ELSEVIER

The Journal of Logic Programming 37 (1998) 255–283

THE JOURNAL OF LOGIC PROGRAMMING

Mechanising shared configuration and diagnosis theories through constraint logic programming

Nirad Sharma *, Robert Colomb

Department of Computer Science, The University of Queensland, Brisbane, Qld. 4072, Australia

Received 30 September 1996; received in revised form 3 May 1997; accepted 25 February 1998

Abstract

Configuration and diagnosis problem-solvers are commonly championed as successes of applied artificial intelligence techniques. A common problem is that problem-solvers typically encode task-specific representation assumptions and simplifications in their domain theories, hindering the reuse of the domain theories between the problem-solvers. While model-based reasoning techniques have been shown to provide an interesting approach to sharing component and device specifications, their respective mechanisations are generally too inefficient. We show how constraint logic programming languages provide a flexible environment in which constraint-based specifications can be effectively shared and efficiently mechanised by exploiting constraint solving and propagation techniques tightly integrated with the backtracking search mechanism of logic programming languages. A component specification language is presented and the mappings from the language to the constraint system and strategies for guiding the search are defined for the respective problem-solvers.

Keywords: Consistency-based diagnosis; Constraint-based configuration; Constraint logic programming; Knowledge representation

1. Introduction

An important engineering principle is that a device design embodies a behavioural and structural specification that is optimal w.r.t. some criteria and that an instance of that device obeys these constraints, violations of which should be characterisable by minimal covering sets of offending components. The respective mechanisations are termed the configuration and diagnosis tasks. As devices have become more complex in their structures and behaviours as have the constraints that govern their allowable such forms, the construction of automated diagnosis and configuration tools to help manage the complexity has become increasingly important. Constraint-based

* Corresponding author. Present address: Trilogy Development Group, 6034 W Courtyard Drive, Austin, TX 78730, USA. Tel.: +1 512 794 5900; fax: +1 512 794 8900; e-mail: Nirad_Sharma@trilogy.com.

0743-1066/98/$ – see front matter
PII: S0743-1066(98)10010-9

component specifications facilitate a more natural and maintainable formulation than do traditional rule-based forms by not encoding the problem-solving logic in the model formulation, improving the potential for model sharing. Design descriptions are often available as a result of the design process, readily providing the models for automated configuration assembly and diagnosis engines.

There are significant benefits in being able to exploit device and component specifications used for configuration in a diagnosis problem-solver. Apart from amortising the cost of knowledge elicitation across the target problem-solvers, knowledge-sharing facilitates consistent characterisation of diagnoses for configured devices. There is intuitive overlap in the knowledge requirements of the two tasks. Configuration can be seen as the generation of models that satisfy a finite set of constraints while attempting to optimise some cost criteria. The diagnosis task is the dual in that given a submitted model and set of observations, diagnoses are the minimal sets of constraint violations that explain the faulty observations. Domain model formulations typically embed task-specific assumptions in their representations, however, requiring reformulation of the problem-solvers for the separate tasks.

While model-based reasoning techniques for both diagnosis and configuration provide an interesting approach to sharing component and device specifications, their respective mechanisations are generally too inefficient. In earlier work [25], an approach was presented for providing a common semantic foundation for the diagnosis and configuration tasks where constraints were expressed in a first order logic with distinguished abnormality and connectedness predicates. While an interesting semantic characterisation, the specifications are not directly mechanisable, motivating the investigation of some intermediary language. The design of such a language is a classic knowledge representation design trade-off between expressiveness and efficiency of computability, exploiting properties of the problem domain.

This paper presents such a representation language and an approach for an effective mechanisation of the diagnosis and configuration tasks from a shared component specification using a constraint logic programming system. We show how constraint logic programming languages, particularly ECLiPSe [5], provide a flexible environment in which constraint-based component specifications can be effectively shared and efficiently mechanised for the diagnosis and configuration tasks by exploiting constraint solving and propagation techniques as integrated with the backtracking search mechanism of a logic programming language.

An internetwork cabling scenario is introduced in Section 1. The forms of diagnosis and configuration reasoning to be considered are characterised in Section 2. Constraint logic programming is introduced in Section 3 as an enabling technology for the construction of practically mechanisable problem-solvers from constraint-based specifications. Section 4 presents our component specification language, motivating the key features while Section 5 discusses the relationship between the language and its encoding for the problem-solvers. Implementation experiences are presented in Section 6, highlighting some avenues for further research. Section 7 relates the work to other systems in the literature.

1.1. An internetwork cabling scenario

A common problem when setting up an internetwork as illustrated in Fig. 1. is the proper assembly of the various components to satisfy the constraints of the devices

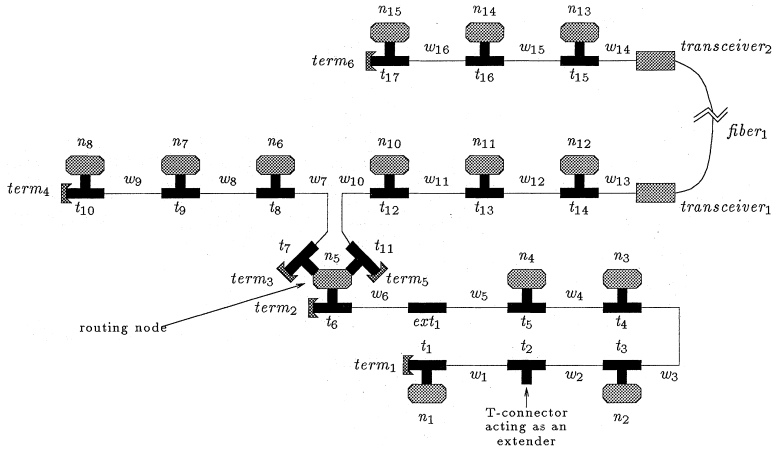

Fig. 1. Sample network to be constructed or diagnosed.

and such requirements as inter-node sequencing and spacing while minimising the expense of the total configuration. Similarly, due to the complexity of such artifacts, locating a fault as characterised by a minimal set of devices that are fault candidates in an internetwork such as where two nodes cannot communicate with each other due to a misconfiguration can be a complex task.

Our cabling scenario is based on ethernet devices. An ethernet segment is a bi-terminated linear form constructed from cabling connected by t-connectors to which nodes are connected. Constraints exist over ethernet segments such as there being a maximum number of nodes that may be connected and the maximum length of cabling in a segment. A fiber-optic channel coupled with transceivers can be used to extend the maximum distance for a segment as can a repeater or bridge. An internetwork consists of a collection of ethernet segments such that every ethernet segment is connected to at least one other by a router. A router is a node that can connect to more than one ethernet segment and thus to more than one t-connector.

The configuration task is to generate the consistent component connectivity configurations such as illustrated in Fig. 1 that minimise some cost criteria given a set of sequences of nodes to be cabled and minimum total cabling lengths between adjacent node pairs. Fig. 2 illustrates such a subnetwork where nodes are to be cabled. For example, the distance between nodes n_4 and n_5 must be at least nine units. The diagnosis task is the identification of components of a device that have been misconfigured w.r.t. their component specifications.

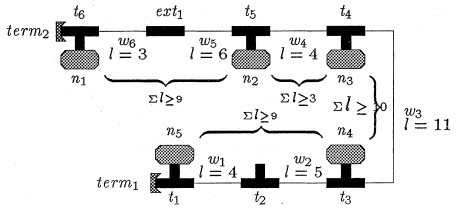

Fig. 2. Sequenced nodes and inter-node constraints on an ethernet segment.

2. The tasks

Before a shared representation language can be designed, a characterisation of the classes of configuration and diagnosis problems to be considered is required. Component knowledge is captured with constraint-based representations rather than through rule-based experiential forms. As the former is a declarative, non-deterministic representation, constraint-based forms are more ably re-interpreted in different contexts whereas rule-based forms incorporate task-specific assumptions by virtue of having to encode problem-solving methods in the representation.

As is typical in the constraint-based configuration literature, "behaviour" considered here will be in terms of properties held by components. If behaviour as propagatable input–output mappings (e.g. [20,25]) were being considered, the complexity of the configuration synthesis task would be beyond the scope of this paper as the problem reduces to the problem of optimising component interconnections to emulate combinatorial behaviour, an NP-hard task.

2.1. Configuration

The configuration task [18] involves the generation of one or more configurations that satisfy a specification of the desired configuration description given a finite set of components, a set of component interconnectivity constraints, and (optionally) optimality criteria. The problem is important as given the increasing complexity of components and devices' internal interconnectivities, automated solutions to a configuration specification can provide more timely and less error-prone solutions than when done manually. The complexity of the task arises from the inherently combinatorial solution spaces that must be managed during search due to component interconnectivity symmetries and typically large numbers of inessential variants.

The earliest and most prominent configurators such as DEC's R1/XCON [15] were rule-based (or heuristic) systems. An advantage of rule-based forms is that control knowledge can be encoded in the formulation of the rule base, providing a mechanism to heuristically manage search through the combinatorial search spaces. Such representations suffer from major maintenance problems, however, as this conflation of domain and control knowledge means it is typically difficult to manage changes to the knowledge base due to the amorphous entanglement of control flow that often results. The natural structure of the device and its components is lost when their descriptions are reduced to the uniform representation of experiential rules. Reuse of the knowledge for different tasks is correspondingly more infeasible.

