From Type Inference to Configuration

Morten Heine Sørensen[1⋆] and Jens Peter Secher[2]

[1] IT-Practice A/S,
Kronprinsessegade 54, 5.,
DK-1306 Copenhagen K, Denmark,
mhs@it-practice.dk
[2] Department of Computer Science, University of Copenhagen,
Universitetsparken 1,
DK-2100 Copenhagen Ø, Denmark,
jpsecher@diku.dk

Abstract. A *product line* is a set of products and features with constraints on which subsets are available. Numerous *configurators* have been made available by product line vendors on the internet, in which procurers can experiment with the different options, e.g. how the selection of one product or feature entails or precludes the selection of another product or feature.
We explore an approach to configuration inspired by type inference technology. The main contributions of the paper are a formalization of the configuration problem that includes aspects related to the interactive dialogue between the user and the system, a result stating that the configuration problem thus formalized has at least exponential complexity, and some techniques for computing approximate solutions more efficiently. While a substantial number of papers precede the present one in formalizing configuration as a constraint satisfaction problem, few address the aspects concerning interactivity between the user and the system.

1 Introduction

A *product line* is a set of products and features with constraints on which subsets are available. For example, a company may sell a number of different computers (lap-tops, desk-tops, etc.) with different storage device, memory size, etc., but some combinations may be unavailable. For instance, DVD drive and CD-ROM may be mutually exclusive in lap-top models.

Another product line example is computer *software*. What in the beginning is a single software tool often ends up as a family of tools that can be composed in various ways with some constraints. For instance, the software may exist for various platforms (Unix, Windows NT, etc.), but some packages are only available for some platforms.

A third product line example is cars. A car usually comes in various models, where each model can be configured in different ways. For instance, the station

⋆ The work was carried out while the author was employed by Terma A/S.

T.Æ. Mogensen et al. (Eds.): The Essence of Computation, LNCS 2566, pp. 436–471, 2002.

wagon does not come with interior trunk release, the three-door model does not come with automatic transmission, etc.

It is natural for the vendor to make a system available to the procurer in which he can experiment with the different options, e.g. how the selection of one product or feature entails or precludes the selection of another product or feature. Indeed, such systems have appeared in online-shops on the internet—see, for instance, the home pages of Dell, Compaq, and Ford Motor Company. In fact, there is an emerging market for tools devoted to building such systems—one example is ILOG's configurator.[1] In general we shall refer to such tools as *configurators*.

A related idea concerns the generation of requirements to a system that is to be developed, where the system in advance is known to respect a model with constraints on how the product can be built. That is, although the product does not exist yet, there is a model that expresses constraints pertaining to the product. In this case the user produces a requirement specification instead of a procurement order.

The PROMIS[2] project [8] was concerned with the development of a prototype of such a tool, in which the procurement officer of a military organisation can experiment with the constraints pertaining to so-called C^3I-systems.[3] The idea is that a so-called *reference model* is available, containing a specification of all components that can be present in a C^3I-system along with the constraints among these components. The procurement officer first makes a consistent choice of components using the tool. Based on this selection, the tool then generates a requirements specification describing the desired C^3I-system.

The motivation for the present paper came from two challenges that were identified, but left open in the PROMIS project:

1. Only constraints of very simple forms were considered, namely *aggregation* of entities in the usual object-oriented sense, *specialization* (also in the object-oriented sense), and *implication* among selection of entities. In particular, mutual exclusion of selection of entities was omitted. In the present paper, a more general approach accommodating *any* set of logical constraints is considered.
2. It was not possible to *deselect* an entity—selection could only be *undone* (back-tracked), but there was no way to deselect an entity selected 100 clicks ago without also undoing all the intermediate selections. In the present paper deselections are treated as analogous to selections making them first-class citizens of the approach.

More broadly, in this paper we explore an approach to configurators inspired by type inference technology. Our fundamental idea, strongly inspired by the AnnoDomini project [12, 13], is to view the product line constraints or reference

[1] We do not mean to suggest that any of the mentioned home pages or tools use the techniques described in the present paper.
[2] PROMIS is short for PRocurement Officer Reference Model Information System.
[3] C^3I is short for Command, Control, Communication, and Intelligence.

model as a program whose types indicate the selection or de-selection of components. In other words, the user interacts with the system in terms of type-based specification.

Working within this framework provides a natural approach to several problems. For instance, the propagation of selections according to the constraints of the model becomes a type inference problem, the handling of inconsistent desires of the user amounts to resolution of type errors, and the generation of a procurement order or requirement specification is a type-directed translation. Having said this, we should admit that the main advantage of the approach is not the derivation of any new efficient constraint algorithms directly derived from type inference, but rather some insights concerning what type of constraint propagation might be desirable in some configurators and concerning how a user might interact with such constraint propagation algorithms.

In the remainder of the paper we shall be concerned exclusively with procurement from product lines, and leave the connection to the generation of requirements satisfying a reference model implicit. This is not to say that the two problems are identical. There *are* differences between selecting from a product line to produce a purchase specification on the one hand and generating a requirement specification according to a reference model on the other hand. For instance, a requirement specification may be vague, since it is usually the starting point of a discussion with the vendor. In contrast, a purchase specification for an e-shop normally uniquely identifies the product and associated price. Nevertheless, in the remainder of the paper, such differences will be ignored, and we will focus entirely on selection from a product line.

The remainder of the paper is organized in three parts. The first part (Section 2) contains a small *analysis* of the problem to be solved and introduces some concepts relevant for reasoning about product lines—in particular, what objects product line constraints pertain to, and what functionality these constraints must accommodate. The second part (Section 3) presents a *formalization* of the problem by introducing the syntax and semantics for constraints as well a notion of optimal constraint propagation. The third part (Section 4) then presents several *solutions* to the problem in terms of algorithms for propagating constraints, based on type inference, program transformation, and binary decision diagrams. We also outline a configuration system based on these algorithms. The paper ends with a review of related work (Section 5) and some ideas for future work (Section 6).

2 Analysis of Product Line Concepts

This section analyzes some concepts relevant for reasoning about product lines. The two first subsections present the building blocks for the constraints that we shall consider; that is, what do the constraints pertain *to?* What do they constrain? The third subsection sets the limits of our scope, and the last subsection finally discusses how the user can interact with our constraints.

Our inspiration for the analysis comes from the PROMIS project [8] mentioned above as well as various online-shops that offer configuration, see e.g. [30]. There are models significantly more complex than what we shall propose, but our working hypothesis is that there are cases, particularly in online-shops, where this complexity is neither required nor desired.

2.1 Product Lines

What we buy from a product line are *entities*. For instance, when we buy a car, the entities are the engine (size), the (number of) doors, the transmission (type), etc. In a computer, they are the different hardware components, e.g. storage device. In software they are the different software components.

We configure the product we are buying by *selecting* and *deselecting* entities. For instance, when we buy a computer, we select or deselect the DVD entity, and we select or deselect the CD-ROM entity. A specification, for each entity, whether it is selected, deselected, or none of these, will be called a *configuration*. A configuration in which every entity is either selected or deselected (and hence not unspecified) is called *total*.

When we select from a product line, we cannot choose whatever we like: our selection is *constrained* by the vendor. We have already mentioned the possible mutual exclusion of DVD and CD-ROM drive in a lap-top. The constraints may state such properties as mutual exclusion of selection of two entities (if one is selected, the other cannot be selected), implication from selection of one entity to another (if one is selected, the other must be selected too), etc. In general, any logical relation ("if-then," "and," "or," etc.) between selection of any number of entities is possible. If a configuration respects all the constraints it is called *consistent*. The final order from a user to an online-shop is always a total, consistent configuration.

Thus, the setup is as follows:

- The user is confronted with entities.
- The user may select or deselect an entity.
- When the user selects or deselects an entity, constraints pertaining to entities are checked or propagated.

2.2 Views on Product Lines

One can discern at least two different views of the above product lines and constraints. The *domain* view expresses the fundamental constraints of the product line. For instance, it does not make sense to furnish a station wagon with an interior trunk release.

The *sales* view is how the marketing manager sees the product line. This view can be seen as a refinement of the domain view—the marketing manager cannot change the fundamental rules concerning how a car can be composed. However, the marketing manager will add entities and constraints of the following types among others:

- *Packages:* several related entities may be grouped in a package. For instance, a car may be configured with a sports package that selects a powerful engine, rear spoiler, extra front lights, etc.
- *Models:* to make the possible selections simpler for the user, and to standardize the actual construction of the purchased items, models are employed. For instance, computers usually come in various models at various prices. Cars also usually come in various models (Ford Focus, Ford Mustang) and submodels (Sedan, Sedan 3-door, station wagon).
- *Additional constraints:* from a marketing point of view it could be desirable e.g. to exclude fancy features in low-price models. For instance, leather interior might be precluded from cars that include the family package, despite the fact that it is possible for the vendor to furnish all models with leather interior.

The domain view is materialized simply by building a model with the correct entities and constraints. The sales view is then built on top of that by adding:

- Entities corresponding to packages and models.
- Constraints reflecting the aggregation relationship between a package or a model and its constituent entities, i.e. constraints expressing that selection of, say, a model entails selection of all its constituent entities.
- Constraints reflecting the marketing manager's additional constraints.

Thus, both views are accommodated by our simple set-up.

2.3 What the Analysis Does Not Cover

An entity can have a number of *features*. These are not entities but rather properties of the entities. For instance, a CPU has a frequency. The frequency itself is not a an entity (at least not a physical one), but rather a property of the CPU. Similarly, the CPU probably has a price. Each feature has a *value*. For instance, the CPU frequency is some number, as is the CPU price.

