Empirical Testing of a Weight Constraint Rule Based Configurator

Juha Tiihonen[1], **Timo Soininen**[1], **Ilkka Niemelä**[2], and **Reijo Sulonen**[1]

Abstract. In this paper we first describe a configurator implementation based on a practically important subset of a synthesized ontology of configuration knowledge. The underlying configuration modeling language has been provided with a declarative semantics by mapping it to weight constraint rules, a form of logic programs. Three issues important for efficiency of the implementation are addressed: off-line compilation of configuration models, limiting a configuration to a finite size in a semantically justified way, and breaking symmetries in the set of configurations. The second part of the paper takes a step in the direction of thorough empirical testing of configurators. We define a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. We test the configurator empirically on batch-mode sales configuration of four real products with progressively larger and thus more restricting sets of random user requirements. The results indicate that our configurator is efficient enough for practical use.

1 INTRODUCTION

Several formal models of configuration knowledge and tasks based on, e.g., constraint satisfaction problems (CSP) and different logical formalisms have been proposed, e.g., [1,2,3,4] and implemented, e.g., [4,5,6,7,8]. For several of these, the configuration problem has been shown to be at least NP-hard [2,4,9,10]. In other words, the configuration task requires in the worst case at least an exponential amount of time in the size of the problem. However, conventional wisdom in the configuration community is that solving configuration problems is relatively easy and does not exhibit this kind of behavior. There are some documented results on the efficiency of configurators [4,6,7,8], but systematic and wide range empirical testing of configurators on real products that would show whether the wisdom is, indeed, wisdom, is still lacking.

In this paper we take a step in the direction of thorough empirical testing of configurators. We briefly describe a configurator implementation, define a general test methodology for configurators, and provide results on the efficiency of our implementation.

Our configurator uses a modeling language based on a practically important subset of a synthesized ontology of configuration knowledge [11]. The language has a declarative semantics provided by mapping it to weight constraint rules, a form of logic programs [2,12]. The configurator uses this mapping to translate the modeling language to weight constraint rules. It then uses a state-of-the-art implementation of weight constraint rules, Smodels [12], for computing configurations satisfying user requirements. Three issues important for efficiency of the configurator are addressed: off-line compilation of configuration models, limiting a configuration to a finite size in a semantically justified way, and symmetry breaking.

We have modeled four real products from a sales configuration point of view. The case products are characterized and the configurator is empirically tested on batch-mode configuration of these products. We define a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. This is accomplished by randomly generating progressively larger and thus more restricting sets of user requirements that are not locally conflicting. Results are given on the number of correct configurations found and the time it takes to find the first and all configurations satisfying a set of random requirements, or to show that no such configuration exists.

The rest of the paper is structured as follows: In section 2 the modeling language and its semantics are outlined and in section 3 the configurator implementation based on the language is described. In section 4 the testing method and the case products are described and the test results are provided. Finally, in section 5 we discuss and compare our implementation and results with related work and in section 6 we present conclusions and topics for further work.

2 MODELING LANGUAGE

In this section we briefly describe *PCML*, the product configuration modeling language of our configurator, and outline its semantics. For more information on the modeling language and its implementation, see http://www.soberit.hut.fi/pdmg/empirical_cfg/ and [2].

The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define intensionally the characteristics (such as parts) of their *individuals* that can appear in a configuration. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type is specific enough to be used in an unambiguous configuration. A component type defines its direct parts through a set *of part definitions*. A part definition specifies a *part name*, a non-empty set of *possible part types* (*allowed types* for brevity) and a *cardinality*. A component type may define properties that parameterize or otherwise characterize the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition.* Component types are organized in a *taxonomy* or class hierarchy where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner.

