Adding Schedulability Analysis to the Octopus Toolset

Ajith Kumar

August 23, 2011

Master Thesis

Eindhoven University of Technology
Department of Mathematics and Computer Science
Design and Analysis of Systems Group
And
Manipal Institute of Technology
Department of Information and Communication Technology

In cooperation with:
Embedded Systems Institute
Eindhoven, The Netherlands

Supervisor at TU/e:
Dr. Mohammad Reza Mousavi
Assistant Professor
M.R.Mousavi@tue.nl

Supervisor at MIT:
Dr. Balachandra
Associate Professor
bala.chandra@manipal.edu

Graduation Tutor:
Dr. Nikola Trčka
Postdoctoral Researcher
n.trcka@tue.nl

Abstract

Embedded systems are computing systems responsible for performing one or more specific functions within a host device. In the early design phases of embedded systems, multiple design choices need to be considered. Manually analyzing the feasibility of the designs is time consuming, lots of design parameters have to be considered, where exploration of the design space using each parameter is complicated hence impractical. Thus an integrated toolset that could assist in exploring the design space and choosing a near optimal design is of great benefit to the designers. This exploration usually involves analyzing the design choices using various tools with specific analysis methods that focus on one of the chosen design parameters. The design parameters are always application dependent. The Octopus toolset is one such design space exploration tool developed by Embedded Systems Institute in collaboration with various Dutch companies. The toolset assists the users in making a near optimal design choice for a system. It uses a graphical modeling language called Design Space Exploration Intermediate Representation (DSEIR).

Embedded systems typically have (strict) real-time constraints and usually work in a resource-constrained environment. For hard-real time systems with hard-deadlines, meeting deadline requirements is one of the critical factors for making design decisions. This property can be checked by performing schedulability analysis on the designs, which ensures that the resource requirement of the system is satisfied throughout its execution. Schedulability analysis is an important analysis method for analyzing design choices, and hence is a useful addition to the Octopus toolset. The goal of this thesis is to incorporate schedulability analysis as an analysis technique in the Octopus toolset.

We first investigate the possible approaches to add schedulability analysis to the Octopus toolset. During this investigation we observed broader requirements for the accomplishment of the primary goal, thereby dividing it into two phases namely, making schedulability analysis *possible* in the Octopus toolset and making this analysis procedure *scalable* for arbitrary DSEIR models.

The first phase includes devising an approach to perform schedulability analysis based on model checking. Model checking was chosen owing to its merits over the other meth-

ods. The Octopus toolset consists of a branch performing model checking on the DSEIR models using the model checker Uppaal, upon which we build the support for schedulability analysis. Completion of first phase included adding of the deadline feature into DSEIR and creating a deadline detection mechanism in Uppaal and embedding this into the DSEIR to Uppaal translation module in the Octopus toolset.

After the completion of the first phase, we analyze the scalability of our approach to arbitrary DSEIR models, which are tuned to simulation and are sufficiently large models. Given the practical limitations for model checking in Uppaal and general problems like state-space explosion in large models, we required a method to approximate the input models into an abstract form with focus on state-space reduction. We construct an algorithm to perform abstraction based on syntax analysis, applying which we get a sound approximation of the original DSEIR model.

Acknowledgements

This thesis is a part of my master project that I have performed in the group of Design and Analysis of Systems at Eindhoven University of Technology, the Netherlands, in co-operation with Embedded Systems Institute, TU/e Campus, Eindhoven, the Netherlands.

Coming to the end of my thesis work, I would like to take this opportunity to thank each and everyone who have directly or indirectly contributed to this project. Firstly I would like to thank Prof.dr.Twan Basten for providing me this opportunity to pursue my masters project in the Embedded Systems Institute (ESI), under the direct supervision of Dr. Nikola Trčka and Dr. Mohammad Reza Mousavi and also guiding me through this project. Dr. Mohammad Reza Mousavi, my graduation supervisor at TU/e, has provided me with constant support and guidance, ensuring the smooth run of my thesis. Dr. Nikola Trčka, my daily tutor at ESI has walked me through the ups and downs of my thesis, correcting me whenever I deviated from the desired path. I also would like to express my heart-filled gratitude to Dr. Nikola Trčka and Dr. Mohammad Reza Mousavi for their patient review on several versions of my report, helping me improve my documentation skills.

I am thankful to Dr. Balachandra for being my project coordinator from Manipal University and for his evaluation. I am grateful to Prof.dr. Mark van den Brand, the program director for the Manipal Program for his valuable guidance at every step on the way through this master‘s study at TU/e. I would like to thank all the people involved with the Octopus project, Frans Reckers, Dr. Martijn Hendricks, Bram in‘t Groen and all others who made my thesis possible. I have to mention my friend/colleague Ashwini Moily, without whom working on this project would not have been as fun. We have worked together for various parts of our project, acted as reviewers for each other and so on; I owe my gratitude to you. Special thanks to Reinier van Eck, director of ESI for his encouraging support through the thesis period. This project would not have been possible without the support of Prof.dr. Manohara M. Pai, Prof.dr. Jos Baeten and Prof.dr. Mark van den Brand, who gave me the opportunity to come to the Netherlands. I must express my appreciation to my friends Alok Lele, Shreya Adyanthaya, Abhinav Singh, Guru Prasanna, Shravan Shetty, Supriya Dsouza and Vinod Menezes for their patient listening to my presentations and support. Last but not the least; I owe my gratitude to all my other friends and my family who stood by me through all the highs and lows of this course and the project.

Contents

Chapter 1

Introduction

1.1 Preface

Embedded Systems Institute (ESI) is a leading research-based institute that tries to bridge the gap between academia and industry. It focuses on embedded systems technology through open innovation research. This master project is a part of an ESI consolidation project, building on the result of the Octopus project [1].

1.2 Context

During the early stages of the design of an embedded system, choosing among the available design alternatives becomes a huge challenge. The Octopus toolset developed in the Octopus project uses model-driven design space exploration (DSE) approach to overcome this challenge. The model-driven DSE approach aims at providing support for modeling, analyzing and selecting appropriate design alternatives during the early phases of system development. This helps in making a good design decision after considering the various metrics of interest such as timing, energy usage and cost, as well as the multiple design parameters such as the number and type of processing cores, sizes and organization of memories, interconnects, scheduling and arbitration policies. Figure 1.1 shows a design space exploration process in which a near optimal design is selected from an available design space with multiple design choices. Users initially have multiple designs for a single project and need to decide the best one among them. Manually, this is done by analyzing these designs using various analysis tools and depending on the results and requirements deciding the optimum design. Performing these analyses manually poses challenges in maintaining consistency in the designs, also this process is time consuming. A DSE tool could assist the users in making design decisions faster, the Octopus toolset

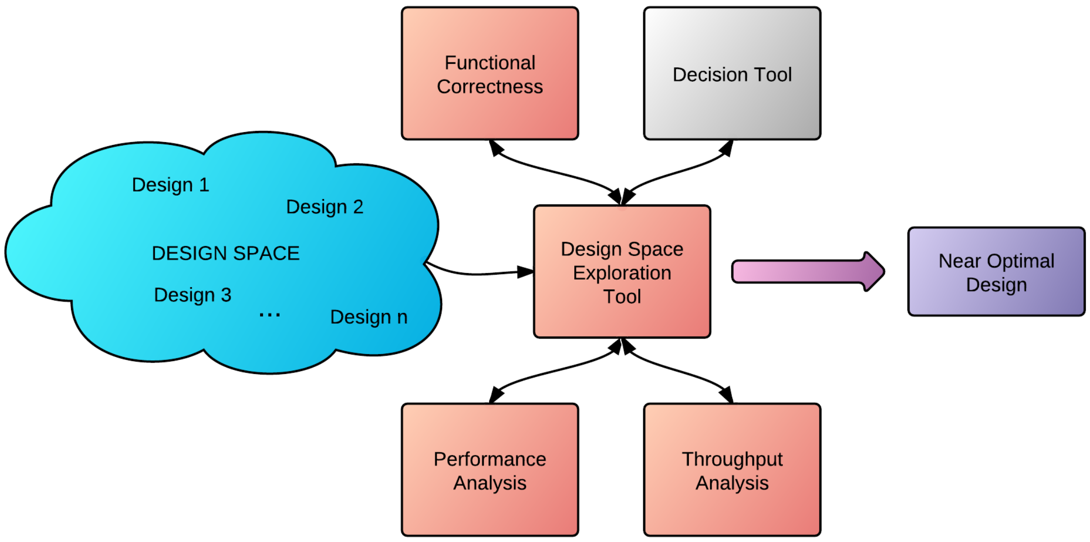


Figure 1.1: The design space exploration process

is one such DSE tool. The toolset currently provides support for:

- High-level modeling of embedded systems, with a clear separation of concerns in application, platform, and the application-to-platform mapping.
- Formal analysis of functional correctness and performance, and
- Exploration of alternatives and synthesis of optimized designs.

Figure 1.2 shows the high level plug-in based architecture of the Octopus toolset. It consists of 5 modules which form the core of Octopus. These modules are:

- **Modeling module:** Interface to input user models, the Octopus toolset provides a graphical editor to design the user models.
- **Kernel module:** The modeling language called Design Space Exploration Intermediate Representation (DSEIR) used to represent the user models.
- **Analysis module:** The input models in the DSEIR language are subject to various analysis altogether via the tools in analysis module.
- **Diagnostic module:** Diagnostic tools for detecting the cause of failure (if any).
- **Search module:** The tools for searching through the entire design space for near

optimal design options by checking the models against various properties via the analysis tools.

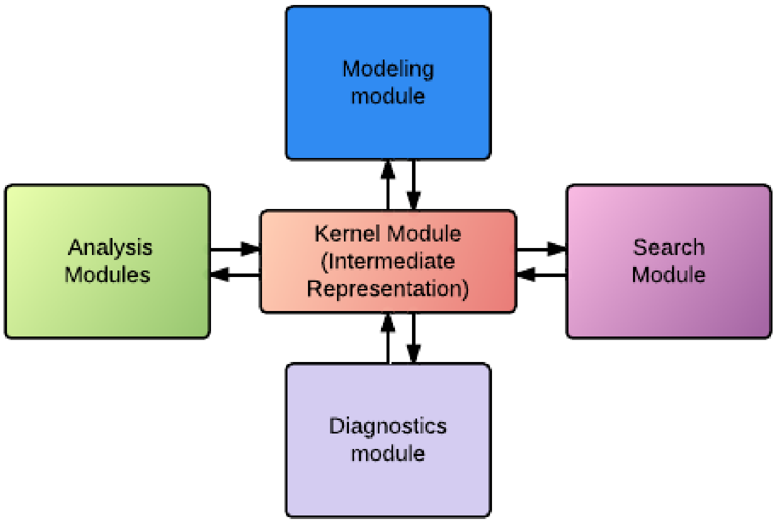


Figure 1.2: A high level plug-in based architecture of the Octopus toolset [1]

The various analysis tools can be seen in the main architecture in Figure 1.3. The idea of Octopus is not to provide new methods but rather to provide connections between different tools, all based on the assumption that no single tool/method can answer all design questions. This architecture depicted allows for:

- Easy reuse of models among different tools, while providing model consistency
- Systematic and combined use of different tools and
- Domain-specific abstractions to support different application domains and easy reuse of tools across domains.