Features may also pertain to a group of entities, rather than a single entity. For instance, the acceleration of a car is not merely a property of the engine: a station wagon probably has a smaller accelaration than a sedan model with the same engine. Similarly, the user is most likely interested in the price of the car he has configured, and the total price is not a property of any of the selected entities.

In this paper we will not consider features or constraints pertaining to them. Of course, in an online-shop, there must be an approach to calculating the price(!), but this can be handled in an ad-hoc manner, by letting each entity have a price, and letting the total price be just the sum of all prices for selected entities. In principle, a feature with n different possible values can be encoded as n entities with a constraint expressing that exactly one of the entities must be selected, but this encoding may not be practical if n is very large; however, our working assumption is that there are online-shops whose configuration problem can be implemented without features and values.

The constraints discussed above are those imposed *on* the user *by* the vendor: if an entity is selected, another is precluded; and if a certain selected entity's feature has a certain value, a certain other selected entity's feature must have a certain other value. The converse type is also conceivable: the user might have constraints. For instance, "If I get a station wagon, I want a 2.0 engine" or "I want the cheapest possible station wagon with an acceleration from 0 to 100 km/h better than 20 seconds."

In this paper, we will not deal explicitly with user constraints. As far as we are concerned, these constraints must be satisfied by the user by making appropriate choices among the entities, respecting the constraints developed by the vendor.

There are a number of other model concepts that we also preclude from our analysis. For instance, we do not consider inheritance or other relations between entities—except the relations expressed by the logical constraints pertaining to possible selection and deselection.

2.4 Selecting from Product Lines

Whereas the preceding subsections have been concerned with *what* the user can select, the precent one addresses the *how*. How does the user select or deselect entities? An important aspect of this is the *order* in which the user is asked to select various entities. There are several options, e.g.:

- *Sequential selection and deselection of desired components.* First the user selects or deselects the first entity (in some given order), then the next and so on. For instance, first the user selects model, then submodel, etc.
- *Arbitrary selection and deselection among all components.* In this approach the user can freely select or deselect any entity in any order, respecting the constraints along the way, of course.
- *Combinations.* The above can be combined. For instance, first the user selects model, then submodel, then he selects among the available packages, and finally he has a chance to review the whole list of selected entities and change any selection, respecting the constraints.

We believe that the choice among these should not be a property of the constraint engine, but rather of a presentation layer built on top of that, following the usual *Model-View-Controller* pattern [17]. It should be possible to use different selection orders with the same underlying constraint engine. Thus, in the present paper, we will only be concerned with how the underlying constraint engine works; any of the above orders can be used by building a presentation layer above the constraint engine, guiding the user through the different choices in the desired order.

Each selection can also be presented in various ways, e.g. with

- *check boxes*: choosing an option or not.
- *radio buttons*: choosing exactly one of the options from a list.
- *list boxes*: choosing zero or more of the options from a list.

For instance, a mandatory choice between a number of mutually exclusive entities can be presented with a radio button—this way, the mutual exclusion will be enforced in the presentation layer. Moreover it is presented in a form likely to be familiar to the user.

But again, the choice between these different types of presentation should not be a property of the constraint engine. That is, the engine should not *rely* on the presentation layer working in one way or another. The presentation layer should collect input in a structured form (e.g. a radio button) and submit it to the engine in a flat form (selection, deselection, or unspecified status for every entity).

It is important to understand that even a check box can be used for different forms of choice:

1. checked means "select entity," unchecked means "unspecified whether to select entity."
2. checked means "select entity," unchecked means "deselect entity."
3. checked means "deselect entity," unchecked means "unspecified whether to select entity."
4. checked means "deselect entity," unchecked means "select entity."

In other words, the constraint engine can receive three types of user input concerning an entity: *select*, *deselect*, and *unspecified*, and the two options of a check box can be used by the presentation layer to indicate any combination of two of these.

Concerning the amount of work done by the system based on a selection, there are at least the following three types of solutions:

- *checking:* The system verifies that the user's current selection is consistent and reports errors if there are selections or deselections that contradict the set of constraints.
- *propagation:* The system, in addition to verifying the consistency of the selections and deselections, also infers all consequences of these, thereby selecting and deselecting a number of other entities. The system also reports inconsistent selections and deselection (both those of the user and those inferred) as in the checking approach.
- *propagation and default selection:* The system, in addition to verifying the consistency of the selections and deselections and in addition to inferring all consequences of these, makes some default choices concerning selection and deselection of entities in cases where a unique solution cannot be inferred from the user's selection. For instance, if a constraint requires selection of one among a number of entities, the system could arbitrarily select one of these.

The choice among these is a true property of the constraint engine. We believe that the second is preferable over the first one. The user should know the consequences of these selections as soon as possible—it is annoying to find out that some selections have undesirable consequence long time and many considerations after the selection has been made.

The choice between the second and third is, at least partly, a matter of taste, we believe. The advantage of the third is that it can be used in such a way that at any given time, the system has a total, consistent configuration: those choices the user has not made himself, have been made by the system. On the other hand, it may be difficult for the user to distinguish between selections made by the system that were dictated by the user's previous selections on the one hand, and default selections on the other hand. It may be relevant for the user to tell the difference, because he can override the latter type, but not the former (at least not without changing some of his previous selections).

We prefer the second option (some of the effect of the third option can be obtained in the second option by having suggestions for the user in addition to the actual propagation). Although we prefer the second option—propagation—we use in fact a solution which is an intermediate step between checking and propagation, because in our set-up the full propagation solutions appears to be intractable, as will be explained later.

Finally, there is the question of *when* to do propagation. The following two options are possible:

- *online:* whenever the user makes a selection or deselection, the system checks or progates the configuration.
- *batch:* the user makes a number of selections or deselections, and the system then checks or propagates the configuration.

If the batch version is used by only invoking it when the user has specified a total configuration, then an engine implementing propagation will actually only work as one that checks, since the configuration is total. In any event, the choice between these two possibilities will also be left to the presentation layer; the contrainst engine will be able to handle both.

In conclusion, the choice between all the options of this subsection will be left to the presentation layer, which is not specified in this paper, with the one exception that we will build a constraint engine where the option between checking and propagation is made in favor of propagation—to the extent feasible.

3 Formalization of Constraints

Having established some idea about what form constraints may have, and how the user should interact with them, we now proceed to present a syntax and semantics of entities and constraints and a definition of the constraint propagation problem. This latter definition is novel and one of the main contributions of the paper.

3.1 Syntax of Product Line Constraints

Definition 1. The syntax of the universe of discourse is as follows:

$$\begin{array}{lll} \mathit{Problem} & \ni \mathit{problem} & ::= \mathit{conf} \quad \mathit{constrset} \\ \mathit{Conf} & \ni \mathit{conf} & ::= E_1 : \tau_1 \, , \ldots , E_n : \tau_n \\ \mathit{Indic} & \ni \tau & ::= \mathbf{selected} \mid \mathbf{deselected} \mid \mathbf{unspecified} \\ \mathit{Constrset} & \ni \mathit{constrset} & ::= \mathit{constr}_1, \ldots , \mathit{constr}_m \\ \mathit{Constr} & \ni \mathit{constr} & ::= E \mid \mathit{constr} \Rightarrow \mathit{constr} \mid \neg \mathit{constr} \end{array}$$

We assume a set of entity names, ranged over by E, E_1, E_2, etc. A *constraint problem* consists of a *configuration* and a *constraint set*. The configuration, in turn, consists of zero or more pairs each comprising an *entity name* and a *select indication*, i.e. an indication of whether the user has selected the entity, deselected the entity, or none of these. The configuration with zero such pairs is denoted by ε. A constraint set consists of zero or more constraints each of which is a propositional formula in which the entities are the atomic propositions. The constraint set with zero constraints is denoted by ε. We require that every entity occurring in *constrset* also occurs in *conf* and vice versa.

A configuration is called *total* if no entity has select indication **unspecified**. Similarly, a configuration is called *empty* if *every* entity has select indication **unspecified**.

As can be seen from the definition, we take $\Rightarrow$ and $\neg$ as primitives. Thus, the three forms of constraints are:

- E must be selected.
- If constraint constr_1 is true, then constr_2 must be true.
- Constraint *constr* must not be true.

It is well-known (see e.g. [24]) that the remaining connectives can be derived, and we shall use disjunction, conjunction, etc., as well as the atomic formulas **false** and **true**, freely in the rest of the paper. In practice, such derivations should be handled by a mapping layer between the presentation layer and the constraint engine.

Some common binary constraints are:

1. The entity E_1 is an aggregation of among others E_2.[4]
2. The selection of entity E_1 precludes the selection of E_2.
3. The selection of entity E_1 implies the selection of E_2.

More precisely, these are informal linguistic constraints. The actual constraints are:

1. $E_1 \Rightarrow E_2$.
2. $E_1 \Rightarrow \neg E_2$.
3. $E_1 \Rightarrow E_2$.

[4] Aggregation is also called *structural decomposition*, see e.g. [39].

Incidentally, note that the first of these is identical to the last one. To express that E is composed of E_1 and E_2 one uses the constraints

$$E \Rightarrow E_1.$$
$$E \Rightarrow E_2.$$

These constraints state that one cannot select a whole without selecting the parts. One can also consider adding the constraint

$$E \Leftarrow E_1 \wedge E_2, \tag{1}$$

which states that one has selected the whole, if one has selected all the parts. However, this latter constraint is not necessarily relevant. For instance, E_1 could be "rear spoiler," E_2 could be "extra front lights," and E could be "sports package." An easy way for the user to select both E_1 and E_2 would be to select E, then the system infers that the two entities must be selected as well. But even if the user has selected E_1 and E_2 it may be irrelevant for him to know that this is the sports package, so it might not be necessary to have the system infer E. Moreover, the user may deselect the sports package, and yet select manually the two constituents. This works fine without (1), but does not work with (1) present. So whether one should include (1) or not, depends on what one wants to express.