A constraint-based representation facilitates a more declarative representation of the domain models by explicating the structure of the device and components as well as constraints over their properties and interconnectivity. Examples of constraint-based configurators include Cossack [6], BEACON [22] and LCON [26]. A naïve formulation of the problem as a bottom up generate-and-test approach can result in a space of $\sqrt{(np)!}$ possible configurations given n components with p connection ports per component [18]. Techniques for efficiently reasoning with constraint-based configuration problems include constraint propagation, the key-component assumption [18] and formulation as a dynamic constraint satisfaction problem [16] where additional variables and constraints are posted/retracted as components are added to/removed from the design. One of the goals of this work is to show that not only do

CLP languages provide an effective framework in which to capture shareable component specifications but that constraint-based configurations are feasibly mechanised using a CLP language by virtue of the constraint propagation and solving facilities (Section 3).

Specification of desired configurations in our approach is through the declaration of a set of partially instantiated component types $\mathcal{SPEC}$ (Definition 2.1). Given $\mathcal{SPEC}$, the configuration task involves finding appropriate valuations for the component variables of $C \in \mathcal{SPEC}$, an important part of which involves establishing component connectivities that satisfy component type topologies and other constraints due to the hierarchical decomposition of the component types of $\mathcal{SPEC}$. The class of configuration problems handled in our framework can be characterised as those that can be represented as partial instantiations of aggregate component types in the component language, detailed in Section 4 (Fig. 3).

Definition 2.1. A *configuration specification* is a tuple $\langle\langle \Sigma, \mathcal{C}, \mathsf{type}\rangle, \mathcal{SPEC}\rangle$ where Σ is the set of component type specifications, $\mathcal{C}$ is a finite set of available component instances each of some type in Σ, $\mathcal{SPEC}$ is a finite set of (partially instantiated) component instances ($\mathcal{SPEC} \subseteq \mathcal{C}$) and the mapping $\mathsf{type} : \mathcal{C} \to \Sigma$ associates component individuals to their type.

The internetwork of Fig. 1 consists of three ethernet segments forming the edges of a hypergraph with nodes being network nodes. Each ethernet segment consists of a sequence of nodes with a minimal distance between them (Fig. 2). The configuration task is to fully instantiate `internetwl` ($\mathcal{SPEC} = \{\texttt{internetwl}\}$) and its constituents using the available components $\mathcal{C}$ while conforming to the appropriate constraints in the component type specifications Σ.

The configuration task as considered here involves dynamic constraint satisfaction over the uninstantiated component variables in the components of $\mathcal{SPEC}$ such that instantiation of port variables may cause new components to be connected with their corresponding variables added to the set of variables undergoing constraint satisfaction. The constraint satisfaction process is presented in Section 5.1.

2.2. Diagnosis

The task of a diagnosis problem-solver is to identify the sets of components that characterise conjectured sources of any discrepancies between a set of observations and the predicted behaviour for a system, commonly some real-world artifact such as a physical device. Models of behaviour as explicit mappings will not be considered here for homogeneity with the configuration task but the approach is simply extended to do so. As devices become more complicated in the number of and types of constraints describing their internal interconnections, automated diagnosis tools are of increasing importance. Similarly to the configuration task, heuristic and model-based reasoning are the two principal approaches to the diagnosis task.

The heuristic approach, typically modelled using rule-based technologies as characterised by early diagnosis systems such as DENDRAL and MYCIN, encode experts' ''rules of thumb'' for how a system would typically fail. While frameworks such as heuristic classification [2] evolved for explicating the diagnosis reasoning strategy of rule-based diagnosis systems, the major problems that arise are those

of robustness and maintainability. The time and cost of accumulating enough expertise in a rule-based diagnosis system that adequately covers the range of possible faults may be prohibitive for non-trivial artifacts. A maintenance nightmare can exist due to rule interactions for even a moderately sized amorphous rule base. A further consequence of capturing experiential knowledge as rules is that the structure of the device and its components is not explicitly modelled. This makes transfer of the model between systems for slightly modified devices or, of particular relevance to this paper, companion configuration systems virtually impossible.

Model-based approaches [3,7], often referred to as "reasoning from first principles", involve the identification of discrepancies between the observed state of the system and that expected based on the structural model of the system. Naturally captured using constraint-based systems, consistency-based systems (e.g. [20]) conjecture that the components will typically adhere to their structural constraints, presuming minimal abnormality. The diagnosis task is the calculation of the minimal sets of components whose abnormality adequately explains the discrepancies.

For the purposes of this paper, the kinds of fault scenarios for the internetwork of Fig. 1 could include the following.

- Node n_4 cannot communicate with node n_{12} due to a t-connector being used as a cable extender having a cable attached to each of its three connectors.
- Node n_2 cannot communicate with node n_5 due a terminator missing from their subnetwork.
- Node n_7 cannot communicate with node n_9 due to a break in the cabling on their subnetwork.

Each of these faults is representative of a fault due to some component(s) not conforming to their structural constraints. The approach to diagnosis described in this work is thus the identification of misconfigured devices. We briefly present the framework through Definitions 2.2–2.4.

Definition 2.2. A *diagnosis specification* is a tuple $\langle\langle \Sigma, \mathscr{C}, \mathsf{type}\rangle, \mathscr{OBS}\rangle$ where Σ is the set of component type specifications, $\mathscr{C}$ is a finite set of the component instances each of some type in Σ, $\mathscr{OBS}$ is a finite set of component instances to be assessed for consistency ($\mathscr{OBS} \subseteq \mathscr{C}$) and the mapping $\mathsf{type} : \mathscr{C} \to \Sigma$ associates component individuals to their type.

While typical approaches to model-based diagnosis such as Reiter's diagnosis from first principles [20] are used to diagnose models of combinatorial behaviours such as the binary adder, the techniques can be applied just as effectively for models with no explicit models of behaviour as in the scenario presented in Section 1. The key principle is abnormality minimisation given a system description SD, a set of components COMP and observations OBS.

Definition 2.3. A *diagnosis* Δ for $\langle\langle$ SD, COMP $\rangle$, OBS $\rangle$ is a minimal subset $\Delta \subseteq$ COMP s.t. the following is consistent

$$\mathrm{SD} \cup \mathrm{OBS} \cup \{ab(c) \mid c \in \Delta\} \cup \{\neg ab(c) \mid c \in \mathrm{COMP} - \Delta\}.$$

The symbol $\underline{\Delta}$ denotes the set of all Δ for a system. The principle of Definition 2.3 (due to [20]) is to characterise the minimal sets of abnormal components that cover

all possible diagnoses. We modify this notion slightly to consider only those diagnoses of minimal *cardinality*, denoted $\underline{\Delta_{\text{Card}}}$.

Definition 2.4. $\underline{\Delta_{\text{Card}}} \subseteq \underline{\Delta}$ s.t. $\forall \Delta_1 \in \underline{\Delta_{\text{Card}}} \nexists \Delta_2 \in \underline{\Delta_{\text{Card}}} \cdot |\Delta_2| < |\Delta_1|$.

This simplification [25] can be justified due to the smallest set of faulty components typically being the first candidates for testing/rectification. Further, $\underline{\Delta_{\text{Card}}}$ is more simply calculated in the CLP($\mathscr{FD}$) framework (Section 5.2.2). Should $\underline{\Delta}$ be required, however, it is naturally reconstructed by iterated calculation of $\underline{\Delta_{\text{Card}}}$ successively assuming normality of candidate components (Theorem 4.4 of [25]).

In the presence of behaviours (e.g. [20]), the combinatorial nature of the behaviours due to component interconnectivity enables the elimination of certain components as fault candidates even with incomplete behaviour observations. Without behaviours, however, less information is available to hasten the elimination of candidate components. When detecting misconfiguration, however, this is of less importance as complete connectivity observations are available with only component attribute variables possibly non-ground due to absent or incomplete information.

3. Constraint logic programming

Constraint logic programming (CLP) languages [11,12] are a generalisation of conventional logic programming languages that extend the notion of term unification by constraint solving over domain variables in a computational domain, replacing the calculation of a most general unifier [13] with the notion of constraint satisfiability. The CLP scheme [11] actually characterises a family of languages parameterised by the algebra formed from the domain of computation and the primitive constraints. An instantiation of the CLP scheme yields a particular CLP language. CLP languages are typified by domain-constrained variables and incremental constraint solving and are thus typically well-suited to the implementation of combinatorial problems due to the tight integration of variable and constraint posting and retraction with the (chronological) backtracking search mechanism of the logic programming paradigm.

For the purposes of this paper, a program in a CLP language is a finite collection of rules of the following form:

$$H \leftarrow A_1 \wedge \cdots \wedge A_m \wedge c_1 \wedge \cdots \wedge c_n$$

where $H, A_1, \ldots, A_m$ are atoms and $c_1, \ldots, c_n$ are constraints in the constraint system. Following the convention of [10], the current state of a constraint logic program is captured by the pair $\langle B, \sigma \rangle$ where B is the conjunction of goals to be solved and σ is the conjunction of constraints accumulated in the constraint store. As a goal is unfolded using such rules and variable substitutions are applied, the atoms $A_1, \ldots, A_m$ and constraints $c_1, \ldots, c_n$ are added to B and σ, respectively, with the former reduced only through successful atom grounding while σ will be augmented or reduced depending on tests carried out by the constraint solver. Backtracking occurs when either a goal or the constraint store becomes unsatisfiable.

The power of a CLP language comes from the constraint solver which is typically required to support the following operations on its constraint system $\mathscr{C}$ (where $\tilde{x}$, $\tilde{\exists}\phi$

and $\mathscr{C} \models \phi$ denote respectively a tuple of distinct variables, existential closure over the formula ϕ and that all models of $\mathscr{C}$ are models of ϕ).

