30])) \\
&= (\{A, \neg B, C, x_1 \geq 0, x_1 \leq 30\}, \{x_1\}).
\end{aligned}$$

The state (S_2, U_2) entails user transition u_2 with selection $\gamma_3 = ([0, 30])$, thus

$$(S_2, U_2) \rightarrow_{(u_2,\gamma_3)} (S_3, U_3)$$

with

$$\begin{aligned}
(S_3, U_3) &= (S_2, U_2) \lhd_U (E_{u_2}, \gamma_3) \\
&= (\{A, \neg B, C, x_1 \geq 0, x_1 \leq 30\}, \{x_1\}) \lhd_U (\{B, x_2 \geq 58, x_2 \leq 90\}, ([0, 30])) \\
&= (\{A, B, C, x_1 \geq 0, x_1 \leq 30, x_2 \geq 58, x_2 \leq 90\}, \{x_1, x_2\}).
\end{aligned}$$

There is a user transition from the initial state using u_2 and $\gamma_4 = ()$,

$$(S_I, U_I) \rightarrow_{(u_2,\gamma_4)} (S_4, U_4)$$

where

$$\begin{aligned}(S_4, U_4) &= (S_I, U_I) \lhd_U (E_{u_2}, \gamma_4) \\ &= (\{\neg A, \neg B\}, \emptyset) \lhd_U (\{B, x_2 \geq 58, x_2 \leq 90\}, ()) \\ &= (\{\neg A, B, x_2 \geq 58, x_2 \leq 90\}, \{x_2\}).\end{aligned}$$

From (S_4, U_4), user transition u_1 is possible with $\gamma_5 = ([58, 90])$,

$$(S_4, U_4) \rightarrow_{(u_1, \gamma_5)} (S_5, U_5)$$

with

$$\begin{aligned}(S_5, U_5) &= (S_4, U_4) \lhd_U (E_{u_1}, \gamma_5) \\ &= (\{\neg A, B, x_2 \geq 58, x_2 \leq 90\}, \{x_2\}) \lhd_U (\{A, x_1 \geq 0, x_1 \leq 45\}, ([58, 90])) \\ &= (\{A, B, x_1 \geq 0, x_1 \leq 45, x_2 \geq 58, x_2 \leq 90\}, \{x_1, x_2\}).\end{aligned}$$

We have the following paths:

$$(S_I, U_I) \rightarrow_{(u_1, \gamma_1)} (S_1, U_1) \rightarrow_{(r_1, \gamma_2)} (S_2, U_2) \rightarrow_{(u_2, \gamma_3)} (S_3, U_3),$$

and

$$(S_I, U_I) \rightarrow_{(u_2, \gamma_4)} (S_4, U_4) \rightarrow_{(u_1, \gamma_5)} (S_5, U_5).$$

Consider now the following user update instances:

$$(u_1, \{x_1 \mapsto 23\}), (u_2, \{x_2 \mapsto 60\}).$$

From the initial state, we can reach (S_3, U_3) with the sequence of substitutions

$$\sigma, \sigma_1, \sigma_2, \sigma_3$$

and their applications

$$S_I\sigma \rightarrow_{u_1} S_1\sigma_1 \rightarrow S_2\sigma_2 \rightarrow_{u_2} S_3\sigma_3,$$

where σ is the empty substitution,

$$\sigma_1 = \sigma_2 = \{x_1 \mapsto 23\}, \text{ and } \sigma_3 = \{x_1 \mapsto 23, x_2 \mapsto 60\}.$$

The same user transitions and user update instances can be used to have another path,

$$S_I\sigma \rightarrow_{u_2} S_4\sigma_4 \rightarrow_{u_1} S_5\sigma_3,$$

where

$$\sigma_4 = \{x_2 \mapsto 60\}.$$

(S_3, U_3) and (S_5, U_5) both have rule-terminal bases as required in the definition. Since they differ from each other, the order in which user transitions happen has an effect on the result, which means $\mathfrak{D}$ is not user confluent.