To sum up, our "model" (the constraints) only allows elements that express boolean relationships between selection and deselection of entities. Other traditional model elements such as aggregation, specialization (inheritance), etc. must be translated to boolean constraints, and such a translation might be handled by a layer between the presentation layer and the constraint engine.

A typical process involving selections and deselections starts with an empty configuration and makes progress towards a total configuration. The following definition formalizes the notion of "making progress" in terms of "refinements."

Definition 2.

1. For select indications τ, τ' define the relation $\tau \leq \tau'$ (read τ' is a *refinement* of τ) as follows:
$$\begin{aligned} \tau &\leq \tau. \\ \textbf{unspecified} &\leq \tau. \end{aligned}$$
2. For configurations $conf, conf'$, define the relation $conf \leq conf'$ (read $conf'$ is a *refinement* of $conf$) as follows:
$$\begin{aligned} \varepsilon &\leq \varepsilon \\ conf\ E : \tau &\leq conf'\ E : \tau' \qquad \text{if } conf \leq conf' \text{ and } \tau \leq \tau' \end{aligned}$$
3. If $conf \leq conf'$ and $conf \neq conf'$ then $conf < conf'$.
4. If $conf$ is the configuration $E_1 : \tau_1\ , \ldots , E_n : \tau_n$ then $conf[E_i := \tau]$ is the configuration
$$E_1 : \tau_1\ , \ldots , E_{i-1} : \tau_{i-1}, E_i : \tau, E_{i+1} : \tau_{i+1}\ , \ldots , E_n : \tau_n.$$

3.2 Semantics of Product Line Constraints

We now proceed to provide the semantics of our constraints. Roughly, the semantics of our constraint sets is obtained by identifying **selected** and **deselected** with the truth values "true" and "false," respectively, and using the usual semantics of classical, propositional logic.

Definition 3.

1. A *valuation* is a map from entity names to $\{t, f\}$.
2. For a valuation ν and constraint *constr*, define $[\![constr]\!]_\nu$ by:

$$\begin{aligned} [\![E]\!]_\nu &= \nu(E) \\ [\![constr_1 \Rightarrow constr_2]\!]_\nu &= \begin{cases} t & \text{if } [\![constr_1]\!]_\nu = f \text{ or } [\![constr_2]\!]_\nu = t \\ f & \text{otherwise} \end{cases} \\ [\![\neg constr]\!]_\nu &= \begin{cases} t & \text{if } [\![constr]\!]_\nu = f \\ f & \text{otherwise} \end{cases} \end{aligned}$$

3. For a valuation ν and constraint set *constrset*, define $[\![constrset]\!]_\nu$ by:

$$\begin{aligned} [\![\varepsilon]\!]_\nu &= t \\ [\![constrset\ constr]\!]_\nu &= \begin{cases} t & \text{if } [\![constrset]\!]_\nu = t \text{ and } [\![constr]\!]_\nu = t \\ f & \text{otherwise} \end{cases} \end{aligned}$$

4. For a valuation ν, an element $b \in \{t, f\}$, and an entity name E, define the valuation $\nu[E \mapsto b]$ by:

$$\nu[E \mapsto b](E') = \begin{cases} b & \text{if } E = E' \\ \nu(E') & \text{otherwise} \end{cases}$$

Remark 1. When we consider a valuation together with some constraint or constraint set, it is always implicitly assumed that the valuation is defined for all the entities occurring in the constraint or constraint set.

Definition 4.

1. A constraint *constr* is *valid* if $[\![constr]\!]_\nu = t$ for all valuations ν.
2. A constraint *constr* is *satisfiable* if there exists a valuation ν with $[\![constr]\!]_\nu = t$.
3. A constraint *constr* is *uniquely satisfiable* if it is satisfiable and any two valuations ν_1 and ν_2 with $[\![constr]\!]_{\nu_1} = t = [\![constr]\!]_{\nu_2}$ satisfy $\nu_1(E) = \nu_2(E)$ for every entitity E occuring in *constr*.
4. A constraint *constr* is *true* in a valuation ν iff $[\![constr]\!]_\nu = t$.
5. The above notions are generalized from constraints to constraint sets by replacing all occurrences of *constr* by *constrset*.

Remark 2. As usual, *constr* is valid iff $\neg constr$ is unsatisfiable, and *constr* is satisfiable iff $\neg constr$ is not valid.

We often need to consider the above notions relative to a given configuration *conf*. For instance, we want to know whether *constrset* is satisfiable in such a way that the corresponding valuation ν that makes *constrset* true respects the assignments in *conf*, i.e. ν assigns true and false to all entitites that are mapped to **selected** and **deselected**, respectively, in *conf*.

Definition 5. Let *conf* be a configuration.

1. The valuation ν_{conf} *determined by* a total configuration *conf* is defined as follows:
$$\nu_{\varepsilon} = \{\}$$
$$\nu_{conf\ E:\tau} = \begin{cases} \nu_{conf}[E \mapsto t] & \text{if } \tau = \textbf{selected} \\ \nu_{conf}[E \mapsto f] & \text{if } \tau = \textbf{deselected} \end{cases}$$
2. A *conf*-valuation is a valuation of the form $\nu_{conf'}$ for some total refinement *conf'* of *conf*.
3. All the notions in Definition 4 are generalized by replacing valuation, valid, satisfiable, uniquely satisfiable, and true by *conf*-valuation, *conf*-valid, *conf*-satisfiable, *conf*-uniquely satisfiable, and *conf*-true, respectively.

Remark 3. The valuation ν_{conf} is only defined for total configurations.

3.3 The Constraint Propagation Problem

The preceding two subsections have presented constraints and their semantics. It remains to address the question: for the actual application of constraints to product line shopping, which properties are we interested in—satisfiability, validity, etc? We now proceed to answer this question.

The following three mutually exclusive and together complete cases could be of interest for a constraint problem *conf constrset*:

1. *constrset* is *conf*-valid. This means that no further conflicts are possible, provided the user from now on only makes choices about entities with indication unspecified. A special case is when there are no more entities with select indication unspecified, i.e. when the selection is total. In this case the user has made a total selection which makes all the constraints true.
2. *constrset* is not *conf*-satisfiable. This means that the user has made a selection which violates one or more constraints. Regardless of how the user handles entities which are currently unspecified, it will not be possible to make all constraints true. A special case is when there are no more entities with select indication unspecified, i.e. when the selection is total. In this case the user has made a total selection which does not make all the constraints true.
3. *constrset* is *conf*-satisfiable, but not *conf*-valid. This means that the user still has a chance to arrive at a total selection that makes all the constraints true. In this case we can distinguish two subcases:

(a) *constrset* is *conf*-uniquely satisfiable. In this case, the remaining selection is dictated by what the user has already selected.
(b) *constrset* is not *conf*-uniquely satisfiable. In this case, there are several different possible selections the user can make.

In addition to these properties we are interested in *propagating* selections. The user does not simply want to know "the constraint set now has a single solution;" he also wants to know what the solution is, that is, which selections are forced by other selections. In fact, even when there is not a single unique solution for the constraint set, the user still wants to be informed if the selection or deselction of one or more of the entities participating in the constraint set can be inferred. Moreover, when there are conflicts, he wants to know what the conflicts consist of and, if possible, have suggestions concerning how to resolve the conflicts. In short, the above cases must be ammended by the following:

- *Inference:* propagation of select indications.
- *Errors:* reporting of inconsistent select indications.
- *Corrections:* suggestions for changes in inconsistent select indications.

Thus, our inference algorithms will not compute validity or satisfiability directly, but rather a mixture in which propagation plays a role, and it will report errors and suggest corrections. We now formalize these considerations in terms of an *optimal inference algorithm.*

Definition 6.

1. An *inference algorithm* is an algorithm
$$I : Conf \times Constrset \to Conf \cup \{\mathbf{fail}\},$$
where $conf \leq I(conf, constrset)$ when $I(conf, constrset) \neq \mathbf{fail}$.
2. We say that *conf makes constrset true* if $[\![constrset]\!]_{\nu_{conf}} = t$, for a total *conf*. We also call *conf* a *solution to constrset.*
3. An inference algorithm is *optimal* if the following two conditions are satisfied:
   (a) If *constrset* is *conf*-satisfiable, then
   $$I(conf, constrset) = conf'. \tag{2}$$
   Moreover,
   i. For all total $conf''$ such that
   $$conf \leq conf''$$
   and $conf''$ makes *constrset* true, it holds that $conf' \leq conf''$.
   ii. There is no $conf'''$ with $conf' < conf'''$ such that for all total $conf''$ with
   $$conf \leq conf''$$
   and $conf''$ makes *constrset* true, it holds that $conf''' \leq conf''$.[5]

[5] Condition ii states that the configuration *conf* cannot be further refined and still satisfy property i.

(b) If *constrset* is not *conf*-satisfiable, then

$$I(conf, constrset) = \textbf{fail}. \tag{3}$$

The definition of optimality is inspired by *completeness* of polymorphic type inference.[6]

3.4 Properties of Optimal Inference Algorithms

Have we succeeded with the preceding definition in capturing the desired form of inference algorithm? We aim to show that the answer is *yes*. We will do so with some informal considerations pertaining to the definition combined with a series of propositions stating some of the desired properties.

The following shows that there is at most one optimal inference algorithm. This can be viewed as a sort of completeness of the definition in that it does not make sense to develop additional desired properties of optimal inference algorithms: those developed already uniquely identify the algorithm (if it exists at all). Of course, only the input-output behaviour of the algorithm is uniquely identified. It may *work* in many ways.