- *Satisfiability*: If $\mathscr{C} \models \tilde{\exists}\sigma$ fails, no satisfiable valuation for the constraint store σ exists and so the most recent application to σ should fail.
- *Entailment*: If for a new constraint c, $\mathscr{C} \models \sigma \rightarrow c$ holds, c will not strengthen σ.
- *Generalisation*: A generalisation c can be computed for a set of (possibly conjunctions of) constraints $\{c_1, \ldots, c_n\} \subseteq \sigma$ if $\mathscr{C} \models c \rightarrow c_i$ $(1 \leqslant i \leqslant n)$.
- *Grounding*: A variable x is *grounded* in constraint c if $\mathscr{C} \models \sigma \rightarrow [\exists z\ \forall x\tilde{y}\ c(x, \tilde{y}) \rightarrow x = z]$, that is there is only one consistent valuation for x w.r.t. c.

Naïvely, constraint satisfaction methods such as simplex might be considered however such techniques are best suited for static sets of constraints. In a CLP program, it is important to be able to curtail execution dynamically due to the dynamic nature of a constraint store σ which incrementally accumulates constraints as goal resolution progresses. Incremental constraint solving techniques as introduced in [14] are more appropriate due to the requirement for repeated applications of the constraint operations described. As many constraint operations such as satisfiability and entailment are NP-complete for interesting CLP instantiations such as finite domains, such constraint solvers are necessarily incomplete and often use approximation techniques such as interval reasoning.

The work presented in this paper will use the finite domains package of the ECLiPSe [5] CLP environment, subsequently referred to as CLP($\mathscr{FD}$). In terms of the CLP scheme of [11], CLP($\mathscr{FD}$) can effectively be characterised by a computation domain of the integers $\mathbb{Z}$ with the generators for terms $\Sigma_{\text{term}} = \{+, *\}$ and relations $\Sigma_{\text{rel}} = \{\{\in [m, n]\}_{m \leqslant n}, =, \neq, \leqslant\}$ where $x \in [m, n]$ denotes $m \leqslant x \leqslant n$, the other symbols have their usual meaning and all domain variables participate in at least one $\in [m, n]$ constraint. The actual constraints directly supported in CLP($\mathscr{FD}$) are presented in Table 1. While many of the constraints may be seen as syntactic sugar over the base constraints, the representation of constraints using such abstractions is a well-established means for enabling constraint-solvers to take advantage of the stronger properties held by and more concise representation of the constraints than if reasoning in the base primitive constraints were required.

Consider the following simple example using CLP($\mathscr{FD}$):

```
[X, Y, Z]::[1..10],
X + Y#=10, X - Y#=Z, X*Y#>20,
doit(X, Y, Z),
```

Before `X`, `Y` and `Z` are unified in `doit(X, Y, Z)`, the constraint solver establishes by constraint propagation that the only satisfiable values for `(X, Y, Z)` are (6,4,2) and (7,3,4), obviating the need to exhaustively enumerate each of the 10^3 possible valuations for the variable tuple `(X, Y, Z)`. Search space pruning is effected through constraint solving, extending the flexibility of LP languages for constraint-based specification of solution spaces by avoiding the costs of "generate-and-test" cycles of conventional LP treatments. The principle is to defer choice (variable grounding) to as late as possible to reduce the amount of expensive backtracking required.

Extensible CLP environments such as ECLiPSe extend such reasoning to constraints such as simple cardinality bounds for value occurrence in a set of variables and boolean connectives through encoding as numerical inequalities (Table 1). For example, representing booleans as domain variables ranging over $\{0, 1\}$, the constraint $b_1 \vee b_2$ is captured as the numeric inequality `B1 + B2 #>= 1` where `B1` and

Table 1
Principal ECLiPSe CLP ($\mathscr{FD}$) constraints

CLP($\mathscr{FD}$)	Semantics
Term comparisons	
T1 # \= T2	$t_1 \neq t_2$
T1 #< T2	$t_1 < t_2$
T1 #<= T2	$t_1 \leqslant t_2$
T1 #= T2	$t_1 = t_2$
T1 #> T2	$t_1 > t_2$
T1 #>= T2	$t_1 \geqslant t_2$
Propositional expressions	
# \+ Phi	$\neg\phi$
Phi #/\ Psi	$\phi \wedge \psi$
Phi #\/ Psi	$\phi \vee \psi$
Phi #=> Psi	$\phi \subset \psi$
Phi #<=> Psi	$\phi \equiv \psi$
Miscellaneous constraint predicates	
V::[Vl,...,Vn]	$\bigvee_{v\in[v_1,\ldots,v_n]} V = v$
V::[Vl,...,Vn]	$\bigvee_{v\in\{v_1,\ldots,v_n\}} V = v$
alldistinct([Vl,...,Vn])	$\bigwedge_{i,j\in[1..n],i\neq j} V_i \neq V_j$
atmost(N,[Vl,...,Vn],Val)	$\sum_{i\in[1..n]} b_i \leqslant N$, where $b_i = \begin{cases} 1 & \text{if } V_i = Val \\ 0 & \text{otherwise} \end{cases}$
#(I, [Phil,...,Phin], J)	$I \leqslant \sum_{i\in[1..n]} b_i \leqslant J$, where $b_i = \begin{cases} 1 & \text{if } \sigma \models \phi_i \\ 0 & \text{otherwise} \end{cases}$

B2 capture the respective boolean truth values of propositions b_1 and b_2. This treatment enables complex constraints to be constructed and reasoned within a homogeneous constraint framework.

While conventional Prolog-style representations are declarative, determinism is inherent to the operational semantics for non-definite programs. Further, the inability to reason from unsatisfiability of the head of a rule to its tail means that while the form a:- b, c may naturally encode the intuition of consequence for the classical form $b \wedge c \subset a$, the former is not a complete representation of the latter as it precludes application of *modus tolens* for the deduction $\{b \wedge c \subset a, \neg a\} \vdash \neg(b \wedge c)$. Such reasoning is particularly important for efficient model computation and constraint satisfaction as it is capturing the idea of domain reduction and constraint simplification. Thus, an additional benefit of encoding the propositional constraints algebraically as numeric inequalities is a more natural and flexible interpretation of the constraints due to its non-deterministic form.

The key contributions of CLP to the mechanisation of the shared theories for the tasks of Section 2 are the following.

1. Support for the representation of constraints written using propositional connectives with classical interpretation, thus supporting bidirectional constraint propagation. As the constraints are reduced to algebraic inequalities over finite

domains, efficient constraint solving procedures can be used to reason with this declarative representation.
2. Integration of the posting and retraction of active domain variables and constraints with backtracking search of a logic programming paradigm facilitates more efficient management of inherently combinatorial search spaces for constraint-based configuration tasks than would a classical constraint-satisfaction formulation [16].
3. As logical constraints are encoded as numeric inequalities, the truth or falsehood of a constraint can itself be encoded as a domain variable, providing a natural treatment of constraints for consistency-based diagnosis reasoning.

4. Component specification

Recall that the principal goal of this work is to facilitate sharing of component specifications between the configuration and diagnosis tasks such that the respective problem-solvers can be mechanised reasonably efficiently. While the work presented in [25] provides an interesting semantic characterisation of the tasks for shared component specifications, the approach is not directly mechanisable due to the intractability of computing in a framework with the full generality of first order logic. For example, while model generation from a set of first order formulae over a finite domain is decidable, it is intractable in the general and, in our experience, typical cases. The task here, then, is the classic knowledge representation language design trade-off of designing a representation language expressive enough to represent a significant class of scenarios for the tasks of interest while being computationally tractable for these tasks and having a clean semantics. Fundamental to the design of the language is recognition and explication of the structure inherent in the domain of discourse and a natural mapping to CLP($\mathscr{F}\mathscr{D}$) facilities.

A representation language for capturing constraint-based component type specifications is presented in Fig. 3. [1] Individual components are organised into *component types*. A component type has associated with it a collection of named *ports* or *port sets* that are abstractions of connectivity between components and are thus associated with a set of component types that can permissibly be connected to the particular port(s). *Attributes* are properties of a component captured as named variables that are assigned a value or are merely restricted in range (*ground_flag* in Fig. 3) during constraint satisfaction and, as such, are domain variables with standard CLP domains. The following examples illustrate these notions:

Example 1.

```
component_type wire {
   isa:          [top]
```

[1] Conventional regular expression notation has been employed. $[\theta]$, $\{\theta\}$, θ^* and θ^+ denote sequences of tokens of length $\leqslant 1$, 1 (scope clarification), $\geqslant 0$ and $\geqslant 1$, respectively. $\theta_1 \mid \theta_2$ denotes choice between the tokens θ_1 and θ_2. Terminals (or n-ary functors) appear in sans serif face. The non-terminal tokens *clp_domain*, *clp_constr*, *clp_atom* and *value* (numeric constant) are grounded in the syntax of the CLP system of implementation.

```
device_spec      == comptype_spec+ compindiv_spec*

comptype_spec    == component_type comptype {
                      isa:                 comptype*
                      ports:               {portname: n × comptype+}*
                      attributes:          {attrname: clp_domain, ground_flag}*
                      {
                         cost:             name_var*, clp_expr
                         local_constraints: {extname_var*, clp_constr}*
                      } | {
                         aggregates:       aggr_spec
                         aggr_constraints: aggr_constr*
                      }
                   }

aggr_spec        == {
                      parts:               {compvar_label, comptype+}*
                      local_constraints:   {augmname_var*, clp_constr}*
                   } | {
                      topology:            graph_type
                      edge:                comptype
                      node:                comptype
                   }

compindiv_spec   == component comp_indiv {
                      component_type:      comptype
                      local_constraints:   {extname_var*, clp_constr}*
                      aggr_constraints:    aggr_indiv_constr*
                   }

extname_var      == portcardbound_var | name_var
portcardbound_var == type(port_name), comptype, min, max, clp_var
name_var         == name_ref, clp_var
name_ref         == name | type(port_name)
name             == port_name | attr_name
port_name        == clp_atom†
attr_name        == clp_atom†
ground_flag      == true | false

augmname_var     == name_var | {subcompref(self | compvar_label, name_ref), clp_var}
compvar_label    == clp_atom†
aggr_indiv_constr == edges(comp_indiv+) | sequence(comp_indiv comp_indiv+) |
                     aggr(count(comptype), comparison, value, betwspec) |
                     aggr({sum | min | max}(attr_name), comparison, value, betwspec)
betwspec         == between(comp_indiv, comp_indiv)
comp_indiv       == clp_atom†

aggr_constr      == aggr(count(comptype), comparison, value) |
                     aggr({sum | min | max}(attr_name), comparison, value)
comparison       == < | ≤ | > | ≥ | =

graph_type       == hypergraph | tree | line | other graph structures as necessary
```

† - pairwise-disjoint sets of symbols

Fig. 3. Abstract specification for device modelling language.

```
  ports:          [(conn, 2, [connector, terminator])]
  attributes:     [(length, 1..20)]
  cost:           [(length, LengthVar)], 10+2*LengthVar
}
```

A `wire` has two ports which may be connected to either a `connector` or `terminator` (or any specialising component types) and has an attribute `length` which can take a value between 1 and 20 during constraint satisfaction. A `wire's` cost is a function of its length. To declare the cost formula, an appropriate binding between the variable `LengthVar` and the symbolic name `length` must be declared (*name_var* in Fig. 3) so that the configuration and diagnosis constraint handlers can ensure that for all instances of `wire`, the appropriate local variable for `length` is bound.

Component types are organised in a closed *isa* lattice according to their parent types with inheritance of port/attribute definitions and component constraints following the subsumption ordering, facilitating incremental component specification. Signature and constraint inheritance is monotonic due to its representational simplicity and, in our experience, adequacy (although future work will extend the isa lattice to multiple partial orders for generality [23]). Multiple inheritance is supported though without name clash resolution. An exception is a component type's cost expression, chosen from the most specific ancestor defining a cost expression.

4.1. Local constraints

The active constraints over a variable will be a component type's *local constraints* and any active *aggregate constraints* (Section 4.2) due to the component being a constituent to some aggregation. Local constraints may be written using any of the constraints available in the CLP($\mathcal{FD}$) system and may be defined over port values, cardinalities of port sets, or over attribute values defined for the component type. Name-variable pairing is similarly required for constraint declarations.

Example 2.

```
component_type t-connector {
  isa:              [connector]
  ports:            [(conn, 3, [empty, node, terminator, wire])]
  cost:             [ ], 10
  localconstraints: [([(type(conn), wire, 0, 2, MaxW)], MaxW),
                     ([(type(conn), node, 0, 1, MaxN)], MaxN),
                     ([(type(conn), wire, 2, 2, NumWire),
                       (type(conn), terminator, 0, 0, NumTerm)],
                       NumWire #=> NumTerm)]
}
```

A `t-connector` has three ports in the port set `conn`, each of which may be connected to any of `node`, `terminator`, `wire` or left disconnected (the distinguished component type `empty`) subject to constraints including that at most two `wires` and at most one `node` may be connected. Each constraint declared in Example 2 exploits the *port value cardinality bound* constraint (Section 5.1.1). The

(type(*Port_set*), *Comptype, Min, Max, V*) form (*extname_var* in Fig. 3) binds the boolean variable V to the truth of the occurrence cardinality bounds constraint:

$$V \equiv Min \leqslant |\{p \cdot p \in Port\_set \wedge subsumes(Comptype, type(p))\}| \leqslant Max, \tag{4.1}$$

where $type(p)$ is the current component type for a port p and $subsumes(t_1, t_2)$ holds iff t_1 is a more general component type than t_2 in the isa lattice.

The last cardinality constraint of Example 2 states that if two `wires` are connected, no `terminators` may be connected, demonstrating the flexibility of being able to construct constraints from references to component ports and attributes in conjunction with the available constraints of the CLP system, in this case the ECLiPSe CLP($\mathcal{FD}$) logical constraint `#=>` (material implication – $\supset$).

4.2. Aggregation constraints

Local constraints are, however, restricted to the port and attribute variables of a component type. Often, it is desirable to state constraints over a group of interconnected components, typically for constraints over resource usage or component occurrence and to declare interconnection topologies. As discussed in Sections 2.1 and 2.2, the ability to state aggregation (hierarchical structural decomposition) of concepts provides a mechanism for, respectively, managing the scope of backtracking during configuration constraint satisfaction and identifying violated constraints at an appropriate granularity level during consistency-based diagnosis. Two forms of aggregation are distinguished – *part-of* and *topology* aggregations.

Part-of aggregation is appropriate for stating a structural decomposition of a "virtual" component into subcomponents, particularly where a more complicated component configuration is substitutable for another. The ports and attributes declared for the aggregate component represent the external *form* for the aggregated components. Component variables (parts in *aggr_spec*, Fig. 3) may be associated with an aggregate component and are existentially quantified over the component individuals of the variables' permissible component types. Any components within the scope of an aggregation can be connected (via their ports) only to other components scoped within the aggregation or via the aggregate component's ports. The fibre extension in Fig. 1 could be modelled:

Example 3.

```
component_type fibre_ext {
    isa:          [top]
    ports:        [(coax_portl,1,[wire]),
                   (coax_port2,1, [wire])]
    aggregates: { parts:              [(transl,1,[transceiver]),
                                       (trans2,1,[transceiver])]
                  localconstraints:   [([(transl,T_idl),
                                         (trans2,T_id2)],
                                         T_idl #<> T_id2),
                                      ([(subcompref(transl,
                                          coax_port),CoaxPl),
```

```
                                    (subcompref(self, coax_portl),
                                       Self_CoaxPl)],
                                         CoaxPl #= Self_CoaxPl),
                                    ([(subcompref(trans2,
                                         coax_port), CoaxP2),
                                    (subcompref(self,coax_port2),
                                         Self_CoaxP2)],
                                         CoaxP2 #= Self_CoaxP2)] }
}
```

A `fibre_ext` is an aggregated concept that allows connectivity to `wires` through the (virtual) ports `coax_portl` and `coax_port2`. A `fibre_ext` component implies the existence of two `transceivers` with identifiers `transl` and `trans2` which, by virtue of the localconstraints, refer to distinct components and whose `coax_ports` are bound, respectively, to `coax_portl` and `coax_port2` of `fibre_ext`.

Topology aggregation is motivated by the common requirement for non-local constraints to capture certain geometries for components' interconnections. A number of observations can be made regarding the topologies of the `ethernet segments` comprising the internetwork of Fig. 1. The individual nodes on a subnetwork (Fig. 2) are linearly sequenced. Further, the individual `ethernet_segments` and their constituent `nodes` can be visualised as edges and nodes, respectively, in a (possibly cyclic) connected hypergraph. An assumption of our language is that the interconnection topologies commonly found in devices can often be captured simply by a collection of such graph-theoretic structures [1] whose nodes and edges abstract specified component types, capturing a restricted but practical class of non-local constraints. The general notion of an internetwork can be specified as:

Example 4.

```
component_type internetwork {
   isa:              [top]
   aggregates: {     topology:      hypergraph
                        edge:       ethernet_segment
                        node:       node }
}
```

In principle, such connectedness relationships can be characterised very generally as first-order expressions. We wish to avoid such characterisations as for this and most common such topologies, however, as the characterisation is necessarily recursive, rendering the characterisation effectively useless for interpretation in different contexts due to the intractability of reasoning in such a general language. Representing such topologies using their more abstract graph-theoretic notions enables specialised handlers to be encoded without necessarily suffering the vagaries of reasoning in, for example, the more general first-order language yet still affords a clean semantics and reasonably efficient mechanisation. The linearity of an ethernet segment can similarly be represented as a trivially simple graph – a line:

Example 5.

```
component_type ethernet_segment {
   isa:              [top]
   aggregates: {     topology:     line
                     edge:         wire
                     node:         node }
   aggr_constraints: [aggr(count(node), <, 30),
                      aggr(sum(length), <, 100),
                      aggr(count(terminator), =, 2)]
}
```

For part-of and topology concept aggregation constraints, *set-wise aggregation constraints* can be declared (*aggr_constr* in Fig. 3). Aggregation constraints over the occurrences of a particular component type within the scope of an aggregation can be declared as comparisons ($<$, $\leqslant$, $>$, $\geqslant$, $=$) between some constant and one of the aggregation functions sum, min and max for component attributes or count for component types. The set-wise aggregation constraints of Example 5 state, respectively, that for within the scope of an `ethernet_segment` aggregation, a maximum of 30 `nodes` may occur, the total `length` of cabling must be less than 100 and two `terminators` must exist.

