

# A Generative Constraint Model for Optimizing Software Deployment

Mihai Nica and Bernhard Peischl and Franz Wotawa<sup>1</sup>

**Abstract.** In this article we propose a constraint model for automated software deployment for embedded automotive systems. Unlike to specific algorithmic approaches, our purely model-based approach is applicable in an early system development stage as there is no need to obtain specific measurements from a prototype. Besides of this advantage, the constraint model offers unprecedented flexibility: By taking account of generative constraints, we not solely extend the application scope of CSPs but provide novel model enhancements that allow for further optimizations. Notably, the presented model extension allows for capturing synchronous and asynchronous ECU tasks and is detailed enough to capture the (optimal) alignment of function blocks to ECU tasks under presence of rigorous point to point timing constraints. The article thus further motivates the application of CSPs in engineering embedded automotive software.

## 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 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 provision of a prototype for measurements, this considerably hampers the seamless integration into the model-based development paradigm.

In [17] we address the prevalent complexity of automated software deployment in a resource constrained setting even catering stakeholders at an early development stage. In this article we extend the CSP model presented in [17] by incorporating (1) rigorous point-to-point timing requirements. Moreover, we propose a model enhancement for (2) capturing the optimal alignment of function blocks to synchronous or asynchronous tasks. Notably, as generative constraints typically provide even more optimized solutions with reasonable computational effort, (3) we illustrate how to employ this powerful technique to further optimize our software deployment approach.

This article is organized as follows. Section 2 briefly introduces CSPs and, in particular, generative constraints. Afterwards, in Section 3, we briefly introduce the problem of automated software deployment on a conceptual level and present a general CSP model capable of handling resource-, quality-, cost-, and timing constraints in Section 4. In this section we also outline how to deal with point-to-point timing constraints and address the alignment of function blocks to specific (asynchronous as well as synchronous) tasks. Notably we exploit the powerful mechanism of generative constraints to achieve an optimal solution. Section 5 discusses related work and finally concludes this article.

## 2 Constraint Satisfaction Problems

Constraints systems are a natural and straightforward way of describing specifications and requirements for hardware and software systems. A *Constraint Satisfaction Problem* (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. 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 constraints solver are available for solving CSPs. More information about CSPs can be found in Rina Dechter's book on constraints [5].

Due to poor or contradictory specifications, it is possible for the CSP associated to a given problem to find itself in an *inconsistent state*. We say that a CSP,  $(V, D, CO)$ , is in an inconsistent state, if for its corresponding variables set  $V = (v_1, \dots, v_n)$ , there exists no valid initialization set from the domain  $D$  such that all the constraints from the constraints set  $CO = (c_1, \dots, c_k)$  are simultaneously fulfilled. A CSP is *consistent* when there exists at least one valid initialization of the variables set  $V$  such that all constraints from the set  $CO$  are simultaneously fulfilled.

Sometimes we want to extend the system's functionalities or in-

<sup>1</sup> Authors are listed in alphabetical order, Technische Universit<sup>ät</sup> Graz, Austria, email: {nica,peischl,wotawa}@ist.tugraz.at

sert new components into it. In order to do that we need to expand the current consistent CSP configuration. We do this by means of generative constraints. However, the CSP can now find itself in an inconsistent state. This is why after each modification undertaken to a consistent CSP, a consistency check has to be performed.

The consistency check mechanism verifies if a given CSP is in an inconsistent state. There are several consistency check algorithms that can be successfully applied. The most popular of them are arc-consistency, path consistency and n-consistency check. A description of these algorithms is found in [5]. An optimized algorithm for consistency check is the Max-Restricted Path Consistency algorithm [9].

