Improving the Efficiency of Dislocality Constraints for an Automated Software Deployment in Safety-Critical Systems

Dr. Robert Hilbrich
German Aerospace Center
Rutherfordstr. 2
12489 Berlin, Germany
Email: robert.hilbrich@dlr.de

Dr. Michael Behrisch
German Aerospace Center
Rutherfordstr. 2
12489 Berlin, Germany
Email: michael.behrisch@dlr.de

***Abstract*—Mapping software components to hardware resources is a central part of the systems engineering process. This task can be automated by formalization and transformation into a Constraint Satisfaction Problem and the subsequent application of a constraint solver. The toolsuite ASSIST demonstrates the feasibility of this concept. In ASSIST, dislocality requirements can be specified for software components to constrain the set of valid mapping solutions and to ensure reliability and fault tolerance of the system. Three approaches to model these dislocality requirements with constraints are presented. They are compared to each other based on twenty synthetic mapping examples.**

I. Introduction

Engineering complex and safety-critical systems, such as flight control systems aboard an airplane, is still challenging and costly. Despite recent advancements in our model-based tool suites and engineering methods, the design of these systems still bears risk and uncertainties with regard to its outcome.

The formalization and automation of crucial engineering tasks appears to be a promising approach to tackle these challenges [1], [2]. Systems in these areas are engineered to implement a complex interplay between mechanical elements, electronic components, as well as (embedded) software. Therefore, their design has to mirror this interplay and requires the development of a hardware and software architecture.

In practice, these architectures can often be developed independent of each other, but their integration in the final system requires a *link* between the software components and their hardware resources. Creating this *link* is referred to as the *deployment* of the software components. Constructing a deployment requires the systems engineer to *map* software components to resources and to *schedule* the access to shared resources. Therefore, mapping refers to a *spatial allocation*, while scheduling refers to a *temporal allocation* of software components.

The construction of a deployment is an engineering task, which not only affects the fulfillment of functional requirements by providing the necessary resources, but also affects the satisfaction of non-functional requirements, such as safety and reliability. Redundancy and fault tolerance can only be achieved, if critical software components are deployed accordingly. Deploying software components is a very intricate task with zero tolerance for errors as they may jeopardize the correctness of the system. At the same time, it requires a detailed understanding of the requirements of all software components and the capabilities of all hardware resources in the system. Due to the sensitivity and complexity of this task, its formalization and automation is a valuable research goal.

II. Automated Construction of Deployments

In order to achieve an automated construction of a deployment and to argue its correctness, a formalization of the mapping problem is required. For smaller mapping problems, this has been successfully achieved based on Linear Integer Programming [3], [4], SMT-based solvers [5] or evolutionary algorithms [6]. However, these approaches reach their limits when larger, real-world mapping problems are considered. On the one hand, methods based on integer programming techniques are typically unable to provide solutions in a short timespan (20 minutes), when the complexity of the system reaches the level of real-world systems. Heuristic approaches based on evolutionary algorithms on the other hand are usually unable to cope with limited gradient information to guide a search process, because the design space of real-world systems often contains many discontinuities.

The authors instead chose to transform a mapping problem into a semantically equivalent *Constraint Satisfaction Problem (CSP)* [7] and solve this CSP with *Constraint Programming* techniques [8]. The advantages of using Constraint Programming in comparison to other techniques lie in the availability of powerful modeling elements, such as an `allDifferent` constraint, and the ease with which custom search heuristics can be implemented.

A. Constraint Satisfaction Problems

Constraint Programming refers to a set of techniques in artificial intelligence and operations research. These techniques assist in finding solutions for problems based on variables, which are affected by constraints. Each constraint defines valid or invalid solutions for a subset of these variables. In this paper, a subclass of constraint satisfaction problems is used to express mapping problems: *finite domain integer constraint satisfaction problems* in which each variable has a finite integer domain. Solutions for this problem class can be obtained by applying a combination of *search* techniques – including backtracking – and constraint *propagation* techniques for value elimination.

