SMFF: System Models for Free

Moritz Neukirchner, Steffen Stein and Rolf Ernst
Technische Universität Braunschweig
Braunschweig, Germany
neukirchner|stein|ernst@ida.ing.tu-bs.de

***Abstract*—Evaluation of scheduling, allocation or performance verification algorithms requires either analytical performance estimations or a large number of testcases. In many cases, e.g. if heuristics are employed, extensive sets of testcase systems are imperative. Oftentimes realistic models of such systems are not available to the developer in large numbers.**

We present SMFF (System Models for Free) - a framework for pseudorandom generation of models of real-time systems. The generated system models can be used for evaluation of scheduling, allocation or performance verification algorithms. As requirements for the generated systems are domain-specific the framework is implemented in a modular way, such that the model is extendible and each step of the model generation can be exchanged by a custom implementation.

I. Introduction

During the development of e.g. scheduling or allocation algorithms or algorithms for performance verification, test-case systems are required to evaluate the applicability and performance. If formal proofs of correctness or analytically derived perfomance estimations can be given a small set of such systems is sufficient. However, in many cases this is not possible e.g. if heuristics are employed. In this case the algorithm has to be tested with an extensive set of testcases. For many algorithm developers, especially in academia, system models are not available in large numbers. Manually creating such system models is very time-consuming and might not respect requirements on randomness.

Consider the following example. A developer has implemented a heuristic algorithm for optimized priority assignment in distributed real-time systems. As the algorithm is based on a heuristic, no analytical estimation of the performance of the algorithm can be given. Thus the developer has to evaluate the algorithm against a set of testcases. As no extensive set of testcases is available to the developer, the system models have to be generated automatically. These models however need to resemble real-world systems, as typical for the targeted domain. Furthermore, they need to be sufficiently random, in order not to bias the evaluation.

In this paper we adress this issue and present SMFF - a framework for parameter-driven generation of models of distributed real-time systems. The generated models incorporate a description of the platform, of the software applications mapped onto the platform and the associated scheduling and timing paramters, thus covering the entire model specification.

As system models, that are used for algorithm evaluation, have to resemble real-world systems, requirements on testcase systems may be highly domain- and problem-specific. The presented framework provides a high degree of modularity, allowing the user to extend the system-model and to replace algorithms for system model generation, thus making the framework a universal tool for testcase generation. The algorithms presented in this paper are example implementations and were developed for the evaluation of an algorithm to find execution priorities in static-priority-preemptively (SPP) scheduled systems under consideration of timing constraints [1].

The key features of the SMFF framework as presented in this paper are:

- Parameter-driven generation of complete system models for use as testcases
- A Modular framework architecture to allow exchange of generation algorithms
- Extendible data structures to allow customization of the system model

The SMFF framework is no simulation or benchmarking environment. Thus, we do not adress the issues of simulation or performance monitoring. We rather provide models as input for such tools.

This paper is structured as follows. First we will discuss how the process of system model generation can be structured into discrete steps. We will then give an overview of related work and how previous approaches relate to this structure. In section IV we will define the main terms and the system model used throughout the paper. The following sections will address the single steps of the model generation and the implemented algorithms. Section IX covers aspects of the implementation of the framework and its modularity. In section X we will present an example of system model generation with the SMFF framework. Then we will conclude the paper.

II. General Approach

In this section we will outline the general approach to generate system models of distributed real-time systems.

We propose to devide the process of model generation into six steps as depicted in figure 1. The six steps can be grouped into the categories *structural definition*, *real-time property definition* and *constraint definition*. While some steps can be executed independent of each other, some require other steps to be performed beforehand (indicated by arrows).

The structural definiton is composed of the *platform model generation*, the *application model generation* and the *application mapping*. During platform model generation the architecture graph is constructed. Thus this steps defines the topology

of the platform. In the application model generation step, application graphs are created defining the logical structure of software applications. The final step of the structural definition maps the application models onto the platform model, defining the distribution of the application on the hardware platform. Naturally this step can only be performed after architecture and application models have been created. All steps of the structural definition are highly domain-specific, as hardware platforms, application structure and their distribution on the platform are generally very diverse for different application domains.

Real-time property definition is composed of the steps of *assignment of timing properties* and of *assignment of scheduling parameters*. In the first step tasks and communication between tasks are assigned an execution model such as best-case execution and worst-case execution times. Furthermore tasks can be assigned an activation model (e.g. activation period and activation jitter). Thus this step defines the timing of each entity of an application in isolation. This step can only be performed after the application models have been generated. In the second step scheduling parameters, such as execution priorities for static priority preemptively (SPP) scheduled resources, are assigned to the tasks and communication. This step can only be executed after the application mapping, as scheduling paramters depend on the scheduling strategy of the resources tasks and communication have been mapped to. After these two steps the timing behavior of the system is completely specified.

The last category is composed of only one step - the *assignment of timing constraints*. In assignment of timing constraints limitations on e.g. end-to-end path latencies or ouput jitters can be defined. Also this step can only be performed after the application model has been created, as constraints are specified for entities of an application.