1.3 Motivation and Related Work

Schedulability Analysis is defined in [2] as follows: *The problem of real-time schedulability analysis involves establishing that a set of concurrent processes will always meet its deadlines when executed under a particular scheduling discipline on a given system.* Embedded systems often involve monitoring and controlling of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. These resources include memory, processing power as well as the time allotted for execution. Such systems could be part of larger systems which can be sometimes life-critical systems, for example in an airplane. For a real-time system with hard dead-

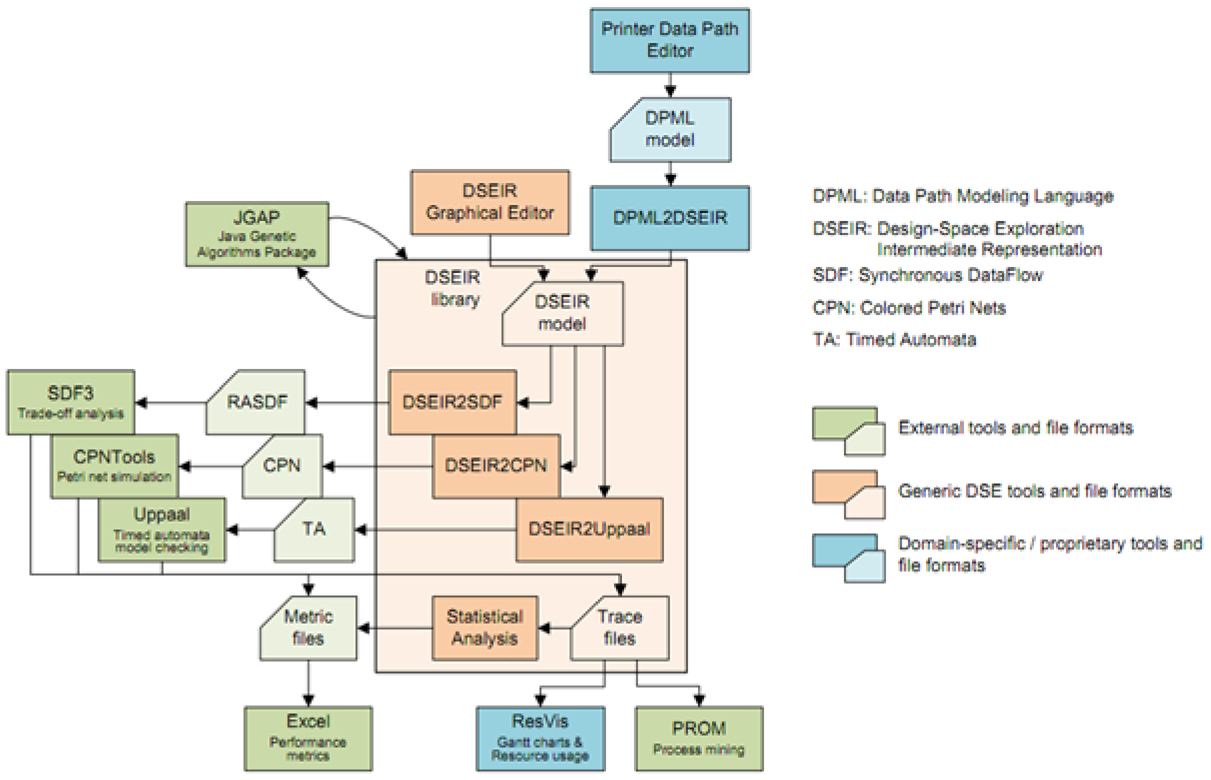

Printer Data Path Editor
DPML model
DPML2DSEIR
DSEIR Graphical Editor
JGAP Java Genetic Algorithms Package
DSEIR library
DSEIR model
SDF3 Trade-off analysis
RASDF
DSEIR2SDF
CPNTools Petri net simulation
CPN
DSEIR2CPN
Uppaal Timed automata model checking
TA
DSEIR2Uppaal
Metric files
Statistical Analysis
Trace files
Excel Performance metrics
ResVis Gantt charts & Resource usage
PROM Process mining
DPML: Data Path Modeling Language
DSEIR: Design-Space Exploration Intermediate Representation
SDF: Synchronous DataFlow
CPN: Colored Petri Nets
TA: Timed Automata
External tools and file formats
Generic DSE tools and file formats
Domain-specific / proprietary tools and file formats

Figure 1.3: Low level architecture of the Octopus toolset [1]

lines, it is critical to ensure that the deadlines of all the tasks are met. The challenge of schedulability analysis is to guarantee that the scheduling principle to be implemented avoids any deadline violations. Thus we need a method to ensure its smooth working.

In the Octopus toolset context, among the various questions affecting the decision making in DSE is the schedulability of the design. To obtain a proper design that ensures smooth running of the system by satisfying all the requirements, it would also require the tool to ensure schedulability. The problem can be solved by imposing task deadlines and automatically checking if they are always met. This would help the tool to immediately eliminate infeasible design options. Traditionally, there are two methods to determine the schedulability of multiprocessor systems:

- Utilization bound tests: Utilization bound tests are theoretical tests based on certain formulas, which depend on the number of processors and the utilization of the tasks. This method is safe but pessimistic.

- Simulation: Simulation of a system execution is the artificial representation of the temporal behavior exhibited by the system during its runtime. This method is unsafe and does not perform exhaustive exploration of the state space.

In view of the drawbacks of utilization bound tests and simulation, it would be valuable if we could have a method for exact schedulability analysis without the pessimism of the utilization bound tests. The method suggested by Guan et al. [3] was to analyze schedulability without any pessimism by transforming the schedulability problem into reachability analysis problem of timed automata. Uppaal [4], a tool for timed automata model checking is used for this purpose.

1.4 Problem Statement

The core problem of this project is to perform schedulability analysis of models designed in Octopus via model-checking using Uppaal, thereby avoiding pessimistic and non-exhaustive approaches.

Though the overall problem statement only mentions about schedulability analysis and model-checking, the actual task is much broader. In the Octopus toolset users can define their models using the DSEIR language. This model is entirely in the DSEIR syntax and then needs to be translated into corresponding Uppaal syntax. Uppaal, or in general any model checking tool, is prone to the *state-space* explosion problem. In the Octopus toolset, models are tuned for simulation, making them less applicable for exhaustive analysis methods. Thus these models can have large number of instances causing state-space explosion. So in a broader sense the problem statement can be divided into two phases:

- To make schedulability analysis *possible* in Octopus.
- To make schedulability analysis *scalable* for arbitrary DSEIR models.

The first point refers to providing an option in the Octopus toolset to perform schedulability analysis on the models designed. The second point refers to making Schedulability analysis *scalable* to be applied on arbitrary models represented in DSEIR, with few syntax restrictions applied on the model. In Section 1.5 we describe the approach used to solve these problems.

1.5 Approach

This section describes our approach towards solving the problems described in Section 1.4. The DSEIR language provides necessary syntax support for the users to represent their system in the form of models. A model usually represents a finite system. We build the solution over the infrastructure provided by the Octopus toolset, to perform translations to the Uppaal syntax. The problems addressed in this thesis are:

- Making schedulability analysis *possible* in Octopus :
  - Extending the DSEIR language to include the *deadline* feature.
  - Modifying the existing Uppaal translation from the DSEIR language by adding a *mechanism* to *detect violation of deadlines.*
  - Automation of the entire process.
- Making schedulability analysis *scalable* for arbitrary DSEIR models in Octopus by performing data abstraction.

The first part is achieved by extending the DSEIR language syntax with deadline which is essential for performing schedulability analysis. The Octopus toolset previously consisted of a module (DSEIR translator) that translates the user defined model into a Uppaal model, but lacks the deadline violation detection mechanism, which is required to perform schedulability analysis. Thus we model a deadline violation detection mechanism in Uppaal and test it. As a next step, the DSEIR translator module is extended to support the deadline violation detection mechanism. These additions make schedulability analysis possible in the Octopus toolset. Hence we have an additional analysis tool, achieving the first goal of the thesis goals. This part is treated in Chapter 5.

The second part focuses on the modeling part of the DSEIR language. Using Octopus toolset, problems can be represented generically in terms of three components namely, application, platform (resources) and mapping of these tasks to respective resources. The representation consists of finite/infinite systems with non-deterministic behavior because of the different possibilities for task execution. A task model in DSEIR has large number of task instances but tends to repeat same behavior in different task instances, considering these repetitions every time, tends to be an overhead to the model-checking tool and also increases the *state-space.* Hence the second part of our approach is to automatically extract the repetition patterns and similarity between different task instances thus identifying similar task instances and combining them. We consider methods like simulation and syntax analysis that could be used for performing the abstraction. The methods and the procedure for abstraction are explained in Chapter 6. The new model created from this automatic extraction is an abstract approximation of the original model, usually an over-approximation. The over-approximation is the price paid to obtain reduced *state-space.* We also require to make necessary changes in the verification procedure to omit the false-negative's (obtaining non-schedulable results for schedulable systems) that are obtained owing to the over-approximation. These necessary changes are described in Chapter 7. In Chapter 8 we then discuss a method to check the fitness of the parameter abstractor method.

1.6 Example

In this section we describe an example which demonstrates some aspects of the schedulability problem. We describe this example in the DSEIR language with brief description of the terms used; a detailed description is given in Chapter 2. We have 2 tasks, where A executes for 5ms and B executes for 4ms except for a value of (b % 1000) == 805 where it executes for 5ms. This execution time is obtained from the load shown in Figure 1.5. (It is actually the product of load and processing time of the resource which is required, but for simplicity we consider processing time of constant 1.) Both these tasks share a single CPU and A has a priority higher than B. The task model is shown in Figure 1.4. We have, A with parameter “a” which is also its port binding. It is initialized with the value 1. The guard over the edge from A to itself states that “a” is continuously incremented by 1 until it reaches a value of 1000 indicating we have 1000 instances to execute. We have also defined the deadline for A and B in Figure 1.5, with 6ms for both A and B. (Note that deadline was not initially a part of the DSEIR language, it was added on later as a part of the solution.) Also there is a delay on the edge from A to itself (>> 4) which is set to 4ms. The working will be, A starts and executes for 5ms then starts again after a delay of 4ms. During this delay B executes. But for one condition when B executes for 5ms, after 4ms of its execution it is pre-empted by A since there is only one CPU and A has higher priority. Thus B will miss its deadline since it will have to wait for 5ms to start again making the count 4ms(initially executed) + 5ms of A thus crossing the deadline of 6ms. Thus B fails to meet its deadline, given the solution for the first part of the problem this failure can be detected using the schedulability analysis technique.

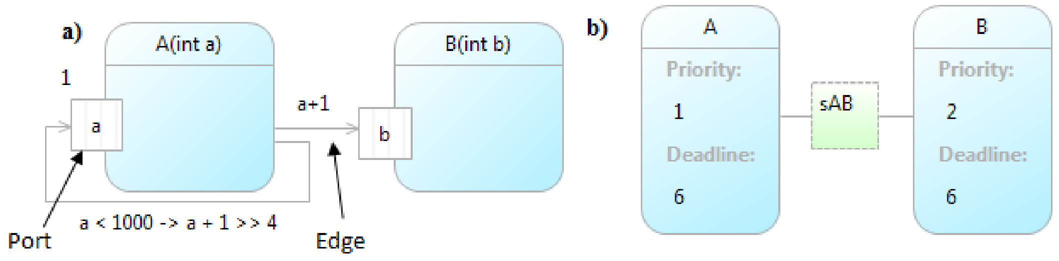


Figure 1.4: a)Example task-set in DSEIR syntax. b) The priority, deadline and scheduler of the tasks

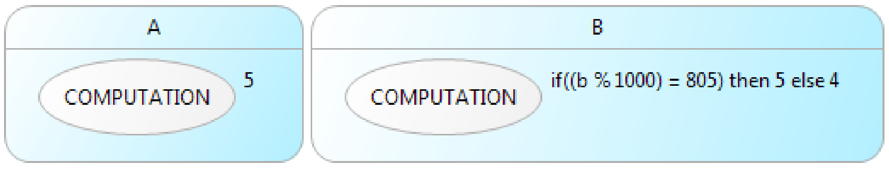


Figure 1.5: Load of the tasks in the example task-set

These task-models can also have the *state-space* explosion problems. Consider a case where we try to verify this system directly on Uppaal. We have to check for all 1000 instances, hence the *state-space* has 2000 different states at least (accounting for both A and B with simple addition, actual value could be much higher). If we consider each task instance has 3 states during execution, the state space would be still 6000 states. Accounting for worst case combinations, the *state-space* growth would be exponential,for instance the combination could be something like 1000 of A $\times$ 1000 of B. Thus the system could become intractable for a bigger system like this with large number of instances for each task. The second part of this thesis aims at solving this problem by performing data abstraction and considerably reducing the *state-space*.

1.7 Overview

This section describes the structure of the report. The structure and the syntax of the DSEIR language are explained in detail in Chapter 2. The model checking tool, Uppaal, is described in Chapter 3. The existing DSEIR translation is described in Chapter 4. Chapter 5 describes the solution provided to the first problem of designing the deadlock detection mechanism in Uppaal and extending the existing translation of DSEIR to Uppaal. We also treat the running example here to show how schedulability checks on the system can be performed in the Octopus toolset. Chapter 6 realizes the second goal of the thesis, namely, performing abstraction of the model in order to make model-checking scalable. Also an algorithm that is used to perform abstraction and its intermediate steps are explained in detail in the same section. The application of algorithm is then demonstrated by applying it on the running example. Chapter 7 explains the verification procedure for schedulability analysis, this includes some revision in the model checking scheme. Chapter 8 describes the methods to measure the fitness of the abstraction procedure used for providing scalability to the schedulability analysis technique. Finally, we conclude the thesis in Chapter 9 and present some recommendations for future work in Chapter 10.