## 2.1 Generative Constraints

The definition of a generative constraints system is given in [16]. The paper states that a generative constraint satisfaction problem is a tuple  $GCSP(X_0, \Gamma, T, \Delta_0)$  where  $X_0$  is the initial variable set,  $\Gamma$  the set of generative constraints,  $T$  the set of variables types and  $\Delta_0$  is a relation that has the property that each of her tuples  $(x, (t, i))$  associate a variable  $x$  from the set of variables to its type  $t$  and to a unique identifier that indicates that  $x$  is the  $i$ -th variable of type  $t$  [16] states that a *generic constraint* is a constraint over meta variables, where the meta variables are replaced, after the preprocessing step is over, with concrete variables having the same type as the meta variables. A *meta variable*  $M^t$  of type  $t$  is a place-holder for all concrete variables of type  $t$ .

## 3 Problem Statement

By *partitioning* we understand the process of dividing a set of function blocks into groups of subsets of function blocks. Each group of function blocks is executed on an available control unit (CU). We denote such a group as *cluster*. By function block we understand a specific task that has to be executed by the system such that a certain required functionality can be provided. For example the signaling function of a car can be abstractly seen as a 3-tasks set: switch on the signal-commutator, signaling an interrupt to the signal-light controller and switching on the signaling light.

The function blocks deployed on a control unit exchange informations with the function blocks from other control units by means of the main data bus. How the function blocks are partitioned into clusters is decisive for the performance of the entire system. A data-bus load of more than 70% is already critical and its an indicator for a bad partitioning schema. Several algorithms were proposed for computing the partitioning schema of a set of function blocks [4].

We propose a partitioning algorithm based on the CSP representation of the system requirements and on the quality specifications. Moreover we use generative constraints in order to expand our initial CSP such that it can incorporate each new added component.

In order to build the CSP we need to define the system's parameters. We configure the software deployment strategy for an automotive system. Within the auto industry an electronic control unit (ECU) is the computational part on which the software functions associated to different requirements of the system, e.g. DVD-play, ABS, torque vectoring, or control of the attitude angle, are executed. The problem of partitioning into clusters is equivalent to the problem of assigning each ECU a set of function blocks that are periodically executed on it.

In order to exchange data, the function blocks have to communicate with each other. We can see the set of the function blocks as a network where the nodes are the functions. Each connection that is

established between two function blocks has associated an weight. This denotes the communication frequency between the connected functions. One criteria of the partitioning algorithm is taking into account this weights. It is recommended to group together those function blocks that communicate very often.

## 4 A Constraint Model for Software Deployment

The system's final CSP represents a union of three categories of constraints:

1. *Resource Constraints*: The resources of the CU, on which the cluster is executed, give us the resource constraints system. The memory of the CU and the processing power, are criteria which impose restrictions on the clusters that can be executed on the given CU.
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 a quality criteria is a bus load that is always under 40%.
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 type of constraints are strongly connected with an arbitrary parameter that 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 the system's performance.
4. *Time Constraints*: Within an automotive system each software functionality has to be executed within an amount of time. It will be a catastrophe if the braking function would take 30s to execute. When partitioning function blocks into clusters this criteria must also be considered with respect to the ECU on which the deployment is made.

We use the following set of definitions:

**Definition 1 (Function Block)** Any function block (of  $t$  function blocks) is associated with a unique identifier  $f_i$ , its processing requirements  $pow(f_i)$ , the memory requirements  $mem(f_i)$ , and the worst case execution time  $w_{tc}(f_i)$ .

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

**Definition 3 (Bus System)** Every bus system  $B$  is associated with a worst case execution time. We assume a function  $t_{wc}(B)$  returning the worst case execution time given a bus system  $B$ .

**Definition 4 (Point-to-Point Requirement)** Any temporal requirement  $Req_{(i,j)}$  specifies the maximal execution time that is allowed between the source  $i$  and the sink  $j$ .

**Definition 5 (Gateway)** A gateway  $G$  connects two different bus systems. For the underlying conversion process we assume a worst case execution time. Given an gateway  $G$ , a function  $t_{wc}(G)$  returns this worst case execution time.

