

A Constraint Model for Automated Deployment of Automotive Control Software

Mihai Nica

Bernhard Peischl

Franz Wotawa*

Institute for Software Technology
Graz University of Technology
8010 Graz, Austria
{mnica,bpeischl,wotawa}@ist.tugraz.at

Abstract

In this paper we address automated software deployment for embedded automotive systems in terms of a constraint satisfaction problem (CSP). Our purely model-based approach allows for fully automatic deployment of software functions in a resource-constrained system (exemplified in terms of memory and bus load). Besides of its applicability in an early stage of development, most notably, our model incorporates optimization criteria from algorithmic approaches proposed recently. Capturing the problem-relevant aspects in terms of a CSP is straightforward and thus easily extendable to complex scenarios like, for example, temporal requirements or the diverse bus protocols in the automotive domain.

1 Introduction

Today's upper class cars contain up to 80 ECUs (Electronic Control Units), several bus systems, and about 55 percent of all failures are caused by electronics, software, cables and connectors [1], [2]. More and more functions in today's cars involve electronics and software, 80-90 percent of the new innovative features are realized by distributed embedded systems. Following this mainstream trend, even highly safety critical mechanical and hydraulic control systems will be replaced by electronic components.

In recent years, the focus in engineering embedded automotive systems has been on rather detailed abstractions primarily dealing with implementation related issues like models for code generation. Model-based optimization techniques typically take a back seat in the overall design

*Authors are listed in alphabetical order. The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT) and by the "Fonds zur Förderung der wissenschaftlichen Forschung" (FWF). Special thanks to the reviewers and to our colleague Willibald Krenn for their useful feedback.

process since they lack suitable, standardized notations, methodologies, and integration into the model-driven tool chain.

As today's embedded automotive software is highly distributed, the automotive industry devotes increasing efforts to develop tools for automated software deployment [3]. The underlying foundations comprise techniques like genetic algorithms and various other clustering techniques [3]. However, to our best knowledge, none of the current approaches addresses automated software deployment in terms of a model-based approach. Relying on an algorithmic approach one has to perform measurements to obtain meaningful metrics for certain parameters as, for example, a reference value for the bus load. Besides of the (often) painstaking provision of a prototype to obtain concrete measurements, this considerably hampers the seamless integration into the model-based development paradigm.

In this paper we address the prevalent complexity of automated software deployment in a resource-constrained setting even catering stakeholders at an early development stage, where no reference measurements for a concrete ECU might be available. Our approach relies on modeling software deployment in terms of a constraint satisfaction problem (CSP). Most notably, this model allows for incorporating optimization criteria from algorithmic approaches proposed recently [4]. Moreover, the model-based approach to automated software deployment directly supports an iterative refinement of the model down to the level of protocols, gateways, of software drivers.

This article is organized as follows. In Chapter 2 we present the CSP and its associated parameters, in Chapter 3 we explain how the partitioning problem is solved by means of a CSP, in Chapter 4 we discuss some constraint solvers available on the market. Finally Chapter 5 presents our conclusions and discusses related work.

2 Constraint Satisfaction Problem

Constraint systems are a natural and straightforward way of describing specifications and requirements for hardware and software systems. A *Constraint Satisfaction Problem*

1. $\{ \text{var}_1 = (x_0 < y_0) ;$
2. $\text{min}_1 = x_0 ;$
3. $\text{min}_2 = y_0 ; \}$

Figure 1. Program for computing the minimum between two numbers

Variables: $V = \{ \text{var}_1, x_0, y_0, \text{min}_1, \text{min}_2 \}$
Domains: $D = \{ D(x) = \mathbb{N} \mid x \in V \}$
Constraints:

$$CO = \left\{ \begin{array}{l} \text{var}_1 = (x_0 < y_0), \\ \text{min}_1 = x_0, \text{min}_2 = y_0 \end{array} \right\}$$

Figure 2. The CSP of the program from Fig. 1

(CSP), (V, D, CO) , is characterized by a set of variables $V = (v_1, \dots, v_n)$, each variable having a domain D , and a set of constraints $CO = (c_1, \dots, c_k)$ which defines a relation R between variables. The variables in a relation $R \in CO$ are called the scope S_R of the relation.

Having the program given in Fig. 1 then its corresponding CSP is the given in Fig. 2

There are very effective reasoning algorithms available for CSP, e.g., for computing solutions. A solution of a CSP is an assignment of values to the CSP's variables which does not contradict any given constraint. State of the art constraint solvers are available for solving CSPs. More information about CSPs can be found in Rina Dechter's book on constraints [5].