To illustrate the modeling approach of Constraint Satisfaction Problems, consider the well-known *Map Coloring* problem as an example. This problem asks, whether it is possible to color a map with only four colors in such a way, that neighboring countries have different colors. It can be formulated as a CSP by assigning an integer variable x_i for each country with the index i. The domain of each variable corresponds to the four colors: $D_{x_i} = \{0, 1, 2, 3\}$. In order to model the restrictions of this problem, a constraint is added for each pair of adjacent countries. If country x_i is adjacent to country x_j, then $x_i \neq x_j$ is required. The search algorithm is now responsible to select a variable and test a value of its domain. Assuming a simple "first variable, first value" strategy, the variable x_0 would be chosen and set to the value 0 as a test. This would be *propagated* to all variables which are directly linked to x_0 by a constraint, so that the value 0 gets removed from their domains. This removal may lead to other value removals in indirectly linked variables and is processed until a fix point is reached. If a contradiction is encountered or the domain of a variable becomes empty, *backtracking* is initiated, so that the next value of the variable x_0 is tested. Otherwise, the search algorithm continues with the next uninstantiated variable.

This example also shows, that the propagation of the NOTEQUAL constraint is *weak*, because it affects only two variables and invalidates only 4 out of the 16 possible value combinations between two variables.

B. Toolsuite ASSIST

As a proof of concept for the ongoing research toward an automated construction of deployments based on Constraint Satisfaction Problems, the toolsuite *Architecture Synthesis for Safety-Critical Systems (ASSIST)* was developed by the authors. It is publicly available and uses the constraint solver *Choco 4.0.6* [9] internally.

ASSIST allows a systems engineer to automatically construct and optimize mappings based on textual specifications of the software components and hardware resources, *dislocality*, *dissimilarity* and *colocality* requirements, and also optimization goals. The textual specifications in ASSIST conform to a domain-specific language which allows to hide the intricacies of a formal specification.

III. ENSURING FAULT TOLERANCE BY REQUIRING DISLOCALITY

In order to achieve fault tolerance and reliability, it is essential to support "significant differences" in the choice of resources to which critical software components are deployed to. For example, a simple redundancy requirement between two software components, may force the systems engineer to allocate these software components to different processing boards in different locations aboard an airplane. Furthermore, systematic errors and undetected design flaws in hardware components may be addressed by choosing dissimilar hardware resources, processors or memory blocks from different vendors for example.