Except for the outlined dependencies the steps of the system model generation can be performed in arbitrary order and independent of each other. This gives great flexibility to the developer as it is possible to e.g. load manually defined platform models and only perform the remaining steps of the testcase generation with the provided platform model. Also the developer may introduce further dependencies to tailor the testcase generation process for his specific needs. E.g. one may require testcases with certain load-distributions on the platform resources and thus make the assignment of timing properties depend on the application mapping.

III. Related Work

We will now review previous approaches to generation of testcase system models and highlight how these approaches relate to the structure of testcase generation as described above.

For evaluation of real-time algorithms (e.g. scheduling, allocation or performance verification) many developers rely on handcrafted example systems, to highlight strengths and shortcomings of different approaches (e.g. [2], [3], [4]). Others rely on benchmark models of real applications such as MPEG2 decoders (e.g. [5]) or on more comprehensive benchmark suites as e.g. [6]. A third approach many developers take for evaluation and comparison of their algorithms is parameter-driven testcase generation (e.g. [7], [8]).

If any of the first two approaches is taken, the results of an algorithm evaluation tend to be fairly well reproducible as the set of system models is well specified and oftentimes publicly available. However the number of testcases typically is low. As a result statements about average performance of an algorithm may be inaccurate, as the examples might not be representative of the targeted domain of application or might not cover common corner-cases properly. This may bias the evaluation of an algorithm.

The approach of pseudorandom parameter-driven testcase generation can provide more accurate results on average performance, as an algorithm can easily be evaluated against a large set of system models. Many developers use custom algorithms to generate testcases automatically. Common approaches are to select a system topology (i.e. platform model and application model) manually and to assign timing properties in specified bounds (e.g. [8]) or to use a fixed platform and generate task sets (i.e. application models) automatically (e.g. [7], [9]). In both cases common corner-cases, that might only occur for e.g. certain platforms, might be neglected. To the best of our knowledge there exists no single approach that addresses all steps of parameter-driven pseudorandom system model generation. Instead parts of the generation process have been addressed.

A fairly comprehensive tool for automatic testcase generation is task-graphs for free (TGFF) [10]. TGFF generates task graphs based on a parameter set that allows detailed influence on the topology. Furthermore it allows to generate timing properties for all tasks (periods and execution times) and latency constraints on paths. Thus it adresses the three steps of application model generation, assignment of timing properties and assignment of timing constraints. TGFF has been widely used for generation of task sets (e.g. [11], [12], [13]). However, if the algorithm under evaluation targets distributed systems with communicating tasks, platform topology and application mapping may be relevant. TGFF is not able to perform these steps of model generation, though.

The steps of application model generation and assignment of timing properties have also been studied thoroughly in the scope of single und multi processor schedulability analysis. Here benchmarking includes generation of task sets and assignment of timing parameters such as execution times and activation periods. In many cases timing parameters of fixed-size task sets are assigned so that a specific processor load is accomplished by randomly assigning activation periods and setting the execution times to match the required utilization (see e.g. [14], [8]). [15] gives an analysis of properties of commonly used algorithms to accomplish this task and discusses their respective properties with respect to the task set parameters generated. It shows that the chosen algorithm can bias the benchmark favoring one schedulability test or the other and proposes new algorithms for timing parameter generation that have been widely adopted in the community [16], [17], [18]. These findings should be considered when designing an algorithm for timing parameter generation in the

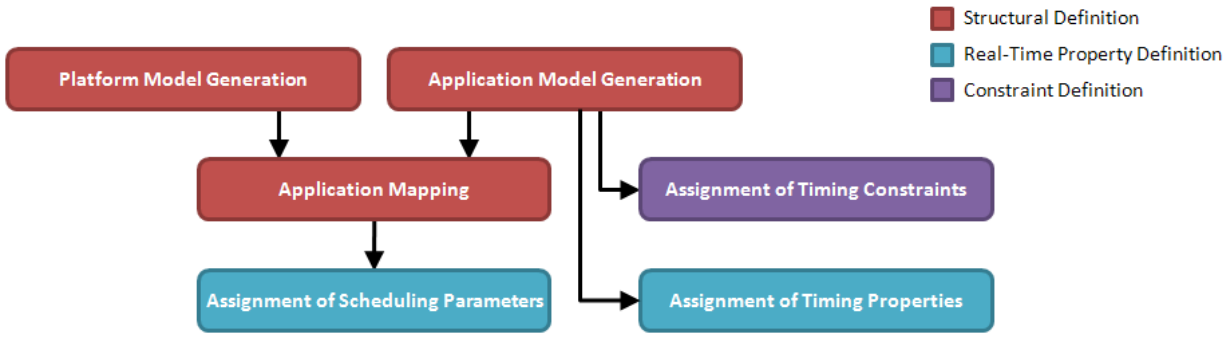


Fig. 1: General testcase generation flow

described flow of pseudorandom system generation.

We are not aware of any tool capable of generating complete system models covering the entire testcase generation flow. With the SMFF framework we aim to incorporate all steps of the testcase generation into a single framework, allowing developers to generate complete system models to evaluate their algorithms. In the following we will first provide a detailed description of the system model and then address the single steps of the testcase generation flow.

IV. System Model

In this section, we will briefly introduce the main terms used throughout the paper and elaborate on the system model.