Figure 1 illustrates the concepts through an example. A server PC has 1 to 2 storage subsystems. There are two kinds of storage subsystems, SSA and SSB. A storage subsystem has 1 to 4 hard disks. The hard disk types in use are HDA and HDB. This is modeled as follows (the upper part of Figure 1): Concrete component type `PC` has a part definition with part name `sto`, cardinality 1..2 and allowed type `SS`. Component type `SS` is abstract and has two concrete subtypes `SSA` and `SSB`. `SS` has a part definition `msu` (mass storage units) with cardinality 1 to 4 and allowed type `HD`. Type `HD`

[1&2] Helsinki University of Technology, Dept of Computer Science and Eng.,
[1] Software Business and Engineering Institute, P.O.B. 9600, FI-02015 HUT
[2] Lab. for Theoretical Computer Science, P.O.Box 5400, FI-02015 HUT
[1&2] {Juha.Tiihonen, Timo.Soininen, Ilkka.Niemela, Reijo.Sulonen}@hut.fi

is abstract and has two concrete subtypes `HDA` and `HDB`. The lower part of Figure 1 shows a configuration where individual `pc-1` of type `PC` has as a part with part name `sto` two storage subsystems of type `SSA` (`ssa-1` and `ssa-2`). The individual `ssa-1` has as a part with part name `msu` one hard disk of type `HDA` (`hda-1`), while `ssa-2` has as a part with part name `msu` one hard disk of type `HDA` (`hda-5`) and one hard disk of type `HDB` (`hdb-5`).

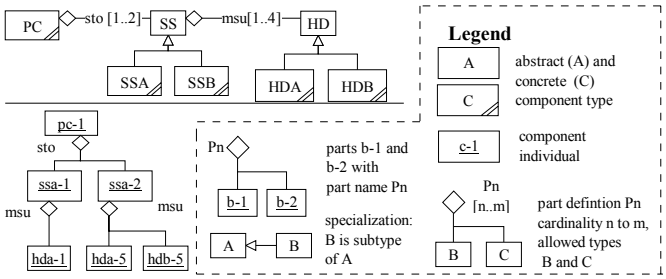

Figure 1. Example product

Constraints associated with component types define conditions that a correct configuration must satisfy. The first level building blocks of the constraint language are *references* to access parts and properties of components, and constants such as integers. References can be used in succession, e.g. to access a property of a part. *Tests* returning Boolean values are constructed using references, constants, and arithmetic expressions. Tests include predicates for checking if a particular referenced individual exists or is of a given type. Property references can be used with constants in arithmetic expressions that can be compared with the usual relational operators to create a test. Test can be further combined into arbitrarily complex *Boolean expressions* using the standard Boolean connectives.

The semantics of the modeling language is provided by mapping it to weight constraint rules [2]. The basic idea is to treat the sentences of the modeling language as short hand notations for a set of sentences in the weight constraint rule language (*WCRL*) [11]. A *configuration model* thus corresponds to a set of weight constraint rules consisting of ontological definitions defining the semantics of the modeling concepts and a set rules representing the configuration model. A *configuration* with respect to a configuration model is defined as a subset of the Herbrand base of the configuration model. A *requirement* is for simplicity defined as an atomic fact. A *correct configuration* is a stable model of the set of rules representing the configuration model and a *suitable configuration* is a correct configuration that also satisfies the set of requirements.

3 IMPLEMENTATION

This section first describes relevant parts of our prototype configurator and off-line compilation of configuration models. Then we discuss individual generation that limits a configuration to a finite size in a semantically justified way. Finally we discuss symmetry breaking that is important for the performance of the configurator.

3.1 Overview and implementation scope

The configurator architecture is outlined in Figure 2. The configurator is implemented in Java programming language except components smodels and lparse of the Smodels system, described below.

The configurator compiles a PCML program to a general WCRL program with variables and further to simple basic rules (*BCRL*) that contain no variables. This potentially costly two-phase compilation process is performed off-line. In the compilation, *PCML core* in the *Model manager* loads a PCML configuration model and checks it for consistency. This includes parsing the PCML file, type checking expressions and checking the configuration model for validity with respect to the language specification. The *Smodels interface* in Model manager translates the configuration model to a WCRL program. Data structures representing the PCML configuration model are saved for later use. The generated WCRL program includes sentences for ontological definitions, component type hierarchy, compositional structure, properties, and constraints as described in [2]. In addition, rules for component individuals and symmetry breaking are included, described in Sections 3.2 and 3.3. Finally, the WCRL program is translated to BCRL.

