Automatic Construction of Models for Analytic System-Level Design Space Exploration Problems

Seyed-Hosein Attarzadeh-Niaki
Faculty of Computer Science and Engineering
Shahid Behehsti University
h_attarzadeh@sbu.ac.ir

Ingo Sander
School of Information and Communication Technology
KTH Royal Institute of Technology
ingo@kth.se

***Abstract*—Due to the variety of application models and also the target platforms used in embedded electronic system design, it is challenging to formulate a generic and extensible analytic design-space exploration (DSE) framework. Current approaches support a restricted class of application and platform models and are difficult to extend. This paper proposes a framework for automatic construction of system-level DSE problem models based on a coherent, constraint-based representation of system functionality, flexible target platforms, and binding policies. Heterogeneous semantics is captured using constraints on logical clocks. The applicability of this method is demonstrated by constructing DSE problem models from different combinations of application and platforms models. Time-triggered and untimed models of the system functionality and heterogeneous target platforms are used for this purpose. Another potential advantage of this approach is that constructed models can be solved using a variety of standard and ad-hoc solvers and search heuristics.**

I. Introduction

Mapping application models to flexible platforms in multi-processor real-time system design involves design space exploration (DSE) for seeking optimal solutions subject to a set of constraints and optimization criteria [1], [2]. Due to the variety of application and platform models used in this domain, it is extremely beneficial to have a generic framework for automatic construction of DSE problem models for each system design scenario. Also, it is useful to be able to investigate different search heuristics for a DSE problem model [3].

As only the structure of the application and platform models are captured explicitly—typically as graphs annotated with performance and cost metrics [4]—the semantics of the application and platform services are implicitly hard-coded in the DSE tools. Additionally, annotating metrics which depend on mapping decisions, such as application mapping to networks-on-chip (NoCs), in such a representation is not straightforward.

As a result of lacking general and explicit representations for the application and platform models, most of the available analytic DSE methods follow a monolithic formulation of the DSE problem where the semantics of the application, the composition rules of the platform components, and the binding policy are inseparably intermixed to a single problem. Consequently, the current DSE frameworks are restricted to particular classes of application and platform models and are not easily extensible with new models.

To address this problem, we present a framework for automatic construction of system-level DSE problems as the composition of application, platform, and binding sub-problems (Fig. 1). The input models can be provided independently by even different vendors and used by the system designer to create the DSE tool. Being based on the declarative constraint programming (CP) paradigm, all the aspects of the DSE problem are captured coherently as constraints and the problem model is separated from the search heuristics used to solve it. A meta-model is presented to explicitly capture the application, the flexible target platform, and the binding models. A method and a tool are developed for automatic generation of the corresponding DSE problem based on these models. As a key contribution, constraints on logical clocks are used to explicitly express the semantics of application and platform services. The applicability of the proposed method is demonstrated using a set of representative DSE scenarios.

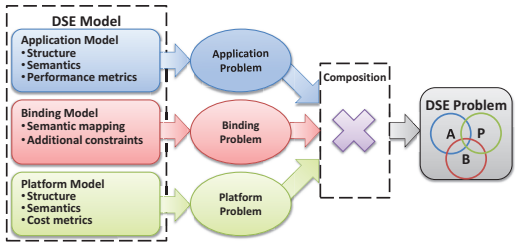

Fig. 1. The proposed approach to construction of DSE problem models.