Theorem 5.44. The state space $\mathcal{V}_{\mathfrak{S}+}$ and transition space $\mathcal{T}_{\mathfrak{S}+}$ of each interpretation of an admissible specification $\mathfrak{S}_+$ are finite sets.

Proof. The states in PIDL+ satisfy the subterm property, as can be seen in the following. A state consists of propositional literals and arithmetic atoms. The propositional literals that can occur in a state are based on the finite set Π given by the specification. Hence, the number of possible propositional literals is finite. Arithmetic atoms appearing in a state can come from

(i) an update set, or

(ii) an atomic representation of a selection.

Update sets are finite and there are finitely many of them, since they are part of user and rule transitions, both finite sets. The set of all possible selections occurring in states is finite because the relevant variables are bounded integers via the constraints C in admissible specifications. All in all, we have that there are only finitely many ways to write a state that can be induced given an admissible specification, so $\mathcal{V}_{\mathfrak{S}+}$ is finite. Due to the finiteness of $\mathcal{V}_{\mathfrak{S}+}$, of the possible selections and of the sets of transitions, $\mathcal{T}_{\mathfrak{S}+}$ is finite as well. □

Theorem 5.45. Let $\mathfrak{S}_+ = (\Pi, X, S_I, U_I, \mathsf{C}, T_U, T_R)$ be an admissible PIDL+ specification. Let b be the length of the largest integer interval that is represented by the bounds with respect to a transition variable,

$$b := max\{d_x - c_x + 1 | x \geq c_x \wedge x \leq d_x \text{ appears in } \mathsf{C}\}.$$

The state space $\mathcal{V}_{\mathfrak{S}+}$ has worst-case space complexity

$$O(3^{|\Pi|} \cdot (1 + |T_U| + |T_R| + (\frac{b^2 + b}{2}))^{|X|} \cdot 2^{|X|}).$$

Proof. A PIDL+ state has the form (S, U), where S is a set of propositional literals and simple atoms, and U is a subset of the set X of arithmetic variables.

In S, a propositional variable P can be present as P, $\neg P$, or not not be present at all. Hence, there are $3^{|\Pi|}$ configurations of propositional literals with respect to S.

Simple atoms $x \circ t$ occurring in S can be associated with their left-hand-side variables x. There are three cases with respect to the appearance of simple atoms $x \circ t$ in S:

(i) There are no simple atoms $x \circ t$ in S.

(ii) There are simple atoms $x \circ t$ in S, where the atoms come from the update sets of the transitions in T_U and T_R. There are at most $|T_U| + |T_R|$ such update sets with respect to x.

(iii) There are simple atoms $x \circ t$ in S, where the atoms are elements from the atomic representations $at(\gamma)$, with γ being the selection that is used in an update operation. This means that the update set of the corresponding transition do not contain atoms with x as a left-hand-side variable, otherwise the update set would take precedence, as defined for the update operators (Definition 5.21 and 5.27) and as is the case in (ii).

Since x is involved in a transition, it is bounded in the constraints by virtue of atoms $x \geq c_x$ and $x \leq d_x$, represented by the integer interval $I_x := [c_x, d_x]$. Every set of simple atoms $x \circ t$ from $at(\gamma)$ represents a subinterval $I \subseteq I_x$ by Definition 5.8. Assume that the size of I_x is n. If the size of I is 1, then there are n ways I could look like. If the size of I is 2, then there are $n - 1$ ways I could look like. This can be continued up to a size of n of I, in which case there is 1 way to form I. To sum up, there are

$$\Sigma_{i=1}^{n} i = \frac{n^2 + n}{2}$$

possibilities to have subintervals of I_x. If we take b, the length of the largest interval represented by the bounds in the constraints, we have at most

$$\frac{b^2 + b}{2}$$

possibilities to have simple atoms in S that come from selections.