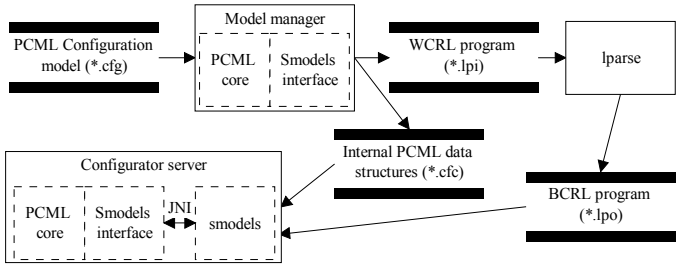

Figure 2. Configurator architecture

The configurator uses as the configuration engine an implementation of the weight constraint rule language called *Smodels*[12]. The main functionality of the Smodels system is to compute a desired number of stable models for a WCRL program. The system allows a user to give further requirements (through so-called *compute statements*) to constrain the stable models to be computed.

The Smodels system is based on a two-level architecture where in the first phase a front-end, *lparse*, compiles a WCRL program into a BCRL program. Lparse exploits efficient database techniques but does not resort to search. The search for models of BCRL programs is handled using a special purpose search procedure, *smodels*, taking advantage of special features of BCRL. The search procedure works in linear space and employs efficient search space pruning techniques and a powerful dynamic application-independent search heuristic. Smodels is implemented in C++ and offers APIs through which it can be directly integrated into other software. Smodels is publicly available at http://www.tcs.hut.fi/Software/smodels/.

The BCRL form of the configuration model is used to repetitively configure a product. The configurator server loads the data structures representing the PCML configuration model into PCML core and the BCRL program into smodels. A compute statement representing requirements is set through smodels API.

3.2 Individual generation

In this work, we take the approach that the set of individuals out of which the configuration can be constructed is pre-defined. This limits the configuration to a finite size. We decide in advance in a semantically justified way the number of individuals of each concrete type available for use in a configuration. The available individuals are represented as a set of facts stating for each individual which type it is an individual of. These facts are added to the WCRL rules representing the configuration model.

There is exactly one component type that can serve as the root of the compositional structure, referred to as the configuration type. An individual of this type, the configuration individual, serves as the root of the compositional structure. Because a maximum cardinality is defined for every part definition, we can calculate an upper bound of the number of needed individuals. First the configuration

individual is generated. For each part definition of the configuration type, the number of individuals defined by maximum cardinality of the part definition is generated for each allowed concrete part type. This is performed recursively to generate part individuals for all part definitions of the types of the newly generated individuals.

The number of needed individuals can grow exponentially. For example, increasing the number of levels in the part hierarchy leads to exponential growth in the number of generated individuals when maximum cardinality at each level is at least 2. The implementation does not try to optimize the number of individuals on the basis of constraints or mutually exclusive branches of the part structure.

3.3 Symmetry breaking

Individuals of a concrete type are equivalent except for their names. *Equivalent configurations,* i.e. configurations identical except for naming, can be created by selecting different individual(s) of a concrete type as a part. This freedom of selection creates unwanted symmetries. Next, we describe two forms of symmetries and present a method used in our configurator that allocates individuals to specific part names of specific individuals in a way that breaks these symmetries.

The first form of symmetry arises when several individuals directly of a type are possible parts with a part name for an individual. For example, in Figure 3(a) type `A` has part definition `P` with cardinality 1 to 2 with type `B` as the only allowed type. There are two individuals `b-1` and `b-2` of type `B` that can be as a part with part name `P` in `a-1`. The configurations in Figures 3(b) and 3(c) are equivalent. In general, individuals can be picked in a combinatorial number of ways creating a potentially huge number of symmetries. The idea of symmetry breaking is that the possible part individuals directly of the same type are always used in a fixed order. The individuals are ordered by giving them priority rankings. A lower priority individual is not allowed in the configuration if all the higher priority individuals are not in the configuration. Symmetry breaking would thus allow only the configuration in Figure 3(b).