It is possible, due to a complex constraints system, for a CSP to become *inconsistent*. A CSP is *inconsistent* when there exists no assignment to its variables such that all constraints are simultaneously satisfied. There are several methods for testing the consistency of constraints system. The best known of them are arc consistency, path consistency and n-consistency check. A description of these methods is found in [5]. An optimized method for consistency check is the Max-Restricted Path Consistency method [6].

3 CSP Partitioning

When grouping functions into clusters the partitioning problem appears. For every cluster, the partitioning algorithm must assure that the quality criteria, e.g., time, bus load, together with the resource limitations, e.g., CPU, memory, are fulfilled with respect to the *control unit* (CU) where a cluster is executed. The CSP representation assures a natural way of depicting and combining all these requirements. When we build the CSP of the system we take into account the following types of constraints:

1. *Resource Constraints:* The resources of the CU, on which the cluster is executed, give us the resource con-

straints system. The memory of the CU and the processing power, are criteria which impose restrictions on the cluster that can be executed on the given CU. We cannot execute for example a cluster which needs 500Kb of memory on a CU which only has 300Kb of memory available.

2. *Quality Constraints:* Using quality functions we define the quality constraints. They assure that the system will behave within the given quality criteria. For example, if we want to have the bus load always under 50% then we have to define the quality functions such that this limit is never exceeded. From the quality functions we extract the quality constraints.
3. *Cost Constraints:* The cost constraints are given by the implementation's cost of the CUs. There can be more types of CUs with different properties and different implementation costs. It is possible that although a certain CU is expensive to implement it offers an all around smaller cost than when using 10 CUs that perform the same task. An optimal cost is hard to achieve. These types of constraints are strongly connected with an arbitrary parameter which we call *desired general cost* (DGC). We define the cost constraints such that they always assure that the 'all around system's costs' is smaller than the DGC. We also try to have the costs as low as possible without cutting off too much from the system's performance.

By combining these constraint systems we successfully build the CSP of the analyzed system. A solution to this CSP is a valid cluster partitioning of the system's function blocks.

Observation. It is possible, after combining the above constraints, that the resulted CSP is in an inconsistent state; that is a solution cannot be successfully computed. For example, if through the cost constraints system we specify that the DGC is k and through the resource constraints system we specify that we only have components that cost p , where $p < k$ and $p > 1$. We know we need at least k CUs so that the system can function correctly. Then we have the following constraints system: $(k * p < k) \wedge (p < k) \wedge (p > 1)$. It can be seen that these constraints system has no valid solution. In this situation we have to revise those constraints that can be adapted such that the inconsistent state is removed. In our example if we set the DGC level to $(p * k + 1)$ the constraints system leaves the inconsistent state.

3.1 Resource Parametrization

In order to build the system's CSP we first define the parameters that describe the system's behavior. There are two types of resources that we parameterize: the CUs and the functions that have to be executed by the system.



Figure 3. An abstract representation of the functions's network

We want to define the function blocks distribution for an automotive system over a set of available electronic control units (ECUs).

Within an automotive system there are different functions that have to be implemented. These functions have different safety levels. Some of the functions are safety critical, the ABS function, torque vectoring, or control of the attitude angle, and other have a lower importance degree, e.g., entertainment functions like DVD playing. All these function blocks are connected to each other by means of messages and data exchange mechanisms like bus protocols. Due to this, an automotive system can function correctly. Let's presume that we have t function blocks, $F = \{f_1 \dots f_t\}$, that have to be executed on a minimal number of ECUs. We build the network functions as follows: the nodes of the network are the function blocks that have to be executed. Between functions that communicate there exists a connection in the network. Each connection has a label which denotes the communication frequency between the connected function blocks.

Let $CF = \{CF_1 \dots CF_t\}$ be the set that denotes the communication frequency sets of the system's functions; e.g., $CF_i = \{cf_{i1}, \dots, cf_{it}\}$ is the set that describes the communication frequency of function f_i with respect to all other functions from the system. If there exists cf_{ij} in the set CF_i of a function f_i such that $cf_{ij} = 0$ then it means that there exist no network connection between function f_i and function f_j . A graphical depiction of such a network is given in Fig. 3.