Proposition 1. *Let I_1 and I_2 be optimal inference algorithms, and consider the problem conf constrset. Then for all conf and constrset:*

$$I_1(conf, constrset) = I_2(conf, constrset).$$

Proof. Let I_1, I_2 be optimal inference algorithms, and let *conf constrset* be a problem. If *constrset* is not *conf*-satisfiable then by optimality

$$I_1(conf, constrset) = \textbf{fail} = I_2(conf, constrset).$$

If *constrset* is *conf*-satisfiable then, again by optimality,

$$I_1(conf, constrset) = conf'_1$$

and

$$I_2(conf, constrset) = conf'_2,$$

[6] There are two changes compared to completeness of polymorphic type inference. The first is clause (ii) of part (a), which is added to force as much refinement as possible. Indeed, without this addition, the inference algorithm could simply return the argument *conf* unaltered. The reason is that we do not require that the output of an optimal inference algorithm be a solution (in particular a total configuration), since most general solutions do not exist. In contrast, in type inference there is a requirement that the answer actually be a type for the term. This means that the analogous situation—that the algorithm simply returns a type variable as the most general type—does not occur there.

The second change is the phrase "total" in clause (i) and (ii) of part (a). The reason for this is that we only speak of solutions among total configurations. In contrast, a type for a term may contain type variables.

where

$$conf'_1 \geq conf \leq conf'_2.$$

Since *constrset* is *conf*-satisfiable, there are a number of total configurations $conf''$ with $conf \leq conf''$ such that $conf''$ makes *constrset* true. By optimality, each $conf''$ is in fact a refinement of both $conf'_1$ and $conf'_2$, i.e.

$$conf'_1 \leq conf'' \geq conf'_2.$$

Let E be an entity in *constrset*. We split into two disjoint cases.

1. Every total refinement $conf''$ of *conf* that makes *constrset* true assigns the same select indication (**selected** or **deselected**) to E. In this case, $conf'_1$ must assign the same indication to E. [Reason: if another indication is assigned to E, no $conf''$ is a refinement of $conf'_1$, and if **unspecified** is assigned to E, there is a proper refinement of $conf'_1$ that still has all total configurations making *constrset* true as refinements. Both cases contradict optimality.] Similarly, $conf'_2$ must assign the same indication to E.
2. There exist two total refinements $conf''_a$ and $conf''_b$ of *conf* that assign **selected** and **deselected** to E, respectively, and which both make *constrset* true. In this case, $conf'_1$ must assign **unspecified** to E. [Reason: otherwise there would be a total refinement of *conf* making *constrset* true that is not also a refinement of $conf'_1$, contradicting optimality.] Similarly, $conf'_2$ must assing **unspecified** to E.

Thus, for every entity, the indication is uniquely determined. □

The equations (2) and (3) state that the algorithm must return **fail** if, and only if, the current configuration is such that there does not exists a refinement of this configuration that makes the constraint set true. That is, we have:

Proposition 2. *Let I be an optimal inference algorithm, and consider the problem conf constrset. Then*

$$I(conf, constrset) = \textbf{fail}$$

if, and only if, constrset is not conf-satisfiable.

Proof. Immediate from the definition. □

The following proposition shows that if an optimal inference algorithm returns a configuration $conf'$ at all (i.e. not **fail**) for a contraint set *constrset*, then *constrset* is *conf*-satisfiable. Moreover, although the returned configuration does not need to be total and make the constraints true, it must be such that all total configurations that make the constraints true are obtained by further refinements, and such refinements do exist.

Proposition 3. *Let I be an optimal inference algorithm, and consider the problem conf constrset. Then*

$$I(conf, constrset) = conf' \tag{4}$$

if, and only if, constrset is conf-satisfiable. Moreover, when (4) holds, constrset is conf′-satisfiable.

Proof. The only part that does not follow immediately from the definition is that (4) implies that *constrset* is *conf′*-satisfiable. Suppose that (4) holds. Since *constrset* is *conf*-satisfiable there is a total refinement of *conf* that makes *constrset* true. By optimality, this refinement must also be a refinement of *conf′*. Hence *constrset* is *conf′*-satisfiable. □

Clause (i) in part (a) of the definition of completeness states that all total refinements of the current configuration that make the constraint set true, must also be refinements of the computed configuration. That is, an optimal algorithm must not refine so much that solutions that were refinements of the configuration we started with, can no longer be obtained—the refinement must preserve all solutions. Another way of putting this is that the algorithm must not make arbitrary choices. Clause (ii) states that there must not be a proper refinement of the computed configuration that also preserves all solutions—in this case the proper refinement should have been the result. Clause (i) and (ii) together then state that the algorithm must make all the non-arbitrary choices, thereby computing a sort of *most specific generalization.*

As a special case, if the configuration has a single total refinement that makes the constraint set true, the algorithm must find this one.

Proposition 4. *Let I be an optimal inference algorithm, and consider the problem conf constrset. If constrset is conf-uniquely satisfiable, and ν is the corresponding conf-valuation, i.e.*

$$[\![constrset]\!]_\nu = t = [\![constrset]\!]_{\nu_{conf'}} \quad ,$$

for some total refinement conf′ of conf, then

$$I(conf, constrset) = conf'.$$

Proof. Since *constrset* is *conf*-satisfiable,

$$I(conf, constrset) = conf',$$

for some refinement *conf′* of *conf*. Since *constrset* is *conf*-uniquely satisfiable there is exactly one total refinement *conf″* that makes *constrset* true. We must then have *conf′* = *conf″*; otherwise, there would be a contradiction with the criterion (a), part (ii), in the definition of optimality. □

Conversely, we have the following.

Proposition 5. *Let I be an optimal inference algorithm, and consider the problem conf constrset. If*

$$I(conf, constrset) = conf',$$

where conf′ is total, then constrset is conf-uniquely satisfiable, and $\nu_{conf'}$ is the corresponding valuation.

Proof. If

$$I(conf, constrset) = conf',$$

where $conf'$ is total, then every total refinement of *conf* that makes *constrset* true is also a refinement of $conf'$. Since $conf'$ is already total it must be the only total refinement of *conf* that makes *constrset* true. Thus, *constrset* is *conf*-uniquely satisfiable. The corresponding evaluation must be $\nu_{conf'}$. □

As another special case, if all total refinements make the constraint set true, the algorithm is not able to make any propagation without user intervention.

Proposition 6. *Let I be an optimal inference algorithm, and consider the problem conf constrset. If constrset is conf-valid, then*

$$I(conf, constrset) = conf.$$

Proof. If *constrset* is *conf*-valid, it is *conf*-satisfiable, so

$$I(conf, constrset) = conf'$$

for some refinement $conf'$ of *conf*. Let E be some entity that *conf* assigns **unspecified**. Then there is a total refinement of *conf* that assigns **selected** to E and there is another total refinement of *conf* that assigns **deselected** to E. Hence, *conf* must assign **unspecified** to E as well, by optimality. □

The converse does not hold; that is, it may be that

$$I(conf, constrset) = conf$$

and yet *constrset* is not *conf*-valid. Consider, for instance, the problem:

$$E_1 : \textbf{unspecified}$$
$$E_2 : \textbf{unspecified}$$
$$E_1 \Leftrightarrow \neg E_2.$$

We have

$$I(conf, constrset) = conf,$$

eventhough the constraint is not valid. The point is that it is non-uniquely satisfiable.

It follows from the preceding propositions that we can recognize satisfiability (getting **fail** or not) and unique satisfiability (getting a total configuration back or not). We therefore have:

Proposition 7.

1. *Computing an optimal inference algorithm is as hard as SAT (the general satisfiability problem).*
2. *Computing an optimal inference algorithm is as hard as USAT (the unique satisfiability problem).*

Proof. With *conf* equal to the empty configuration (i.e. all entities mapped to **unspecified**), *conf*-satisfiability and *conf*-unique satisfiability degenerate to satisfiability and unique satisfiability, respectively. Hence the result follows from Propositions 3, 4, and 5. □

It follows that it may not be feasible to compute an optimal algorithm.

4 Solution of Contraints

In this section we finally provide several algorithms for solving constraint problems. The first subsection describes several conceptual algorithms, and the second subsection outlines the elements of a configuration system based on these algorithms. The last two subsections explore ideas for implementation of the conceptual algorithms based on program transformation techniques and binary decision diagrams, respectively.

4.1 Conceptual Algorithms

We will assume that the overall constraint set may be quite large (say several hundred entities and a similar number of constraints). It follows from the last result in the preceding section that it is probably intractable to compute an optimal inference algorithm on the entire constraint set.

In contrast we will assume that each constraint is typically quite small, say involving less than 10 entities. Therefore, it may be tractable to compute an optimal inference algorithm at the level of the individual constraints. The following is the obvious such algorithm; we call it I_l (l for "local").

Definition 7. For $b \in \{f, t\}$ define

$$\underline{b} = \begin{cases} \textbf{selected} & \text{if } b = t \\ \textbf{deselected} & \text{if } b = f. \end{cases}$$

Algorithm 1 *Define*

$$I_l : \mathit{Conf} \times \mathit{Constr} \to \mathit{Conf} \cup \{\textbf{fail}\}$$

by:

1. *input: configuration conf and constraint constr.*
2. *let* $E_1, \ldots, E_n$ *be the entities in conf mapped to* **unspecified**.
3. *let* $c_1 = \ldots = c_n =$ **unspecified**.
4. *let sol* = **false**.
5. *for each conf-valuation* ν*:*
   if $[\![constr]\!]_\nu = t$ *then*
   if sol = **true** *then for* $i \in \{1, \ldots, n\}$*:*
   if $c_i \neq \overline{\nu(E_i)}$ *then* $c_i =$ **unspecified**.
   if sol = **false** *then*
   sol = **true**.
   for $i \in \{1, \ldots, n\}$*:* $c_i = \underline{\nu(E_i)}$.
6. $conf' = conf[E_1 \mapsto c_1, \ldots, E_n \mapsto c_n]$.
7. *if sol* = **false** *then return* **fail** *else return* $conf'$.