Chapter 2

Design Space Exploration Intermediate Representation

The Octopus toolset follows the Y-chart approach (Figure 2.1). The essence of the Y-chart approach is an orthogonal specification of application and platform and their combination in the mapping. It supports clear separation of application and platform. In the Y-Chart approach [5], initially an architecture is designed in the specific composition as shown in Figure 2.1, implemented, analyzed and then diagnosed. These analysis results are then diagnosed to find the cause of failures, if any. The diagnosis results are used by the designers to improve the initial design by making necessary modifications to prevent the observed failures. The Octopus toolset has a similar work-flow. We create an initial design and improve it with analysis and diagnostics feedback.

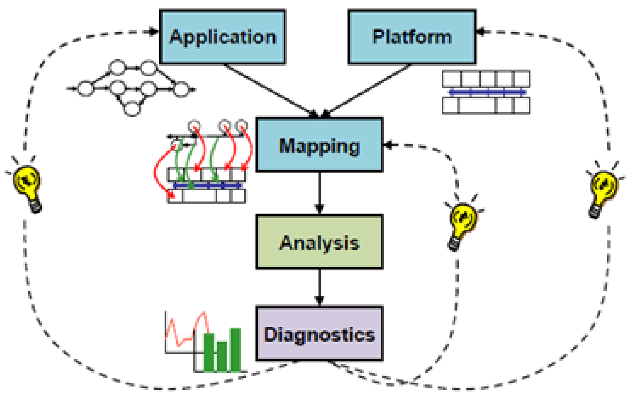


Figure 2.1: Octopus toolset using Y-Chart approach[5]

Using the Design Space Exploration Intermediate Representation(DSEIR) language any generic problem can be modeled as an input to the Octopus toolset. The DSE basically involves defining a model in terms of three components: *Application*, *Platform* and the *Mapping* of the *application* onto the *platform.* This problem-specific model (specific to the user) is then analyzed and possible flaws detected through the analysis are diagnosed. After analysis and diagnostics, if any problems are found, the cycle is repeated by redefining the initially defined model and proposing solutions to the problems found. The diagnostic information helps in proposing the solution. This entire process can be manual or automatic. In this chapter we introduce the DSEIR language and its syntax constructs required for modeling in detail, in Section 2.1. We then define a running example in the DSEIR constructs in Section 2.2 which will also be used in the rest of this thesis to explain the application of various approaches used in this thesis.

2.1 DSEIR Language

This section is largely derived from the Octopus documentation [6], we here describe the syntax of the DSEIR language in terms of its three basic parts, which are, *application*, *platform* and *mapping.* Each part is explained in detail with both syntax and example diagrams.

2.1.1 Application

In DSEIR an application is modeled in terms of atomic tasks as the basic building blocks. Moreover, an application consists of global variables, local variables, a task-flow graph, start-statements, end-statements, edges, ports, task load, and task handovers between various tasks. We explain each of these concepts with its syntax. Figure 2.2 shows the generic syntax of DSEIR models. An instance of this syntax is given in Figure 2.3. Below, we explain the concept of local and global variables and task-flow graph followed by its tasks and other entities of the application.

Global Variables are variables that are used to store data and are visible to every entity present in the task-flow perspective. The declaration syntax of global variables is shown in Figure 2.2, they are declared along with the name of the application (SyntaxofApp()). An example of a global variable can be seen in Figure (2.3) where “c” is a global variable initialized with an integer value 1.

A *local variable* is a variable that is declared within any task and is local to that particular task. The syntax for declaration of the local variables is “Type Variable” as shown in Figure 2.2. The example model in 2.3 shows “Int x” as a local declaration for task A.

The *task-flow graph* specifies the flow of *tokens* between individual tasks. The token

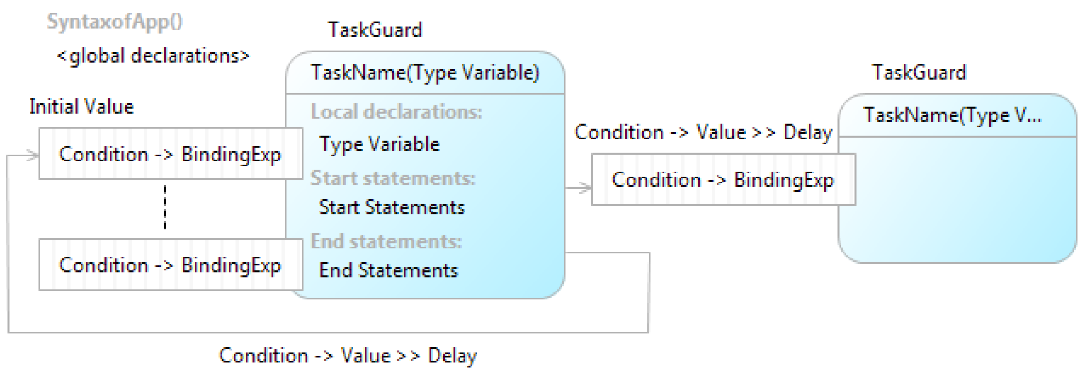


Figure 2.2: Task-flow perspective syntax diagram

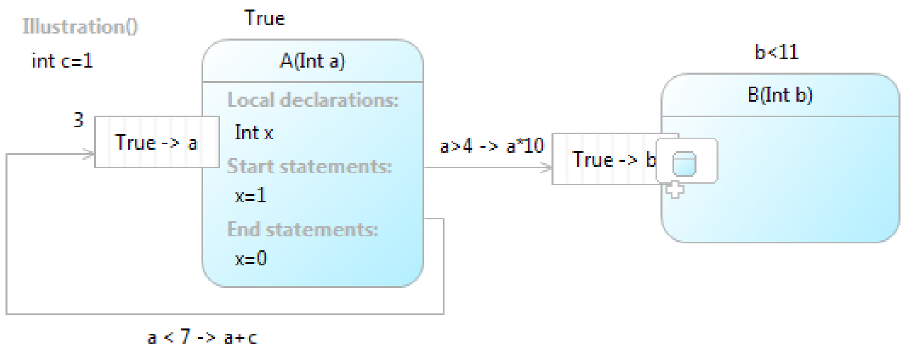


Figure 2.3: Task-flow perspective example

flow represents the flow of control and data between individual tasks, thus representing data dependency and execution sequence as in general scheduling terms. On a global perspective the tasks and their connections with other tasks via edges can be viewed as a task graph, the edges represent the dependency (data or control). For example, "a * 10" on the edge from task A to B is the value passed onto B by A.

The *task* in DSEIR is comparable to a task in real-time systems. It has several attributes and it executes (fires) to complete some actions for the system. In DSEIR task is identical to a function in a programming language, having zero or more parameters of given types. When each parameter of a task is given a value, a task instance is obtained. A task instance executes for a required amount of time depending on its load (explained later), during its execution it consumes some tokens and produces some. A task is said to be in the "ready" state in scheduling terms when it has enough tokens in its ports. Each task has a task guard associated with it, which is an expression that evaluates to a boolean value and needs to be true for the task to execute. The guard is by default true; if not specified. Task declaration syntax can be seen in the model of Figure 2.2. For example, in Figure 2.3 A and B are tasks, the task guard of A is true and for B it is $b < 11$. The task parameters of A and B are respectively "a" and "b", which are both of type

Integer.

The tasks can also contain a set of start-statements and end-statements. Like the name itself suggests they are executed in the beginning and end of task execution and usually affect the local variables. In the example, shown in Figure 2.3, $x = 1$ is the start statement assignment and $x = 0$ is the end-statement assignment.

Edges are used to connect a task and the port of another task. They represent transfer of data and control from one task to another via tokens. As explained before, tokens are passed through edges and this passing is guarded by a condition on the edge. Condition is an expression which evaluates to a Boolean value. For example, in Figure 2.3 the edge from A to B has the condition as $a > 4$ and the value to pass is $a * 10$. The condition is considered to be true if absent. The edges are also associated with a delay, the time taken by the tokens to pass through the current edge. The value of delay is zero by default and is represented as ">> Delay" as shown in the syntax describing model (Figure 2.2).

A *port* acts as a destination to an edge coming from another task and is contained inside a task. Port binds (assigns) the value from the input edge to its binding expression. In cases where the initial value of the port is set, the initial binding happens using this initial value. The binding expression is a mathematical expression that consists of task parameters, but usually it contains only one task parameter to which the value from the edge or initial value is directly assigned. Ports also have conditions associated with them which are similar to conditions on edges. If the condition on the port evaluates to false then binding does not happen. For example, in Figure 2.3 the condition for the port in Task A is true and the binding expression consists of only a parameter "a".

A task that has all the required tokens in its ports is said to be "enabled" and will begin executing when it is allocated with resources (platform) by the schedulers. A port can contain more than one token at a time and requires ordering schemes. Port can be *unordered*; in such a case the task can non-deterministically choose any token and use it to bind. It can alternatively be *FIFO*, in which case the token that has first arrived must be used first. The ports can be *unbounded* i.e., can have arbitrary number of tokens. After the firing of the task the tokens are said to be consumed and hence removed from the port (The binding expressions are assigned to a default value, e.g. 0 indicating that the tokens are removed). The tasks can also produce tokens while firing, which are then sent through the output edge from the task.

Along with a task, certain properties of the task can also be defined such as the *Task load* which defines the resource requirements of a task. Task load also partially decides the execution time of the task. The load does not change during the execution of a task instance and also that unit of load is considered the same for each service type. In other frameworks tasks may directly request for resources, but in DSEIR, resources are handled by the schedulers and they provide services to the tasks. The tasks here request services rather than resources. Services can be of type COMPUTATION, STORAGE, TRANSFER or any other user-defined type.

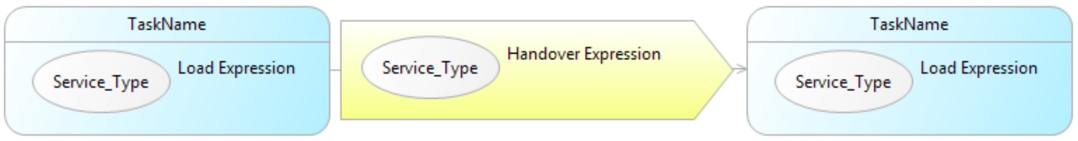


Figure 2.4: Load perspective diagram syntax

Figure 2.4 shows the declaration of load and handover between two tasks in the DSEIR syntax. Load expression is a mathematical expression which results to a constant or a distribution over a specified range when evaluated, it is usually dependent on the parameters of the task or sometimes simply a constant or range. Figure 2.5 shows an example of the load perspective. We can see the mathematical expression for load over service type INTERNAL_STORAGE in task A, which is dependent on a parameter "a". The remaining expressions are just constants.

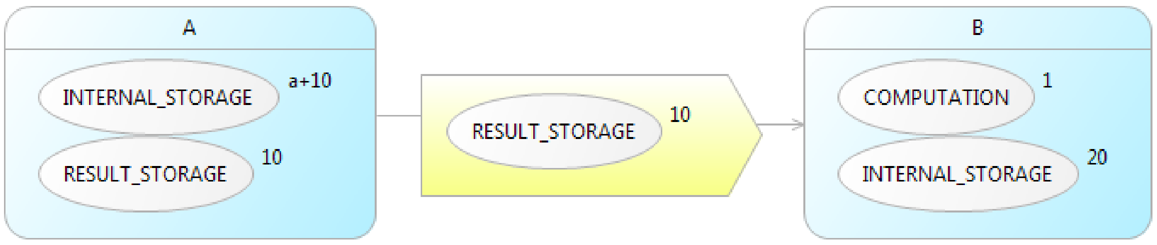


Figure 2.5: Load perspective diagram example

The *handover* expression is also similar to the load expression. To model resource reservations and situations where one task needs to transfer rights over some resource to another, DSEIR introduces the notion of task handover. A handover is an integer expression associated to each edge (considered zero if absent). This expression is evaluated at the end of the execution of the current task instance; it can depend on the task parameters and other local or global variables. The amount of resource that has been handed over is then released by the resource receiving it or is transferred over to another task until it is released.