The CF set helps us build the quality functions. The quality function receives as input-parameter the CF_i set of every function f_i , where $1 \leq i \leq t$. After we have built the network of functions we start the parameterizing process for the ECU's. They help us construct the Resource Constraints System. Let $ECU = \{ECU_1 \dots ECU_k\}$ be the set of the available ECUs, then for each $ECU_i \in ECU$ such that $1 \leq i \leq k$. We define mem_i as being the available memory of module ECU_i and $proc_i$ as being the processing power of module ECU_i . Let the set $MEM = \{mem_1 \dots mem_k\}$ be the set of ECU's memories and let $PROC = \{proc_1 \dots proc_k\}$ be the set that describes the processing performances of the available ECUs. The sets MEM and $PROC$ represent the most simplified description of the available resources of an ECU. They suffice to describe how the resource constraints building process takes place. Each distributed system can have supplementary re-

sources that have to be parameterized, but these basic resources are characteristics of every CU found on the market.

3.2 Building the CSP

In order to build the CSP of the system we have to build the three constraint systems: the resource constraints system, the quality constraints system and the cost constraints system. For this purpose we use the parameters introduced earlier: the memory and the computational power available on a given ECU, the memory and the computational power that a function block requires and the communication frequency that exists between functions.

We give the following formal definitions:

Definition 1 (Function Block) Any function block (of t function blocks) is associated with a unique identifier f_i and its processing requirements $pow(f_i)$.

Definition 2 (ECU) Every Electronic Control Unit ECU_i is associated with a processing capacity $max_{ECU} pow_i$.

We start building the *resource constraints system*. The following equations define the constraints system.

1. The overall memory consumption of the function blocks is smaller or equal to the available memory. Usually not all function blocks are executed at the same time, but in the worst case scenario, this trivial safety constraint assures us that no jamming occurs in the function execution process.

$$\sum_i mem(f_i) \leq \sum_j max_{ECU} mem_j$$

2. An adjacent memory constraint is the *maximal function block memory constraint*. That is, let f_{max} be a function block such that the memory requirement of f_{max} , $mem_{f_{max}}$, is the maximum from all functions' memory requirements. There exist an ECU, $ECU_k \in ECU$, with the available memory mem_k , such that $mem_k \geq mem_{f_{max}}$.

3. After we decide to deploy a cluster of functions, $C_j = \{f_i \dots f_{i+n}\}, i \geq 1$, on an ECU, ECU_j , then ECU_j must provide enough memory and processing power to host the deployed functional blocks. The function $deploy(ECU_j)$ returns the indices of the function blocks deployed on ECU_j .

$$\sum_{i \in deploy(ECU_j)} (mem(f_i) \leq max_{ECU} mem_j) \wedge \sum_{i \in deploy(ECU_j)} (pow(f_i) \leq max_{ECU} pow_j)$$

4. A function block is deployed on a single ECU only. $\forall i, j \in \{1..n\}, i \neq j \cdot deploy(ECU_j) \cap deploy(ECU_i) = \emptyset$

5. Any function $deploy$ that distributes all functional blocks f_i on max ECUs is a solution.

$$\{1..n\} = \bigcup_{j=1}^{max} deploy(ECU_j)$$

By unifying the above constraints system we derive the resource constraints system (RCS):

$$RCS : \left\{ \begin{array}{l} 1. \sum_i mem(f_i) \leq \sum_j max_{ECU} mem_j; \\ 2. \exists f_i | f_i \in F, i \in [1, t], \forall j \in [1, t], \\ \quad i \neq j, mem_{f_i} \geq mem_{f_j} \\ \quad \Rightarrow \exists ECU_l \in ECU, l \in [1, k] : \\ \quad \quad mem(ECU_l) \geq mem_{f_i}; \\ 3. \sum_{i \in deploy(ECU_j)} (mem(f_i) \leq max_{ECU} mem_j) \wedge \\ \quad \sum_{i \in deploy(ECU_j)} (pow(f_i) \leq max_{ECU} pow_j); \\ 4. \forall i, j \in \{1..n\}, i \neq j, deploy(ECU_j) \\ \quad \cap deploy(ECU_i) = \emptyset; \\ 5. \{1..n\} = \bigcup_{j=1}^{max} deploy(ECU_j); \end{array} \right.$$

The *quality constraints system* are the most important factor when we partition the function blocks into clusters. In order to build these constraints system we use a set of functions, named *quality functions*. The quality functions offer us a metric for computing the optimal partitioning of the function blocks. The constraints are created by imposing output values that these functions should not exceed for a given cluster. The constraint solver tries to find a set of function blocks such that all the quality constraints are fulfilled. When it finds such a set it creates the cluster.