Certain aggregate constraints such as the sequencing of nodes (edge in a topology aggregation) and other constraints specific to the sequence are only specifiable when declaring component instances:

Example 6.

```
component ethersegl {
   component_type:  ethernet_segment
   aggr_constraints:  [sequence([n1,n2,n3,n4,n5]),
                       aggr(sum(length), >, 9, between(n1,n2)),
                       aggr(sum(length), >, 3, between(n2,n3)),
                       aggr(sum(length), >, 10, between(n3,n4)),
                       aggr(sum(length), >, 9, between(n4,n5))] }
}
```

The linearity constraint states that a sequence of instances of a component type must appear in the explicated order within the applicable scope of aggregation. That is, no other instances of the component type may appear on the shortest path between the components adjacent in the linearity constraint. For example, in Example 6 the constraint `sequence([n1,n2,n3,n4,n5])` states that the nodes will be so ordered in the generated configurations as illustrated in Fig. 2. In the presence of sequencing constraints, set-wise aggregation constraints may be given an additional `between` (e_1,e_2) argument (*aggr_indiv* in Fig. 3) to further restrict the sets of components under consideration to only those components between elements e_1 and e_2 in the sequence. The constraints of Example 6 capture the requirement that the minimum distances between consecutive node pairs as illustrated in Fig. 2 are 9, 3, 10 and 9 units. Note that the `length` attribute referred to pertains uniquely to the `wire` component type.

A component instance of a type defined in terms of topology aggregation may have a closed set of constituent edges. Such declarations can provide a convenient means of specifying configuration specifications (Definition 2.1):

Example 7.

```
component internetwl {
   component_type: internetwork
   aggr_constraints: [edges([etherseg1,etherseg2,etherseg3])]
}
```

The language presented here facilitates a natural representation of components through a clean integration with the CLP constraint environment and enables a natural mapping to CLP($\mathscr{FD}$) structures. The following sections present the reasoning strategies and mappings of the component specifications into the internal representations of the configuration and diagnosis problem-solvers for execution in a CLP language.

5. Mechanising the problem solvers

As the configuration and diagnosis tasks can be viewed as variants on model generation over a finite set of formulae with finite domains [25], the use of a CLP language can be seen as applying constraint posting, propagation and solving techniques to finite model generation over a domain language that is more structured than the general first order calculus and reflects the natural structure of the components.

5.1. Deriving the configurator

Recall (Section 2.1) that a desired configuration is specified by partially instantiated component instances. In this framework, a configurator carries out constraint satisfaction over the variables associated with the uninstantiated attributes and ports while minimising some cost criterion. Configuration is distinguished from classical constraint satisfaction in that instantiation of a port variable corresponds to connecting the variable's host component to the instantiating component. Heuristics for component selection are used such as preference for components currently participating in the constraint satisfaction over the introduction of new components to the partial configuration. We show how these notions are naturally represented as domain variables with CLP($\mathscr{FD}$) constraints and that the backtracking search of the logic programming paradigm cleanly integrates the dynamic nature of variable activity with the management of set-wise aggregate and topology constraints.

5.1.1. Converting to CLP variables and constraints

CLP($\mathscr{FD}$) constraints allow a natural formulation of many of the configurator's internal data structures. Each component individual participating in a configuration has associated with it a number of variables corresponding to the ports and attributes of its component type (Fig. 3).

An attribute is captured by a domain variable with its CLP($\mathscr{FD}$) domain as declared in the component specification. A port set as declared with port has a port individual variable pv_i and component type variable pvt_i associated with each of the n ports in the port set. A type domain closure constraint is asserted for each port for each possible type. Consider a port variable pv_i of type pvt_i with component type domain $\{terminator, wire, node\}$ whose respective instances are $\{term_1, term_2\}$, $\{w_1, w_2, w_3, w_4\}$ and $\{n_1, n_2, n_3, n_4\}$. These constraints are naturally expressed over the domain variables by CLP($\mathscr{FD}$) constraints:

```
PVTi::[terminator,wire,node],
PVi::[term1,term2,w1,w2,w3,w4,n1,n2,n3,n4],
PVTi#=terminator#<=>PVi::[term1,term2],
PVTi#=wire#<=>PVi::[w1,w2,w3,w4],
PVTi#=node#<=>PVi::[n1,n2,n3,n4].
```

Prior to asserting these constraints over `PVTi`, the component type set is expanded to include all subsumed component types according to isa relationships stated in Σ (Definition 2.1). Correspondingly, the associated component instance set `PVi` reflects the subsumed types in its domain and by additional type domain closure (#<=>) constraints.

This approach of co-relating instance and type variables facilitates type-based reasoning such that should a particular component type be removed from the type variable's domain at a point in the search, its associated component instances can be removed from consideration. The cost of redundant and expensive backtracking over component instances guaranteed to fail is reduced. Through constraint propagation and incremental constraint solving techniques, the CLP($\mathscr{FD}$) system can exploit these constraints along with other constraints posted over PVi and PVTi during search to reduce active variable domains and thus eliminate the exploration of whole classes of futile configuration search spaces.

A local_constraint is a pair of a set of name-variable pairs `([([(transl, T_idl), (trans2,T_id2)]` of Example 3 and a constraint `(T_idl #<> T_id2))` in the language of the appropriate constraint system, in this case CLP($\mathscr{FD}$), over variables local to a component. All variables appearing in the CLP constraint must appear in the name-variable pairs so that they may be correctly bound to the port and attribute variables of the appropriate component instance prior to the constraint being posted. Without the rebinding, all subsequent applications of a CLP constraint would result in the (ineffectual) reapplication of the constraint over the variables bound from the first application of the constraint, that is for the first instance of the component type.

The ability to state local constraints using the constraints of the underlying CLP system has the additional benefit of providing extensibility to the component specification language. User-defined constraints may be used in the constraint declaration *clp_constr*. The *port value cardinality bound* constraint (Eq. (4.1), *portcardbound_var* in Fig. 3) is an example of a user-defined constraint associated with a boolean variable which is true iff the port set identified by *port_name* has at least *min* and at most *max* occurrences of a component type *comptype* (or any of its subsumed component types), building on the cardinality constraint of ECLiPSe CLP($\mathscr{FD}$). By virtue of our approach, the boolean variable can then participate in other available CLP constraints as for `t-connector` in Example 2.

The distinction between local_constraint and aggregate_constraint is explicated in the representation language due to the different mechanisms for handling the espective constraints during configuration. Part-of and topology aggregation constraints operate over sets of components and, as such, are not directly representable as CLP($\mathscr{FD}$) constraints, requiring handling by specially encoded handlers (Section 5.1.2).

Within the scope of a part-of aggregation such as Example 3, however, local_constraints can be declared over port or attribute variables as associated with the components identified by the existential variables of the aggregation (parts in Fig. 3). The purpose of such constraints is typically to relate port or attribute variables of aggregating components *higher* in the hierarchical scoping of the aggregation hierarchy as illustrated for `fibre_ext` in Example 3. Such `local_constraints` (*aggr_spec* in Fig. 3) are sets of pairs of name-variable pair sets and CLP($\mathscr{FD}$) constraints as previously. As well, the names are extended (*aug-name_var* in Fig. 3) to enable reference to a port or attribute variable *var* within a component identified by the component individual variable name *comp* using the form `subcompref`(*comp*,*var*) and to allow reference to the aggregating component individual using the name `self`. For simplification and with no significant loss of generality, names of port and attributes are required to be unique down a scope of aggregation, obviating need for any name-space resolution.

5.1.2. Relationship to dynamic CSP

Classical constraint satisfaction problems (CSPs) involve finding valuations to a fixed set of variables V with domains $dom(v_i), v_i \in V$ that satisfy a finite set of constraints Γ each over a subset of V. The configuration task is generally recognised as being a *dynamic constraint satisfaction problem* (DCSP – [16]). DCSPs extend CSPs by enabling new variables to be introduced depending on variable activity. In a configurator, the set of variables under consideration, the *active variables list* (AVL), may be extended during the search due to port variables being instantiated to new components, necessitating the creation of new variables associated with the component. In principle, this dynamic nature of variable and constraint activity can be encoded by using 1/0 activity variables but this is typically impractical as the encoding massively increases the number of variables under consideration and is limited to configurations with a predefined upper bound on size.

CLP languages provide a natural environment in which to express incremental variable and constraint posting integrated with backtrackable variable valuation. Unlike in dedicated constraint solving environments, the AVL in CLP systems is a standard Prolog list and so can be readily extended by conventional list manipulation. More customised procedures for manipulating the AVL and constraints over its variables can thus be encoded that leverage properties of the problem domain.

DCSPs provide a very general framework in which to discuss the idea of dynamic introduction and retraction of variables in the constraint satisfaction process. The configuration task can be mechanised in a CLP language (algorithm sketched in Fig. 4) by adopting an extended DCSP-style search for valuations customised to exploit certain innate characteristics of the task.