Finally, there are $2^{|X|}$ possibilities for the user variables U, which is a subset of X. All in all, the number of PIDL+ states in $\mathcal{V}_{\mathfrak{S}+}$ is bounded by

$$O(3^{|\Pi|} \cdot (1 + (|T_U| + |T_R|) + (\frac{b^2 + b}{2}))^{|X|} \cdot 2^{|X|}).$$

□

The results make it possible for PIDL+ to, in principle, embed configuration systems such as LEEGOO, which also consider arithmetic constraints. Non-arithmetic parameters and constraints are modeled in the same way as done in PIDL, since propositional variables and formulas stay an essential part of PIDL+. The arithmetic terms and their integration in PIDL+ specifications have been chosen and done in such a way that the arithmetic part of LEEGOO is sufficiently represented, paving the way for the formal verification of properties of LEEGOO models.

Chapter 6

Conclusions

We have presented PIDL and PIDL+, two decidable logics that have been developed to formalize rule-based configuration systems in a comprehensive way. The semantics is defined so that the whole configuration process can be represented, forming a basis for useful verification of important properties. Rule-terminal states correspond to the fixpoint behavior of configuration systems such as DOPLER (Dhungana and Grünbacher, 2008; Dhungana et al., 2011) and LEEGOO (Struck, 2012), and the transition update scheme of replaceable literals in states gives flexibility in the modeling of those systems. While PIDL deals with purely propositional configuration, PIDL+ integrates arithmetic in the framework, with a compact representation of instances by using bounds and selections. For both logics, we have provided calculi and algorithms to show they are amenable to decision procedures connected to investigating important consistency properties.

With this thesis demonstrating first experiments in the case of PIDL, it would be desirable to be able to have more real-world configuration data that can be used for further experimentation. Corresponding further arrangements would likely have to be made with partners in industry. Furthermore, there is always room for extensions to expressivity and features covered by the logics. The present work is motivated by DOPLER and LEEGOO and refers to a particular state of the systems. It would be interesting to see whether additional developments in the area of configuration systems in general that have happened since can be represented by modifying and extending the logics accordingly. Nevertheless, PIDL and PIDL+ show how a formalization of relevant elements of interactive product configuration can look like and how it opens up possibilities for the automated verification of those systems.

Bibliography

Alur, R., Henzinger, T. A., and Kupferman, O. (2002). Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713.

Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency restoration and explanations in dynamic csps application to configuration. *Artificial Intelligence*, 135(1-2):199–234.

Bachmair, L. and Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247.

Bachmair, L. and Ganzinger, H. (2001). Resolution theorem proving. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier.

Balbiani, P., Herzig, A., and Troquard, N. (2013). 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.

Behjati, R. and Nejati, S. (2014). Interactive configuration verification using constraint programming. Lyon, France.

Belardinelli, F. and Herzig, A. (2016). On logics of strategic ability based on propositional control. In Kambhampati, S., editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 95–101. IJCAI/AAAI Press.

Benavides, D., Segura, S., and Cortés, A. R. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

Bonzon, E., Lagasquie-Schiex, M., Lang, J., and Zanuttini, B. (2006). Boolean games revisited. In Brewka, G., Coradeschi, S., Perini, A., and Traverso, P., editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 265–269. IOS Press.

Booch, G., Rumbaugh, J. E., and Jacobson, I. (2005). *The unified modeling language user guide - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley.

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

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, MA, USA.

Classen, A., Heymans, P., Schobbens, P., Legay, A., and Raskin, J. (2010). 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.

Czarnecki, K. and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness OCL constraints. In Jarzabek, S., Schmidt, D. C., and Veldhuizen, T. L., editors, *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, pages 211–220. ACM.

Czarnecki, K. and Wasowski, A. (2007). 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.

Davis, S. M. (1989). From "future perfect": Mass customizing. *Planning Review*, 17(2):16–21.

de Moura, L. and Bjørner, N. (2008). *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg.

Dhungana, D. and Grünbacher, P. (2008). Understanding decision-oriented variability modelling. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 233–242. Lero Int. Science Centre, University of Limerick, Ireland.

Dhungana, D., Grünbacher, P., and Rabiser, R. (2011). The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1):77–114.

Dhungana, D., Tang, C. H., Weidenbach, C., and Wischnewski, P. (2013). 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.

Felfernig, A. (2007). Standardized configuration knowledge representations as technological foundation for mass customization. *IEEE Trans. Engineering Management*, 54(1):41–56.