This algorithm is inefficient in that it traverses all possible (and impossible) valuations for the entities in each constraint. This can be remedied by including in the language some predefined constraints such as:

$$E_1 \wedge \ldots \wedge E_n \Rightarrow E'_1 \wedge \ldots \wedge E'_n,$$

where all the entities are distinct. The propagation along this constraint can be done more efficiently than by traversing all possible valuations and checking whether only one is possible, namely in the following way:

1. if $E_1, \ldots, E_n$ are **selected** and $E'_1, \ldots, E'_n$ are **selected** or **unspecified**, then those that are **unspecified**, must be refined to **selected**.
2. if at least one of $E'_1, \ldots, E'_n$ is **deselected**, and all of $E_1, \ldots, E_n$ are **selected**, except E_i which is **unspecified**, then E_i must be refined to **deselected**.

By having such special treatment for certain constraints, we are pruning the search for solutions in the space of all valuations. In the next two sections, other techniques for improving the efficiency of I_l are introduced.

By viewing a constraint set as the conjunction of all its constraints, and using the above algorithm, we obtain the following result.

Proposition 8. *There exists an optimal inference algorithm.*

Proof. The algorithm corresponds to the cases in the proof of Proposition 1. □

However, for efficiency reasons we might not wish to use the algorithm (or some other implementation of it) at the constraint set level. Instead we might use the following algorithm I_g (g for "global"), which calls I_l for each individual constraint.

Algorithm 2 *Define*

$$I_g : Conf \times Constrset \rightarrow Conf \times Constrset$$

by:

1. *input:* $conf_0$ *and* $constrset = constr_1, \ldots, constr_m$.
2. *let* failures= {}.
3. *for* $i = 1, \ldots, m$
   let $I_l(conf_{i-1}, constr_i) = result$.
   if $result =$ **fail** *then*
   let failures=failures $\cup\{constr_i\}$.
   let $conf_i = conf_{i-1}$.
   else let $conf_i = result$.
4. *return* $conf_m$ *and* failures.

The algorithm returns two objects. The first is the refined configuration $conf_m$, which should be considered as a suggestion from the system concerning how to refine the selections. The user can either accept these, or make other selections. The second object returned by the algorithm is the set *failures*, which is

the list of constraints that have no solution. One can imagine that the algorithm, in addition to this set, produces a list of suggested *corrections*, i.e. suggestions for changing the user's selections in such a way that the constraints become satisfiable. Among all the different possibilities, the system might present the first few with smallest distance to the configuration $conf_0$, for some notion of distance.

The following example, that could not be handled by the techniques developed in the PROMIS project [8], was the original motivation for the research reported in the present paper.

Example 1. Consider the constraint problem *conf constrset*:

$$\begin{aligned} &E_1 : \textbf{unspecified} \\ &E_2 : \textbf{unspecified} \\ &E_3 : \textbf{unspecified} \\ &E_1 \Rightarrow E_2 \end{aligned}$$

Suppose the user selects E_1 and runs I_g, i.e. let

$$conf' = conf[E_1 \mapsto \textbf{selected}].$$

Then $I_g(conf', constrset)$ will assign **selected** to E_2, so this assignment will be suggested to the user. Suppose that the user selects entity E_3, so that he has now selected E_1 and E_3, i.e. let

$$conf'' = conf'[E_3 \mapsto \textbf{selected}].$$

Then $I_g(conf'', constrset)$ will again assign **selected** to E_2, so this is again suggested to the user. Now suppose the user deselects E_1, i.e. let

$$conf''' = conf''[E_1 \mapsto \textbf{deselected}].$$

Then $I_g(conf''', constrset)$ will not assign **selected** to E_2, since the propagation from E_1 to E_2 no longer happens, so the suggestion to select entity E_2 will go away.

This example illustrates that our set-up can accommodate interactions with the user that are not possible with an undo-feature, because in the above example, undoing instead of deselecting E_1 would also roll back the selection of E_3, i.e. the whole history of interactions is rolled back a number of steps. This is generally not the desired behaviour.

The algorithm I_g has at least two shortcomings. The first is that the propagation depends on the order in which the constraints are processed, and correct propagation may be lost due to this order. For instance, if the constraint problem

$$\begin{aligned} &E_1 : \textbf{unspecified} \\ &E_2 : \textbf{selected} \\ &E_3 : \textbf{unspecified} \\ &E_1 \Leftrightarrow \neg E_3 \\ &E_2 \Rightarrow E_1 \end{aligned}$$

is processed in the order in which the constraints are listed, then one execution of I_g will refine E_1 to selected, but E_3 will not be refined, although clearly E_3 should be refined to be deselected.

A simple way to remedy this is by iterating I_g until no further refinements are obtained. While such compromises are ugly from a theoretical point of view, they can be necessary and useful in practice. A more efficient remedy is to not iterate I_g on the entire constraint set, but only on constraints containing an entity which was refined in the previous iteration.

The second shortcoming of I_g is that some propagation is left out, regardless of the order or number of times the constraints are processed. For instance, consider the constraint problem $conf\ constr_1, constr_2, constr_3$:

$$\begin{array}{l} E_1 : \textbf{unspecified} \\ E_2 : \textbf{unspecified} \\ E_3 : \textbf{unspecified} \\ E_1 \Leftrightarrow \neg E_2 \\ E_2 \Leftrightarrow \neg E_3 \\ E_3 \Leftrightarrow \neg E_1 \end{array}$$

All three constraints are satisfiable (though not uniquely), and we have that $I_l(conf, constr_i) = conf$ for each $i = 1, 2, 3$. It follows that we also have $I_g(conf, constr_1\ constr_2\ constr_2) = conf$. However, for an optimal algorithm I we have $I(conf, constr_1\ constr_2\ constr_2) = \textbf{fail}$. This cannot be remedied: we are paying a price for the fact that we are using an approximate solution.

The consequence of this example is that the user may start from a situation without any failures, but regardless of how he refines the configuration, he ends up with failures. In other words, the reporting of some inconsistent selections are postponed until the user makes more choices. We do not know how rare or typical such phenonema are in actual models.

One way of minimizing the damage is to collect all such constraints in a set *warnings* and return them as warnings to the user. This does not make the algorithm optimal, but it high-lights the risks of non-optimality.

Algorithm 3 *Define*

$$I'_g : Conf \times Constrset \to Conf \times Constrset \times Constrset$$

by:

1. *input: conf and constrset* $= constr_1, \ldots, constr_m$.
2. *let* failures= {} *and* todo= $\{constr_1, \ldots, constr_m\}$.
3. *while there is a constr* $\in$todo*:*
   let todo=todo$\backslash\{constr\}$.
   let $I_l(conf, constr) = result$.
   if $result = \textbf{fail}$ *then* failures=failures $\cup\{constr\}$ *else*
   let $E_1, \ldots, E_t$ *be the entities refined from conf to result.*
   let $constr'_1, \ldots, constr'_s$ *be all constraints in which one or more of* $E_1, \ldots, E_t$ *occur.*

let todo=todo$\cup\{constr'_1, \ldots, constr'_s\}$.
let $conf = result$.

4. *let* $F_1, \ldots, F_k$ *be the entities mapped to* **unspecified** *in conf.*
5. *let* $constr_{i_1}, \ldots, constr_{i_l}$ *be all constraints in which one or more of* $F_1, \ldots, F_k$ *occur.*
6. *let* warnings=$\{constr_{i_1}, \ldots, constr_{i_l}\}$.
7. *return conf,* failures*, and* warnings.

As mentioned above, I'_g is not optimal, since some propogation is left out. However, the algorithm is also non-optimal in another sense: it never returns **fail**! In fact, the same applies to I_g. Rather than reporting **fail** when an unsatisfiable contraint is encountered, the algorithm reports an error and proceeds with the remaining constraints. In practice, we believe that this is the desired behaviour. In this sense, the algorithm resembles a type inference algorithm; the latter does not abandon the typing of a whole program when a sub-expression cannot be typed.

4.2 System Overview

The following should be the main functionality of a configuration system based on the above algorithm; the formulation of the three points are inspired by type-based specification, type-inference, and type-directed translation, respectively.

- *Editing*: There should be an editor in which the user can browse the current configuration, and in which the user can select or deselect an entity, or leave unspecified the select indication of an entity. This should happen through some presentation layer, as previously explained.
- *Inference*: There should be an inference engine that the user can invoke (or it is invoked automatically whenever the user changes the select indication of an entity). It propagates choices using I'_g, reporting suggestions (the refined configuration), failures (constraints that could not be satisfied), and possibly corrections and warnings. These should be mapped back through the presentation layer so that they are presented to the user at the same level as he made the original choices on.
  Notice that the inferred select indications are offered to the user as *suggestions.* He might decline to accept them. For instance, he might prefer to change some of the select indications that led to these suggestions. In other words, one can distinguish the following five select states:
  1. unspecified.
  2. selected.
  3. deselected.
  4. selected by the system as part of a propagation.
  5. deselected by the system as part of a propagation.

  In the two latter states the system has made a selection or deselection, which is presented to the user as a suggestion which he can accept or reject. Thus, an acceptance by the user of selection proposed by the system, corresponds to a shift from, say, state 4 to state 2.