The term *platform* refers to the hardware software will run on. We assume that a platform $\mathcal{P}$ consists of set of processors (*computational resources*) $\mathcal{R}_{comp} = \{\rho_{comp,0}, \ldots, \rho_{comp,i-1}\}$ interconnected by a set of communication media (*communication resources*) $\mathcal{R}_{comm} = \{\rho_{comm,0}, \ldots, \rho_{comm,j-1}\}$. The platform will be modeled as a bipartite graph

$$G_{Arch} = \{\mathcal{R}_{comp}, \mathcal{R}_{comm}, E\}$$

with the two types of resources $\mathcal{R}_{comp}$ and $\mathcal{R}_{comm}$ being the two types of vertices. The undirected edges $e_q \in E$ denote connectivity between two resources.

Applications are running on the platform. An application $\mathcal{A}_p$ consists of a set of *tasks* $\Gamma_{\mathcal{A}_p} = \{\tau_0, \ldots, \tau_{n-1}\}$ and a set of communication channels (*task links*) $\Lambda_{\mathcal{A}_p}$. By task we understand a piece of code that will be scheduled by a micro kernel. By a task link $\lambda_k \in \Lambda_{\mathcal{A}_p}$ we understand a communication entity that may be established/scheduled on computational and communication resources. An application will be modeled as a directed bipartite graph

$$G_{App} = \{\Gamma_{\mathcal{A}_p}, \Lambda_{\mathcal{A}_p}, C\}$$

with the task and task links being the two types of vertices. Edges $c_p \in C$ denote communication of a task via a task link while information flows in the direction of the edge. Furthermore an edge denotes a precedence relation between adjacent vertices. All task links have an in-degree and out-degree of 1. This model slightly deviates from the common model of task graphs, where tasks are modeled as vertices and communication between tasks as edges. We chose to modify this model to provide better expressiveness for mappings of task links. Regular task graphs can easily be transformed to the modified model by replacing edges by a task link vertex and two edges. For the modified model we additionally define adjacency of tasks. Two tasks are *adjacent* to each other if a task link exists that is adjacent to both tasks. This notion of adjacency is identical to adjacency of tasks in regular task graphs.

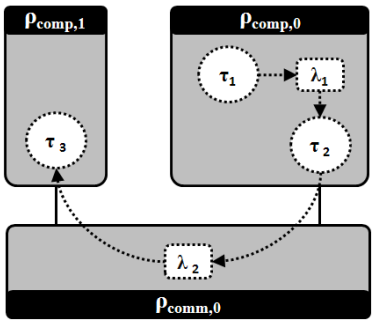


Fig. 2: System Model

Tasks are mapped on resources, signifying that a task is executed on that resource. Task links can be mapped on computational resources or communication resources indicating that communication is established via/on this resource. Thus, the mapping of an application

$$m : \Gamma_{\mathcal{A}_p} \cup \Lambda_{\mathcal{A}_p} \rightarrow \mathcal{R}_{comp} \cup \mathcal{R}_{comm}$$

assigns each task and task link of the application to a resource.

An example system is depicted in figure 2. It is composed of a platform of two computational and one communication resource and one application of three tasks that communicate over two task links. Task link λ_1 is mapped on a computational resource, while λ_2 is mapped on the communication resource.

As we are dealing with real-time systems timing properties can be defined. A task τ_i can be assigned an activation model and an execution model. An activation model may be a specification of e.g. an activation period P_{τ_i} and an activation jitter J_{τ_i}. An execution model can be specified by e.g. best-case and worst-case execution time (BCET and WCET) C_{b,τ_i} and C_{w,τ_i}, respectively. Task links that are mapped on a communication resource can also be assigned an execution model.

In addition to the timing properties a set of constraints $C =$

$\{\chi_0, \ldots, \chi_{k-1}\}$ on timing properties can be provided. This can include e.g. constraints on end-to-end path latency or output jitter.

In the following sections we will explain the single steps of system model generation. We will present possible characterizations for the system models and present examplary algorithms for each step as used in [1].

V. Platform Generation

When generating platform models for use as testcases, the models should resemble real-world systems as closely as possible. Real-world embedded systems however exhibit a diversity of platform architectures depending on application domain. Possible criteria for platform characterization are e.g.:

1) **Size**
   Embedded real-time systems differ in size to a large degree. Very small-size systems of only one or a few processors (e.g. heating control) as well as large-scale systems such as an avionic platform exist.
2) **Degree of connectivity**
   The communication structure of systems may be diverse. Bus architectures, in which all computational resources can directly communicate with each other, are common as well as architectures where communication between two resources can only be established via a gateway.
3) **Degree of heterogenity**
   Systems may be composed of a homogenous set of computational and communication resources or may consist of a large number of different types of resources.

When generating testcase systems, the developer should know how real-world systems of his targeted domain can be characterized e.g. with respect to the above criteria. The employed platform generation algorithm should then be capable of producing platform models with this characterization.

We have implemented an examplary algorithm for generation of homogenous platforms that is parametrizable w.r.t. size and degree of connectivity. The generated platforms exhibit the following properties:

- G_{Arch} is a connected component.
- G_{Arch} is bipartite.
- $\forall \rho_{comm,i} \in \mathcal{R}_{comm} : deg(\rho_{comm,i}) \geq 2$. I.e. each communication resource is connected to at least 2 computational resources.