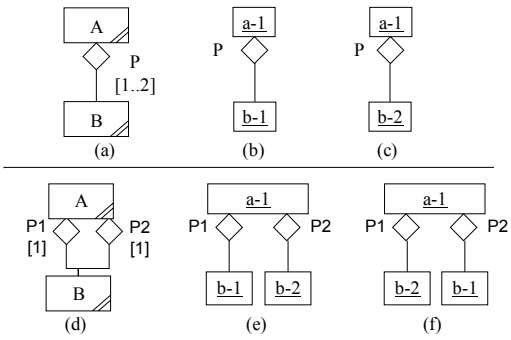

Figure 3. Symmetry breaking

The second form of symmetry is shown in Figures 3(d) to 3(f). Without symmetry breaking any individual of type B could be as a part with any part name in any individual whose type has a part definition that has B as an allowed type. For example, individuals `b-1` and `b-2` of type `B` could both be as a part with part name `P1` or `P2` in individual `a-1`. Our solution for breaking this form of symmetry is to allocate each individual to a specific individual and part name. After allocation either (e) or (f) is allowed, but not both.

4 EMPIRICAL TESTING

In this section we discuss empirical testing of configurators in general and our configurator implementation in particular. We first describe a general test methodology for configurators and continue by describing and characterizing the products used as test cases. We then specify our test setup. Finally, we provide results on the efficiency of our configurator.

4.1 Testing method

In principle, one could test a configurator by using real configuration models or by using randomly generated configuration models. Random configuration model generation could be synthetic or use real products as a seed. Another dimension is the selection between fixed and randomly generated requirements. We chose for this work real configuration models with random requirements.

There is a risk that random models without a large set of real products as a seed do not reflect the structured and modular nature of products designed by engineers. In addition, it is hard to attain a level of difficulty representative of real problems. Knowledge acquisition and modeling for a sufficient seed of real models for random model generation in a justified way would be a major task.

Random requirement generation with progressively larger and thus more restrictive sets of requirements allows one to investigate how well the configurator performs with varying sizes of requirement sets. A dramatic increase in time to find a configuration with some requirement size indicates that the problem becomes critically constrained at that point. The existence of hard configuration problems would then be revealed.

For generating random sets of requirements, we consider how the configuration model appears to the user configuring a product. There are menus (possibly multi-choice), radio buttons or check boxes to select between different alternatives. Guided with these, it is probable that the user will not break the "local" rules of the configuration model, e.g. by requiring alternatives that do not exist or by selecting a wrong number of alternatives. However, a naïve user can easily break the rules of the configuration model that refer to the dependencies of several selections.

We follow this idea by considering the configuration model as consisting of a set of “local” requirement groups. A *requirement group* (*group* for brevity) represents a set of potential requirements that a user could state. For example, a group could represent the selection of a value for the power property of an engine, or the selection of the cooling system in a compressor out of the allowed component types. Each group has a number of *requirement items* each representing a potential requirement. The number of requirements that can be generated from a group is defined by *minimum* and *maximum cardinality*. Note that cardinality applies only if the group is selected to generate requirements.

In our tests, a requirement group is created for each property and part definition of the type of each individual. For each property definition a group with maximum and minimum cardinality of one is created. A value in the domain of the property corresponds to one requirement item. If the property is optional, a requirement item that denies a value for the property is included. A part definition corresponds to one group with maximum cardinality of the part definition. Minimum cardinality is the maximum of one and the minimum cardinality of the part definition. Each potential part individual corresponds to one requirement item. If the cardinality of the part definition includes 0, a requirement item that denies all part individuals is included.

A *test case* contains a number of requirement items related to a configuration model. When generating a test case, a group is randomly selected to generate the number of requirements specified by the minimum cardinality. A group can be selected again to generate one new requirement. Group selection is repeated until the desired

number of requirements has been generated. A group cannot be selected to generate requirements, if the desired number of requirements or maximum cardinality would be exceeded, or if all requirement items are already in the generated requirements.

A requirement is generated from a group representing a property by choosing randomly one requirement item. Generating a requirement for a part is slightly more complex. In our implementation, the order in which the individuals of a given type may be chosen as requirements is important due to the symmetry breaking. Therefore, a requirement is generated by randomly selecting the direct type of the part or the requirement item that denies all part individuals. If a type is chosen, the highest priority individual of that type that has not been required yet is set as the requirement. For example, `hda-1` of Figure 1 would be required before `hda-2`, as they are allocated to the same part name (`msu`) of a component individual (`ssa-1`).