However, it is important to understand that the system's reported failures for some constraints and the absence of failure for the other constraints assumes that all suggestions are accepted.

– *Code generation*: There should be a component translating a total configuration which makes all the constraints true into a procurement document (the totality and truth should be checked by the component).

The system can be tuned by only calling I'_g with those constraints that contain one or more of the entities that the user has changed select indication for in the *todo* set. Another obvious improvement is to allow some short-cuts for the user such as a "deselect all unspecified entities" option. This is useful when the user is done selecting and deselecting, and just want to deselect everything else.

4.3 A Program Transformation Approach

The authors have recently presented a *program inversion* method called *explicitation* [32]. The method can invert a first-order function-oriented program with respect to a particular output. In general, the result of an inversion is a grammar that approximates the set of inputs that would result in the particular output. We can attempt to obtain an optimal inference algorithm by inverting a program that checks whether a constraint set is true in a given valuation:

```
 data Term = And Term Term | Or Term Term | Not Term |
Imply Term Term | Val Bool eval (And x y) = if eval x then
eval y else false eval (Or x y) = if eval x then true else
eval y eval (Imply x y) = if eval x then eval y else true
eval (Not x) = if eval x then false else true eval (Val x)
= x
```
(5)

This program expects as input a constraint set *constrset* in which the entities have been replaced according to a valuation ν. It returns **true** or **false** depending on the value of $[\![constrset]\!]_\nu$.

Now, if we invert the checker program w.r.t. the output **true** and a constraint set in which the entities have been replaced by program variables, the result will be a description of all ν such that $[\![constrset]\!]_\nu = \mathbf{t}$.

Example 2. For instance, consider the constraint set

$$\begin{aligned} z &\Rightarrow v \wedge (y \vee x \vee x_2) \\ x_2 &\Rightarrow x_1 \wedge \neg w \\ w &\Rightarrow v \vee y \vee x \end{aligned} \tag{6}$$

where x, y, z, v, w, x_1 and x_2 are entities. The encoding of the above constraint would be

```
 And (Imply (Val z) (And (Val v) (Or (Val y) (Or (Val x)
(Val x2))))) (And (Imply (Val x2) (And (Val x1) (Not (Val
w)))) (Imply (Val w) (Or (Val v) (Or (Val y) (Val x)))))
```
(7)

where `x`, `y`, `z`, `v`, `w`, `x1`, and `x2` are program variables. The result of inverting the checker program (5) w.r.t. constraint (7) will give us a grammar like the following.

$$S ::= [\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}] \,|\, [\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{f}] \,|\, \ldots \,|\, [\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}] \quad . \tag{8}$$

The grammar explicitly describes all valuations making the constraint set true. In all assignments, the entities are implicitly represented by a position in the list.

Remark 4. Inversion of the checker program w.r.t. any constraint set in which the entities have been replaced by program variables will always be a grammar that precisely lists the possible of the entities. Since the search space is finite, the same result could be obtained by using a functional-logic programming language directly.

Having obtained the set of valuations making the constraint set true, it is a trivial task to translate it into a set of total configurations by making the entities explicit and replacing **t**/**f** by **selected**/**deselected**. Let us call this set of configurations *valids*. The inference algorithm is then quite simple:

Algorithm 4 Define

$$I_1 : \mathit{Conf} \times \mathit{Conf\,list} \to \mathit{Conf} \cup \{\mathbf{fail}\}$$

by:

I$_1$ (*conf*,*valids*) =
  case filter (*conf* $\leq$) *valids* **of** *c* :: *cs* $\longrightarrow$ foldl msg *c cs*
    | [] $\longrightarrow$ **fail**
  where msg [][]= []
    msg ((*E*:*x*) ::*xs*) ((*E*:*y*) ::*ys*) =
      (*E*:(**if** *x* = *y* **then** *x* **else unspecified**)):: msg xs ys

The functions 'filter' and 'foldl' are the standard functions on lists:

filter : ($\alpha \to$ Bool) $\to$ α List $\to$ α List
filter *f* []= []
filter *f* (*x* :: *xs*) = **if** *f x* **then** *x* :: (filter *f xs*) **else** filter *f xs*
foldl : ($\alpha \to \beta \to \beta$) $\to$ $\beta \to \alpha$ List $\to$ β
foldl *f z* []= *z*
foldl *f z* (*x* :: *xs*) = foldl *f* (*f x z*) *xs*

This algorithm is very similar to I_l. The main differences are:

- the set of valuations making the constraint set true is computed in advance by inversion.
- among the resulting configurations those that are not refinements of the starting configuration are subsequently filtered away.
- the overall algorithm is expressed in functional rather than imperative style.

The *valids* set is in general exponential in the number of entities in the constraint set, so the space bound on the algorithm is

$$O(|conf|\,|valids|) = O(2^{entities}) \ , \tag{9}$$

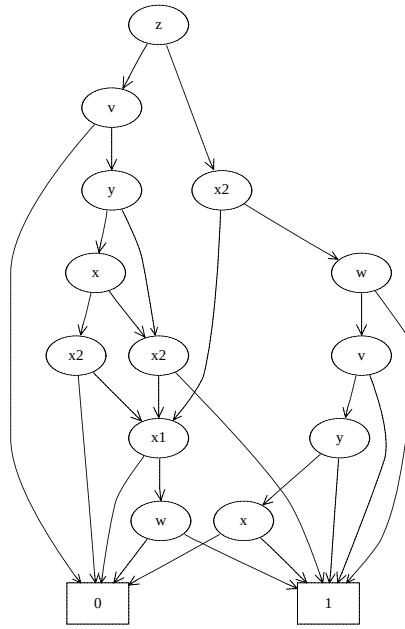
which makes this solution intractable, even though *valids* can be represented compactly as a vector of bit vectors.

Instead of obtaining all valuations making the constraint set true explicitly by inverting the checker program, we could *specialise* it with respect to the constraint set to get a program that can accept or reject total configurations. If the specialisation is done by one of the techniques collectively called *partial evaluation* [20], the result will be the following simple program:[7]

```
cond a b c = if c then cond0 a b else true
cond0 d e = if e then if d then false else true else false
main x y z v w x1 x2 =
    if z then if v
         then if y then cond w x1 x2
              else if x then cond w x1 x2
                   else if x2 then cond0 w x1
                        else false else false
    else if x2 then cond0 w x1
         else if w
              then if v then true
                   else if y then true else x
              else true
```

The interpretative overhead from the checker has been completely eliminated, and the program has become a simple, recursion-free, read-once boolean program. In fact, the program can be represented as the following *free binary-decision diagram* [40]:

[7] The specialised program has been produced by a variant [35, 33] of so-called *super-compilation* [36].



Every interior node is labelled with a variable and has two "legs," a solid and a dashed. Selecting a solid leg means assigning the value **t** to the variable, whereas selecting a dashed legs means assigning the **f** to the variable. Each path in a the diagram thus represents an assignment of truth values to the variables. If a path leads to the leaf **0**, it means that that particular assignment result in the value **f**; conversely **1** means **t**. If a variable is not mentioned on a particular path, it means that the value can be either **f** or **t**.

The merit of representing the constraint set this way, is that several total configurations can share common subconfigurations. This is the key observation that prompts us to cast the problem of optimal inference in a binary-decision-diagram setting in the next subsection.

4.4 Inference by BDDs

We will now encode the constraint set as a *reduced ordered binary-decision diagram* [7] (from now on simply called BDDs). The advantages of using BDDs are that there exist numerous efficient implementations, and that the inference problem can be solved by using only basic BDD operations, as we will explain below.

It is well known how to construct a BDD from a propositional formula, although the chosen variable (i.e. entity) order is crucial for the size of the BDD—in extreme cases a difference between linear and exponential size. But also optimal variable ordering is a hard problem: The best known algorithm runs in $O(n3^n)$, and even improving the variable ordering is NPTIME-complete, see [6, 16]; so heuristics are used in practice. In the following, we will simply

assume that a near-optimal variable order has been obtained by some standard method (see, e.g.[21, 29, 11]). Given a constraint set C, we will denote the resulting BDD by B_C.

Remark 5. Calculation of a good variable ordering can be done once and for all when the constraint set C is fixed.

The operations we will need are the 'restrict' operation, that specialises a BDD to a variable assignment, and the 'vars' operation that returns the list of variables in a BDD. Both run in time $O(|B_C|)$. We refer to the unique unsatisfiable BDD by '**0**', and similarly we refer to the unique valid BDD by **1**. Comparing other BDDs to **0** is a constant time operation. An optimal inference algorithm can now be cast as follows. An explanation follows the algorithm.

Algorithm 5 Define

$$I_2 : \mathit{Conf} \times \mathit{BDD} \to \mathit{Conf} \cup \{\mathbf{fail}\}$$

by:

```
I2 (conf, bdd) =
    case restrict bdd ν_conf
    of 0 ⟶ fail
    | bdd' ⟶ foldl speculate conf (vars bdd')
        where speculate E conf =
                  if restrict bdd' {E ↦ t} = 0
                  then conf[E:=deselected]
                  else if restrict bdd' {E ↦ f} = 0
                       then conf[E:=selected]
                       else conf
```

where ν_{conf} is the translation of *conf* into a partial valuation, omitting the entities that are **unspecified**.

In the above algorithm, we first specialise the given BDD to the given configuration, possibly aborting if the resulting bdd' is unsatisfiable. We then deduce impossible variable assignments by speculatively assuming that each variable E is **t** or **f**: If, say, $E = \mathbf{t}$ results in an unsatisfiable BDD, then E *must* be given the indication **deselected** to fulfil the constraints; we say that (the indication of) variable E is *forced*. The forced variable indications are returned together with the original configuration.