Related Work: Analytic performance models of application and platforms, such as service curves and event stream [5], [6] address only one aspect of the DSE problem, namely the performance analysis. COSI [7] targets synthesis of optimized on-chip NoCs using integer linear programming. Based on a library of fine-grained platform components and their composition rules more complex communication topologies are constructed. However, the scheduling of the application with different semantics is not considered as a part of the DSE problem. Kuchcinski [8] presents a finite-domain constraint-based model of the classic system-level mapping and scheduling and high-level synthesis problems. The frameworks [9], [10] present a complete approach using CP to perform a throughput-optimum mapping of synchronous dataflow (SDF) graphs of practical size onto a multi-core platform in reasonable time. DesertFD [11] structures the design spaces as an AND-OR-LEAF tree and uses finite-domain solvers to prune the design space. A property composition language is provided to specify constraints on different parameters. In comparison to the above methods, in our framework the DSE problem model is captured as three separate models and is generated automatically. Also, different semantics are supported by exploiting clock constraints.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

II. Preliminaries

1) Constraint programming (CP): CP is a declarative programming paradigm for solving combinatorial constraint satisfaction and optimization problems. Problems are modeled as constraint satisfaction problems (CSPs) and solved by propagation and search. A constraint satisfaction problem (CSP) (X, D, C) is expressed as a set of constraints C which must be satisfied over a set of variables X with given domains D [12]. Each constraint $c \in C$ involves some variables $x_j, x_k, \ldots \in X$ and is a subset of possible values that each variables can take i.e., $c \subseteq d_j \times d_k \times \ldots$. By adding a set of objective functions over the variables a constraint optimization problem can be formulated. Constraints are usually expressed as expressions containing variables of interest, such as $(x_1^2+2x_2 \leq 5) \wedge (x_1 \neq x_2)$ or taken from a global constraint catalog. Propagators implement the constraints by filtering out the values which lead to inconsistent solutions. Afterwards, searching in the resulting solution space is performed by selecting values for each variable based on a given heuristic.

2) Clock Constraint Specification Language (CCSL): CCSL is a companion language to the UML profile MARTE used to express and annotate the timing and causality aspects of events and actions in a model. [13]. In the *multiform* time model of CCSL, several parallel time-lines, modeled by logical clocks, can progress non-uniformly. This matches the behavior of parallel and distributed systems where only a partial causal order between the events of a system exists. The syntax and semantics of CCSL are formally defined in [14]. A CCSL clock is a totally ordered set of instants, one of them highlighted as the *current* instance. When a clock *ticks*, the succeeding instance in that clock becomes the current instance. A CCSL specification consists of a set of clocks and the constraints on them. Clock constraints are either *clock relations*, which are *precedence* or *coincidence* constraints on the instances of two clocks, or *clock expressions*, which are used for defining new clocks. The TimeSquare tool set [15] uses a binary decision diagram (BDD)-based solver to simulate a CCSL specification. We adopt CCSL for describing the semantics of applications and platforms services and use a finite-domain constraint-based formulation to coherently express them together with other types of DSE constraints. This is detailed later in Section III-1.

III. The DSE Construction Framework

A DSE meta-model is proposed to explicitly capture models of the application functionality, the flexible target platform, and the binding policy. A CP formulation of CCSL constraints is used to capture the semantics of applications and platform services. A method and tool is presented to support automatic generation of the corresponding problems and compose them to construct the model of an analytic DSE problem. The declarative style of its constraint-based formulation also separates the problem model from the solving method, enabling reuse of different search heuristics or even complete search methods for the same DSE model.

We use a constraint-based formulation of the DSE problem models to achieve composability. A DSE problem is modeled as a CSP and solved using finite domain constraint solver kits. Assume two CSPs $P_1 = (X_1, D_1, C_1)$ and $P_2 = (X_2, D_2, C_2)$. The composition of P_1 and P_2, denoted by $P_1 \times P_2$ is also a CSP which has the variables $X_1 \cup X_2$ and constraints $C_1 \cup C_2$. If P_1 and P_2 share variables, then we call $X_1 \cap X_2$ the *interface* between P_1 and P_2. In the same style, a DSE problem model P_{DSE} can be defined as

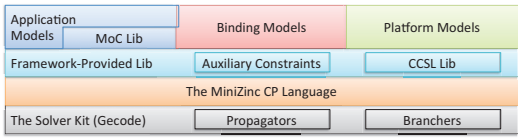

Fig. 2. CP modeling facilities provided by the proposed framework.