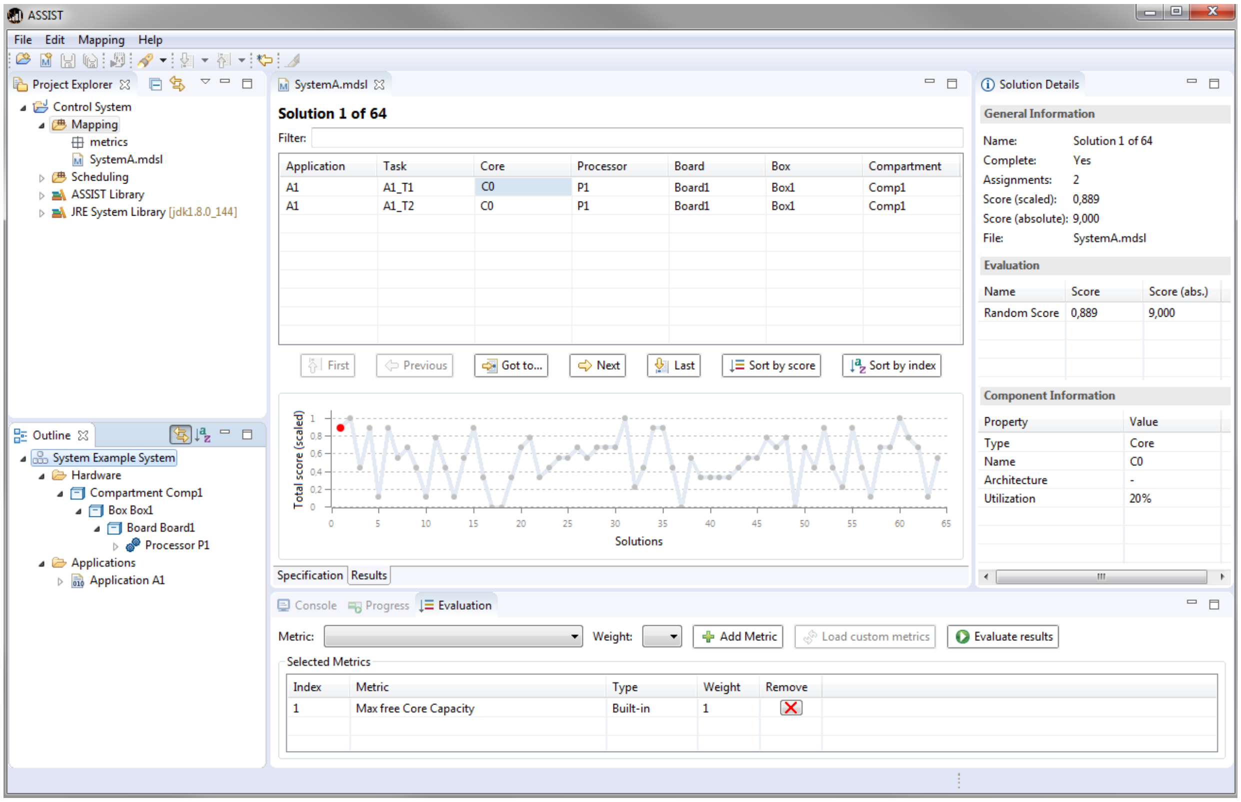
Due to the importance of choosing "different" resources for fault tolerance and reliability in safety-critical systems, engineering tools for an automated construction of deployments need to be able to fully support these choices. ASSIST supports the engineer by offering *dislocality* and *dissimilarity* requirements as part of the domain specific language. They can be used to enforce "differences" for the choice of resources during the mapping process. Finding an *efficient* formulation to express the semantics of each requirement as a Constraint Satisfaction Problem is challenging, but also essential in order to provide an effective toolsuite for the engineering of safety-critical systems.

In order to illustrate the challenges and to present specific modeling improvements, the *dislocality* requirement is used as an example in this paper. Please note, that the concepts developed for *dislocality* requirements can also be applied to *dissimilarity* requirements.

The semantics of a dislocality requirement can be illustrated with the following deployment problem (see Figure 2). In this example system, there are three *applications* consisting of one or more *tasks*. Each task has to be mapped to exactly one of the *processors* in the system. It is assumed, that the processor contains multiple cores, so that multiple tasks can be mapped to a single processor. However, in order to ensure fault tolerance, a *dislocality* requirement is added for all applications. This means, that the applications must not share a processor, so that a faulty processor affects only one application.

Expressing the basic deployment problem with constraints is straight forward. Each task i in the system is represented by an integer variable X_i. The domain of each variable X_i corresponds to the indices of the n processors in the system ($X_i \in \{0, 1, \ldots, n-1\}$). In order to add the constraints for the dislocality requirements, two cases have to be distinguished. In the simple case, in which all applications consist of only one task, a dislocality can be enforced with a single `allDifferent` constraint (see [10]) over all task variables X_i. This case is depicted in Figure 2 (a). Fortunately, Choco 4 already contains an implementation for an `allDifferent` constraint based on the algorithm of Rgin [11], so that the simple case can be implemented.

In real-world systems, applications usually consist of more than one task. Furthermore, tasks of the same application usually share the same processor. This situation is more complex and depicted in Figure 2 (b). Unfortunately, in this case, the previous approach of applying a single `allDifferent` constraint over all task variables X_i can no longer be used. It would prevent solutions in which a processor is shared by



Solution 1 of 64

Filter:

Application	Task	Core	Processor	Board	Box	Compartment
A1	A1_T1	C0	P1	Board1	Box1	Comp1
A1	A1_T2	C0	P1	Board1	Box1	Comp1

Metric:
Weight:

Selected Metrics

Index	Metric	Type	Weight	Remove
1	Max free Core Capacity	Built-in	1	☒

General Information

Name: Solution 1 of 64
Complete: Yes
Assignments: 2
Score (scaled): 0,889
Score (absolute): 9,000
File: SystemA.mdsl