Felfernig, A., Friedrich, G., and Jannach, D. (2000). Uml as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):449–469.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., and Zanker, M. (2003). Configuration knowledge representations for semantic web applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):31–50.

Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J. (2014). *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition.

Fischer, M. J. and Ladner, R. E. (1979). Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., and Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68.

Fränzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. (2007). Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling, and Computation*, 1:209–236.

Harrenstein, P., van der Hoek, W., Meyer, J.-J. C., and Witteveen, C. (2001). Boolean games. In *Proceedings of the Eight Conference on Theoretical Aspects of Rationality and Knowledge*, pages 287–298.

Herzig, A., Lang, J., and Marquis, P. (2013). Propositional update operators based on formula/literal dependence. *ACM Transactions on Computational Logic (TOCL)*, 14(3):24.

Herzig, A., Lorini, E., Moisan, F., and Troquard, N. (2011). A dynamic logic of normative systems. In Walsh, T., editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 228–233. IJCAI/AAAI.

Jackson, P. (1999). *Introduction to Expert Systems, 3rd Edition*. Addison-Wesley.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Lauenroth, K., Pohl, K., and Toehning, S. (2009). 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*, pages 269–280. IEEE Computer Society.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):383–397.

Marek, V. W. and Truszczyński, M. (1999). *Stable Models and an Alternative Logic Programming Paradigm*, pages 375–398. Springer Berlin Heidelberg, Berlin, Heidelberg.

McDermott, J. P. and Bachant, J. (1984). R1 revisited: Four years in the trenches. *AI Magazine*, 5(3):21–32.

McGuinness, D. L. (2003). Configuration. In Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 388–405. Cambridge University Press.

Mendonça, M., Wasowski, A., and Czarnecki, K. (2009). SAT-based analysis of feature models is easy. In Muthig, D. and McGregor, J. D., editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 231–240. ACM.

Niemelä, I., Simons, P., and Soininen, T. (1999). Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '99, pages 317–331, London, UK, UK. Springer-Verlag.

Osborne, M. J. and Rubinstein, A. (1994). *A Course in Game Theory*. The MIT Press, Cambridge, USA. electronic edition.

Pine, B. (1993). *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press.

Pnueli, A. (1977). 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.

Rosa, M. L., van der Aalst, W. M. P., Dumas, M., and ter Hofstede, A. H. M. (2009). Questionnaire-based variability modeling for system configuration. *Software and System Modeling*, 8(2):251–274.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. international edition)*. Pearson Education.

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

Sinz, C. (2004). *Verifikation regelbasierter Konfigurationssysteme*. PhD thesis, Eberhard Karls University of Tübingen.

Soininen, T. and Niemelä, I. (1999). Developing a declarative rule language for applications in product configuration. In Gupta, G., editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer.

Struck, D. (2012). Leegoo builder applications. `http://www.eas-solutions.de`. Accessed: 1 February 2017.

Suda, M. (2011). Single plan reasoning. unpublished.

Tang, C. H. and Weidenbach, C. (2016). A dynamic logic for configuration. In Benzmüller, C. and Otten, J., editors, *Proceedings of the 2nd International Workshop Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2016) affiliated with the International Joint Conference on Automated Reasoning (IJCAR 2016)., Coimbra, Portugal, July 1, 2016.*, volume 1770 of *CEUR Workshop Proceedings*, pages 36–50. CEUR-WS.org.

Tsang, E. P. K. (1993). *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press.

van der Hoek, W. and Wooldridge, M. (2005). On the logic of cooperation and propositional control. *Artificial Intelligence*, 164(1-2):81–119.

Winslett, M. (1990). *Updating Logical Databases*. Cambridge University Press, New York, NY, USA.

Zhang, W., Yan, H., Zhao, H., and Jin, Z. (2008). A BDD-based approach to verifying clone-enabled feature models' constraints and customization. In Mei, H., editor, *High Confidence Software Reuse in Large Systems, 10th International Conference on Software Reuse, ICSR 2008, Beijing, China, May 25-29, 2008, Proceedings*, volume 5030 of *Lecture Notes in Computer Science*, pages 186–199. Springer.