$P_{DSE} = P_{app} \times P_{plt} \times P_{bnd}$, where P_{app} is the model for the *application problem*, P_{plt} for the *platform problem*, and P_{bnd} for the *binding problem*. Each of the above three sub-problem models can be automatically generated from the corresponding application, platform, and binding models (Fig. 1). These models comply to the meta-model presented in this Section.

1) CP-based formulation of CCSL Constraints: In this work, CCSL is formulated as a library in the constraint programming language MiniZinc [16] (Fig. 2). A full list of CCSL constraints is presented in [14].

a) Clocks and Time Structure: A clock c is modeled as an array of 0/1 decision variables with the length of the maximum scheduling steps. A *schedule* for a CCSL specification is a function of the form $\mathbb{N} \rightarrow 2^C$. For a schedule σ and an *execution step* $n \in \mathbb{N}$, $\sigma(n)$ is the set of clocks that tick in n. $c[n]$ denotes if c ticks in a scheduling step n. A *clock configuration* χ_σ is a function $C \times \mathbb{N} \rightarrow \mathbb{N}$ which for a schedule σ gives the number of times that a clock $c \in C$ has ticked until step $n \in \mathbb{N}$. More precisely:

$$\chi_\sigma(c,n) = \begin{cases} 0 & n = 0 : \text{the initial configuration} \\ \chi_\sigma(c, n-1) & n > 0, c \notin \sigma(n) \\ \chi_\sigma(c, n-1) + 1 & n > 0, c \in \sigma(n) \end{cases}$$

b) Clock Relations: Clock relations are either synchronous or asynchronous and are used to constrain two clocks with respect to each other. For example, the clock relations *sub-clocking*, *exclusion*, and *synchrony* are synchronous, while the *precedence* constraint is asynchronous. Asynchronous relations are index-dependent and are defined in terms of the clock configurations. For a consistent schedule σ the relation (non-strict) precedence is defined for instance as

$$c_1 \text{ Precedes } c_2 \Leftrightarrow \forall n \in \mathbb{N}, \chi_\sigma(c_1, n) \geq \chi_\sigma(c_2, n)$$

c) Clock Expressions: Clock expressions are used to define implicit new clocks. For example the *union* and *intersection* expressions define the ticks of a new clock point-wise in each step. The *sampling* clock expression defines a new clock which ticks with a base clock that immediately appears after a tick of the trigger clock. The *defer* clock expression is a generic form for implying delay constraints which is specialized to simpler constraints in practice. For example, the *delay on* constraint, delays the ticks of a trigger clock on a base clock a constant number of ticks. A similar constraint *filter by* accepts a clock and a $0/1$ string as its parameter and creates a clock which ticks whenever the input clock ticks and its corresponding entry in the constructed $0/1$ string is a 1. Regular expressions might be used to construct complex $0/1$ strings; for example, by concatenating an initial s_i and a periodic s_p string as $s_q = s_i.(s_p)^w$. This constraint is captured in our library in the following style.

$$c_f \triangleq c_1 \text{ FilteredBy } s_i, s_p \Leftrightarrow \forall n \in \mathbb{N}, \chi_\sigma(c_f, n) = \chi_\sigma(s_q, \chi_\sigma(c_1, n))$$

2) The Meta-model: Fig. 3 presents the framework meta-model. The fundamental elements of this meta-model are: **Decision variables** which represent quantities in the design space and are determined during the DSE (e.g., slot length of a time-division multiplexed (TDM) bus); **Logical clocks** are

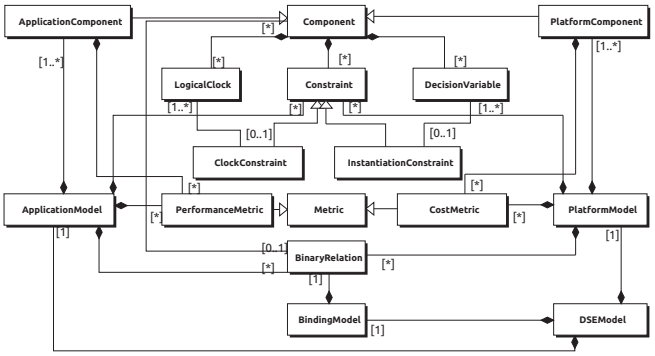

Fig. 3. The DSE metamodel.