Besides, as an extra quality constraint, we try to keep the output values of the quality functions to a level close to optimal (such that the cost is minimal). Each quality function receives as input parameter the *CF* set. How this set is built depends on the user and on the described system. There are more solutions proposed for building this set; one, given in [4], proposes a representation of the *CF* set by means of a geometrical matrix. It is beyond the scope of this paper to discuss how the *CF* is created. We presume that the set is already given and use it directly as input for the quality functions.

We build the quality constraints system based on the quality functions set. We use the quality functions presented in [4].

We define the following:

Definition 3 (Cluster's external cost) *It represents the frequency with which the function blocks within a cluster C_i , $i \in [1, c]$, communicate with the rest of the function blocks from the network. We denote this metric through E_i and we compute it as the average *CF* between the function blocks within the cluster and the external function blocks.*

Definition 4 (Cluster's internal costs) *It represents the frequency with which the function blocks communicate with each other within a given cluster C_i , $i \in [1, c]$. We denote this metric by I_i and it represents the average of all *CF* within the cluster C_i .*

Definition 5 (Cluster's diameter) *It represents, based on the *CF* of the function blocks, the average distance between the function within a given cluster C_i , $i \in [1, c]$. We denote this metric through $diamC_i$.*

Definition 6 (Distance between Clusters) *It represents, based on the *CF* of the function blocks, the average distance between a cluster C_i and a cluster C_j , $i, j \in [1, c]$, $i \neq j$. We denote this metric by $d(C_i, C_j)$.*

Definition 7 (External costs between clusters) *It represents, based on the *CF* of the function blocks, the external cost between a cluster C_i and the function blocks of a cluster C_j , $i, j \in [1, c]$, $i \neq j$. We denote this metric by $E(C_i, C_j)$.*

Definition 8 (Cluster's Nodes) *It represents the number of function blocks within a cluster C_i , $i \in [1, c]$. We denote this metric by N_i .*

The quality functions are defined below. Detailed informations about these functions can be found in [4].

1. *The External-Internal Ratio* is a ratio between the external and the internal costs must be as low as possible. That is, a good cluster is a cluster which communicates as little as possible with the other function blocks from the network and that has the internal communication frequency as high as possible. We define for every cluster a communication ratio limit, CRL_{max} , which represents the qualitative limit that every cluster must respect.

$$\forall C_i, i \in [1, c] \frac{E_i}{I_i} \leq CRL_{max}$$

2. *The Davies Bouldin Criteria* shows a good partitioning when the factor is as low as possible. The Davies Bouldin (DB) factor is computed only after all the clusters are formed. We set a limit, DB_{max} that should never be surpassed by the final cluster partitioning. After computing all the clusters c , we compute the DB factor. If it is greater than DB_{max} then the constraint is violated and a new partitioning of the function blocks is performed. If the constraint holds a valid configuration with respect to the DB factor was found.

$$DB = \frac{1}{c} \sum_{i=1}^c max_{j \neq i} \left[\frac{diam(C_i) + diam(C_j)}{d(C_i, C_j)} \right]$$

3. *The Modularization Factor (MF)* is an indicator of a compact clustering of the function's blocks. The value of this factor should be as high as possible. For our constraints system we settle a minimal value, MF_{min} , below which the optimality criteria is violated. If, after computing all the clusters, we observe that the value of MF is smaller than MF_{min} , then the constraint is violated and we discard the partitioning. If the value of MF is greater than MF_{min} then we found a valid solution.

$$MF = \frac{\sum_i I_i}{\sum_i \frac{N_i(N_i - 1)}{2}} - \frac{\sum_{i < j} E(C_i, C_j)}{\sum_{i < j} N_i N_j}$$

4. *The SILHOUETTE factor* (Sh) verifies the correctness of the distribution of a function f_i within a cluster C_i with respect to a neighbor node C_j . The domain of the Sh value of the function f_i is $[-1, 1]$. A good distribution of the functions f_i within a cluster C_i , has the Sh value in the vicinity of 1. For every function f_i , we compute $Sh(f_i)$. If this value diverges with more than δ_{max} from 1 then the constraint is violated, the function is not distributed within cluster C_i and we start the search for a new cluster.