- *Component ports/attributes*: The addition of a new component to a configuration causes new variables to be added to the AVL and constraints posted according to the component's type declarations (Fig. 3) as per Section 5.1.1.

```
CalcConfig(ConfigSpec, Comps) → Cost
≜
  _ ← DCSPish(ConfigSpec, Comps, {}, Cost)
  return Cost.

DCSPish(ActiveCompSet, AvailCompSet, Constraints, Cost) → ComponentsIntroduced
≜
  % initialisation defaults
  IntroducedCompSet ← ∅

  % decompose aggregated components
  AggrCompSet ← {c | c ∈ ActiveCompSet s.t. type(c) contains an aggregates clause}
  for c ∈ AggrCompSet do {
    (EdgeSeqIndivs, LocConstraints) ← extract_aggr_info(c)
    IntroducedCompSet ∪ ← DCSPish(EdgeSeqIndivs,
                                  AvailCompSet − IntroducedCompSet,
                                  Constraints ∪ LocConstraints, Cost)
  }
  % carry out DCSP-style constraint satisfaction
  AVL ← gen_comp_type_vars(ActiveCompSet − AggrCompSet) % components' variables with
  while non-nil(AVL) {                                  % constraints posted (§5.1.1)
    AVL ← AVL − v where v is most constrained variable in AVL
    if attribute-var(v) ∨ porttype-var(v) {
[CP1] choose value for v from its domain
      test any applicable sum|min|max set-wise aggregation constraints from Constraints
    } else {
      % v's port type variable is vt
[CP2] choose value for vt from its domain
[CP3] attempt to bind to v a pre-existing component with a free type-compatible port {
        test any applicable topology constraints in Constraints not violated
      } else {   % no more choices at [CP3]
        create a new component c   % will be of type vt
        test any applicable count set-wise aggregation constraints from Constraints
        if type(c) contains an aggregates clause{
          (NewEdgeSeqIndivs, NewLocConstraints) ← extract_aggr_info(c)
          IntroducedCompSet ∪ ← DCSPish(NewEdgeSeqIndivs,
                                        AvailCompSet − IntroducedCompSet,
                                        Constraints ∪ NewLocConstraints, Cost)
        }     % if
      }     % attempt
    }     % if
  }     % while
  return IntroducedCompSet.
```

extract_aggr_info:
extracts 1. the component individuals referred to in the edges or sequence terms and 2. the local_constraint ∪ aggr_constraint terms from the component type declaration for a component individual c

gen_comp_type_vars:
returns the set of attribute, port and port type variables according to the component type of c with domain closure and local_constraints posted as per §5.1.1

Fig. 4. Sketch of algorithm used for the extended DCSP-style search.

- *Part-of aggregate constraints*: The selection of a component with a component type containing a part-of aggregation declaration causes the addition of component individual and type variables to the AVL as for the port declarations. Having posted any local_constraints defined within this scope of aggregation, constraint satisfaction continues (recursively calling DCSPish, Fig. 4) except that the component individual variables may only be bound to components within the scope of aggregation, that is only to components not already participating in other subassemblies.
- *Topology aggregate constraints*: If within a current scope of aggregation (current invocation of DCPish, Fig. 4), a topology aggregation constraint is active ($\in$ *Constraints*), it is necessary to test that that topology is being correctly honoured and, as such, the binding of any component to another (CP3 in Fig. 4) must be verified by the appropriate handler for the active topology. Violation according to the handler causes the algorithm to backtrack.
- *Set-wise aggregate constraints*: The sum, min, max and count constraints work over sets that typically grow dynamically as the configuration search proceeds and so cannot be entirely handled by posting CLP($\mathscr{FD}$) constraints. Consider the constraint count(`terminator`) = 2 for `ethernet_segment` (Example 5). If during the configuration search, a partial configuration of the aggregate instance `etherseg2` contains only one terminator but uninstantiated port variables remain (connectivity satisfaction still not closed), it would be premature to claim the aggregation constraint had failed. Alternatively, for a constraint such as sum(`length`) < 100 for `ethernet_segment`, as the constraint is required to hold monotonically with an increase in the connected `wire lengths`, the constraint is postable over the sum of the `length` attribute variables as components are added. Indeed, it is often advantageous to post such constraints when over attribute variables (sum, min, max) immediately an applicable component is added as the relevant attribute domain variable participating may be non-ground and may have its domain reduced to satisfy the constraint. Set-wise aggregation constraints can be separated into two classes based on the comparison operator ($<$, $\leqslant$, $>$, $\geqslant$, $=$) of the constraint:

1. *Defer until aggregation completion*: $>$, $\geqslant$ and $=$ can be checked only once all port variables have received a valuation during the configuration search within the scope of aggregation for the constraint.
2. *Immediately postable*: $<$, $\leqslant$ and $=$ (posted as $\leqslant$) can be posted as constraints over the set of applicable attribute values (or number of applicable component occurrences for count) to possibly facilitate some constraint propagation.
   The $=$ comparison falls into both classes as $v_1 = v_2 \equiv v_1 \geqslant v_2 \wedge v_2 \leqslant v_1$.

The constraint-based configuration task can be viewed as a special-case DCSP honouring these characteristics. The algorithm as sketched in Fig. 4 contains a number of *choice points* indicated CPx which represent points during the algorithm where non-deterministic choice from a collection of values is being made. The assertion of a constraint subsequently may fail due to the constraint solver failing to find a satisfiable variable valuation, indicating that the component addition is in violation of this constraint and causing the search to backtrack to the last choice point. Any new variables or constraints posted since this last choice point are retracted, an implicit feature of backtracking in a CLP language.

A cost expression is a normal CLP($\mathscr{FD}$) expression whose variables are first bound to the appropriate variables of the new component and then added to the accumulated cost of the assembly so far. The ECLiPSe CLP($\mathscr{FD}$) system provides various `minimize` meta-predicates to invoke our `CalcConfig/3` to only return configurations of least cost, carrying out an optimising search that forces failure of any partial generated solutions whose cost exceeds that of the best-cost solution generated during the search.

5.2. Diagnosing with component specifications

The diagnosis task considered is essentially the detection of misconfigured components with respect to the component specifications. As the task is, by nature, one of verifying propositional constraints rather than design synthesis, the mechanisation is simpler than for configuration. The key idea is that all constraints from the component specification are reinterpreted as consequences of an assumption of component "normality". Component interconnectivity (device structure) is assumed to be fixed although component attributes may be non-ground due to observations for attributes being absent or incomplete. Through the use of CLP($\mathscr{FD}$) constraints, a network of boolean expressions can be built from which parsimonious diagnoses is subsequently calculated. The concepts of abnormality are first related to component specifications after which the CLP($\mathscr{FD}$) encodings are discussed.

5.2.1. Relating component specifications to abnormality

A component specification constraints the set of possible configurations for a component's port and attribute variables as well as, where aggregating other components, the allowable topologies and set-wise aggregate constraints. The principle of consistency-based diagnosis is that a component is behaving (in our case, configured) correctly iff its actualisation conforms to its specification. Boolean variables can thus be associated with the individual constraints over the components.

The general form for relating the correct behaviour ϕ_t for a component of type t ($comp_t(x)$) can be written [25]:

$$\forall x \cdot comp_t(x) \wedge \neg ab(x) \supset \phi_t(x), \tag{5.1}$$

where $ab(x)$ denotes the abnormality of the component x. Eq. (5.1) presumes that violation of $\phi_t(x)$ is only one of the ways that a component of type t can misbehave. If a sufficient set of behaviours $\phi_t^1, \ldots, \phi_t^n$ can be identified for a component type t, the corresponding abnormality predicates $ab_t^1, \ldots, ab_t^n$ can be written (where $n(t)$ is a slight abuse of notation denoting the number of constraints for type t):

$$\forall x \cdot \bigwedge_{i \in 1,\ldots,n(t)} ab_t^i(x) \equiv comp_t(x) \wedge \neg \phi_t^i(x), \tag{5.2}$$

$$\forall x \cdot ab(x) \equiv \bigvee_{i \in 1,\ldots,n(t)} ab_t^i(x). \tag{5.3}$$

In our framework, ϕ_t^i corresponds to the individual port types, attribute domains, local_constraints and aggregate_constraints for a component type t as specified in a component specification (Fig. 3). Aggregate constraints necessitate a generalisation of Eq. (5.3) due to the intuition that the abnormality of an aggregating component should reflect not only its immediate constraints but also those of its constituents.

Where $part\_of(x, y)$ denotes that x is a constituent component of y, Eq. (5.3) can be generalised for each component type t to

$$\forall x \cdot ab(x) \equiv \left[\bigvee_{i \in 1,..,n(t)} ab_t^i(x) \right] \vee \exists y \cdot [part\_of(y, x) \wedge ab(y)]. \tag{5.4}$$

As the actual components participating in the diagnosis will be fixed for a particular diagnosis calculation, Eqs. (5.2) and (5.3) can be simplified by removing the quantification and the $comp_t(x)$ conjunct as constraints will be posted on a per-component basis (Section 5.2.2), effectively becoming propositional. Due to the natural integration into CLP($\mathscr{FD}$) of propositional constraints (Section 3), this framework for calculating ab_t and ab_t^i extends naturally to the constraints available in CLP($\mathscr{FD}$).

Unlike for the configuration task where components were dynamically being added (Section 5.1.2), the truth of set-wise aggregate constraints can be directly encoded with CLP($\mathscr{FD}$) constraints as sum, min and max constraints are over finite sets of values as is count which will be ground due to the fixed device structure. Further, it is reasonable to assess the compliance of an aggregating component's constituents to a topology constraint using a Prolog predicate external to the constraint system as the truth of such a constraint will be constant irrespective of the grounding of the other component variables as the interconnectivity of the components is ground.