2.1.2 Platform

The execution of a task requires resources. In DSEIR resources and tasks are not directly mapped onto each other. Instead resources provide services and tasks requests for these services. These requests for services are allocated by the scheduler on the basis of the scheduling policies and the availability of the resource. The types of resources considered here are usually CPU, GPU, FPGA and MEMORY. Along with the resources their attributes, given below, are also defined:

- Service Type: Defines the type of services provided by the resources in the model. The possible service types are COMPUTATION, STORAGE, TRANSFER and any other user-defined types.
- Resource Name: Defines the name of a resource used in the model, e.g., CPU or MEMORY.
- Capacity: Defines the capacities of the resources managed by the scheduler, e.g., the capacity of CPU is defined to be 1.
- Processing Time: Defines the processing time required by the respective resource to process unit load. E.g.Processing time of CPU is 4 time units per 1 unit of load. So if the load on the service type provided by CPU is 10 then the total time taken by CPU to process this load amount is 40 time units.

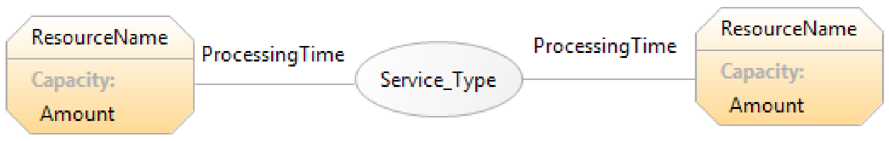


Figure 2.6: Resource perspective syntax diagram

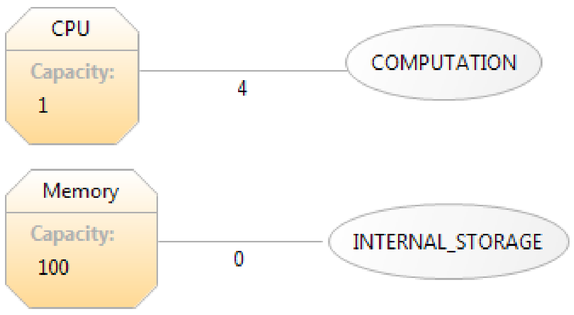


Figure 2.7: Resource perspective diagram example

Figure2.6 shows the declaration of two resources which provide the same service type. Figure 2.7 shows a declaration of a resource named as Memory with capacity 100 units and processing time as 0 time units. This resource provides a service of service type INTERNAL_STORAGE.

Tasks put some load on their required service types which are granted by the scheduler and mapped to the resources that provide them (service type). The mapping of a service type to the resource that provides that particular service type, by the scheduler is shown in Figure 2.8. The amount of the resource to be provided is decided dynamically during run time and is usually problem specific. Some of the resources like processing units (CPUs) are also associated with attributes like pre-emptivity. This attribute of pre-emption is defined in the scheduling perspective in Figure 2.8. In Figure 2.8, the white circle connecting the resource to a service type means that the resource is pre-emptive and the black circle connecting the second service type represents non pre-emptiveness. The amount on the edge from the white circle to the resource describes the amount of resource to be allocated to that specific task requesting this service type. This amount can either be a constant value or the load value defined by the requirement of the task. For the concept of pre-emption, consider a scenario where two tasks X and Y are contending for the same service type provided by one single resource R. Task Y has a priority higher than X. Task X is currently executing and has finished 50% of its execution, when task Y is enabled (has enough tokens in its ports) and is ready to execute. At this moment task X is pre-empted and Task Y is allowed to execute. Task X can continue only after task Y has finished execution.

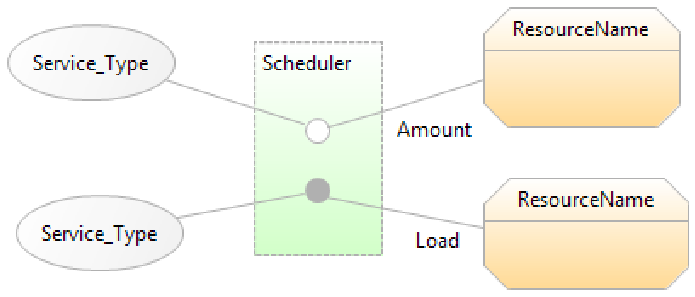


Figure 2.8: Scheduling perspective syntax diagram

Figure 2.9 shows an example of a scheduler mapping resources to the respective service type they provide. Here, amount allocated in case of CPU is either 1 or none and is not dependent on the value of any parameters. But in the case of M1 and M2 the amount allocated is set to be the amount of load requested by task A for the particular service type. The load expression can be dependent on the task parameters, hence this allocation also becomes dependent on parameters.

2.1.3 Mapping

A *scheduler* provides tasks with resources requested by them. In DSEIR, a scheduler is mapped onto a task or a set of tasks depending on the service type they require. Consider an example, tasks A and B have requirements for service types COMPUTATION and a resource called CPU that provides the COMPUTATION service type. Thus these 2 tasks can share a scheduler like in the example shown in Figure 2.10.

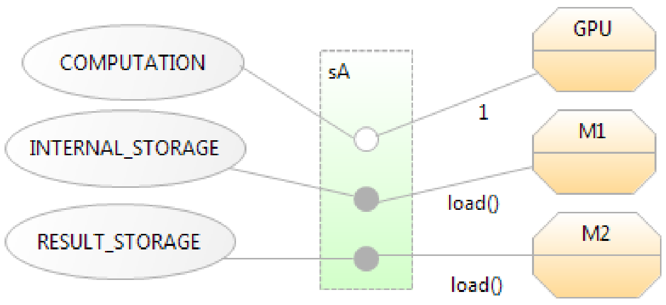


Figure 2.9: Scheduling perspective diagram example

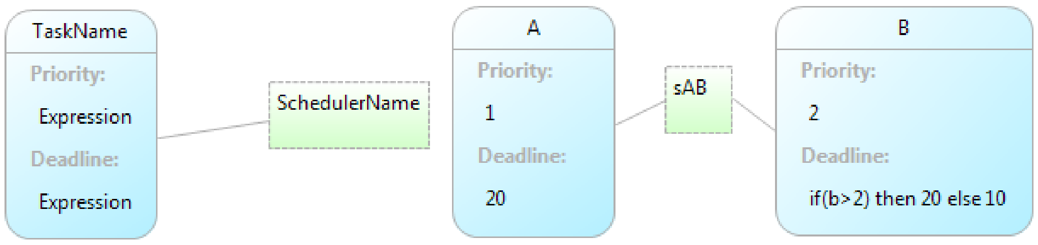


Figure 2.10: Mapping perspective syntax diagram

This mapping is defined along with certain attributes of the task which are set in the mapping perspective, they are:

Priority: Defines the priority of each task. It is a mathematical expression which results into an integer constant when evaluated. The expression can be used to specify dynamic priorities. For static priorities we can define a constant. The priority of the task will determine the task that will be granted a resource when there is an ambiguity like two resources are ready at the same time and requesting for the same service type.

Deadline: Defines the relative deadline of a task, in a set of tasks. It is a scheduling term which defines the latest time until which a task can execute. It is a mathematical expression which results into an integer constant when evaluated or a distribution. Note: This feature was not existing previously in the DSEIR syntax and has been added later on in the context of this thesis. The schematic view of mapping a task to a particular scheduler along with an example is shown in Figure 2.10. The concrete example shows the integer constants declared for priority and deadline of Task A, for the deadline of task B "if b> 2 then 20 else 10" is an expression which evaluates to an integer constant depending on the value of the parameter "b" local to Task B. If the "condition" in the "If statement" evaluates to true the deadline will be 20 or else 10. Both the tasks share the same scheduler (sAB).

2.2 Running Example in the DSEIR Syntax

The Octopus toolset provides some example models specified in the DSEIR language. We use one of this to describe the problems solved by this thesis. First we take one of these example models and use that to illustrate various concepts of DSEIR syntax.

This example consists of a task-set with 3 tasks namely, scan, image processing and print as well as resources and schedulers which maps these tasks into required service types. The basic job of this system is to process the pages to be printed and then printing them. The task-set execution is initiated by initializing scan which is to be executed first in the sequence. The data (page numbers) about the pages to be printed and their corresponding page sizes are transferred to scan task which performs a set of operations given this data and transfers the results to the next task (image processing) after finishing its execution. Image processing task also performs some operations and transfers the page data to the next task (print), which prints the page.

2.2.1 Application

The DSEIR model for this example is shown in Figure 2.11 and explained in detail in this section. The ports of tasks scan, image processing and print task are respectively named as “inA”, “inB” and “inC”. Scan task is initialized by providing it with a token; this action is executed by setting the initial values for the port of scan task. In the current example the initial value for the port binding is [1, dist(sizeDist)] where sizeDist is a probability distribution for the appearance of elements (array elements), with probability of each element in the distribution. It is defined as an array “int sizeDist[] = [1, 20, 2, 50, 3, 30]” which means that constant 1 appears with 20% probability, 2 with 50% probability and 3 with 30% probability, the declaration can be seen in the global declarations. The file name has been set to ODSE.

Once the ports have enough tokens, scan task waits for its execution until its resource requirements and priorities are satisfied. After scan task has finished execution it produces two tokens, of which one is sent over an edge to scan task itself and re-consumed for the execution of its next instance and the other token is sent over the edge to port “inB” and is consumed by scan task for its execution. The token to be sent from scan task to itself, reaches only if the guard condition over the edge from scan task to its port “inA” is satisfied.

After execution, image processing task produces a token which is passed on to print task for the execution of its current instance. Print task executes but does not produce any tokens. Note that each task here has its parameter name as *pageSize*, since it is a local variable the names can be the same. The edges, conditions, expressions, handover and the port bindings are defined as follows:

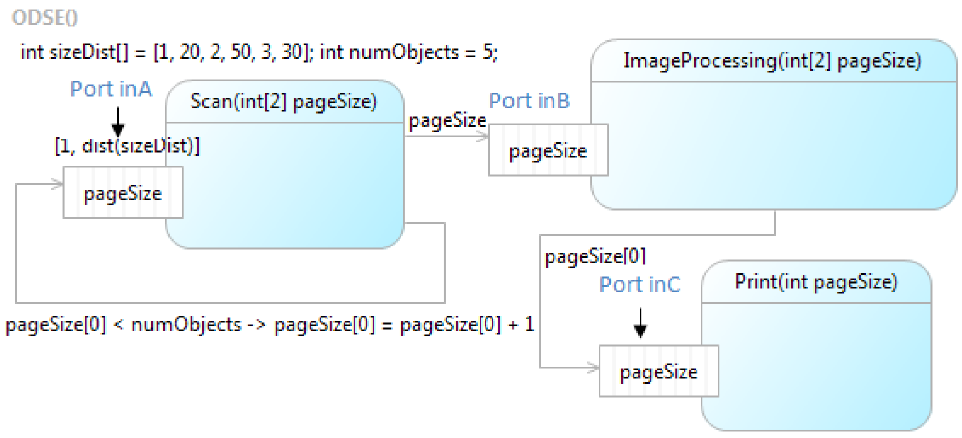

ODSE()
int sizeDist[] = [1, 20, 2, 50, 3, 30]; int numObjects = 5;
Port inA
Scan(int[2] pageSize)
[1, dist(sizeDist)]
pageSize
pageSize
Port inB
pageSize
ImageProcessing(int[2] pageSize)
pageSize[0]
Port inC
Print(int pageSize)
pageSize
pageSize[0] < numObjects -> pageSize[0] = pageSize[0] + 1

Figure 2.11: Task-flow perspective diagram for the printing example

- Scan task:
  - Edge to “inA”: Here numObjects is a global variable which is set to 5. This in combination with the guard on the edge from scan task to its port “inA” defines that scan task executes for 5 instances, since after that the guard over it evaluates to false. The value expression ($pageSize[1] = pageSize[1]+1$) over the edge has the values that are used to bind the variables of port “inA”. This expression changes the element at index 1 of *pageSize* array and passes the new array onto the port.
  - Edge to “inB”: The value expression (*pageSize*) over the edge has the values that are used to bind the variables in port “inB”. There is also some amount of handover that takes place from scan task to image processing task. In this case it is the RESULT_STORAGE and 20 units are handed over. The handover is shown in Figure 2.12. Scan task initially claims 20 units of service type RESULT_STORAGE, after its execution it does not release this resource but transfers the same amount (since handover expression and load expression of scan task are same) to image processing task which uses this storage to perform its operations, and does not have to explicitly claim it.
  - Port “inA”: Binds pageSize (binding Expression of “inA”) array with the values provided by the value expression (pageSize array with one of its elements altered) on the edge to “inA”. This binding happens always whenever tokens are produced by scan task and the guard over the edge evaluates to true, since the condition for port binding is explicitly specified as true. In the beginning “inA” is initialized by binding its variables with the initial values onto an integer array of size (*pageSize*) which stores this information.