Algorithm 1 shows the pseudo-code for the platform generation. The algorithm requires only two parameters. `numRes` directly defines the number of computational resources (`numRes`$= |\mathcal{R}_{comp}|$) in the system. It allows the user to scale the overall system size. The second parameter (`CRes%`) controls the degree of connectivity of the platform. It allows to set the average number of communication resources as percentage of the number of computational resources. E.g. if set to 5%, on average the generated platforms will have 0.05 times as many communication resources as computational resources. As a result low values of `CRes%` will tend to generate bus-like architectures, while higher values will tend to create architectures with gateways.

Algorithm 1 Platform Generation

INPUT: `numRes`, `CRes%`
1: `numCRes` = value from Binomial Distribution(n=`numRes`, p=`CRes%`)
2: **for** $i = 1$ **to** $i =$`numCRes`-1 **do**
3: connect $\rho_{comm,i-1}$ and $\rho_{comm,i}$ to random $\rho_{comp,rand}$
4: **for** all $\rho_{comp,l} : deg(\rho_{comp,l}) = 0$ **do**
5: connect $\rho_{comp,l}$ to random $\rho_{comm,rand}$
6: **for** all $\rho_{comm,p} : deg(\rho_{comm,p}) < 2$ **do**
7: connect $\rho_{comm,p}$ to random $\rho_{comp,rand}$
8: Random pruning of superfluous connections

In the first step of the algorithm (line 1) the number of communication resources is determined based on a binomial distribution with n=`numRes` and p=`CRes%`. The following step (lines 2-3) iterates over all comm. resources and connects two consecutive comm. resources to a random comp. resource. This step assures that all communication resources are contained in one connected component. The following two steps (lines 4-5 and lines 6-7) assure that all comp. resources are connected to at least one comm. resource and that all comm. resources are connected to at least two comp. resources, respectively. In a last step (line 8) connections are pruned, such that no two computational resources are connected to the same two communication resources, if this does not violate any of the required platform criteria.

VI. Application Generation

Also generated application models should resemble real-world systems as closely as possible. As for platform architectures, application graphs of real-world embedded systems exhibit a large diversity in topology. Possible characterization metrics for application graphs are:

1) **Size**
   Applications differ in size. This can be characterized e.g. by the number of vertices in the application graph.
2) **Connectivity**
   Applications may be diverse in terms of communication dependencies. Some applications might be highly parallel, while others might be strictly parallel. This can be characterized e.g. by maximum in-degree and out-degree of vertices of the application graph.
3) **Cyclicity**
   Some applications e.g. control algorithms are cyclic in nature. As oftentimes this is not supported by analysis algorithms it can be sensible to exclude these systems from evaluation.

TGFF [10] already provides sophisticated algorithms for parameter-driven task graph generation. It has been used extensively in other projects (e.g. [11], [12], [13]). The exemplary algorithm of SMFF is based on TGFF but extends the functionality by support for cyclic task graphs. The remaining parameters are identical to the old algorithm of TGFF (for a complete description of TGFF's functionality refer to [19]).

Algorithm 2 shows the application model generation in pseudo-code. As input the algorithm accepts 5 parameters. `numTasks` and `diffNumTasks` define the number of tasks in the task graph and the allowed deviation from that number, respectively. With `taskMaxDegrIn` and

Algorithm 2 Application Generation

INPUT: `numTasks`, `diffNumTasks`, `taskMaxDegrIn`, `taskMaxDegrOut`, `cyclicGraph`

1: get task graph from TGFF(`numTasks`, `diffNumTasks`, `taskMaxDegrIn`, `taskMaxDegrOut`)
2: **if** `cyclicGraph` **then**
3: undirect edges
4: find cycles
5: redirect edges
6: convert task graph to application model

`taskMaxDegrOut` the maximum in-degree and maximum out-degree of the tasks can be specified. These parameters are passed directly to TGFF (line 1). TGFF can only create non-cyclic task graphs. If cycles are allowed in the created applications (parameter `cyclicGraph`, line 2), the direction of all edges of the task graph is deleted (line 3). A cycle search is performed on the undirected graph (line 4) and edges are directed, such that the cycles found, are also cycles for the redirected graph (line 5). In a final step (line 6) the task graph is transformed to the framework's application model.

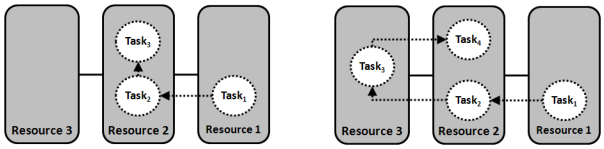
VII. Application Mapping

The application mapping completes the structural definition of a system model and determines the degree of distribution of an application on the platform. Applications may be either clustered, i.e. all tasks are located on only a few resources, or widely spread, i.e. tasks are distributed across many resources. However some domains may have very specific requirements on application mapping. E.g. if the generated system model should resemble a client-server setup, a path of the application graph should start and end on the same resource while some intermediate task has to be mapped on a different resource. I.e. there should be a task $\tau_i : deg_{in}(\tau_i) = 0$, a task $\tau_j : deg_{out}(\tau_j) = 0$ that are mapped onto the same resource and at least one more task $\tau_l : deg_{in}(\tau_l) \neq 0, deg_{out}(\tau_l) \neq 0$ that is mapped onto another resource. This example illustrates, that the application mapping of generated system models has highly domain-specific requirements, that have to be taken into account in implementation and parametrization of mapping algorithms for testcase generation.