## 4.1 Resource Constraints

We start building the *resource 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 in 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 function's 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):

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

## 4.2 Timing Constraints

Regarding the timing requirements, for sake of simplicity, we assume that is only a single path from the source  $x$  to the sink  $y$  and that there are no loops in the structural model. Under these assumption the constraint model for the point-to-point timing requirements map to a constraint model as follows.

1.  $t_{xy} \leq \sum_i t_{wc}(f_i) + t_{wc}(B_i) + t_{wc}(G_i)$ .

For every timing requirement  $Req_{(a,b)}$  we instantiate these constraints. We thus can handle the timing annotation in the model in a straightforward an intuitive manner.

A system's software functionality is divided in a finite number of tasks (software blocks) that have to be sequentially executed. Each ECU executes in one execution cycle a number of tasks. This tasks

are not necessarily part of the same software functionality and have to be periodically executed. Each software functionality has to be executed within a certain prior known time. Because of that, time requirements with regard to the task execution are enforced. From this we extract the *time constraints system* (TCS) of the system.

Given a software functionality  $SF$  consisting of a sequential set of tasks  $T = \{t_1, \dots, t_k\}$  and a required maximum execution time  $T_{max}$ . Then if function  $time(t_i)$  with  $i \in \{1, \dots, k\}$ , returns the time necessary to execute task  $t_i$  on the assigned ECU then  $\sum_i time(t_i) \leq T_{max}$ . This constraint is verified after all tasks  $t_i$  of the software functionality  $SF$  are deployed.

Within an ECU, each task as an assigned execution time within the execution cycle, it possible that a task is completely executed after more execution cycles. For example if for a task  $t_i$  we need 4 ms but the execution slice for this task is of 1s, we need 4 execution cycles in order to complete the task. If one cycle takes 10ms to complete, then, if the task is programed to be execute first on the execution cycle, we need 31s in order to execute it. This can lead to a violation of the global time assigned for the software functionality which includes this task. We have to be careful not to neglect this possible inconsistency. Let  $t_{ECU_i}$  be the time that a ECU needs in order to execute a task  $t_i$ , then  $t_{ECU_i} \leq time(t_i)$

An *asynchronous task* of a software functionality is not programed to be periodically executed and is triggered by special events from the outside world. Each ECU has in its execution cycle a slice that is specially assign to this type of tasks. The slice has a fixed amount of time in which it can execute the asserted task. The possible asynchronous tasks are known for each type of software functionalities, e.g for braking we have as possible asynchronous task the ABS functionality; triggered only in special cases. The asynchronous tasks are also included in the partitioning process. The constraints associated to this type of tasks are the same as in the case of synchronous tasks.

## 4.3 Quality Constraints

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 constraints 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 are 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 6 (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 trough  $E_i$  and we compute it as the av-*

erage  $CF$  between the function blocks within the cluster and the external function blocks.

**Definition 7 (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 8 (Cluster's diameter)** It represents, based on the  $CF$  of the function blocks, the average distance between the function blocks within a given cluster  $C_i, i \in [1, c]$ . We denote this metric through  $diamC_i$ .

**Definition 9 (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 10 (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 11 (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 cluster are formed. We set a limit,  $DB_{max}$  that should never be surpass 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 the all 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  $\delta_{max}$  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.

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

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

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

#### 4.4 Cost Constraints

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 with the CCS and the TCS we derive the CSP associated to the system:

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

## 4.5 Generative Deployment of Clusters

When the partitioning process is finished we start to deploy the clusters on the available ECUs. We use the generative constraints in order to choose the best available ECU that suits the requirements of the cluster.

We have different types of ECUs on which we must assign the clusters. We select the cluster that we want to deploy. The requirements of this cluster are generated by means of the generative constraints, e.g. memory, processing power, time to compute the assigned tasks. We choose from the set of ECUs the cheapest ECU and try to deploy the cluster. If, after replacing the meta variables with the concrete variables, an inconsistency happens between the resource provided by the ECU and the requirements of the cluster, we drop the selected type of ECU and chose another one. If the memory was not sufficient then we choose another type of ECU that provides the same processing power but enough extra memory to accommodate the requirements. We repeat this technique until all clusters are deployed. The case when a cluster can not be deployed, because there is no type of ECU to accommodate it, is eliminated in the cluster's building process. When generating clusters we generate them such that no cluster's requirements exceed the maximal resource capacity provided by the best type of ECU.