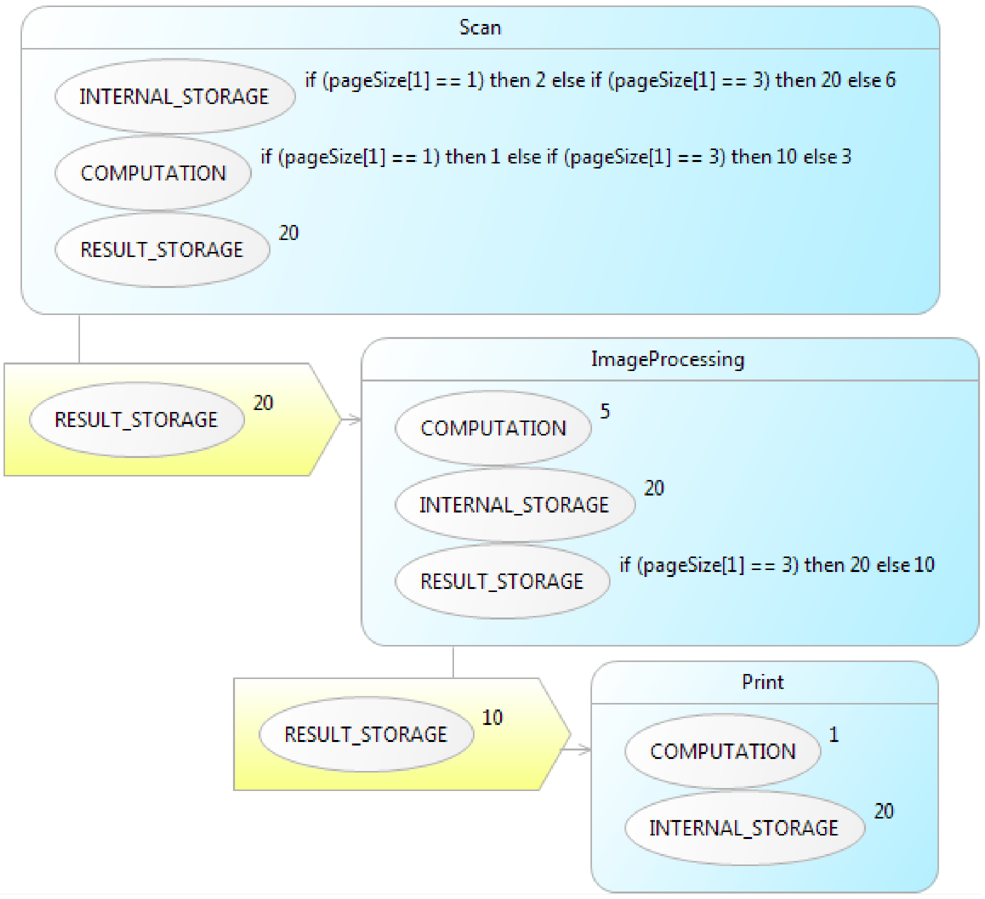

Scan
INTERNAL_STORAGE
if (pageSize[1] == 1) then 2 else if (pageSize[1] == 3) then 20 else 6
COMPUTATION
if (pageSize[1] == 1) then 1 else if (pageSize[1] == 3) then 10 else 3
RESULT_STORAGE
20
RESULT_STORAGE
20
ImageProcessing
COMPUTATION
5
INTERNAL_STORAGE
20
RESULT_STORAGE
if (pageSize[1] == 3) then 20 else 10
RESULT_STORAGE
10
Print
COMPUTATION
1
INTERNAL_STORAGE
20

Figure 2.12: Load perspective diagram for the printing example

- Image processing task:
  - Edge to “inC”: The value expression over the edge to “inB” (pageSize) produces the value that is used as binding to “inC”.
  - Port “inB”: Binds *pageSize* array with the values provided by the value expression on the edge to “inB”.
- Print task:
  - Port “inC”: Binds pageC an integer variable with the value provided by the expression (*pageSize[0]* array access of element at index 0) on the edge to “inC”.

Scan task requires the services of type COMPUTATION, INTERNAL_STORAGE and RESULT_STORAGE, the amount of load to be provided to the task is decided by the scheduler. The behavior of the scheduler is problem specific. If the available amount of a resource is less than the requested amount by the task then the scheduler grants the resource only if its possible for the task to execute with the allocated amount (This is known apriori and amount defined as a constant in this case is irrespective of the load request of the task, thus load could be more than or equal to the constant). If it is not possible, scheduler does not grant the resource and waits for the requested amount to be available. The load expression defined for each service type decides the amount of load at each instance. The detailed definition of load of each task is given in Figure 2.12.

2.2.2 Platform

The execution of the task requires services of different types, which are provided by the resources and are allocated to the tasks by the scheduler. Allocation is done by the scheduler on the basis of the scheduling policies and the availability of the resource. There are four resources considered here, namely:

- CPU
- GPU
- FPGA
- MEMORY (M1 and M2)

The resource definitions are shown in Figure 2.13.

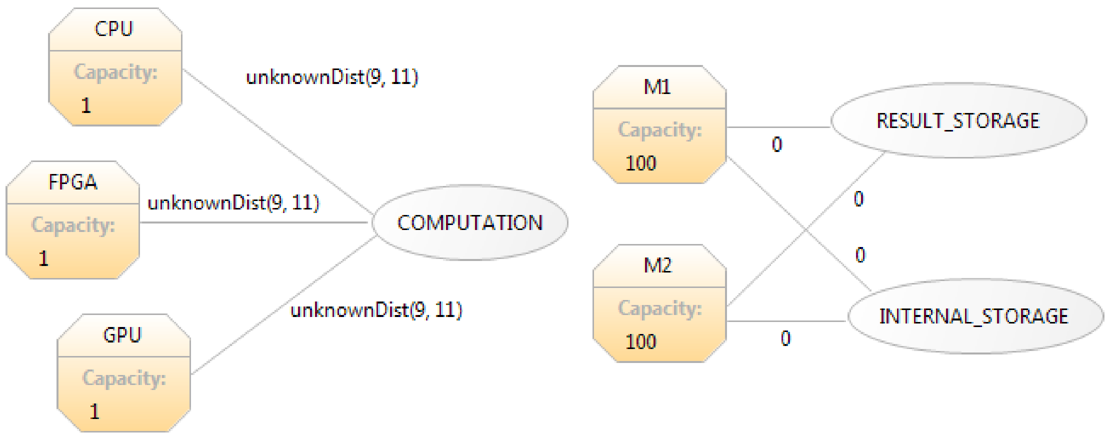


Figure 2.13: Resource perspective diagram for the printing example

The resources are defined along with their attributes such as resource name, processing time, capacity and service type. The processing time for the CPU, GPU and FPGA in the current example ranges over an unknown distribution from 9 to 11. An unknown distribution is a type of distribution where the values are non-deterministically chosen from the given interval, i.e. the probability of the elements are unknown. The capacities of these resources are pre defined with CPU and GPU defined as 1 unit, FPGA has 1 unit and each memory has 100 units. The processing time of the memory resources M1 and M2 is 0. The tasks are mapped onto their requested resources by schedulers. The allocation of service types will be done dynamically during the execution depending on the requirement of tasks and their priority.

In this particular example there are schedulers that manage a particular set of tasks. These schedulers then map the tasks onto the resources that they manage. Further the mapping of the service types required by the tasks to the service types provided by the actual resources is also done by the schedulers. This is shown in Figure 2.14.

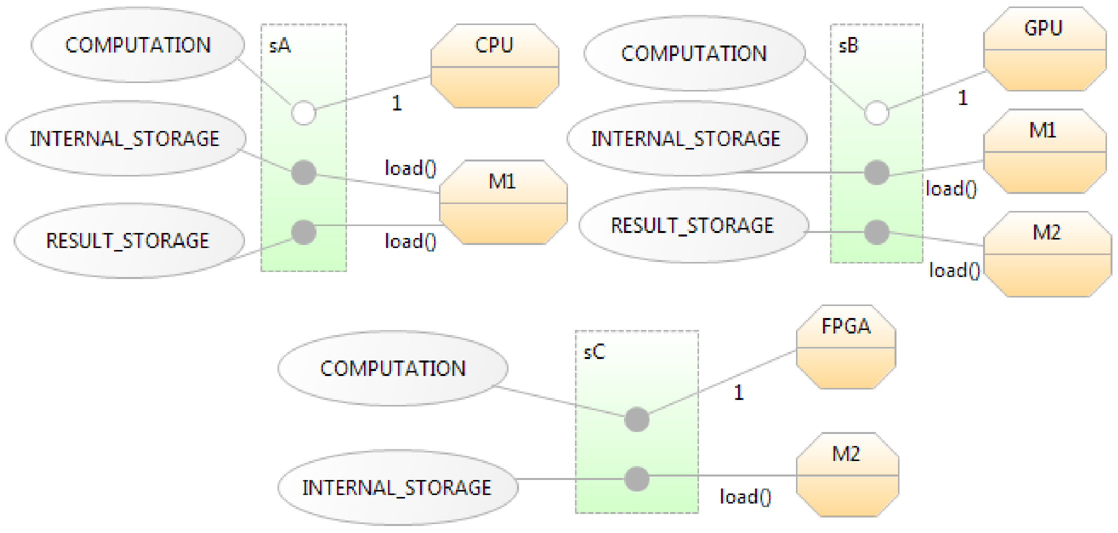


Figure 2.14: Scheduling perspective diagram for the printing example

2.2.3 Mapping

The mapping of each task to scheduler is shown in Figure 2.15. "sA", "sB" and "sC" are schedulers. The tasks are defined with their respective priority and deadline expression.

Scan task requires 3 types of services COMPUTATION, INTERNAL_STORAGE and RESULT_STORAGE. The scheduler "sA" for scan task maps the services to the resources CPU and Memory M1. The actual load is defined in the application part in Figure 2.12. Similarly image processing task and print task use scheduler "sB" and "sC" respectively, for the mapping to services required by them. In this example the scheduler grants the amount requested by the task if available, and otherwise no service is granted until the requested quantity becomes available.

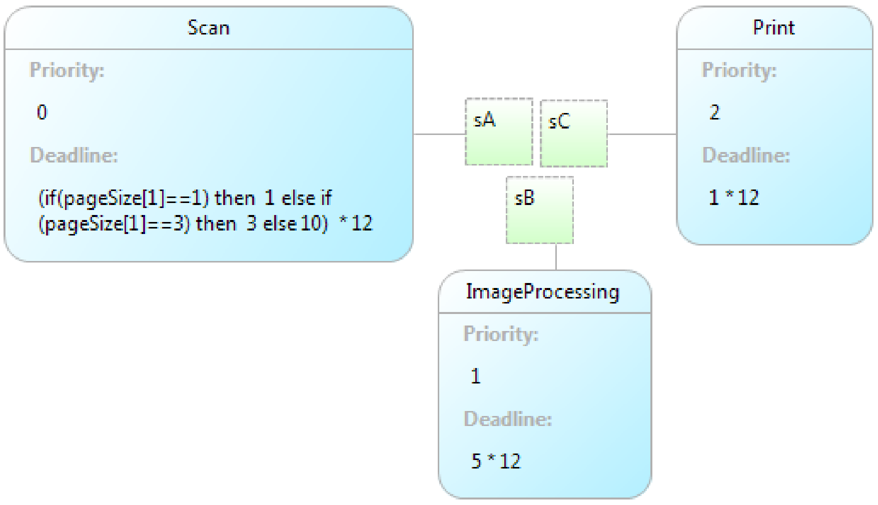


Figure 2.15: Mapping perspective diagram for the printing example

The mapping also describes the priorities. Here scan task has the highest priority, followed by image processing task and print task. Due to task dependency (Figure 2.11 the edges describe dependency) usually the execution starts with scan task followed, respectively by image processing task and print task. Along with the priorities, deadline is also defined in the mapping perspective.

Chapter 3

Uppaal

Uppaal is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University [7]. In Uppaal, the concept of timed automata is used to model processes in a system. A network of communicating timed automata forms the entire system. The definition of these processes also includes various attributes and functions with data types such as integer or user defined structured data type. Channel synchronization are used to synchronize two or more processes. The purpose of using Uppaal in this thesis is to perform schedulability analysis via model checking. One of the main problems we address in this thesis is *state-space* explosion, which is also a problem in any model checking tool and also is extensively addressed in the literature regarding Uppaal [7]. Uppaal provides a graphical user interface which can be used by the users for creating, editing and simulating the input models followed by verifying the input models. Verification involves specifying the property of the model in a logical query language and exhaustively searching the state space to make sure that the specified property is satisfied. For more information about Uppaal and its components refer [8].