The example mapping algorithm, as implemented for evaluation of the algorithm presented in [1], is tailored to generate mappings for "sensor-actuator-like" applications. It only supports task chains without forks or joins and tends to distribute these chains across several resources. In particular, we assume that such sensor-actuator applications are characterized by the following features:

1) **Task chains do not traverse a computational resource more than once**
If two tasks are mapped on the same resource, all tasks on the path between these tasks are also mapped on the same resource.
More formally, let I_{τ_i,τ_j} be the set of tasks that are part of the path between tasks τ_i and τ_j. Then

$$\forall \tau_i, \tau_j \in \Gamma_{\mathcal{A}_p}, m(\tau_i) = m(\tau_j) : \forall \tau_l \in I_{\tau_i,\tau_j}, m(\tau_l) = m(\tau_i)$$



(a) Valid Mapping (b) Invalid Mapping

Fig. 3: Valid and Invalid Mappings

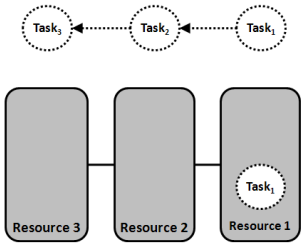


Fig. 4: Mapping to "spread" applications

Figure 3 shows examples of valid and invalid mappings according to this criterion. Figure 3a shows a valid mapping, as for every pair of two task mapped on the same resource, all tasks between are mapped on the same resource as well. Figure 3b shows an invalid mapping. Task 2 and task 4 are both mapped on resource 2 although their intermediate task 3 is mapped on a different resource.

2) **Applications are distributed across several resources**
The probability of a task being mapped on a resource increases with the distance to other resources that have a task of the same application mapped on them.
Figure 4 shows an example for this criterion. The upper part depicts the task graph of the application to be mapped. At a point in the mapping process only task 1 is mapped to resource 1. If the next task to be mapped is task 3, its probability to be mapped on resource 3 is higher than that of resource 2 which is again higher than that of resource 1. This assures that applications consisting of several tasks are not clustered on only a very small set of resources.

The mapping is performed by a probabilistic algorithm that enforces the first criterion and lets the user control the degree of distribution with a parameter. Its pseudocode is shown in Algorithm 3.

As long as the application is not completely mapped (line 2) the algorithm alternatingly selects the first/last task of the task chain, that is not yet mapped (lines 3-6). In the following step the set of resources this task can be mapped on is calculated based on distances in the platform graph and the distances to already mapped tasks of the application graph. For the first task this set includes all resources. Then the probabilities with which the task is mapped to any resource of this set are calculated (line 8). This step is based on a weighted random number generator, which selects a value from a given set with a probability corresponding to the value's weight (non-normalized probability mass). If a resource violates criterion 1 its weight is set to 0. The probabilities for all other

Algorithm 3 Application Mapping

```
INPUT: kPredecessor, kSuccessor, kResDist
 1: temp=0
 2: while not all tasks mapped do
 3:     if (temp++)%2==0 then
 4:         get first unmapped task of task chain
 5:     else
 6:         get last unmapped task of task chain
 7:     calculate set of possible resources
 8:     calculate probability of all possible resources
 9:     if temp ==1 then
10:         apply kResDist to probabilities
11:     map task on resource based on probability distribution
```

resources are initialized with 1 and are then modified by the three parameters that are passed to the mapping algorithm. `kPredecessor` and `kSuccessor` are factors applied to the weight of a resource, if the predecessor or successor of the task to be mapped, is mapped on that resource. The third parameter `kResDist` is applied only to the end of the task chain (line 9). For each resource the distance to the resource that the first task was mapped to is calculated. Then its weight is multiplied by $\texttt{kResDist}^{\text{distance}}$. Thus, depending on the value of `kResDist` the application tends to be either spread across many resources (for `kResDist` > 1) or clustered on a small set of resources (for `kResDist` < 1). In a final step a resource is chosen from the set of possible resources based on probability mass and the task is mapped to that resource.

VIII. Definition of Real-time Properties and Constraints

The three steps for definition of real-time properties and constraints typically are closely related. Again consider the example of the developer who has implemented an algorithm to assign execution priorities in SPP scheduled systems w.r.t. to end-to-end latencies. To evaluate the algorithm the developer might require testcases, which violate some path latency constraints while other constraints are not violated. This can only be accomplished if the steps of assignment of timing properties, of scheduling paramters and of constraints are implemented to work together.

The three example algorithms for definition of real-time properties and constraints as presented below were also implemented in the scope of the evaluation of the algorithm from [1].

A. Assignment of Scheduling Parameters

Along with the framework we provide an examplery algorithm for assignment of scheduling parameters for SPP scheduled resources. Execution priorities for tasks and task links, that have no scheduling parameter assigned yet, are set randomly. Already assigned execution priorities remain unchanged.

B. Assignment of Timing Properties

In this step of testcase generation the timing properties of all tasks and task links are defined. In SMFF this comprises assignment of activation period P_{τ_i}, activation jitter J_{τ_i}, BCET C_{b,τ_i} and WCET C_{w,τ_i}. SMFF uses the UUniFast algorithm presented in [15] for this step. UUniFast assigns task execution times, such that the resource utilization assumes a specified value, while the distribution of the task execution times is uniform.