This approach specifies two collections of abnormality predicates – $ab_t^i(x)$ for the abnormality of constraint i for component individual x of type t; and $ab(x)$ for the overall abnormality of component individual x. As we are interested in the minimal sets of abnormal components (Section 2.2), $ab(x)$ will be of interest for calculating diagnoses while $ab_t^i(x)$ will identify individual constraints within components. Given the possible labellings for the non-ground variables, those extensions of ab (as calculated by the network of boolean abnormality variables) of minimal cardinality are thus the diagnoses. Further, as abnormality in a constituent component of an aggregation is reflected up the aggregation hierarchy by Eq. (5.4), minimisation of ab will prefer faults at higher levels of the aggregation hierarchy.

5.2.2. CLP constraints and abnormality

The diagnosis problem-solver is reasonably straightforward to encode given the relationships between component type specifications and abnormality. The mechanisation involves the accumulation of constraints as instantiations of Eq. (5.2) and Eq. (5.4) to form a network of boolean constraints from the components' respective abnormality variables. The characterisation of abnormality propositions as negated constraint consistency checks has a simple treatment in CLP($\mathscr{FD}$), leveraging the power of the constraint solver.

For each of the components in $\mathscr{OBS}$ (Definition 2.2), port, port type and attribute variables are created in the same manner as in Section 5.1.1 except that constraints other than type domain closure constraints are not posted. The attribute and port variables are populated to reflect the ports and attributes of the components of $\mathscr{OBS}$. It is not necessarily the case that all attribute variables will be ground as their values may be either unknown or a range for a component individual.

The ECLiPSe CLP($\mathscr{FD}$) environment provides an evaluation predicate `isd/2` which, given the sentence `B isd Expr`, assigns a boolean value of `1` to `B iff Expr` is true, otherwise `0`. In principle, the sentence could be written using the `#<=>`

constraint. The `isd` predicate is preferred in keeping with the CLP philosophy of using the highest abstraction to benefit from future developments in CLP technologies.

The term `Expr` for `isd` is limited in CLP($\mathscr{FD}$) to the term comparison, propositional operator and domain restriction (`V::[vl..vn]`) constraints of Table 1. As such, special handling is required for deriving a boolean variable for consistency for the other constraint forms of Table 1 and the port value cardinality bound constraint (Section 4). The consistency tests for a component type's individual constraints (Fig. 3) are posted according to the following rules.

- port *types*, attribute *domains*: As these constraints are just domain restriction constraints, isd is used to relate the boolean consistency variables although the port type list requires expansion to the closure of the *subsumes* relation as applied to each element.
- local_constraint *term comparison and propositional expressions*: Use of `isd` is permissible after appropriate variable rebinding for name-variable pairs has been effected.
- local_constraint *miscellaneous constraint predicates*: The miscellaneous constraint predicates of Table 1 other than domain restriction have appropriate analogues defined that associate a boolean consistency variable with the consistency of the predicates in terms of the constraints' underlying semantics.
- *Set-wise* aggregate_constraint: The set of applicable components participating in a sum, min, max or count constraint will be of a fixed length due to the ground component interconnectivity. Given the fixed set of components, a count expression will be a constant and thus the corresponding boolean variable will be ground to the truth of the comparison. For sum, min or max, the set of variables can be unfolded into a single expression (e.g. `Vl` $+ \cdots +$ `Vn` for sum) and the comparison in the aggregate constraint represented using the term comparison constraints of Table 1. The resultant expression may then be associated with its corresponding boolean variable using `isd` in the standard way.
- *Topology* aggregate_constraint: As discussed in Section 5.2, component interconnectivity is ground so the evaluation of an aggregating component's topology is a boolean constant. The associated boolean variable for the topology constraint will thus be ground and can be evaluated by an external Prolog predicate e.g. `hypergraph(internetwl,B)`.

Having constructed the appropriate boolean variables for the consistency of the constraints for each of $\phi_t^1, \ldots, \phi_t^n$ for component type t, the appropriate abnormality variable as per Eq. (5.2) can be constructed by application of the `#\+` (negation) operator, throwing away the consistency variable.

The component abnormality variable corresponding to Eq. (5.4) can then be constructed by unfolding the list of boolean abnormality variables for the component's local constraints and the abnormality variables for the constituent components, repeatedly applying the `#\/` constraint to yield their cumulative disjunction. For the purposes of posting the boolean abnormality variables, the components of $\mathscr{OBS}$ can be seen as being organised into a disjoint tree representing component aggregation where the edges reflect the *part_of* (x, y) relation (y is parent/aggregating node of x). The tree is traversed in a left-to-right post-order as the posting of abnormality variables corresponding to Eq. (5.4) requires that the abnormality variables for constituent components have first been created. Due to the left-to-right post-order tra-

versal of the tree, the appropriate constituent abnormality variables are guaranteed to be available for this unfolding.

The network of constraints over the boolean abnormality variables has been constructed. As discussed in Section 5.2.1, the diagnoses for a system are the smallest sets of abnormal components given the possible valuations of any free component variables in the system. This corresponds naturally to the optimisation notion of those solutions (abnormality variables) for which a cost (number of abnormal components) is minimal. An additional *cost* variable is thus encoded:

$$cost = \sum_{c \in \mathcal{CBS}} ab_c, \tag{5.5}$$

where ab_c is the abnormality boolean ([0, 1]) variable for component c.

The generation of the sets of minimal diagnoses $\underline{\Delta_{\text{Card}}}$ (Definition 2.4) becomes the application of the ECLiPSe CLP($\mathcal{FD}$) meta-predicate `minimize` predicate over the labelling of the free variables in the system with the cost variable as constructed by Eq. (5.5) as the optimisation objective subject to the network of constraints over the boolean variables derived from the component specifications.

Consider the misconfigurations described in Section 2.2 for Fig. 1. If the t-connector `t2` had wires connected to each of its three ports, the port value cardinality constraint for `wires` in Example 2 would be violated, grounding the corresponding abnormality variable (ab_t^i) for the constraint to true. This would cascade to the abnormality variable for `t2`, the ethernet segment `ethersegl` and internetwork `internetwl`. As an alternate scenario for Fig. 1, consider where a terminator was missing from `ethersegl` and various valuations for attribute variables of components constituent to `ethersegl` suggested some of these components to be abnormal while other valuations suggested not. The effect of the `minimize` predicate on the `Cost` variable of Eq. (5.5) would be to calculate the diagnoses to only be the abnormality of the constraint that count (`terminator`) = 2 (Example 5) as well as components `ethersegl` and `internetwl` (cascaded abnormality). This demonstrates preference of failure diagnoses at higher levels of the aggregation hierarchy.

6. Implementation experiences

The configuration and diagnosis engines have been developed in the ECLiPSe constraint logic programming environment using the Finite Domains (FD) library [5]. The emphasis of the implementation of ECLiPSe has been on environment extensibility, with many of the key predicates available in source form in libraries to simplify extensibility and prototyping of techniques for constraint manipulation. Timing figures were generated on a SUN SS20-612 (dual 60 MHz SuperSparc processors) with 256 megabytes of memory and minimal user load.

The `deleteff/3` predicate for selecting the most constrained variable has been modified to carry the context of the source component and name of the variable selected from a list of variables similarly carrying their contexts. `deleteff` is relevant when variables are to be chosen from the AVL according to the *first-fail* principle (smallest domains [9,5]) to maximise constraint propagation. The intuition is that the most constrained variable should be chosen first for enumeration to make likely failures appear earliest in the search tree in an effort to reduce the expense of back-

tracking. This modification was necessitated due to DCSP-style search (Section 5.1.2) requiring knowledge of which variable has been selected for incremental variable and constraint introduction. The modification was feasible due to ECLiPSEe's extensible design and addresses Mittal and Falkenhainer's [16] key criticism of CLP languages for DCSPs by enabling the constraint handler to identify the source component and name of a chosen variable, that is reason about variable activity. This approach could be used to handle the RV constraint of [16] for removing variables from the AVL however the constraint has not been deemed useful in our configurator case studies to date.

Despite the exponential nature of naïve constraint-based configuration in general [18], the configuration of Figs. 1 and 2 over an increasing numbers of nodes exhibited linear growth. Fig. 5 illustrates how as the number of computer nodes (n) to be cabled was increased, the number of iterations of (attribute and port) variable valuation and elapsed user CPU time also increased linearly due to strong constraint propagation. For example, a solution for the configuration of a network involving 11 computer nodes that resulted in 126 port connections (Fig. 5, left) required 209 variable valuations. The difference represents unsuccessful variable valuations attempted and would have been substantially higher but for valuation possibilities implicitly ruled out through constraint propagation.

Key factors that contribute to the effectiveness of constraint propagation include the (i) strong interconnectivity of component types; and (ii) enforcement of explicated topology graphs (Section 4.2). A detailed study of the kinds of constraints that propagate well is in progress. Other case studies including Searls and Norton's [22] computer configuration also performed well due to high interconnectivity.

As the diagnosis task involves a once-off construction of the network of constraints over the boolean abnormality variables, the execution times for even large configurations ($n > 50$) were similarly stable. The introduction of behaviour mappings ([20] and Section 3.1 of [25]) not considered in this work (Section 2.2) requires labelling of a finite set of variables. These mechanisations are currently under investigation.