4.2 Case products

We have modeled four real product families using PCML. Three products are screw compressors manufactured by Gardner Denver Oy. Each configuration model represents a complete sales configuration view of a compressor family. The models are detailed to production quality, except for some constant values. The fourth product is a 4-wheel vehicle anonymized by renaming. It was modeled for demonstration purposes and represents about half of the sales view of the product. Numerous optional parts and some constraints were excluded. Despite inaccuracies the model reflects quite well the nature of sales configuration of this configurable product.

The configuration models are characterized in Table 1. Row "Comp. types" gives the number of concrete, abstract and all component types. "Properties" specifies the total number of properties and the number of component types that specify at least one property. "Domain size" indicates minimum, maximum and average domain size of the defined properties. It also gives the number of properties with "small" domain size of 2 or 3, as the average domain sizes are strongly affected by the few large domain properties. "Part defs" specifies the total number of part definitions, the number of component types with part definitions, and the average number of allowed concrete component types. "Cardinality" specifies the number of part definitions with different cardinalities: 0 to 1, exactly 1, and others. "Constraints" specifies the number of constraints and the average number of parts or properties referenced by a constraint. In every compressor model all constraints except one had 2 or 3 references. The exceptional constraints had 262 to 347 references to enumerate allowed combinations of four to five properties. These huge constraints dominate the averages. In all configuration models, the configuration type defined all the constraints and most properties.

Compressor configuration models use almost solely properties. The most complex of these, ESVS, has 3 part definitions. Optional components without properties were modeled as properties.

Configuration model	ESVS	FS	FX	Vehicle
Comp. types c / a / tot	7 / 2 / 9	3 / 0 / 3	1 / 0 / 1	25 / 4 / 29
Properties tot / ct	24 / 5	22 / 3	18 / 1	8 /3
Domain size min – max avg / 2-3	2 – 61 5.9 / 17	2 – 51 5.5 / 14	2 – 44 5.7 / 10	2 - 22 5.8 / 5
Part defs tot / ct / allow	3 / 2 / 2	1 / 1 / 2	0 / 0 / -	16 / 3 / 1½
Card. 0-1 / 1 / max >1	0 / 3 / 0	0 / 1 / 0	0 / 0	12 / 4 / 0
Constraints tot / refs	17 / 20	14 / 25	21 / 12	7 / 2.0

Table 1. Properties of the configuration models

4.3 Test setup

Test setup is illustrated in Figure 4. A WCRL program was generated off-line for each PCML configuration model. For each test case, a new process was created to execute a batch file that executed lparse (version 1.0.4) to generate a BCRL program with a compute statement with the requirements of the test case. The output of lparse was piped to smodels version 2.26 with modifications that suppressed the output of found configurations. Suppressing the output was needed to avoid the configuration task to become I/O bound due to a large number of atoms printed for each configuration. Instead, just the number of found configurations was reported.

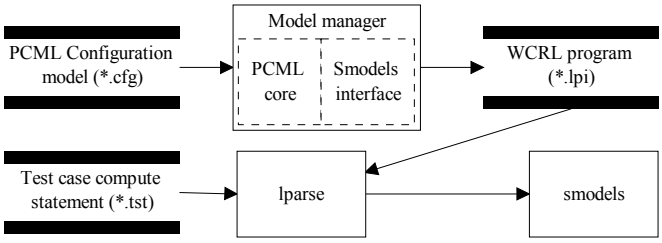

Figure 4. Test setup

For each configuration model, we generated 100 test cases with 2 requirements, 100 test cases with 4 requirements, etc, for each even number of requirements up to the total number of groups. The random requirements in a test case were expressed as a smodels compute statement. If a configuration was found with the requirements of the test case, the test case was considered *satisfiable*, otherwise it was considered *unsatisfiable*.

The tests were run on a laptop PC with 1 GHz Mobile Pentium III processor, 512 MB RAM, and Windows 2000 Professional. All timings were performed using the test system's built-in clock. All times are reported in seconds. A Java based test generator and driver was used to generate and execute the test cases.