Using this technique we avoid searching for a best fit ECU. Instead of performing a (expensive) number of comparisons each time a cluster is deployed on a specific ECU, by means of generative constraints we focus exactly on the needed ECU. We do that by using the nogoods provided by the consistency checker. This saves time and computational power.

## 5 Related Work and Conclusion

Constraints are a natural way of representing problems. They are often used in the area of configuration and reconfiguration of system. We use this straightforward and natural way of representing information and by means of a CSP solver we compute a valid solution which satisfies all the CSP criteria. Because of this approach, we do not have to generate all the cluster combinations but just wait for the first  $n$ ;  $n \geq 1$  solutions that the CSP solver delivers. Choosing a constraints solver proves to be a hard task to fulfill. As future work we also want to do an in depth comparison of the constraints solvers available on the market. Representing the partitioning problem as a CSP problem is a good extension to any other clustering algorithm. CSPs are also successfully applied in the area of configuration and reconfiguration of large software systems over a network, e.g. CAWICOMS which is presented in [15]. Other application areas for CSP representation are large business tasks planning.

This article proposes a constraint model for automated software deployment in embedded automotive systems. Unlike to specific algorithmic approaches, our purely model-based approach is applicable in an early stage of system development as there is no need for reference measurements on a prototypical implementation. Our novel CSP model notably extends previous models by incorporating (1) rigorous point-to-point timing requirements, (2) captures the optimal alignment of function blocks to synchronous as well as asynchronous ECU tasks by (3) relying the powerful technique of generative constraints. In terms of a simplified example, our article captures the critical issues in automating software deployment and thus further extends the practical application scope of (generative) CSPs.

## 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.
- [3] R. Henia, A. Hamann, M. Jersak, Razvan Racu, Kai Richter, Rolf Ernst System Level Performance Analysis - the SymTA/S Approach *IEE 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] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On Tractable Queries and Constraints. In *Proc. 12th International Conference on Database and Expert Systems Applications DEXA 2001*, Florence, Italy, 1999.
- [7] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243-282, 2000.
- [8] M. Yannakakis. Algorithms for acyclic database schemes. In C. Zaniolo and C. Delobel (Eds.), *Proc. of the International Conference on Very Large Data Bases (VLDB-81)*, Cannes, pp. 82-94, 1981.
- [9] R. Debruyne and C. Bessiere From Restricted Path Consistency to Max-Restricted Path Consistency *SPrinciples and Practice of Constraint Programming* Berlin/Heidelberg, pp. 312-326, 1997.
- [10] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cortes Using Java CSP Solvers in the Automated Analyses of the Feature Models *GTTSE*, Braga, Portugal, pp. 399-408, 2006.
- [11] I. P. Gent, C. Jefferson and I. Miguel Minion: A Fast, Scalable, Constraints Solver *ECAI* Riva del Garda, Italy, 2006.
- [12] 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.
- [13] F. Laburthe and N. Jussien CHOCO constraint programming system, <http://choco.sourceforge.net>, 2003-2006
- [14] K. Kuchcinski Constraints-driven scheduling and resource assignment, *ACM Transaction on Design Automation of Electronic Systems (TODAES)*, 8(3):355-383, July 2003
- [15] 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).
- [16] A. Felfernig, G. Friedrich, D. Jannach, M. Silaghi, and M. Zanker, Distributed Generative CSP Approach towards multi-site product configuration, *Workshop on Immediate Applications of Constraint Programming (ACP)*, Cork, Ireland, 2003, 100- 123
- [17] To be published within the proceedings of The 20th International Conference on Software Engineering and Knowledge Engineering (SEKE2008)