In this section we briefly describe the modeling language of Uppaal, the specification language used to verify and also how the tool works. For a more detailed explanation of the Uppaal syntax we refer the Uppaal documentation [9]. In explaining the syntax, we mostly confine ourselves to the constructs that are used in the remainder of this report.

To illustrate the syntax we use a *coffee machine* example, depicted in Figure 3.1. The features missing in this example are explained in separate illustrations.

The coffee machine example models the behavior of a system with three components, a *coffee machine*, a *person* and an *observer*. The *person* repeatedly tries to insert a coin, tries to extract coffee after which he will make a publication. Between each two actions the person requires a suitable time-delay before being ready to participate in the next one. After receiving a coin the machine should take some time for brewing the

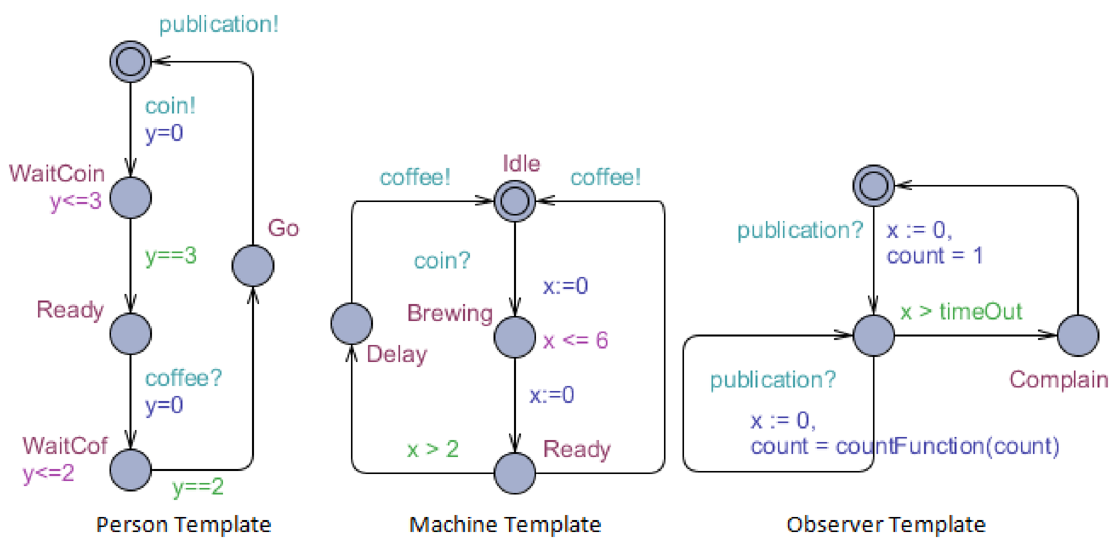


Figure 3.1: Coffee machine example [10]

coffee. The machine should time-out if the brewed coffee has not been taken before a certain upper time-limit. The *observer* should *complain*, if at any time more than 8 time-units elapse (expressed by the constant timeout in the model) between two consecutive publications.

3.1 Uppaal Timed Automata Modeling Language

Uppaal modeling language is based on timed automata, which are finite-state machines, extended with the concept of time [8]. Time is represented by clock variables. All clocks in the model progress synchronously. In Uppaal, a system is modeled as a network of several such timed automata in parallel. Each timed automaton is an instantiation of a template automaton. The Uppaal syntax has 3 sorts of declarations, given below:

- Global declarations: The declarations in the main section which are visible to all templates. The declarations can include clock variables, integers, user defined variables, constants, channel and user-defined functions. The global declarations for the *coffee machine* example are given below:

  ```
  // Only Synchronization channels are declared. - comments
  urgent chan coin, coffee, publication;
  int count; //Used to count the number of coffee's served.
  ```

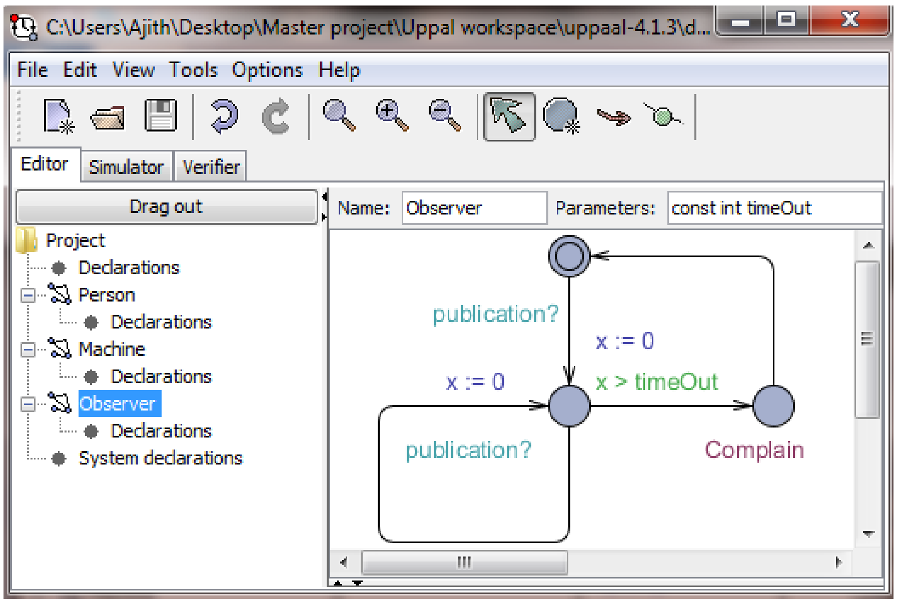
- Local/Template declarations: Local declarations are similar to the global declarations except for the fact that a variable defined in this section is local to that particular template. An example of a local definition for the *person* template in the *coffee machine* example is given below:

```
clock y; // A clock variable used in the template
```

- System declarations: System declarations are used to instantiate the templates into processes and execute them in parallel. A process instance of a template is defined by giving it a process name and instantiating it parameters. Templates without any parameter can be readily used as processes. System declaration starts with the keyword *system* and continues by naming all process to be composed in parallel. For the *coffee machine* example the system declarations are given below:

```
Obs = Observer(8);// Process instantiation
system Person, Machine, Observer; //Calling all the processes
```

In this example we can see that *observer* is a parameterized template with a parameter timeout, this is instantiated in the system declarations with the value 8. Figure 3.2 shows the parameterized template of *observer*.



Name: Observer Parameters: const int timeOut

Figure 3.2: Parameterized template *Observer* in coffee machine example

Other elements of the Uppaal modeling syntax are:

- Synchronization: There are three types of synchronization in Uppaal, described below:

- Binary Synchronization: Channels are used for message passing between edges of different templates. They can be declared using the syntax "chan *variable_name*" either in the local declarations or in the global declarations. An edge labeled with *variable_name*! synchronizes with another edge labeled *variable_name*?, where *variable_name*! acts as the sender and *variable_name*? acts as a receiver. In the *coffee machine* example the binary channels are coin, publication and coffee. This type of synchronization is also known as hand-shaking.

- Broadcast channels: Channels can also be declared as broadcast channels, using the syntax "broadcast chan *variable_name*". The difference between a binary channel and a broadcast channel is that the broadcast channels synchronize a send with as many "receive" as "enabled". For example, consider the Figure 3.3 which represents a scenario with 3 templates A, B and C, with one broadcast channel test. A sends two messages and both B and C receive them. So in working all these automata work synchronously, i.e, when A moves from A1 to A2 both B and C mimic the same due to the synchronization channel.

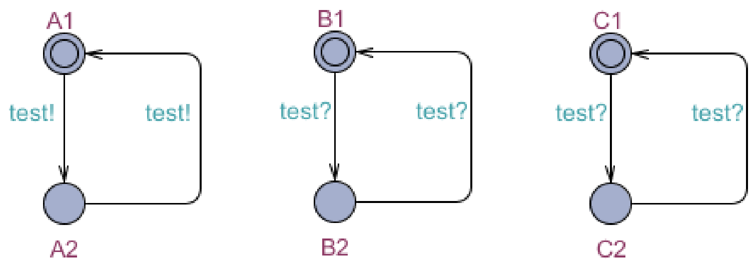


Figure 3.3: Example for broadcast channel

- Urgent channels: These channels enforce a synchronization when the edge is "enabled", i.e., do not allow the time progress when both sending and receiving actions can be taken. Edges having urgent channels cannot have guards with clock variables. Syntax for urgent (respectively, urgent broadcast) channels is "urgent chan name *variable_name*" (respectively, "urgent broadcast chan *variable_name*"). For example, all three channels in the *coffee machine* example, i.e., coin, coffee and publication of the *coffee machine* example are urgent channels.

• User functions: Users can define functions similar to imperative languages such as C++ or C functions except that they cannot use pointers. These functions can have parameters as input and can produce an output. Functions can be global as well as local, depending on their use. For example, in the Observer template the function *countFunction* is declared in its local declarations as:

```
int countFunction(int count){
count = count+1;
return count;
}
//This function is used to count the number of times coffee is served.
```

- Data types in Uppaal:
  - Clock Variables: Clock variables are used in Uppaal to keep track of time. Clock variables are of type real. They can be declared as "clock *variable_name*" as used in the *coffee machine* example. There is a possibility to reset the clock and also pause the clock. The actions are defined as : "x'==1" to indicate the clock running, "x'==0" to indicate the clock is paused and x = 0 to reset the clock. All clocks in Uppaal progress synchronously. The *coffee machine* example uses a clock variable "y" in the *person* template.
  - Structured Data types: Along with the normal data-types such as integers, arrays of integers and clocks, Uppaal also allows user defined structured data-types; record-types can be defined using the keyword typedef struct. For example, consider the definition "typedef struct {int start_time; int end_time;} task". It defines a record type named task which has two components namely, start_time and end_time, both of type integer.
  - Bounded Integers: The Integer definitions can be bounded in Uppaal. E.g., int [0,10] var; specifies that the variable *var* can take values ranging from 0 to 10.
  - Constants: Values that are constant throughout the model can be defined as "const int variable = value", in global declarations.

Next we discuss the structure of a template automaton. Each template consists of a set of locations, transition edges between locations and local declarations. We define each component of the templates in detail:

- Locations: Locations along with the parameters and the valuations of variables and clock are used to describe the state of a system. A location is denoted by a blue circle. Figure 3.4 shows the location which are listed below:
  - Normal: a typical location in an automaton is depicted by a circle.
  - Initial: Defines the starting location of a process template. It is denoted by an encircled location.

  Locations can also have invariants defined. Clocks, integer variables, and constants can be referenced in the invariants [9].

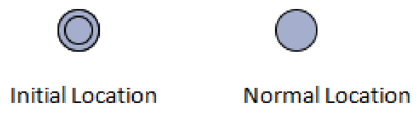


Figure 3.4: Types of locations

In the example model (Figure 3.1), all 3 templates have an initial location and several normal locations. Each location can also be given a name; e.g., in the example model, *Idle*, *Ready*, *WaitCoin*, *WaitCof*, *Go* and *Complain* are location names. The *WaitCof* location has an invariant (y<=2) which is defined over the clock variable and specifies the amount of time the system can be in that location.

- Edges: The edges are used to connect two locations to denote possible transitions. It is defined along with various attributes such as the ones given below:
  - Select: Defines a variable and a range of values from which the variable can non-deterministically take a value. In Figure 3.5, x: int[0,10] defines that "x"obtains a value in the range from 0 to 10 non-deterministically, when the edge from location 1 to location 2 is taken.

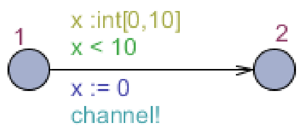


Figure 3.5: Example of an edge

  - Guard: Defines the condition, which has to be true in case the edge associated with the guard is to be taken. A guard is of Boolean type and consists of integer or clock values. Clock values are always compared with integer values. In Figure 3.5 the edge from 1 to 2, has a guard x < 10. This states that this edge can be taken only when x < 10. In the *coffee machine* example, in *person* template the guard "y==2" can be seen over the edge *WaitCof* to *Go*.
  - Synchronization: Defines the synchronization labels that affects a particular edge or is initiated by a particular edge. The label is either in the form ChannelName! Or ChannelName?. The synchronization variables in Figure 3.1 are publication! and coffee!. In Figure 3.5, channel! denotes synchronization variable.
  - Update: Defines update operations on variables. For example, in Figure 3.5 the variable "x" is updated with value 0. In the *observer* template of the

coffee machine example, the count variable is first updated to 1 and then updated using valued calculated by the function *countFunction*(count).