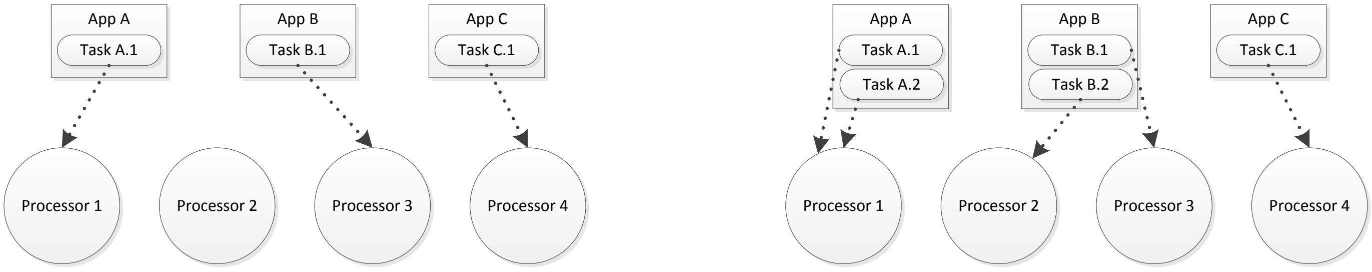
Evaluation

Name	Score	Score (abs.)
Random Score	0,889	9,000

Component Information

Property	Value
Type	Core
Name	C0
Architecture	-
Utilization	20%

Fig. 1. Screenshot of the ASSIST User Interface



(a) Simple Case

(b) Complex Case

Fig. 2. An Example Deployment Problem - Mapping Tasks to Processors

multiple tasks of the same application.

The existing `allDifferent` constraint simply ensures, that values for a list of variables are different. However, the complex case requires an advanced `allDifferent` constraint working with *a list of a list of variables* and ensuring that the combined values for each list of variables are disjunct. Unfortunately, such a constraint is neither part of the Global Constraint Catalog [10] nor available in Choco 4.

IV. Modeling Complex Dislocality Requirements

This section introduces three alternative implementations for the semantics of an advanced `allDifferent` constraint. Their performance and impact on resolution time will be analyzed and compared to each other in the next section.

Before continuing with the description of possible implementations, the semantics of an advanced `allDifferent` constraint should be described more precisely. For this purpose, consider the example system in Figure 2 (b). It consists of the applications A, B and C. Each of these applications contains one or more tasks, i.e. $A = \{A_1, A_2\}$, $B = \{B_1, B_2\}$ and $C = \{C_1\}$. An advanced `allDifferent` for all applications would be working with a list of a list of variables, for example:

$$\texttt{allDifferent}\,\{\{A_1, A_2\}, \{B_1, B_2\}, \{C_1\}\}$$

Assuming that $A^\star$ combines the values for each task in application A:

$$A^\star \in \mathcal{P}(\mathbb{N}) = \bigcup \{A_i : A_i \in A\} \qquad (= A_1 \cup A_2)$$

and $B^\star$ and $C^\star$ do the same for the applications B and C, then the advanced `allDifferent` constraint would ensure, that the sets $A^\star$, $B^\star$ and $C^\star$ are pairwise disjunct.

Element-wise Approach

One option to implement the advanced `allDifferent` constraint semantic is to apply the already existing `allDifferent` constraint for every subset s with

$$s = \{a, b, c \,|\, a \in A, b \in B, c \in C\}$$

For systems with a large amount of applications and more than one task within each application, the amount of constraints, that will be added to the constraint solver by this approach, may significantly prolong the resolution time. However, this approach can be realized with the tools already available in Choco 4 and does not require any additional implementation of custom propagators and constraints.

Instantiation-only Approach

In order to address the drawbacks of the first approach, another option is to implement a custom constraint and propagator for Choco 4, that is able to operate on a list of a list of variables. For the sake of simplicity, the propagator should only react on instantiation events for any of its variables. If an instantiation is detected, then the value of the instantiated variable will be removed from the values of all variables in all of the *other* lists. This approach has the advantage, that only one constraint needs to be added to the constraint solver for each dislocality requirement. However, this instantiation-only approach leads to a weak propagation as it only removes values, when one of the variables is instantiated.