In the SMFF implementation of UUniFast activation periods are assigned uniformly in a specified interval [`minActPeriod`, `maxActPeriod`]. The user furthermore specifies an interval for resource utilization [`minResU`, `maxResU`]. From this interval a random value is chosen for each resource and the task execution times are assigned according to UUniFast to achieve that resource utilization. BCETs are assigned as user-specifiable percentage of WCETs [`bcetPercentage`].

C. Assignment of Timing Constraints

The assignment of timing constraints for a testcase system model is of particular importance when evaluating algorithms for optimization or design space exploration, as it may significantly influence the number of feasible system configurations (i.e. configurations that do not violate any constraint).

The implemented algorithm is again targeted at sensor-actuator-like applications. It defines paths from the first task to the last task of a task chain. The value of the constraint is defined as multiple of the sum of all WCETs along the path. This factor - the constraint *laxity* - is randomly selected from a user-defined interval [`minLaxity`, `maxLaxity`]. Smaller laxity values result in more tightly constrained systems.

IX. Implementation and Modularity

We will now focus on aspects of implementation of SMFF and the modularity of the framework. As previously indicated, requirements on generated system models may be diverse, depending on application domain, scheduling algorithm and algorithm under evaluation. Thus the presented implementations of the generation steps may not be suitable for every user and the system model might not be sufficient.

To account for the diverse requirements the testcase generation framework is implemented in a modular way, allowing extension of the data structures as well as replacement of all generation steps. The framework was implemented in Java for ease of development and platform independency. The next paragraphs give a short overview of the software architecture of SMFF, highlighting the aspects of modularity and extendibility.

A. Model Generation Infrastructure

In order to enable a flexible combination of the system model generation steps described in this paper, SMFFs generation logic is based on an aggregation of factories, each responsible for one step of the generation process. For an overview refer to figure 5. All factories are grouped in a central system generation factory orchestrating the order of the single generation steps. The SMFF model generation core library as depicted in figure 5 does not provide any generation logic, but merely defines the relationship between the single factories and their APIs to ensure seamless integration of different

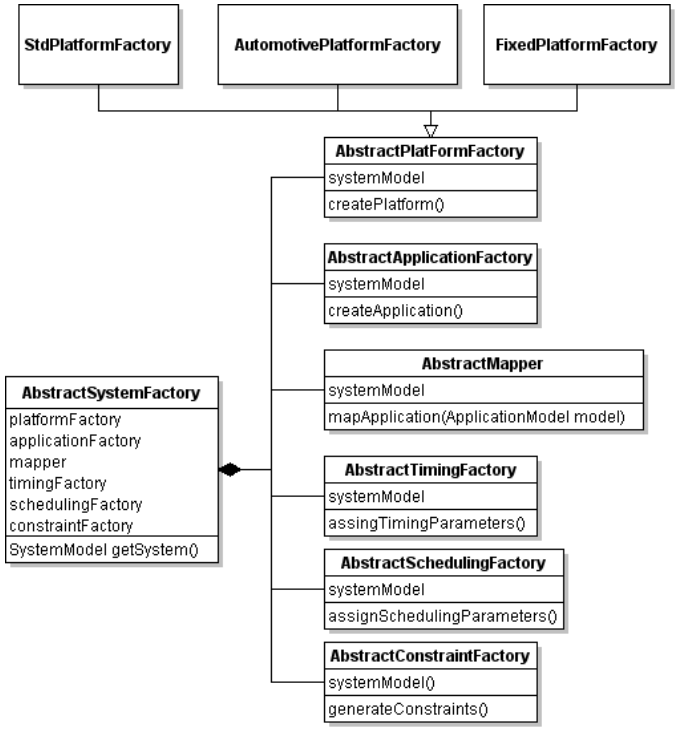


Fig. 5: Organization of System Generation Stages

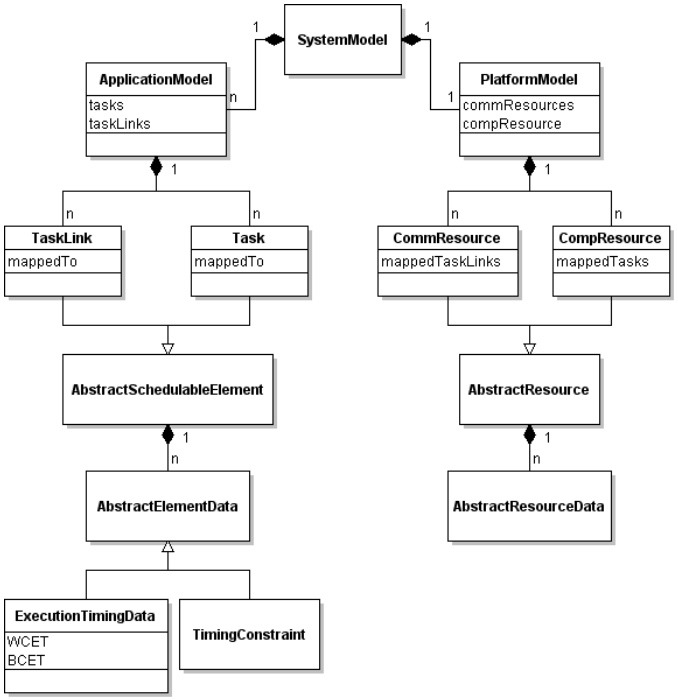


Fig. 6: System Model and Extension Points

implementations of the model generation stages. An actual implementation, as the one discussed in this paper needs to provide an implementation for each of the factories.