3.2 Uppaal Query Language

To perform model checking, we need a logical language to define the properties that the model should satisfy. The query language used in Uppaal is a subset of the TCTL (Timed Computational Tree logic) [11]. Similar to TCTL the query language consists of state formulae and path formulae. Path formulae can be classified into safety, liveness and reachability. We particularly consider the reachability and safety in this project.

- State formulae: A state formula is a logical expression with unary or binary operators combining components that represent the process template. For example, a state formula for the *coffee machine* example can be "y==2". A state formula can also be used to check whether the system is resided in a particular location, by using the syntax "*process.location*". For checking deadlocks Uppaal has a special keyword "deadlock". System reaches a *deadlock* when no outgoing transition is enabled and then the clock variables halt as well.

- Path formulae: A path formula quantifies a logical formula over some or all paths in the system. "E" is used to existentially quantify over paths and "A" universally quantifies over all paths in the system. In combination with the above mentioned quantifiers, one can use the modalities <> and [] respectively, to specify eventuality and invariance of a logical formula. These combinations (required combinations are only specified) are spelled out as follows.

  – A[] ϕ - For all paths ϕ always holds.

  – E<> ϕ - There exists a path where ϕ eventually holds.

Two particular types of formulae that we consider in the remainder of this report are reachability and safety. Reachability is to check whether a particular state formula can be satisfied in some reachable state. A formula of the form E<> ϕ can be used for this purpose. For example in the *coffee machine* example it can be checked whether the complain state can be reached by using the formula "E<> *Observer.Complain*", this shows that the timeout of 8 seconds has happened since *complain* location is reached. If this query returns false, it means that the system never reaches the *complain* location, i.e., two consecutive servings of coffee takes place within 8 time units.

Safety properties are used to ensure that some properties never hold or always hold. Safety properties can also be represented as a reachability problem by checking if the negation of the considered property is reachable. Checking that deadlock never occurs is a kind of safety property. It can be checked using the formula “A[] *not deadlock*”, which means deadlock never occurs along any path.

Chapter 4

Existing DSEIR Translation to Uppaal

The Octopus toolset already has the required infrastructure for translation of the user model in the DSEIR syntax to a model in the Uppaal syntax. Also creation of necessary query files for checking the newly created model. The work-flow of the entire system is shown in flow diagram in Figure 4.1. Given a user model, DSEIR model is created manually. This DSEIR model is then translated to Uppaal syntax using the existing translation support, i.e., the DSEIR translator. A suitable query file is created manually for the corresponding Uppaal model file using the Uppaal query file creator of Octopus. The query file consists of the properties that need to be verified, written in the query language of Uppaal. Using the created Uppaal model file and the query file we apply the Uppaal verifier to check whether the property is satisfied or not. In case the property is satisfied the model is verified otherwise a trace file is generated witnessing the violation of the property. This trace file is then converted into a user readable format and stored. This kind of analysis framework can also be seen in [12], with verification trace provided as output.

In this section, we first describe the existing translation support to Uppaal. This is done using a generic model which is defined in Section 4.1. Using this generic model, the mapping of a task in DSEIR syntax into an automaton in the Uppaal syntax is described in Section 4.2 along with the working of the corresponding Uppaal model. The application of this translation on the running example is treated in Section 4.3.

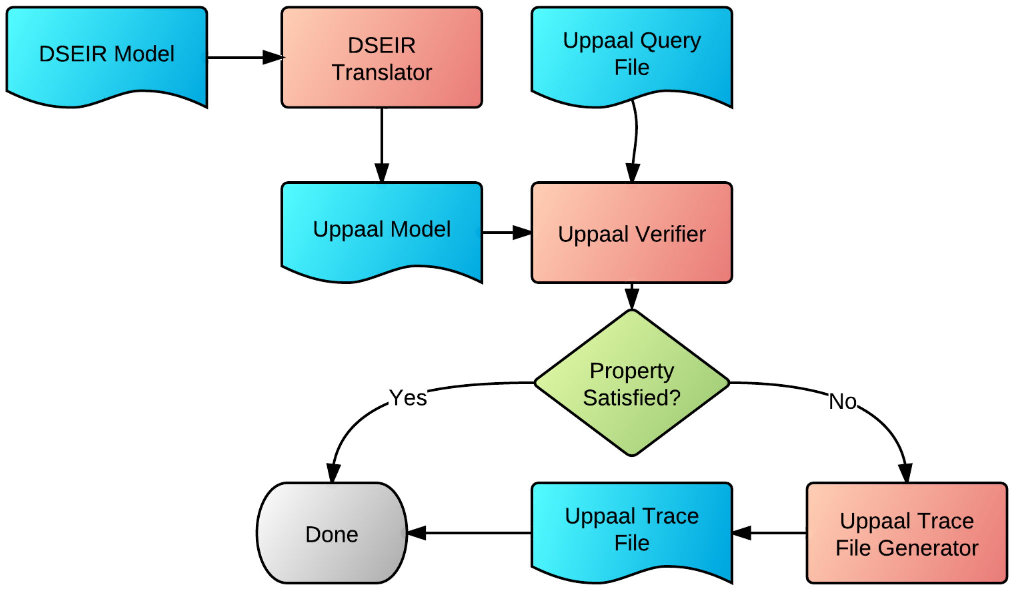


Figure 4.1: The existing work-flow for formal verification in the Octopus toolset

4.1 Generic Model

Consider a generic model with two tasks A and B. Here the two tasks have various parameters, A has “n” parameters and B has “m” parameters. The number of ports in A is “x” and B is “m”. In DSEIR, the number of ports should be at least equal to the number of parameters, with each of those ports having their binding expression as one of those parameters. A has “z” edges in this example and B has no edges here.

4.1.1 Application

The task-flow perspective diagram of the generic model can be seen in Figure 4.2. The task parameters of A and B are “a1 $\cdots$ an” and “b1 $\cdots$ bm” respectively. Task A has “x” ports with “n” ports with only parameters as binding expressions and remaining “(x-n)” ports contain “E” a mathematical expression containing operators, constants and parameters. The distribution of ports with only parameters and the rest are not necessarily in sequence always, hence they can be spread over the “x” ports in any order. The sequence numbers “Aa”, “Ab” indicate this randomness, here “a” and “b” could be any value between 1 and “x”. The operator set for “E” is $\{+, -, *, /, \%\}$ and the logical operator set for “C” is $\{<, >, \leq, \geq, ==\}$. For example, $a + b$ is a mathematical expression, $a + b > 5$ is a conditional expression. The notations “A1 $\cdots$ Ax” describe the initial values of the ports. The initial value of a port is an array with 0 or more

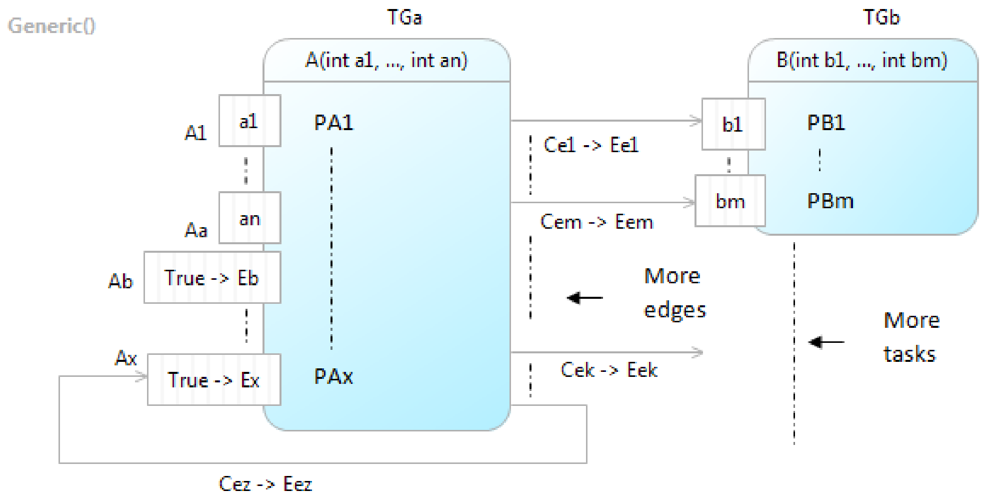


Figure 4.2: Task-flow perspective of the generic model

elements.

Edges that connect the tasks have their own conditional expressions "Ce" and value expressions "Ee". There are "z" (some number) edges for task A in this model. Note that all "m" ports of B may not always have edges from task A, but can have edges from other tasks as well. "TGa" and "TGb" are task guards of A and B respectively.

Figure 4.3 shows the load and mapping perspective of the generic model. Task A requires "p" service types with load amount associated with each of those service types, similarly B requires "q" service types. Service types of A has numbering from 11 to 1p and B has 21 to 2q to specify that both can request for different service types, they could be same as well.

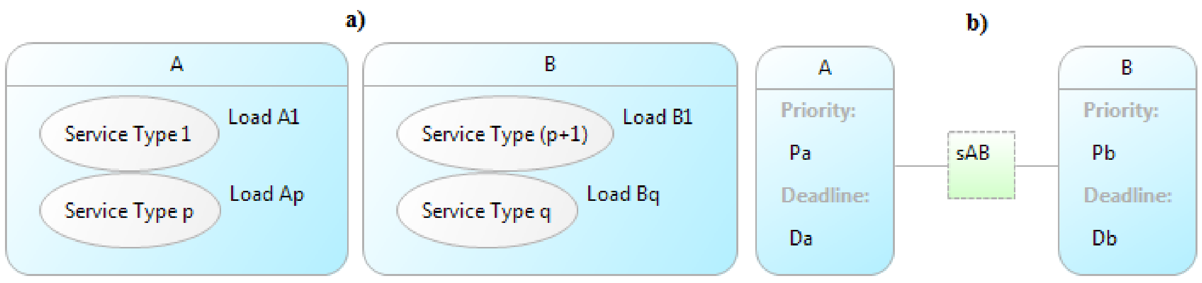


Figure 4.3: a)Load perspective of the generic model. b)Mapping perspective of the generic model

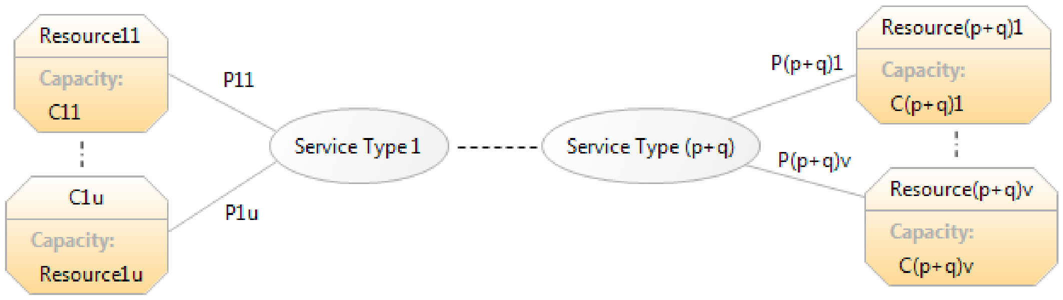


Figure 4.4: Resource perspective of the generic model

4.1.2 Platform

Figure 4.4 shows the resource declaration for the generic model. We have "u" resources providing "service type 1" and "v" resources providing "service type (p+q)". There are total "(p+q)" service types required by tasks A and B. We skip the scheduling perspective diagram here since it does not add any information to the translation description.

4.1.3 Mapping

The last perspective in Figure 4.3 shows the scheduler mapping of the tasks with their priority and deadline defined respectively. Priority and deadline of each task is also defined. They are mathematical expressions which evaluate to an integer.

4.1.4 Restrictions in Translation

The current infrastructure for translation to Uppaal consists of certain restrictions on the input DSEIR model it can translate, they are listed below:

- Ports cannot have conditions.
- Ports can have only single variables in their expressions.
- Delay on edges is not supported.
- Dynamic task priorities are not supported.
- Global Variables are not supported.

The generic model here follows these restrictions and is constructed using a subset of the actual DSEIR construct.

4.2 Syntax Mapping

This section describes the details of mapping a task in DSEIR syntax into an automaton in the Uppaal syntax and creating necessary local, global and system declarations for the same. As described earlier a task is translated into template automaton in Uppaal syntax. From the scheduling theory perspective a task usually has two states; it can be idle or active. The DSEIR translation to Uppaal hence creates two locations, namely idle and active for a particular task automaton. All tasks have these two locations. Since tasks can also be pre-empted and initialized there can be more locations. When a task has its initial values set, it has an extra location in the translation called initialize. When a task has a service type which is provided by a pre-emptable resource then the translated automaton has a location named pre-empted which is reached when this particular task is pre-empted by a higher priority task. Hence there are 4 types of translations possible as listed below:

- Idle, active (only and common for the rest of the types).
- Initialize and pre-empted (as extra states).
- Initialize (only as extra state).
- Pre-empted (only as extra state).