Combined Union-Variables and Instantiation-Only Approach

The third approach tries to build upon the instantiation-only approach and improve the strength of its propagation. For this purpose, a new integer variable $X^\star$ will be added for each list of task variables $X = \{X_1, \ldots, X_n\}$ that were provided to the advanced `allDifferent` constraint. The new variable $X^\star$ will contain the union of the remaining values of all variables in X. This can be achieved with a custom propagator similar to the already existing `PropSetIntValuesUnion` propagator in Choco 4. With these new “union variables” being available, adding a simple `allDifferent` constraint, which links all of the union variables, should improve the effectiveness of the propagation.

Please note, that these “union variables” must not be added to the list of variables for branching as part of the search strategy in Choco 4 as they will not always be resolved to only *one* value if tasks of an application are deployed to different processors. This approach will be combined with the instantiation-only approach from the previous section. In comparison to the instantiation-only approach, the handling of union-variables requires the addition of a new variable and a propagator for each list of variables, so there is some overhead to be expected.

Based on the description of these three approaches to implement the semantics of an advanced `allDifferent` constraint, the next sections will describe experiments conducted by the authors and preliminary results to assess and compare the performance and efficiency of each approach.

V. Experiments

Benchmarking the performance and efficiency of these approaches requires several example to work on. For this purpose, the authors developed a generator for synthetic mapping problems that resemble the typical characteristics of mapping problems encountered in safety-critical systems. Twenty randomized examples with at least one solution were generated for the experiments. They exhibit the following properties.

In each example, there are *two* compartments, each containing between *four* and *six* boxes. Each box contains between *four* and *six* processing boards and every processing board is equipped with *one* or *two* processors each comprising of up to *four* cores. The examples contain between *16* and *24* applications and each of these applications have between *two* and *eight* tasks that need to be mapped to the cores on the processors. All tasks of an application are required to be placed on the same processing board to allow for memory-based communication (colocality requirement). Finally, every example features between *24* and *32* dislocality requirements and each of these requirements refers to at least *four* and up to *six* applications.

Every example was benchmarked on an Apple iMac 5k with 64GB of RAM by using ASSIST 2.3 and Choco 4.0.6. The experiments were done with the *Domain over Weighted Degree* as a variable selector strategy and *Minimum Value First* as a value selector strategy. The examples are publicly available in the ASSIST repository.

VI. Results

The amount of constraints and variables are depicted for each example and each approach in Figure 3 and Figure 4. The results show, that the element-wise approach leads to a substantial increase in the amount of constraints in the solver in comparison to the other two approaches. However, the increase of the variable count due to the addition of union-variables for the implementation of the third approach appears to be rather negligible.

The total resolution time until the first solution has been reached is depicted in Figure 5. Please note the logarithmic scale on the y-axis. The results demonstrate, that the resolution time can be significantly reduced in all examples by adopting the instantiation-only approach.

Further improvements to the instantiation-only approach can be achieved by applying the combined approach, but the additional gains are significantly smaller. In some examples, the combined approach requires slightly more time for a resolution. However, the time differences in these cases are so small, that they may also be induced by external effects,