The algorithms described in this paper are shipped with SMFF as standard implementations, e.g. StdPlatformFactory. As an example, figure 5 shows two additional possible implementations of a platform factory (AutomotivePlatformFactory, FixedPlatformFactory), which could be integrated into the flow. This shows that a user of the tool may replace e.g. the standard platform generation logic with a platform factory specific to his own requirements. The specific custom implementation could e.g. be the generation of one static platform that is used for all experiments (FixedPlatformFactory). The user could then keep using the standard implementations of the other factories for generation of the remaining parameters of the system model.

Similarly, e.g. the default order imposed on the stages of system model generation can be altered by replacing the standard implementation of the system factory by a user-defined one.

B. Model Representation

Next, we present the system model data structure as shown in figure 6. It consists of a platform model and multiple application models. Platform models are comprised of computational and communication resources, represented by the CommResource and CompResource class respectively. Similarly, application models consist of tasks and task links, each represented by a distinct class. Additionally, application and platform models manage adjacency information about their parts defining the bipartite graphs described in section IV. A mapping is specified as a relation between task or task links and communication and computational resources. In the diagram this is indicated by the relevant local variables.

In order to reflect the flexibility of the model generation framework also in the system model data structure, all model elements allow the attachement of additional classes for future extension of data and functionality without modification of the basic data structure. Figure 6 shows that application as well as platform model components may be associated with a set of data elements. Example extensions as have already been mentioned in this paper are shown for a task or a task link. These may be annotated with timing properties (such as best- and worst-case response time) or timing constraints (such as jitter constraints). Although not shown in the figure, similar data extension points also exist for the system model as well as the application and platform model, thus enabling a very flexible extension of the model depending on the specific use-case of SMFF.

C. Framework Extensions

This flexible data structure allows usage of the generation framework for multiple purposes. For example, we implemented an interface to the performace verification tool SymTA/S [20] - the *SymTA/S Adaptor* - to allow verification of timing constraints. This also allows to e.g. only generate systems that are schedulable. All relevant data and functionality needed to transform the system model to a SymTA/S compliant representation as well as retrieving data from the SymTA/S tool is encapsulated in data extensions of model elements.

Another existing extension is a *Visualization Plugin* to the SMFF framework. It enables the display of a graph

Algorithm 4 Sample System Model Generation

```
INPUT: systemFactoryData
1: // create system factory
2: StdSystemFactory systemFactory = new StdSystemFactory(systemFactoryData);
3: // create new system model
4: SystemModel systemModel = systemFactory.generateSystem();
5: // create XML file
6: new ModelSaver("SystemModel.xml").saveModel(systemModel);
7: // create PDF file
8: PdfPrinter.convertToPdf(systemModel, "systemGraph.pdf");
```

representation of the system model as well as the export to a pdf file. This allows the user to quickly grasp the platform and application graphs and the application mappings of the generated system models.

The *XML Load/Store Plugin* completes the SMFF model generation suite, allowing easy integration with other tools. Additionally, when incompletely specifying a system model in XML - e.g. when using another tool to generate a partial model, SMFF can be used to generate the remaining parameters and save the resulting - completely specified - system as XML.

X. Testcase Generation Example

In this section, we give a brief example on how to use SMFF in practice. We assume that all necessary factories for the system generation stages are present. We used the algorithms specified earlier on in this paper for generation of the example systems shown below.

Algorithm 4 shows the necessary code to generate a system from a set of parameters as needed for the different model generation stages, which we assume to be given. In order to generate a system model, one merely needs to instantiate a system factory, in this case the StdSystemFactory supplied with SMFF (line 2). A (potentially new) system model is generated each time the `generateSystem()` function of the factory is called (line 4). The following lines show the code necessary to save the model to an XML file (line 6) and create a pdf file (line 8) containing a graphical representation of the system.

Figures 7 and 8 show two example systems that have been generated by SMFF and exported using the visualization plugin. The relevant parameters to the system model generation algorithms as described in the previous sections are summarized in table I. The parameters in column 1 resulted in the system model shown in figure 7, the model shown in figure 8 corresponds to the parameter set in the second column. Note that multiple application models were generated for both systems. Each application model is depicted in a separate color.

XI. Conclusion

In this paper we have presented *System Models for Free* (SMFF) - a framework for parameter-driven generation of models of distributed real-time systems. SMFF can generate completely specified system models, including specification of platform architecture, of application models, mapping of applications and definition of timing and scheduling parameters and timing constraints.