Proposition 9. *Algorithm 5 is an optimal inference algorithm.*

Proof. By definition [7], there is only one unsatisfiable BDD, namely **0**. Therefore, if $bdd' = (\text{restrict } bdd\ \nu_{conf})$ is unsatisfiable, then $bdd' = \mathbf{0}$, and thus the algorithm **fail**s, as required. Otherwise, bdd' is satisfiable and $\forall E \in (\text{vars } bdd') : conf(E) = \mathbf{unspecified}$. If there exists E such that $\nu(E) = b$ for all valuations ν which make bdd' true, then restrict $bdd'\ \{E \mapsto \neg b\} = \mathbf{0}$, and thus 'speculate

E conf' will infer the corresponding indication for E. Since 'speculate' is called for all variables in bdd', every forced indication will be changed (from **unspecified**) in the original configuration. Conversely, if for any E both restrict bdd' $\{E \mapsto \mathbf{t}\} \neq \mathbf{0}$ and restrict bdd' $\{E \mapsto \mathbf{f}\} \neq \mathbf{0}$, then there exist valuations ν and ν' making bdd' true, where $\nu(E) = \mathbf{f}$ and $\nu'(E) = \mathbf{t}$. Thus, if an indication is *not* forced, it is not changed. Hence, the most specific partial configuration is returned. □

The running time of the algorithm is

$$O(|bdd| + |\text{var } bdd|\,|bdd|) \geq O(|bdd| \log |bdd|) \ , \tag{10}$$

so there is certainly need for improvements, considering that the BDDs can be exponential in size of the constraint set. In practice, we expect constraint sets to be well-behaved, meaning that a good variable ordering can be found to reduce the size of the corresponding BDDs.

Improvements The deduction carried out by the function 'speculate' can be done more efficiently by a single breadth-first traversal of bdd'.[8] The key observation is that, if, say,

$$\text{restrict } bdd' \ \{E \mapsto \mathbf{t}\} = \mathbf{0} \ ,$$

then it holds for bdd' that

1. every branch that leads to **1** will contain a node labelled E, and
2. every node labelled E must have the **t**-leg connected to **0**.

It is relatively easy to see how the forced variable indications can be obtained by traversing the BDD with the help of priority queue.

Algorithm 6 Let Q be a priority queue that can contain nodes of the BDD such that the order of the nodes are dictated by the order of the variables.[9] The front of Q will thus always be the nodes that are closest to the top of the BDD. Initially, Q contains the single root node.

1. If Q is empty, terminate.
2. Let the set N be all the nodes at the front of Q with the same variable label. Remove N from Q.
3. If Q is empty, condition 1 is true. If furthermore all nodes in N have the same leg in **0**, condition 2 is true, and thus we have found a forced indication for a particular variable; output this indication.
4. Let M be all the children of nodes in N. Remove all **0**-nodes from M.

[8] The function 'speculate' was suggested by Ken Friis Larsen (IT-C, Denmark) as a conceptual simplification of the more efficient function that follows.

[9] Recall that the variables in a BDD are totally ordered; the largest variable is the one labelling the root.

5. If M contains **1**-nodes, the algorithm terminates, since condition 1 cannot be fulfilled by any of the remaining variables.
6. Add M to Q, and repeat step 1.

In algorithm 6, each node is inserted at most once into Q, and each such insertion takes time $O(\log |Q_{\max}|)$ where $Q_{\max}$ is the largest Q. The running time of algorithm 6 is thus $O(|bdd| \log |Q_{\max}|)$. If we replace the traversal of all the variables in algorithm 5 with algorithm 6, the total running time of this improved algorithm is

$$O(|bdd| + |bdd| \log |Q_{\max}|) \leq O(|bdd| \log |bdd|) \ , \tag{11}$$

a clear improvement of the previous upper bound (10).

Partitioning the Constraint Set The upper bound $O(|B_C| \log |B_C|)$ is still not tractable, since $|B_C| \leq 2^{|C|}$. We might therefore look for properties of the constraint set that can make the inference more tractable. In particular, we could aim at building a set of much smaller BDDs instead of building one large BDD. Consider a constraint set C. What we are looking for is a partition of C such that

$$\begin{gathered} C = C_1 \wedge C_2 \wedge \cdots \wedge C_n \\ \mathrm{I}_2(conf, B_C) \succeq \mathrm{I}_4(conf, [B_{C_1}, B_{C_2}, \ldots, B_{C_n}]) \succeq conf \end{gathered} \tag{12}$$

where

$$conf \overset{\text{def}}{\succeq} conf' \quad \text{iff} \quad conf \geq conf' \vee conf = \mathbf{fail}$$

and

Algorithm 7 Define

$$I_4 : Conf \times BDD\ list \to Conf \cup \{\mathbf{fail}\}$$

by:

I_4 ($conf$, $bdds$) = fix (runthrough $bdds$) $conf$
where fix f d = **let** $d' = f\ d$ **in if** $d = d'$ **then** d **else** fix f d'
runthrough $bdds$ $conf$ = foldl propagate $conf$ $bdds$
where propagate **fail** bdd = **fail**
propagate $conf$ bdd = I_2 ($conf$,bdd)

Algorithm 7 repeatedly applies algorithm 5 to each part C_i, accumulating new configurations until a fixpoint is reached.

The use of $\succeq$ in (12) allows us to select a partitioning of the constraint set which leads to non-optimal inference.

Remark 6. Any partitioning of the constraint set will fulfil (12), so choosing a particular partitioning only affects the precision of the inference.

Clearly, if the constraint set can be divided into *independent* parts, in the sense that each pair of parts have no variables in common, (12) will hold even if the first occurrence of $\succeq$ is replaced by $=$, and thus optimality is regained. Partitioning into independent parts has apparently been used successfully in the hardware-varification community. At the present time, we have not been able to find better partioning schemes that ensure optimality.

Incremental BDDs Algorithm 5, and thus algorithm 7, can easily be modified to avoid repeated re-specialisation of the initial constraint set B_C by maintaining a state between user interactions. If the algorithm remembers the last configuration $conf_{i-1}$ as well as bdd_{i-1} resulting from specialising bdd' w.r.t. the inferred configuration (cf. algorithm 5) between each interaction with the user, then bdd_{i-1} can be used instead of B_C as long as each received configuration is a refinement of the previous one. Indeed, only the difference between configurations needs to be transmitted between the user and the inference algorithm. The constraint set thus decreases in size with each refinement.

Of course, when the algorithm receives a configuration that is *not* a refinement of the previous one, it is necessary to restart and use B_C again.

Another merit of using the BDDs incrementally, is that default select indications could be represented as a total, satisfiable configuration D : When the user has finished his selections, the defaults D are used to further specialise the constraint set. Since no user-(de)selected entity is mentioned in the specialised constraint set, the default indication for such an entity is effectively ignored; only entities deferred by the user will be affected by the default indications.

Explanations Providing the user with an explanation of why a particular selection is unsatisfiable is also easier when the algorithm maintains a state (cf. previous section). Consider that the configuration $conf_{i-1}$ was accepted, but $conf_i$ is not accepted. In this case, the variables that constitute the difference between the configuration

$$\text{diffvars } conf\ conf' \stackrel{\text{def}}{=} \left\{ E \middle| conf(E) \neq conf'(E) \right\} \tag{13}$$

can be used to provide an explanation of why a configuration could not be accepted: The initial constraint set specialised to the most specific generalisation of the two configurations

$$bdd_{\text{msg}} = \text{restrict } B_C \ (\text{msg } conf_{i-1}\ conf'_i) \tag{14}$$

can be "existentially qualified" w.r.t. to the changed variables:

$$bdd_{\text{explanation}} = \text{exist } bdd_{\text{msg}} \ (\text{diffvars } conf_{i-1}\ conf_i)) \ . \tag{15}$$

Example 3. Consider the constraint set (6) and assume the user has first made the selections

$$x : \textbf{deselected}\ y : \textbf{selected}\ (\text{the rest } \textbf{unspecified})$$

which succeeds. Assume now that the user changes his mind w.r.t. y and makes additional selections

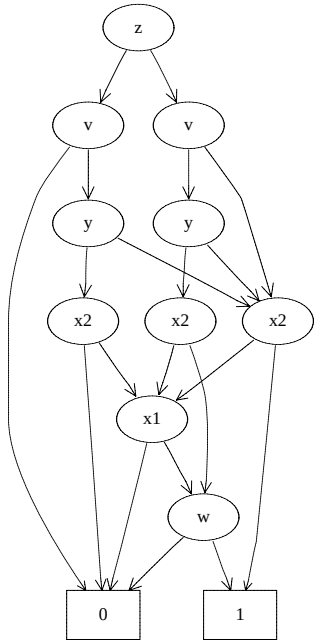
$$x : \textbf{deselected}\ y : \textbf{deselected}\ z : \textbf{selected}\ x_2 : \textbf{deselected}$$
(the rest **unspecified**)

which leads to failure. The msg of the two configurations is simply

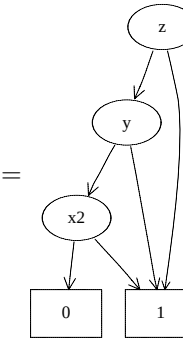
$$x : \textbf{deselected}\ (\text{the rest}\ \textbf{unspecified})$$

and the set of different variables is $\{y, z, x_2\}$. Hence,

$bdd_{\text{msg}} =$



$bdd_{\text{explanation}} =$



$bdd_{\text{explanation}}$ could then easily be converted to

$$z \Rightarrow (\neg y \Rightarrow x2)\ ,$$

which can be post-processed to the more understandable

$$z \Rightarrow y \vee x2\ .$$

5 Related Work

The problems addressed in this work arose in the PROMIS project [8], and the corresponding solutions were developed based more or less on the work in the

AnnoDomini project [12, 13]. The paper [27] reports on a pre-study of a project with similar aims as the PROMIS project.