The configuration models, test cases, test case run logs, full results as well as the modified Smodels source files and the Windows executable are available at http://soberit.hut.fi/pdmg/empirical_cfg/.

4.4 Results

We briefly explain the measurements before proceeding to the results. *Time to translate PCML to WCRL* includes the time needed by a running Model Manager process to load and translate a PCML configuration model to WCLR and to save the output.

Total duration of a test case includes creating the smodels process for the test case, extracting the number of found answers and the duration reported by smodels, and writing the output log. *Smodels duration* includes the time the smodels executable uses for reading the BCRL program and the time used for computation. *Non smodels time* includes the time to start a test case, run lparse and start smodels, and to gather the results from smodels output (= total duration – smodels duration).

Run time characteristics of the configuration models without the effect of test cases are given in Table 2. The results start with the time to translate PCML to WCRL ("PCML WCRL"), the result is the average of 100 executions. In addition, all configuration models were run once on smodels to find all configurations of each model without any requirements. Table 2 lists the number of configurations ("#Configs") after symmetry breaking, the smodels duration ("Smodels"), as well as the rate of configurations found per second (#Configs/s). "Non smodels" is averaged non smodels time from running the test cases.

Cfg. Model	ESVS	FS	FX	Vehicle
PCML WCRL	8,2 s	8.0 s	5.0 s	1.3 s
#Configs	1,841,356,800	36,106,560	1,136,160	268,800,000
Smodels (s)	18401.4 s	377.1 s	17.0 s	2537.2 s
Configs / s	100066	95748	66833	105944
Non smodels	0,12 s	0,13 s	0,14 s	0,12 s

Table 2. Run time characteristics of configuration models

ESVS		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	89	0,37	189441067	88238	0,30
4	61	0,35	18987439	76849	0,28
6	25	0,34	2234799	72687	0,29
8	9	0,33	211432	19957	0,28
10	4	0,31	1920	263	0,29
12	1	0,32	15552	526	0,29
14-28	0	-	-	-	0,30

Table 3. ESVS compressor results with test cases

FS		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	97	0,33	3583976	84436	0,27
4	76	0,31	706695	74793	0,29
6	55	0,30	90602	55829	0,28
8	26	0,29	10985	20411	0,28
10	15	0,30	1512	4269	0,28
12	10	0,33	719	2173	0,28
14	4	0,30	29	109	0,28
16-22	0	-	-	-	0,28

Table 4. FS compressor results with test cases

FX		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	93	0,23	116646	42200	0,20
4	69	0,22	12558	20116	0,25
6	43	0,21	2537	8552	0,19
8	18	0,21	307	1439	0,19
10	9	0,23	47	181	0,22
12	2	0,27	14	50	0,26
14	1	0,19	5	28	0,20
16	0	-	-	-	0,20

Table 5. FX compressor results with test cases

Vehicle		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	95	0,05	37317928	98238	0,04
4	85	0,05	5855831	88913	0,04
6	59	0,05	747205	84642	0,05
8	33	0,05	108638	71394	0,05
10	22	0,09	29136	54358	0,07
12	8	0,07	9526	42361	0,08
14	4	0,06	289	3853	0,08
16	0	-	-	-	0,08
18	2	0,06	9	140	0,07
20-24	0	-	-	-	0,06

Table 6. Vehicle results with test cases

Tables 3 to 6 show our main results from running the generated random test cases. The first run of the test cases evaluated the performance of finding one configuration that satisfies the requirements. The second run evaluated the performance of finding all the configurations that satisfy the requirements. Each row lists the number of requirements ("#req") and the number of satisfiable cases ("#sat"). Note that the sum of satisfiable and unsatisfiable cases is 100. "Find first" gives the average smodels duration of finding one configuration that satisfies the requirements, and "Unsat" gives the average smodels duration to determine unsatisfiability, taken from the second run. "Find all" gives the average number of configurations per satisfiable case ("#cfgs /case") and the average rate of configurations found per second ("#cfgs / s"). Non smodels time from Table 2 can be added to the results to get the average total duration of finding the first configuration or determining unstatisfiability.