CCSL clocks that are subject to scheduling during the DSE (e.g., firing times of an actor); **Constraints** are either instantiation or clock constraints and restrict the possible values of the decision variables and valid schedules of the logical clocks; **Relations** which are represented as matrices of decision variables and can be visualized as directed graphs (e.g., data/control dependencies in a task graph, mapping of application components to platform components, etc.); and **Metrics** appear either as application performance or platform cost metrics and are used for constraining and specifying the objectives of the DSE optimization problem (e.g., throughput of an application, power consumption of a platform, etc.).

a) The Application Model: Both the structure and the semantics of the application models relevant to the DSE problem are captured inn the meta-model. An application model consists of 1) a set of application components, such as dataflow actors or real-time tasks; 2) a set of binary relations on the components, representing a graph; 3) a set of application clock constraints, expressing the timing semantics of the application components; and 4) a set of application metrics, such as performance metrics of the application. Each application component in turn is modeled by a) a set of logical clocks, representing different events associated with it such as invocation, output production, etc.; b) a set of component clock constraints, used to express the timing relation among the events happening within an application component; and c) a set of component metrics.

The current implementation uses an XML-based format to represent the models. The constraints in this model refer to a library provided by the framework (Fig. 2).

b) The Platform Model: In addition to the architectural structure and cost annotations, the platform models also capture the semantics of the services provided to the application. A platform model consists of 1) a set of platform component templates, such as processors, busses, etc.; 2) a set of binary relations on the instantiated components, expresssing an architecture graph; 3) a set of platform instantiation constraints, expressing valid combination of platform components; 4) a set of platform clock constraints; and 5) a set of platform metrics, representing different cost metrics. Each platform component is also modeled by a) a set of variables, used to instantiate and tune a platform component; b) a set of component instantiation constraints, expressing valid instances of a platform component; c) a set of logical clocks, representing different events in a platform; d) a set of component clock constraints, used to express timing aspects of the platform services; and e) a set of component metrics.

c) The Binding Model: The binding meta-model establishes binary relations between the application and platform components and also scheduling the application clocks on top of the platform clocks. More specifically, a binding model consists of 1) a set of binary binding relations from application components to the instantiated platform components; 2) a set of binding constraints on the binding relations; and 3) a set of binding clock constraints which sample application component clocks on the clocks of the mapped platform component.

A standard interface is defined to compose problems generated from arbitrary application, platform and binding models. The application components need to either have a *start* and an *end* clock (computational), or a *read* and a *write* clock (communicational). Platform components have an *exec* clock and a pair of *read* and *write* clocks for each of their ports which are used to schedule the application clocks on them. Also, the binding problem needs to provide a binary relation between the application components domain and the platform components co-domain. For each application/platform component pair in the binding relation, the application clock is sampled on the platform clock. The union of all application clocks mapped onto a platform clock is a sub-clock of the platform clock. To avoid overlapping tasks, the exclusion constraint is posted on every pair of clocks mapped to a platform component.