There is a significant literature on configuration, in particular a body of papers that view the configuration problem as a *constraint satisfaction problem* (CSP)—see, e.g. [31, 39]. The idea is that the configuration problem is formalized as the CSP (X, D, C) where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = D_1 \cup \ldots \cup D_n$ is a domain made up of domains D_i from which the values of variable x_i must be drawn, and where C is a set of constraints limiting the possible values of $x_1, \ldots, x_n$. The setting of the present paper can be seen as the special case $D_1 = \ldots = D_n = \{\mathbf{selected}, \mathbf{deselected}\}$.

However, this formalization does not address the aspects related to the dialogue between the user and the system. We are not simply interested in a solution to (X, D, C). We are also interested in knowing if there is more than one solution, and if some values for variables are the same in all solutions given the user's current selections, and so on. Such issues are addressed in [1], which has similar aims as the present paper. Whereas we consider the user's current selection as a configuration, the former paper formalizes it as a set of unary constraints H. The problem then is to find a subset E of H such that C and H together have a solution. The user assigns weights to the constraints in H indicating how desirable it is for him that a given constraint in H is satisfied, and this is used to find an E with maximal "customer satisfaction."

To this end the CSP is compiled into a finite automaton using the technique described in [38]. This compilation is done off-line, i.e. before questions are asked concerning the constraints, though the automaton must be updated when the set H changes. In principle, the varibles of X are ordered in some way, say, $x_1, \ldots, x_n$, and a search tree is constructed where the nodes at level i correspond to the variable x_i, and where the arcs out of each node correspond to the possible values for the variable of the node, given the values for $x_1, \ldots, x_{i-1}$ determined by the path from the root to the node. This search tree can then be turned into a DFA or NFA, and questions concerning the original constraint set can be rephrased as questions pertaining to the automaton. These questions (at least some of them) can be answered in linear time in terms of the size of the automaton. However, this size can be exponential in terms of the constraint set.

In the present setting, the search tree amounts to a binary tree encoding all the possible valuations of a constraint, and the more compact NFA or DFA is in our setting a binary decision diagram.

A main difference between the work described in [1] and the present line of work is that Amilhastre and Fargier consider the situation that the user has made an inconsistent selection, and they use the weights assigned by the user to the constraints H to find a relaxation of the user constraints that is consistent. In contrast, in the present line of work the user is presented with the conflicts, and he must then resolve the conflicts in some way himself. Both techniques seem to have advantages and disadvantages. This means that the questions asked by Amilhastre and Fargier concerning a given constraint set are somewhat different

than the questions asked in this paper, though some of the questions are the same or closely related.

Another difference is that we consider boolean domains, whereas Amilhastre and Fargier consider more general domains. Again, both techniques seem to have advantages and disadvantages.

Gelle and Weigel [18] represent constraints by relations containing all tuples that satisfy the constraints, and argue that this representation is easier to maintain than a representation using explicity rules. In our setting this corresponds to representing constraints by truth tables.

In constraint terminology, *global consistency* means that for every variable in the CSP, there exists some value for it that participates in a solution to the problem. In our setting, this translates into plain consistency. In constraint terminology one also considers the notion of *arc-consistency* (see [2] for a good survey) meaning, roughly, that for every constraint C involving variables x and y, and every value d_x currently considered valid for x, there exists a value d_y considered valid for y, such that the constraint C is satisfied. Arc consistency algorithms remove elements from the domains so as to ensure that all constraints are arc-consistent. Indeed, Algorithm 3 can be seen as a variation of the arc-consistency algorithm AC-3 [23, 22], where we consider n-ary constraints rather than binary ones, but where the domains are boolean rather than arbitrary finite domains.

There are other techniques for testing satisfiability, notably the *Davis-Putnam procedure* [10, 9]. A comparison between BDD-based techniques and the Davis-Putnam procedure appears in [37].

The company ConfigIt Software (www.configit-software.com) has developed a so-called *virtual table* technique [26] that seems to address the inference problems we have stated here. Their technique is not well described, however, since they have a pending patent on it.

Our inference algorithm on BDDs is akin to the Dilemma Rule used in Stålmarck's proof procedure for propositional logic [34].

6 Future Work

A natural next step of this work is to develop an implementation of the techniques described in the paper using efficient techniques, e.g. based on compilation to binary decision diagrams, to conduct experiments with a realistic case study, and to compare the performance to related techniques.

It would also be interesting to consider subclasses of the general set of contraints for which more efficient optimal algorithms can be devised, e.g. the class of horn formula—see [3]. Another idea is to refine Algorithm 3 inspired by other arc-consistency algorithms [25, 28, 19, 4, 5].

Acknowledgments. This paper grew out of work in the PROMIS project, which is joint work between the following companies: Terma A/S (Denmark), MSI and Thomson CSF (France), Intracom and HAI (Greece), Datamat and Marconi

(Italy), and MRC (Turkey). The authors are indebted to Steffen Mogensen, Terma A/S, for raising some of the issues discussed in the present paper and for discussions about them.

The solutions in this paper were more or less based on the work in the Anno-Domini project, which is joint work with Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, and Mads Tofte.

References

[1] J. Amilhastre and H. Fargier. Handling interactivity in a constraint-based approach of configuration. In *Configuration Workshop at the 14th European Conference on Artificial Intelligence*, pages 7–12. Electronic proceedings available at url `www.cs.hut.fi/ pdmg/ECAI2000WS/Proceedings.pdf`, 2000.

[2] R. Barták. Theory and practice of constraint propagation. In *Proceedings of the 3rd workshop on Constraint Programming for Decision and Control*, 2001.

[3] K. A. Berman, J. Franco, and J.S. Schlipf. Unique satisfiability of horn sets can be solved in nearly linear time. In *Discrete Applied Mathematics*, volume 60, pages 77–91, 1995.

[4] C. Bessiere. Arc-consistency and arc-consistency again. *Aartificial Intelligence*, 65:179–190, 1994.

[5] C. Bessiere, E.C. Freuder, and J.-R. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Aartificial Intelligence*, 107:125–148, 1999.

[6] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] J.F. Challine. *PROMIS Concluding Report - Summary of the Study.* Thomson-CSF Communications, 2000.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[11] R. Drechsler, N. Göckel, and B. Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 730–739, Berlin, 1996. Springer.

[12] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H.B. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM press, 1999.

[13] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H.B. Sørensen, and M. Tofte. AnnoDomini in practice: A type-theoretic approach to the year 2000 problem. In *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science, pages 6–13. Springer-Verlag, 1999.

[14] B. Faltings and E. Freuder, editors. *Configuration. Papers from the 1996 Fall Symposium.* Number FS-96-03 in AAAI Fall Symposium Series. The AAAI Press, 1996.

[15] B. Faltings, E.C. Freuder, G. Friedrich, and A. Felfernig, editors. *Configuration. Papers from the AAAI workshop.* Number WS-99-05 in AAAI Workshop technical report series. The AAAI Press. Electronic proceedings available at url `wwwold.ifi.uni-klu.ac.at/ alf/aaai99/index.html`, 1999.

[16] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.

[18] E. Gelle and R. Weigel. Interactive configuration using constraint satisfaction techniques. In Faltings and Freuder [14], pages 37–44.

[19] P. Van Hentenryck, Y. Deville, , and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[21] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, January 1993.

[22] A. K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[23] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[24] E. Mendelson. *Introduction to Mathematical Logic.* Wadswoth & Brooks/Cole Advanced Books and Software, third edition, 1987.

[25] R. Mohr and T.C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.

[26] J. Møller, H. R. Andersen, and H. Hulgaard. Product configuration over the internet. In *Proceedings 6th International INFORMS Conference on Information Systems and Technology*, Miami Beach, Florida, November 2001.

[27] K. Osvärn and O. Hansson. Prestudy of configuration of a naval combat management system. In Faltings and Freuder [14], pages 138–139.

[28] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[29] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE /ACM International Conference on CAD*, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[30] J. Rahmer, A. Boehm, H.-J. Mueller, and St. Uellner. A discussion of internet configuration systems. In Faltings et al. [15], pages 138–140.

[31] D. Sabin and E. Freuder. Configuration as composite constraint satisfaction. In Faltings and Freuder [14], pages 28–36.

[32] J. P. Secher and M. H. Sørensen. From checking to inference via driving and dag grammars. In Peter Thiemann, editor, *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 41–51. ACM Press, 2002.

[33] J.P. Secher. Driving in the jungle. In Olivier Danvy and Andrzej Filinski, editors, *Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217. BRICS, Springer-Verlag, May 2001.

[34] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design: An International Journal*, 16(1):23–58, January 2000.

[35] H. Sørensen, M and R. Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.

[36] V.F. Turchin. Semantic definitions in Refal and the automatic production of compilers. In N.D. Jones, editor, *Workshop on Semantics-Directed Compiler Generation, Århus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 441–474. Springer-Verlag, 1980.

[37] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J. P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49. Springer-Verlag, 1994.

[38] N. R. Vempaty. Solving constraint satisfaction problems using finite state automota. In W. R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 453–458. American Association for Artificial Intelligence press/MIT press, 1992.

[39] M. Veron, H. Fargier, and M. Aldanondo. From CSP to configuration problems. In Faltings et al. [15], pages 101–106.

[40] I. Wegener. *The Complexity of Boolean Functions.* Wiley Teubner Series in Computer Science. John Wiley and Sons, New York, 1987.