The test arrangement caused occasional random delays of approximately ½ second, possibly due to garbage collection in the Java environment, the functions of the operating system or the virus scanner. Therefore maximum durations are not repeatable and only average results are shown. The maximum non-repeatable smodels time for finding one configuration or determining unsatisfiability was still below 0.7 s. Repeatable times were close to the average, typically approximately within 20% of the average, except for the vehicle model, where average duration was always less than 0.1s causing small absolute errors to show major relative differences.

5 DISCUSSION AND PREVIOUS WORK

In this section we first discuss our implementation and empirical results and compare our empirical results to previous work.

Our results indicate performance adequate both for batch mode configuration and interactive configuration with the simple case products. There were no test cases with repeatable significantly inferior performance. Also, there was no significant change of performance as a function of the number of requirements. The average configurations per second results show weakening with increasing number of requirements. However, this seems to be mostly illusory: because the number of configurations with many requirements is small, Smodels duration comes mostly from reading the BCRL program and from setting up the computation.

Our case products were small but we feel that they are representative of what is needed in sales configuration. We expect that the good performance of our configurator can be generalized to many products suitable for web based sales configuration.

No critically constrained problems were found and no phase transition behavior was apparent. As expected, the number of configurations seems to decrease exponentially as the number of requirements increases. Minor exceptions due to random requirements were encountered in the Vehicle and ESVS configuration models.

The case models had no maximum cardinalities larger than one and a component type was usually used as an allowed type only in one part definition. Therefore the significance of symmetry breaking for performance was low. However, it is evident that several forms of symmetries remain unbroken and new important forms arise when implementing the full ontology, e.g. port and connection oriented concepts.

Our approach in individual generation may create more individuals than needed resulting in unnecessarily large compiled models. On the other hand knowing all individuals can make propagation more efficient in smodels and thus enhance performance. This kind of individual generation was also straightforward to implement in conjunction with our compilation strategy.

Running lparse for each test case conflicts with the knowledge compilation principle and our normal way of using the configurator. As the results indicate, running time of lparse for the case products was small. According to our experiences, the time required by lparse to translate a WCRL program increases significantly with large cardinalities.

We measure performance using execution time due to its practical importance for users and its suitability to searching for phase transition behavior. It would be useful to use metrics that are inde-

pendent of processor power, efficiency of implementation tools and the technology used. Unfortunately such metrics are difficult to define. For example, the number of consistency checks is not commensurate between different technologies such as CSP and logic based approaches. Compromises between propagation and search also significantly affect the number of needed consistency checks.

According to our experiences, the timing result averages are repeatable only to 1/10th of a second. For example, the average time to show unsatisfiability with the Vehicle model differed up to 16 ms between 2 runs making the 1/100th second reading inaccurate. Average times with the FX model varied in one case even by 60 ms between two runs. The repeatability of the results would be improved and maximum durations would become more reliable by executing the test cases several times, excluding the worst results to eliminate the effect of random delays. Fully automatic generation of the test cases from configuration models would make testing easier.

We note that our test methodology can be applied relatively easily to other formalisms like CSP. In CSP, a requirement group could correspond to a CSP variable and a requirement item to one possible assignment for that variable.

We now compare our work briefly with other similar work. Syrjänen configured the main distribution of Debian GNU/Linux using configuration models expressed using an extension of normal logic programs. The configuration task was to select a maximum set of mutually compatible software packages satisfying random user requirements that exclude or include some packages. Average Smodels time for configuration was 1.06 s for Debian 2.0 with 1526 packages and 1.46 s for Debian 2.1 with 2260 packages. The tests were run on a 233 Mhz Intel Pentium II on Linux [4]. The configuration duration was approximately the same as in our largest ESVS model (adjusted for our roughly four times faster processor). Syrjänen's approach seems to perform better than ours as the Debian configuration models are substantially larger.

Sharma and Colomb developed a constraint logic programming (CLP) based language for configuration and diagnosis tasks. Experimental results stem from thin ethernet cabling configuration. The largest 12 node configuration included 126 port connections and required 12 seconds of CPU time on a dual 60 Mhz SuperSparc processor based system to find a configuration [5]. Direct performance comparison to our work is difficult due to port and connection oriented domain, different processor power, and missing details.