3) Realization: We use model-driven engineering techniques for capturing the input models and generating the DSE problem. Particularly, the Eclipse Modeling Framework (EMF) is used to capture the meta-model and individual application, platform and binding models based on it. The ATL transformation language is employed as a model-to-model transformation language to generate the final DSE problem based on the captured models. Finally, the Acceleo model-to-text transformation engine is used to generate code for the DSE CSP problem. MiniZinc and Gecode are used for modeling and solving CSPs (Fig. 2). MiniZinc [16] is a medium-level constraint modeling language which is solver-independent and can model different classes of constraint problems. The framework constructs the DSE problem as a CSP and generates corresponding MiniZinc code for it. The Gecode solver is used afterwards to solve the generated problem using a given heuristic. Gecode [17] is a constraint development environment which is deployed as a C++ class library and can be connected to MiniZinc (FlatZinc) as a back-end. In addition to a set of well-known solving techniques in a library, Gecode enables the developers to program their own propagators, branchers, variables, and search engines.

IV. Experiments

To validate the implementation of the proposed DSE construction framework, two applications expressed in different models of computation (MoCs) and two different target platforms are captured and composed with each other using a simple binding policy.

The first application is a JPEG encoder modeled as an SDF graph with 16 actors and 20 arcs. SDF graphs is an untimed MoC with partial orders between the events and is suitable for modeling signal processing applications [18]. An instantiation constraint `buffer_size` ensures the minimum required buffer size. Each arc is equipped two clocks named `read` and `write`. Throughput of an SDF graph can be captured as an application performance metric and might be a subject of optimization. The `sdf_arc` component clock constraint is provided by the framework's CCSL library and

enforces data causality and buffer size limitation. Throughput of an SDF graph can be captured as an application performance metric and might be a subject of optimization. The second application is a time-triggered periodic task set (TS) with three tasks where each task is characterized by its activation period and offset, execution time, and a deadline. This application model is captured using the framework-provided clock constraints in a similar style described in [19] but using our framework's facilities. The first platform model used is a single-core machine (SC) running a real-time operating system (RTOS) with a preemptive scheduler. It includes a single platform component which is instantiated only once where its associated `exec` clock is constrained to be periodic on the base time clock with the period of the length of the RTOS scheduler ticks. The second platform considered is the TDM multi-processor platform similar to the one described in [20] as described below. Its model includes a processing element to which computational application components are mapped and a bus component which has two variables: one indicating the number of its ports and a variable array representing the TDM table. Our bus is configured with 10 TDM slots. Using a biding policy based on classic system synthesis, where application components are mapped to corresponding platform component instances using a graph relation, DSE problem models of the four possible combinations of the above application and platform models are generated. Note that the TS-TDM combination is not a feasible mapping in practice because the self-timed processors of the TDM platform cannot trigger the tasks of TS on time due to lack of synchronization with a base time. The separation of binding policy as a separate problem is a powerful concept. It enables independent formulation of application and platform problems and also investigation of different binding policies.

The DSE generator framework produces the models instantly. For the purpose of these experiments, a simple generic branch and bound solve heuristic is devised which is automatically generated by the DSE generation engine. To form the search tree, the instantiation variables, the component variables, the binding relation matrix, and the CCSL clock variables are branched in order. To resemble the maximal firing policy [14] in the branching, all the clocks are tried for ticking in each step, unless inconsistencies are detected. In this way, the models can be solved with the objective of satisfaction, to produce all solutions, or optimization for an objective based on the performance or cost metrics. The models are solved with the satisfaction goal and the outputs are verified to ensure the correctness of the results. Fig. 4 depicts a sample output of the generated DSE tool. Efficient solving engines and search heuristics can be considered for a fast and scalable DSE.

V. Conclusion

This paper suggests a new design chain model by automatic construction of system-level DSE problem models for arbitrary combinations of application semantics, target platforms and binding models possibly provided by separate vendors. A framework is proposed and implemented using the CP paradigm which provides a meta-model for capturing and characterizing the application, platform, and binding models. These models are composed and used for generation of a constraint satisfaction problem representing the DSE problem of interest. Experiments with timed and untimed application semantics on different target platforms validates the approach.

This work opens new areas for further research. We have not aimed at optimizing the solving time yet. To achieve the ultimate goal of constructing a full DSE tool automatically, compositional formulation of the searching heuristics used for solving the generated models is a complementary topic. Integration of more advanced search engines to enable for example multi-objective optimization is also interesting.