$$Sh(f_i) = \frac{d(f_i, C_j) - d(f_i, C_i)}{\max(d(f_i, C_j), d(f_i, C_i))}$$

5. *The Cluster Load Deviation* (CLD) is computed after all the clusters c are created. Small values of this function denote a good partitioning of the function blocks. In a good case scenario all the clusters have a similar number of function blocks within them. We have the following constraint: the final CLD value of the network must not be greater than an optimal criteria CLD_{max} . If the CLD of the network is greater than CLD_{max} the partitioning of the function blocks is discarded and we restart the partitioning process. If the value of CLD is smaller than CLD_{max} then we have found a valid partitioning.

$$CLD = \sqrt{\frac{1}{c-1} \sum_{i=1}^c (N_i - \bar{N})^2}, \quad \bar{N} = \frac{1}{c} \sum_{i=1}^c N_i$$

By combining the above criteria we build the Quality Constraints System (QCS). The CRL_{max} , DB_{max} , MF_{min} , δ_{max} and the CLD_{max} must be given by the user with respect to the desired system performances.

$$QCS : \left\{ \begin{array}{l} 1. \forall C_i, i \in [1, c] \frac{E_i}{I_i} \leq CRL_{max}; \\ 2. DB = \frac{1}{c} \sum_{i=1}^c \max_{j \neq i} \left[\frac{diam(C_i) + diam(C_j)}{d(C_i, C_j)} \right] \wedge \\ \quad (DB \leq DB_{max}) \\ 3. MF = \frac{\sum_i I_i}{\sum_i \frac{N_i(N_i - 1)}{2}} - \frac{\sum_{i < j} E(C_i, C_j)}{\sum_{i < j} N_i N_j} \wedge \\ \quad (MF \geq MF_{min}); \\ 4. Sh(f_i) = \frac{d(f_i, C_j) - d(f_i, C_i)}{\max(d(f_i, C_j), d(f_i, C_i))} \wedge \\ \quad ((1 - Sh(f_i)) \leq \delta_{max}); \\ 5. CLD = \sqrt{\frac{1}{c-1} \sum_{i=1}^c (N_i - \bar{N})^2} \wedge \\ \quad (CLD \leq CLD_{max}); \end{array} \right.$$

The *Cost Constraints System* (CCS) is built based on the system's cost criteria. Each ECU has a price and a performance description associated to it. We use the following constraints in order to build the CCS.

1. *The Price Constraint*. Given a network of function blocks F , a set of ECUs and a desired general cost DGC , then we have to distribute all the function set F

over a number N_E of ECUs such that the total cost of these ECUs, P_{N_E} is smaller than DGC

2. The *Bus Load Constraint* (BLD) of the system must be lower than an imposed value, BLD_{max} . That is, we have to choose the ECUs on which we distribute the function blocks, such that the bus load of the system is never greater as the imposed value BLD_{max} .

By combining the RCS with the QCS and the CCS we derive the CSP associated to the system:

$$CSP = RCS \cup QCS \cup CCS$$

4 Constraint Solvers

A solution to a CSP is a valid assignment to all CSP's variables that does not violate any of the constraints from the CSP. For solving a CSP we use *CSP solvers*. A CSP solver is a software tool that, by means of different algorithms, tries to detect and remove the inconsistencies from a constraints system and to offer a valid solution to a given CSP. There are more state of the art CSP solvers which can offer good performance when solving a CSP. In this chapter we present four constraint solvers, CHOCO, JaCoP, MINION and TREE*. Each CSP solver has its weak points and its strong points and it depends on the size and type of the CSP which of the constraint solver is best fit for a given task. Because of that a comparison between the CSP solvers is not always possible.

We find a comparison and a description of **JaCoP** and **CHOCO** in [7]. The authors say that the all around performance of the JaCoP solver is better (54% faster) than of the CHOCO solver. However when talking about small CSPs the CHOCO solver is consireably faster than JaCoP. The two CSP solvers have similar specifications with regard to the type of variables and constraints that they can handle. Both constraint solvers offer a wide range of operations for describing constraints. Both tools are free of charge for academic purposes. We find a description of the CHOCO solver in [10] and of JaCoP in [11].

The **Minion CSP solver** is fully presented in [8]. It allows four type of variable's domains and the input language supports definition of up to three dimensional matrices of decision variables. The set of primitive constraints is smaller than by CHOCO and JaCoP. However it includes the basic constraints like *equalities*, *inequalities*, *sum*, *product* and so on. The MINION project is an open source project and is still under development. A particularity of this CSP solver is the fact that variables representation is optimized at hardware level with respect to the solving algorithm (backtracking).