Mailharro used the Ilog system to configure the instrumentation and control hardware and software of nuclear power plants. Several thousand component individuals were created and interconnected in about an hour of execution time on a Sun Sparc 20 [8]. The case product is larger and more complex than ours. Direct performance comparison is not possible due to limited details available.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have briefly described a configurator implementation based on mapping its modeling language to weight constraint rules, a form of logic programs. The configurator uses a state-of-the-art general implementation of weight constraint rules for computing configurations satisfying user requirements. Our configurator implementation addressed three issues important for efficiency: off-line compilation of configuration models, limiting the size of a configuration to be finite in a semantically justified way, and breaking symmetries in the set of configurations. However, improved symmetry breaking and generative or more optimal individual generation are subjects for further work.

We then aimed to assess the difficulty of configuration problems as well as the efficiency of our configurator through empirical testing. We defined a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. The methodology enables systematic testing using a small set of products.

We modeled four products taken from two domains from a sales configuration point of view. The modeling language of the configurator was found adequate for modeling these products.

The empirical results indicate that our configurator is efficient enough for sales configuration use. The results also support the common wisdom that configuration problems are relatively easy to solve. However, our small sample of relatively small sales configuration models originates from two domains only. Thus, more tests are needed with larger and potentially more difficult to configure products taken from different domains (e.g. telecommunications and electronics), and modeled from engineering point of view.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Technology Development Centre of Finland (Tekes). The work of the third author has been supported by the Academy of Finland (project 53695). We thank Hannu Peltonen for authoring most of the first version of the PCML specification. We thank him, Andreas Anderson and Jan Elfström for implementing significant parts of the configurator. We also thank Tommi Syrjänen and Patrik Simons for developing Smodels with configuration in mind. Finally, we wish to thank Gardner Denver Oy for sharing the configuration knowledge.

REFERENCES

[1] A. Felfernig, G. Friedrich and D. Jannach, 'UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**, 449-469, 2000.

[2] T. Soininen, I. Niemelä, J. Tiihonen and R. Sulonen, Representing Configuration Knowledge With Weight Constraint Rules. *In Proc. of the AAAI Spring 2001 Symposium on Answer Set Programming*, 2001.

[3] S. Mittal and B. Falkenhainer, Dynamic Constraint Satisfaction Problems, *in Proc. of the 8th National Conf. on AI (AAAI-90)*, 25-32, 1990.

[4] T. Syrjänen, Including Diagnostic Information in Configuration Models, i*n Proc. of the First International Conference on Computational Logic, (Volume 1861 of Lecture Notes in Artificial Intelligence)*, 2000.

[5] N. Sharma and R. Colomb, Mechanising Shared Configuration and Diagnosis Theories Through Constraint Logic Programming, *Journal of Logic Programming* **37**, 255-283, 1998.

[6] M. Stumptner, G. Friedrich, A. Haselböck, Generative constraint-based configuration of large technical systems. *AI EDAM,* **12**, 307-320, 1998.

[7] D. McGuinness and J. Wright, An Industrial-strength Description Logic-Based Configurator Platform, *IEEE Intelligent Systems & Their Applications,* **13**, 69-77, 1998.

[8] D. Mailharro, A classification and constraint-based framework for Configuration, *AI EDAM*, **12**, 1998.

[9] Mackworth A.K. Consistency in Networks of Relations. *Artificial Intelligence,* **8**, 99-118, 1977.

[10] T. Soininen, E. Gelle, and I. Niemelä, A Fixpoint Definition of Dynamic Constraint Satisfaction. *In Principles and Practice of Constraint Programming - CP'99 (LNCS 1713)*, 419-433, 1999.

[11] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, Towards a General Ontology of Configuration, *AI EDAM,* **12**, 357–372, 1998.

[12] P. Simons, I. Niemelä, and T. Soininen, Extending and implementing the stable model semantics. *To appear in Artificial Intelligence, Special Issue of Knowledge Representation and Logic Programming.*