6.1. Extending the implementations

The following observations are made in regard to the configuration task due to its generative and combinatorial nature, as opposed to the diagnosis task's verificational form. A problem of a chronological backtracking scheme as embodied in logic programming languages is that the (possibly unproductive) exploration of the search space that ensues the commitment of a component choice (instantiation of compo-

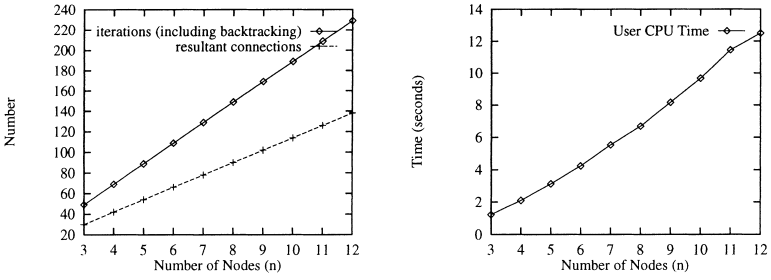

Fig. 5. Observations for configuration tests.

nent variable) may be redundantly re-explored for some subsequent instantiation(s) of the same or other component variable(s). Although all solutions will eventually be found through chronological backtracking, the exploration of such fruitless search sub-trees can be expensive for non-trivial problem statements.

While techniques such as the constraint propagation due to the posting of constraints over component and component type variables and the partial choice ideas of [17] enable the commitment of variables' instantiations to be deferred significantly, situations often still occur where redundant backtracking over possibly very deep search sub-spaces still results. Sometimes this is due to insufficient constraint propagation but more often the "geometry" of the search space is such that a cost-ineffective look-ahead of sufficient depth in the search tree would be required as a result of the class of problem. This is typically due to decision points being made locally to a component during search rather than referring to overall task objectives.

In principle, constraint programming languages that support dependency-based intelligent backtracking would be appropriate. Such languages are few, e.g. [8] and in their infancy, however. It is a non-trivial task to characterise constraint formulations and predicate declarations such that efficient intelligent/non-chronological backtracking strategies can be automatically derived without prohibitive expense.

From our observations of the executions of the constraint-based configuration tasks, a convenient extension to improve the robustness of the search of the solution space is the tagging of certain constraints as *goal constraints*. While, in principle, a goal is just another constraint that holds for a given solution, the explication of goal constraints enables the configuration engine to recognise and prune sub-trees of a search space when a search is deviating too far from a path likely to lead to a solution. Interestingly, the issue of goal-directedness does not seem to appear in the constraint-based configuration literature.

These issues further strengthen our choice of using a CLP language as ongoing research in CLP implementation techniques can be leveraged for the configuration and diagnosis problem-solver mechanisations.

7. Related work

The literature on constraint-based configuration and diagnosis systems tends to be disjoint. We briefly review prominent constraint-based approaches that share similarities to our mechanisations. As the diagnosis and configuration tasks have been discussed as combinatorial constraint satisfaction problems, the reader is referred to [9,4] for extensive presentations of the general relationship between such problems and CLP languages.

A number of systems have been designed that effectively characterise the configuration solution space and constituent components in terms of sets of constraints such as Cossack [6], BEACON [22] and LCON [26]. These systems required individual constraint handlers to be implemented and tested whereas we are able to benefit from CLP technology and ongoing research on constraint-solving techniques. The LCON system, for example, is implemented in Smalltalk and, as such, required a complete constraint system for posting, propagation and retraction to be implemented.

The BEACON system, implemented using a logic programming language, encodes a semantic network formalism KNET that supports similar constructs to

our component specification language (Section 4). The constructs of the KNET language are effectively re-interpreted by a meta-interpreter constructed on top of the base Prolog language. Our approach, in contrast, directly translates the component specification language into the CLP($\mathcal{FD}$) constraint system, leveraging the underlying constraint-solving facilities. Further, our language is extensible as it supports the constraints supported by the underlying constraint engine.

Features of our system not apparently available in these systems include the ability to linearly sequence components of a particular component type, the expression of aggregation constraints and the ability to express cost expressions for component types in terms of the component's port values and/or attribute values which may, themselves, be variables participating in constraint satisfaction. Some preliminary results were presented in [24].

While the literature on model-based diagnosis is extensive [7], only the work of Freuder [21] and Mozetič [19] is considered here. The work in [21] is interesting as the work considers the diagnosis of software misconfigurations in the domain of networking systems where abnormality is identified by the violated constraints describing correct configurations, similarly to our approach. The diagnosis task is treated as a *partial constraint satisfaction process*, applying a DCSP-style approach to formally integrating the acquisition of observations as needed with the constraint satisfaction process.

Mozetič [19] describes a treatment of model-based diagnosis in CLP languages by encoding the input and output values of components as CLP variables which are shared between components to capture interconnectivity. Apart from the key distinction of Mozetič's work from ours being that we do not consider behavioural models, a more subtle distinction between this work and ours is that in [19], the problem requires rephrasing to exploit the constraint system whereas in our work (Section 5.2.2), the component specifications can be used to derive the appropriate constraints. Similar observations are made to ours that the diagnosis task can naturally take advantage of the underlying constraint solving systems.

8. Conclusions

Constraint logic programming languages provide a natural integration of the backtracking search mechanism of conventional logic programming and non-deterministic constraint representations and solving techniques. As such, CLP languages provide a flexible environment in which variable and constraint posting is tightly integrated with backtracking search to enable reasonably efficient mechanisations of combinatorial problems such as constraint-based configuration. CLP languages also allow more declarative representations of domain knowledge than do conventional logic programming languages, making it the preferred environment in which to mechanise different problem-solvers working from shared domain representations.

A component specification language has been presented from which appropriate CLP($\mathcal{FD}$) variables and constraints can be derived to populate engines for the mechanisation of the constraint-based configuration and diagnosis tasks. We believe this work is novel as: (i) it demonstrates that CLP languages as a viable platform for knowledge sharing and; (ii) the component specifications are translated directly into

CLP($\mathcal{FD}$) constraints for both tasks to gain maximum leverage from the underlying constraint solvers.

Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centres Programme through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

References

[1] C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, 1973.
[2] W.J. Clancey, Heuristic classification, Artificial Intelligence 27 (3) (1985) 289–350.
[3] R. Davis, W. Hamscher, Model-based reasoning: Troubleshooting, in: Exploring Artificial Intelligence: Survey Talks from the National Conference on Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, 1988, pp. 297–346.
[4] M. Dincbas, H. Simonis, P. van Hentenryck, Solving large combinatorial problems in logic programming, J. Logic Programming 8 (1) (1990) 75–93.
[5] ECRC, ECLiPSe Extensions user manual 3.5, December 1995.
[6] F. Frayman, S. Mittal, COSSACK: A constraints-based expert system for configuration tasks, in: Knowledge Based Expert Systems in Engineering: Planning and Design, Computational Mechanics, 1987, pp. 144–166.
[7] W. Hamscher, L. Console, J. de Kleer (Eds.), Readings in Model-based Diagnosis, Morgan Kaufmann, Los Altos, CA, 1992.
[8] W.S. Havens, Intelligent backtracking in the Echidna CLP reasoning system, Int. J. Expert Systems: Res. Appl. 5 (4) (1992) 21–45.
[9] P. van Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, MA, 1989.
[10] P. van Hentenryck, V. Saraswat, Y. Deville, Design, implementation, and evaluation of the constraint language `cc(fd)`, in: Constraint Programming: Basics and Trends, Lecture Notes in Computer Science, vol. 910, Springer, Berlin, 1994, pp. 293–316.
[11] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL), Munich, 1987, pp. 111–119.
[12] J. Jaffar, M.J. Maher, Constraint logic programming: A survey, J. Logic Programming 19/20 (1994) 503–581.
[13] J.W. Lloyd, Foundations of Logic Programming, 2nd extended edition, Springer, Berlin, 1987.
[14] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[15] J. McDermott, R1 (“XCON”) at age 12: Lessons from an elementary school achiever, Artificial Intelligence 59 (1993) 241–247.
[16] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problems, in: Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, 1990, pp. 25–32.
[17] S. Mittal, F. Frayman, Making partial choices in constraint reasoning problems, in: Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), Seattle, 1987, pp. 631–636.
[18] S. Mittal, F. Frayman, Towards a generic model of configuration tasks, in: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89), 1989, pp. 1395–1401.
[19] I. Mozetič, C. Holzbaur, Model-based diagnosis with constraint logic programs, in: Proceedings of the Seventh Austrian Conference on Artificial Intelligence (ÖGAI-91), Vienna, Springer, Berlin, 1991, pp. 168–180.
[20] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1) (1987) 57–95.
[21] D. Sabin, M.C. Sabin, R.D. Russel, E.C. Freuder, A constraint-based approach to diagnosing configuration problems, in: Proceedings of the IJCAI-95 Workshop on Artificial Intelligence in Distributed Information Networks, Montreal, August 1995.

[22] D.B. Searls, L.M. Norton, Logic-based configuration with a semantic network, J. Logic Programming 8 (1) (1990) 53–73.
[23] N. Sharma, Partial orders of sorts and inheritances (or placing inheritance in context), in: Proceedings of the Fifth International Conference on Knowledge Representation and Reasoning (KR-96), Cambridge, MA, 1996, pp. 280–290.
[24] N. Sharma, R. Colomb, Constraint-based configuration in a constraint logic programming environment, in: Proceedings of the Fourth Australian and New Zealand Conference on Intelligent Information Systems (ANZIIS-96), Adelaide, IEEE Press, 1997.
[25] N. Sharma, R. Colomb, Towards an integrated characterisation of model-based diagnosis and configuration through circumscription policies, Technical Report 364, Department of Computer Science, The University of Queensland, May 1996.
[26] M. Stumptner, A. Haselböck, G. Friedrich, COCOS – A tool for constraint-based, dynamic configuration, in: Proceedings of the Tenth Conference on Artificial Intelligence for Applications (CAIA-94), San Antonio, 1994, pp. 373–380.