The **TREE* CSP solver** is presented in [9]. The TREE* solver is best suited for binary constraint systems. It can also work on integer variables but it takes a long time to compute a solution. That makes it not a valid choice for big CSPs. It implements the basic operations needed for

creating constraints. A particularity of this CSP solver is that it does not use backtracking in order to get a solution, but simulates instead, by means of tables, all the possible solutions of the constraints from the CSP. It then joins all the constraints and the result is a valid CSP-solution. The TREE* solver is an open source product and still under development.

As our model contains arithmetic as well as logical operators and we are required to provide a rather detailed model, JaCoP and Minion CSP solvers appear to be a reasonable choice for our specific task.

5 Conclusions and related work

Constraints are a natural way of representing complex problems and are often used in the area of configuration and reconfiguration of diverse systems. Constraint Satisfaction Problem (CSP) representations are successfully used in diverse areas from software engineering like configuration and reconfiguration of large systems [13], recommender systems like CAWICOMS which is presented in [15], and software task planning [7].

In this article we outline a novel modeling approach that allows for deployment of embedded automotive software. Our purely model-based approach allows for fully automatic deployment of software functions in a resource-constrained software system. For ease of discussion, we exemplify this for general constraints like memory consumption and bus load, however, our approach can be extended in a natural vein.

Besides of its applicability in an early stage of development, most notably, our model incorporates well-known quality criteria from algorithmic software deployment approaches.

In addition to the straightforward and natural problem representation our model allows for computation of a valid solution satisfying the outlined criteria (resource constraints, quality constraints, cost constraints ...) by relying on standard CSP solvers. Moreover we do not have to generate all the cluster combinations but rely on the first n , $n \geq 1$, solutions that the CSP solver comes up with.

References

- [1] Henrich Druck & Medien GmbH, Challenges for the automotive supply chain, *Association of German Car Manufacturers (VDA) HAWK2015*, Frankfurt am Main, 2003.
- [2] E. Schoitsch, Design for Safety AND Security of Complex Embedded Systems: A Unified Approach, *NATO Advanced Research Workshops, Cyberspace Security and Defense: Research Issues*, p. 161-174, Springer Dordrecht, Berlin, Heidelberg, New York, 2004.
- [3] R. Henia, A. Hamann, M. Jersak, Razvan Racu, Kai Richter, Rolf Ernst, System Level Performance Analysis - the SymTA/S Approach, *IEEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.
- [4] S. Brummund, N. Kehl, P. Nenninger and U. Kiencke, ISODATA Clustering for Optimized Software Allocation in Distributed Automotive Electronic Systems, *SAE World Congress & Exhibition*, Detroit, MI, USA, Session: In-Vehicle Networks, 2006.
- [5] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
- [6] R. Debruyne and C. Bessiere, From Restricted Path Consistency to Max-Restricted Path Consistency, *Principles and Practice of Constraint Programming*, Berlin/Heidelberg, pp. 312-326, 1997.
- [7] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cortes, Using Java CSP Solvers in the Automated Analysis of the Feature Models *GTTSE*, Braga, Portugal, pp. 399-408, 2006.
- [8] I. P. Gent, C. Jefferson and I. Miguel, Minion: A Fast, Scalable, Constraints Solver, *ECAI/Riva del Garda*, Italy, 2006.
- [9] M. Stumptner and F. Wotawa, Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems, In *Proc. 18th International Joint Conf. on Artificial Intelligence*, pages 388–393, Aca-pulco, Mexico, 2003.
- [10] F. Laburthe and N. Jussien, CHOCO constraint programming system, <http://choco.sourceforge.net>, 2003-2006.
- [11] K. Kuchcinski, Constraints-driven scheduling and resource assignment, *ACM Transaction on Design Automation of Electronic Systems (TODAES)*, 8(3):355-383, July 2003.
- [12] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, R. Schafer, M. Zanker, A Framework for Rapid Development of Advanced Web-based Configurator Applications, *AI Magazine*, 24(3), 93-110, 2003.
- [13] G. Fleischanderl, G. E. Friedrich, A. Haselbck, H. Schreiner, M. Stumptner, Configuring Large Systems Using Generative Constraint Satisfaction *IEEE Intelligent Systems*, Volume 13, Issue 4, pp. 59 - 68, 1998.