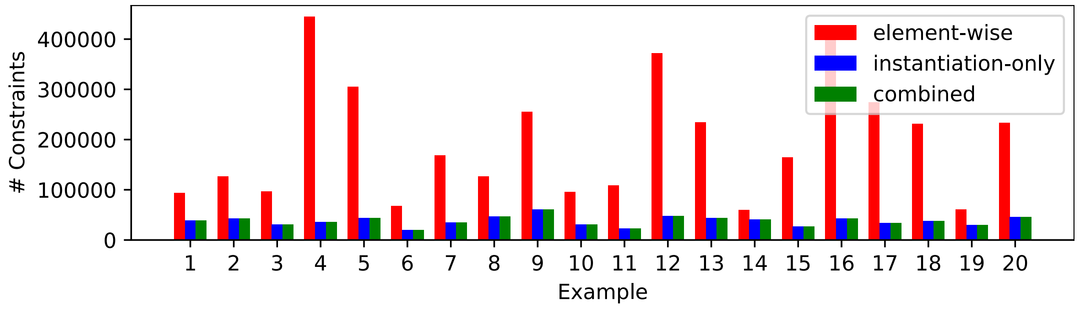


Fig. 3. Amount of constraints in the Choco solver in each example

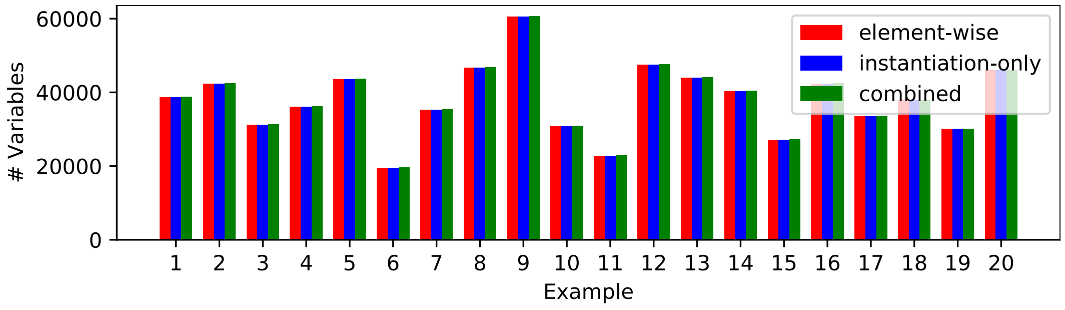


Fig. 4. Amount of variables in the Choco solver in each example

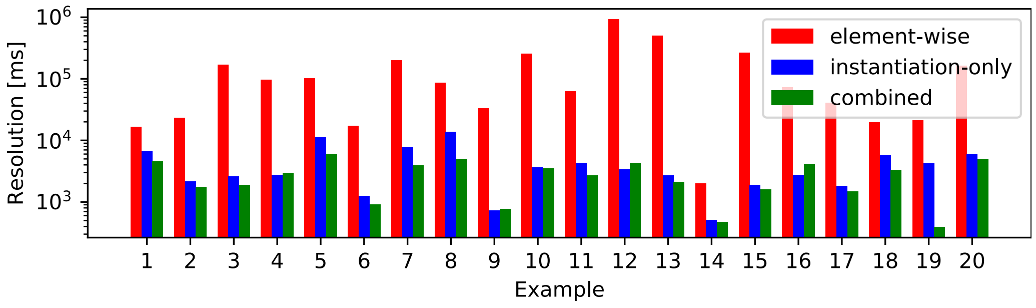


Fig. 5. Resolution time (in ms) to reach the first solution in each example

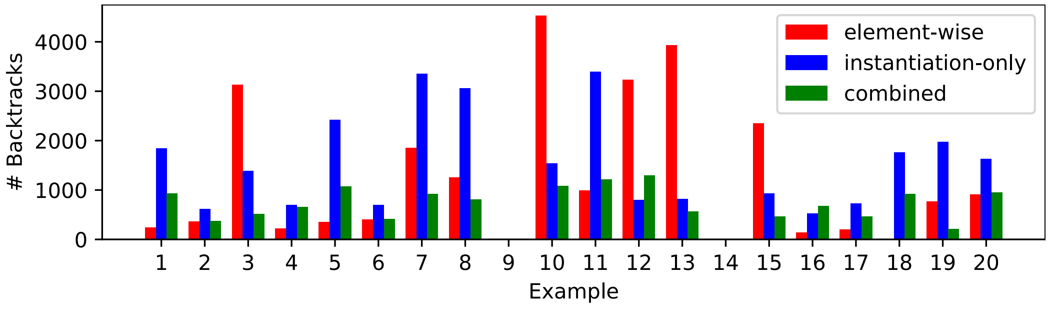


Fig. 6. Amount of backtracking occurring in each example

such as garbage collection in Java or scheduling interferences in the operating system.