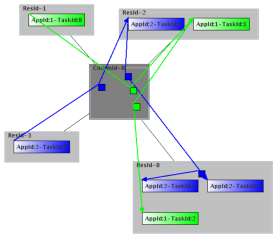


Fig. 7: Small Example System Model

Param. Set	1	2
numRes	4	12
Cres%	35%	35%
minTasks	2	3
maxTasks	4	7
kPredecessor	3.0	3.0
kSuccessor	3.0	3.0
kResDist	1.5	1.5

TABLE I: SMFF Factory Parameters

As illustrated with examples, the user can easily and quickly generate pseudorandom system models for use in his field of application, thanks to supplied standard implementations of parameter-driven factories. More advanced users can take advantage of the flexible infrastructure of SMFF, by interchanging implementations of system generation steps or extending the system model to tailor SMFF to fit their specific needs.

If you would like to use SMFF in your project, feel free to contact any of the authors.

References

[1] M. Neukirchner, S. Stein, and R. Ernst, "A lazy algorithm for distributed priority assignment in real-time systems," in *under submission*, 2010.

[2] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour, "Influence of different abstractions on the performance analysis of distributed hard real-time," *Design Automation for Embedded Systems*, vol. 13, no. 1-2, pp. 27–49, June 2009.

[3] J. Real and A. Crespo, "Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal," *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, March 2004.

[4] J. G. García and M. G. Harbour, "Optimized priority assignment for tasks and messages in distributed hard real-time systems," in *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1995, pp. 124–132.

[5] T. Cucinotta and L. Palopoli, "QoS Control for Pipelines of Tasks using Multiple Resources," *IEEE Trans. on Computers*, vol. 59, pp. 416–430, 2010.

[6] A. R. Weiss, "The standardization of embedded benchmarking: pitfalls and opportunities," in *Computer Design, 1999. (ICCD '99) Int'l. Conf. on*, 1999, pp. 492 –508.

[7] T. D. ter Braak, P. K. F. Hölzenspies, J. Kuper, J. L. Hurink, and G. J. M. Smit, "Run-time spatial resource management for real-time applications in heterogeneous mpsocs," in *Proc. of DATE'10*, 2010.

[8] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proc.. 22nd IEEE*, dec. 2001, pp. 193 – 202.

[9] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *RTSS*, 2006, pp. 111–126.

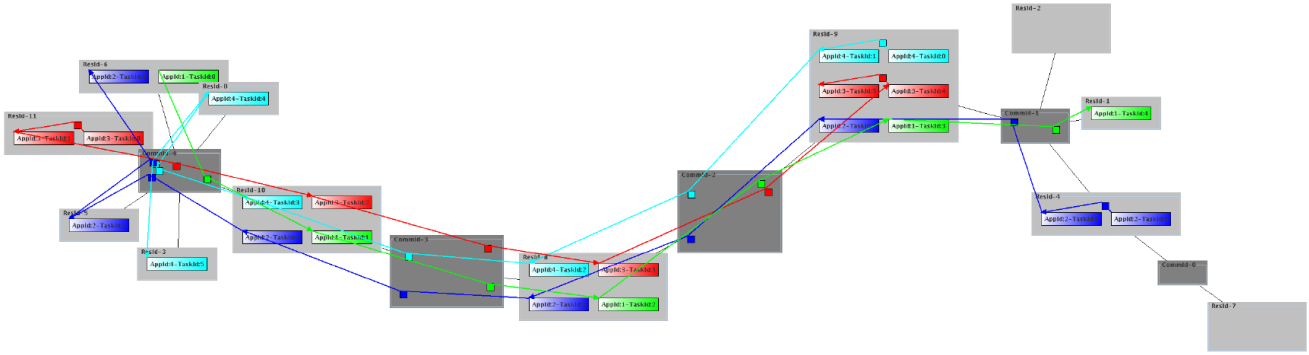


Fig. 8: Large Example System Model

[10] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *CODES/CASHE '98: Proc. of the 6th international workshop on Hardware/software codesign*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 97–101.

[11] V. Kianzad, S. Bhattacharyya, and G. Qu, "Casper: an integrated energy-driven approach for task graph scheduling on distributed embedded systems," *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE Int'l. Conf. on*, pp. 191–197, July 2005.

[12] M. Schmitz, B. Al-Hashimi, and P. Eles, "Energy-efficient mapping and scheduling for dvs enabled distributed embedded systems," in *DATE '02: Proc. of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2002, p. 514.

[13] L. Shang and N. K. Jha, "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable fpgas," in *VLSI Design*, 2002, pp. 345–.

[14] E. Bini, G. Buttazzo, and G. Buttazzo, "A hyperbolic bound for the rate monotonic algorithm," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 59–. [Online]. Available: http://portal.acm.org/citation.cfm?id=871910.871919

[15] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[16] E. Bini, T. H. C. Nguyen, P. Richard, and S. Baruah, "A response-time bound in fixed-priority scheduling with arbitrary deadlines," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 279–286, Feb. 2009.

[17] V. Pollex, S. Kollmann, K. Albers, and F. Slomka, "Improved worst-case response-time calculations by upper-bound conditions," in *DATE*, 2009, pp. 105–110.

[18] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with edf scheduling," *Computers, IEEE Transactions on*, vol. 58, no. 9, pp. 1250 –1258, sept. 2009.

[19] K. Vallerio, *Task Graphs for Free (TGFF v3.0)*, April 2008.

[20] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques, IEE Proc. -*, vol. 152, no. 2, pp. 148–166, Mar 2005.