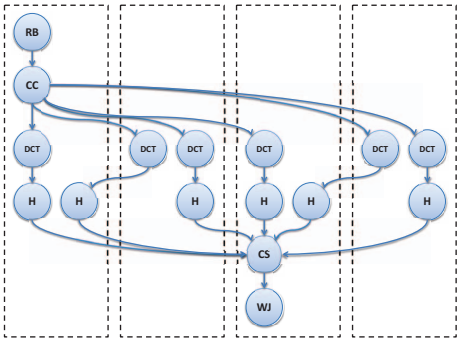

Fig. 4. The JPEG encoder model and a mapping to the individual processing elements of a 4-node TDM bus-based platform, as suggested by the DSE tool.

Acknowledgment: This work has been partially supported by the EU integrated project CONTREX (FP7-611146).

References

[1] A. Gerstlauer *et al.*, "Electronic system-level synthesis methodologies," *IEEE TCAD*, vol. 28, no. 10, Oct. 2009.

[2] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proc. of the IEEE*, vol. 100, May 2012.

[3] S. Bleuler *et al.*, "PISA — a platform and programming language independent interface for search algorithms," in *Evolutionary Multi-Criterion Optimization*, ser. LNCS. Springer, 2003, vol. 2632.

[4] T. Blickle *et al.*, "System-level synthesis using evolutionary algorithms," *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.

[5] L. Thiele *et al.*, "Real-time calculus for scheduling hard real-time systems," in *Proc. of ISCAS*, vol. 4, 2000.

[6] A. Hamann *et al.*, "A framework for modular analysis and exploration of heterogeneous embedded systems," *Real-Time Systems*, vol. 33, 2006.

[7] A. Pinto *et al.*, "A methodology for constraint-driven synthesis of on-chip communications," *IEEE TCAD*, vol. 28, no. 3, March 2009.

[8] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM TODAES*, vol. 8, no. 3, Jul. 2003.

[9] A. Bonfietti *et al.*, "Maximum-throughput mapping of SDFGs on multi-core SoC platforms," *Journal of Parallel and Distributed Computing*, vol. 73, no. 10, 2013.

[10] K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on MPSoCs," in *DATE*, 2014.

[11] B. Eames *et al.*, "DesertFD: a finite-domain constraint based tool for design space exploration," *Design Automation for Embedded Systems*, vol. 14, no. 1, 2010.

[12] F. Rossi *et al.*, *Handbook of Constraint Programming*. New York, NY, USA: Elsevier Science Inc., 2006.

[13] C. André *et al.*, "Modeling time(s)," in *Model Driven Engineering Languages and Systems*, ser. LNCS. Springer, 2007, vol. 4735.

[14] C. André, "Syntax and semantics of the clock constraint specification language (CCSL)," INRIA, Rapport de recherche RR-6925, 2009.

[15] J. Deantoni and F. Mallet, "TimeSquare: Treat your models with logical time," in *Proc. of TOOLS*, ser. LNCS, vol. 7304. Springer, May 2012.

[16] N. Nethercote *et al.*, "MiniZinc: Towards a standard CP modelling language," in *CP 2007*, ser. LNCS. Springer, 2007, vol. 4741.

[17] Gecode Team, "Gecode: Generic constraint development environment," 2013, available from http://www.gecode.org.

[18] F. Mallet *et al.*, "The clock constraint specification language for building timed causality models," *Innovations in Systems and Software Engineering*, vol. 6, 2010.

[19] M.-A. Peraldi-Frati and J. DeAntoni, "Scheduling multi clock real time systems: From requirements to implementation," in *ISORC*, 2011.

[20] S. H. Attarzadeh Niaki *et al.*, "Automatic generation of virtual prototypes from platform templates," in *Languages, Design Methods, and Tools for Electronic System Design*, ser. LNEE. Springer, 2015.