Backtracking occurs, when the search process in the constraint solver encounters a *fail* event, i.e. a variable has an empty domain after value propagation or one of the constraints determines a contradiction. Figure 6 shows the amount of backtracking that occurred before finding the first solution in each example. One would assume, that the element-wise approach exhibits the strongest propagation due to the large amount of additional `allDifferent` constraints, so that the amount of fails and backtracks might be smaller compared to the other approaches. In fact, the results are not as clear-cut. Some examples show, that the propagation of the element-wise approach is significantly stronger compared to the other approaches. However, there are also other examples in which the situation is reversed.

VII. Summary and Conclusions

Ensuring significant differences in the choice of resources to which software components in safety-critical systems are deployed to, is essential for satisfying safety requirements and facilitating fault tolerance. ASSIST is a toolsuite for system engineers, which aims to automate the deployment of software components to hardware resources by transforming this mapping problem into an equivalent Constraint Satisfaction Problem. In ASSIST, there are several means to require "differences" in the choice of resources. *Dislocality* requirements are one of those means and they are used as an example in this paper to illustrate the challenge of finding an efficient constraint model.

Three different approaches to implement the intended semantics of dislocality requirements are presented. They specifically address complex use-cases in which applications consist of multiple tasks. While one of these approaches can be realized without any customized constraints or propagators, the other two approaches rely on custom implementations of constraints and propagators as well as additional variables in the third approach.

Experiments conducted with twenty synthesized mapping examples for ASSIST show, that the implementation of custom constraints and propagators is well worth the effort. They lead to a substantial reduction in the resolution time with only minimal overhead due to the additional constraints and variables. The results also indicate, that the element-wise approach, which requires the highest amount of additional constraints, does not always lead to the lowest rate of backtracks in the examples.

Acknowledgment

The authors acknowledge the financial support for this work by the Federal Ministry of Education and Research of Germany (BMBF) in the project "ARAMiS II" (DLR, grant identifier 01IS16025D).

References

[1] R. Chapman, "Correctness by construction: putting engineering (back) into software," in *Proceedings of the 2007 ACM international conference on SIGAda annual international conference*, ser. SIGAda '07. New York, NY, USA: ACM, 2007, pp. 100–100. [Online]. Available: http://doi.acm.org/10.1145/1315580.1315605

[2] R. Hilbrich, *Platzierung von Softwarekomponenten auf Mehrkernprozessoren: Automatisierte Konstruktion und Analyse für funktionssichere Systeme*. Springer Fachmedien Wiesbaden, 2015. [Online]. Available: https://books.google.de/books?id=qu5rCgAAQBAJ

[3] W. Damm, A. Metzner, F. Eisenbrand, G. Shmonin, R. Wilhelm, and S. Winkel, "Mapping Task-Graphs on Distributed ECU Networks: Efficient Algorithms for Feasibility and Optimality," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 87–90.

[4] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs, "Optimizing Automatic Deployment Using Non-functional Requirement Annotations," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. Communications in Computer and Information Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2009, vol. 17, pp. 400–414. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88479-8_28

[5] S. Voss and B. Schatz, "Deployment and Scheduling Synthesis for Mixed-Critical Shared-Memory Applications," in *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, 2013, pp. 100–109.

[6] J. White, B. Dougherty, C. Thompson, and D. C. Schmidt, "ScatterD: Spatial deployment optimization with hybrid heuristic/evolutionary algorithms," *TAAS*, vol. 6, no. 3, p. 18, 2011.

[7] K. R. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[8] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*. ELSEVIER SCIENCE & TECHNOLOGY, 2006. [Online]. Available: http://www.ebook.de/de/product/5834373/handbook_of_constraint_programming.html

[9] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. [Online]. Available: http://www.choco-solver.org

[10] N. Beldiceanu, M. Carlsson, and J.-X. Rampo, "Global Constraint Catalog," online, SICS, Technical Report T2012:03, Feb. 2014, iSSN: 1100-3154. [Online]. Available: http://www.emn.fr/z-info/sdemasse/gccat/

[11] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, ser. AAAI '94. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1994, pp. 362–367. [Online]. Available: http://dl.acm.org/citation.cfm?id=199288.178024