We discuss the translations with respect to the generic model defined previously. The translation of task A with initial values and pre-emptable resource requirement into a template, is shown in Figure 4.5. This translation in Figure 4.5 has four locations; with invariants shown with colour maroon. The locations of the automaton for task A are:

- *Initialize*: When the initial values of a task are set, it will have this location. Task A in the generic model has its initial values set hence it contains the location initialize.
- *Idle*: From the *initialize* location, Task A goes to the *idle* location where it waits for enough number of tokens in the port, priority requirements to be satisfied and resources to be available.
- *Active*: In the *active* location the task has acquired all the resources and is running. The time for which the task will be active is decided by the processing time (Figure 4.4) and the task load (Figure 4.3). Two functions are used to determine the minimum and maximum duration possible for its execution, namely the *minDuration()* and *maxDuration()* respectively. These functions are declared local to each task template. Invariants are used in this location so as to prevent the CPU from executing the task for a time more than the *maxDuration()*. *minDuration()* is checked to complete at least the minimum duration of execution before returning

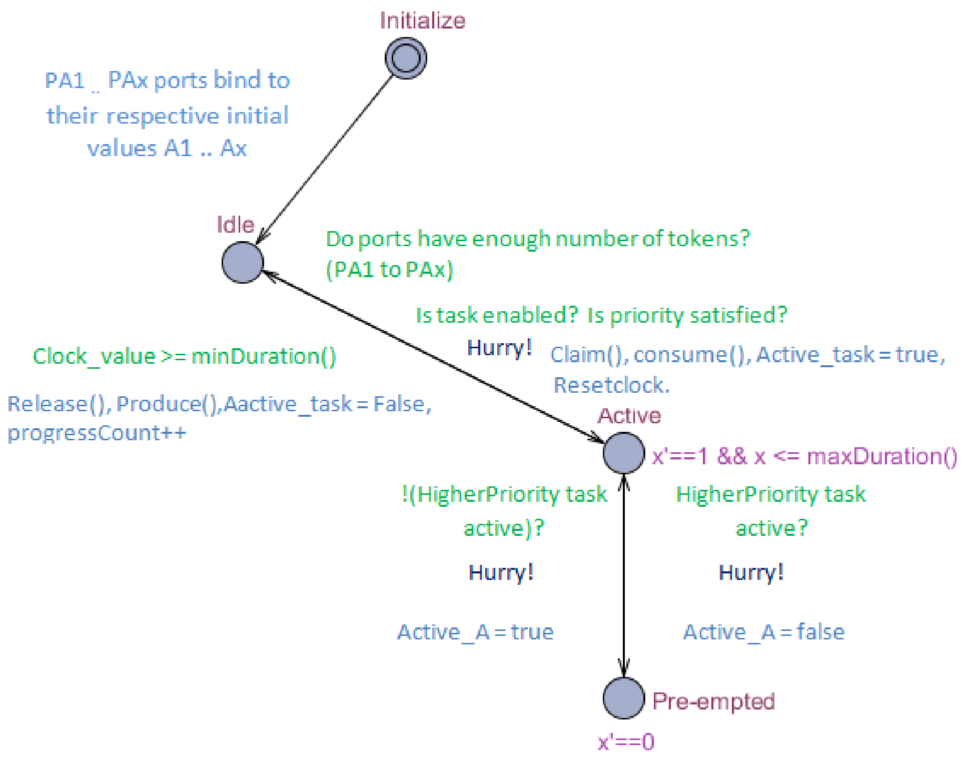

Initialize
PA1 .. PAx ports bind to their respective initial values A1 .. Ax
Idle
Do ports have enough number of tokens? (PA1 to PAx)
Is task enabled? Is priority satisfied?
Hurry!
Claim(), consume(), Active_task = true, Resetclock.
Clock_value >= minDuration()
Release(), Produce(),Aactive_task = False, progressCount++
Active
x'==1 && x <= maxDuration()
!(HigherPriority task active)?
Hurry!
Active_A = true
HigherPriority task active?
Hurry!
Active_A = false
Pre-empted
x'==0

Figure 4.5: A task model with all possible locations

to the idle location by the guard on the edge *active* to *idle*. These functions are defined in the local declarations which are defined later in this section. A clock variable “x” is used to keep track of the time elapsed in executing the task. Here $x` == 1$ symbolizes that the clock is ticking when in *active* location.

- *Preempted*: When the task instance is in the *active* location and is running on a pre-emptable resource like CPU then the task instance can be pre-empted. Hence this location is reached when a task instance is pre-empted. Preemption is determined by checking the active status of other tasks that contend for the same resource. In the current example the task A requires a pre-emptable resource and hence has the location *pre-empted*. The invariant $x` == 0$ of this location pauses the clock variable “x” since task A is not being executed when in this location.

All the resources considered in the generic model providing (p+q) service types (Figure 4.4) are declared in the global declaration section along with their corresponding capacity “C”. Also the parameters of the task are declared and initialized to value 0, the guard “is task enabled?” depends on the Boolean function “enabled()” and the variable “active_task” which is initially set to false and is set to true when the system goes from *idle* to *active* location, are declared in global declarations. The synchronization channel “hurry” is also declared in the global declarations. Note that C here is from the resource perspective and not from task-flow perspective.

```
//Global Declarations
//Channel
urgent broadcast chan hurry;
//Resources
Data_type Resource11 = C11;
Data_type Resource12 = C12;
...
Data_type Resource(p+q)v = C(p+q)v;
//Parameters
a1 = 0;
...
bm = 0;
bool enabled(){
if (required resources available)
return true;
else
return false;
}
bool active_task = false;
```

The guard “is priority satisfied?” and other functions “claim()”, “consume()”, “release()”, “produce()” are declared in the local declarations (local to each template or task). These can be seen in Figure 4.5. The guard “is priority satisfied?” is used to check if priority requirements of the task are satisfied, is true when any higher priority tasks is not in *active* location. The functions “claim()” and “release()” represent the task claiming the required amount of resource from the available amount when it starts execution and releasing it after execution. Functions “produce()” and “consume()” are used to represent consumption and production of tokens, before the task starts firing and after respectively. These are declared in the local declarations, where “consume” is more of a representative function setting the ports to a default value. “produce()” sets the binding expression on the port to the value expression on the incoming edge depending on the condition of the edge. These functions are listed below in a snapshot of local declarations for the generic model.

```
//Local declarations
clock x;
bool is_priority_satisfied(){
return (!active_t1) && ... (!active_t2)
// ( for each t1,t2 that belongs to the set of all higher priority tasks);
}
//For our generic model if B has higher priority than A then the statement
```

```
//would be "return(!active_B)"
void claim(){
Resource11 = Available_amount - LoadA1;
...
}
void release(){
Resource = Available_amount + LoadA1;
...
}
void consume(){
E1 = 0;
...
Ex = 0;
}
void produce(){
if (Ce1)
b1 = Ee1;
...
if (Cem)
bm = Eem;
...//All other edges, maybe to other tasks as well
if (Cez)
Ex = Eez;
}
int minDuration(){
return min(LoadA1) * P11;
}
int maxDuration(){
return max(LoadA1) * P11;
}
}
```

We then discuss the edges and their attributes:

- The edge from *initialize* to *idle*: can contain select and update statements, here the ports are set with initial values.
    - The initial values are bind to the binding expression of the ports, which can be a single variable like “a1” or a binding expression like “Ex” as shown in Figure 4.5. Select is optional and is usually set when the model requires selecting a random value from a given set. For example when the initial value of a port is selected randomly from the range (1 to 4), select statements are used.

- The edge from *idle* to *active*: This edge can have all the 4 attributes namely, select, guard, synchronization and update, set. Select can be present depending on the model behavior, hence we omit it and present the rest.
  - The 3 guards are, "is priority satisfied?", "is task enabled?" are explained previously and "do ports have enough tokens?". The last one checks if ports have sufficient tokens to fire/execute the task. The actual guard condition that can be seen in the task automaton, for the guard "do ports have enough tokens?" is shown in the following example we check for port P1 and P2 in some task.

    ```
    0 < numOfTokensP1 && 0 < numOfTokensP2;
    //This guard depends on the variables numOfTokens (P1 and P2)\\
    which has a count of number of tokens in ports P1 and P2 (respectively).
    ```
  - Includes a urgent broadcast synchronization channel "hurry" that is used to force the model to take this edge, whenever the edge is enabled. Note: Only send is used since the purpose of it is to force the model to take the edge and not to propagate any message.
  - The update part of the edge ensures the execution of the functions defined in the local template declarations of the task automaton namely "claim()", "consume()", "active_task = True" and "x=0". Here "x" is a local clock variable defined for each task and can be seen in the local declarations. It is reset when the task moves from *Idle* to *active*. It is used to keep track of the execution time of each task.
- The edge *active* to *idle*: This edge has a guard and some update statements.
  - The guard is "$x >= minDuration()$" ensures that the task has executed to at least a minimum duration of execution.
  - The functions in the update section are "release()" , "produce()", "active_task = False" and "progressCount++".
- The edge from *active* to *pre-empted*: Contains a guard, a synchronization channel and an update statement.
  - The guard checks if any higher priority task is active. In case a higher priority task is active the current task is shifted from the *active* to *pre-empted* location.
  - The urgent broadcast synchronization channel "hurry".
  - The update statement updates the variable active_taskName to indicate that the task is not active anymore.

- The edge from *pre-empted* to *active*: Similar to the edge from *active* to *pre-empted.*
  - The guard checks if any higher priority task is active. In case no higher priority task is active the current task is shifted from pre-empted to active location.
  - The synchronization "hurry" on the edge ensures that it is taken whenever possible.
  - The update statement updates the variable "active_taskName" to indicate that the task is active again.

The translation for a regular task such as task B without initial values and pre-emptable resource is shown in Figure 4.6. The edge in the task automaton is bidirectional the task can go from *idle* to *active* and vice-versa. The attributes over the edges are similar to the previous translation. Finally the system declarations part calls in all the task automatons to be executed in parallel.

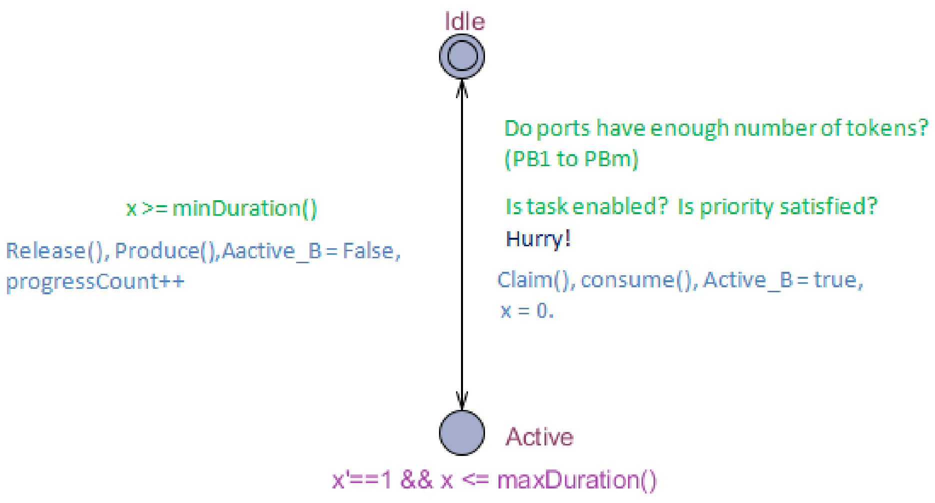

Idle
Do ports have enough number of tokens?
(PB1 to PBm)
Is task enabled? Is priority satisfied?
Hurry!
Claim(), consume(), Active_B = true,
x = 0.
x >= minDuration()
Release(), Produce(),Aactive_B = False,
progressCount++
Active
x'==1 && x <= maxDuration()

Figure 4.6: Normal DSEIR translation for task B

4.3 DSEIR Translation of Running Example

The existing Uppaal translation of the printing example described in Chapter 2 has been implemented in the Octopus toolset. Our translation creates one automaton per task in the form of a template in Uppaal syntax. The translation of scan task in this example model using our translation scheme is given in Figure 4.7, the task is renamed to task A. The translations for the rest of the tasks (task B and task C corresponding to image processing task and print task respectively) are shown in Figure 4.8 and Figure 4.9 respectively.

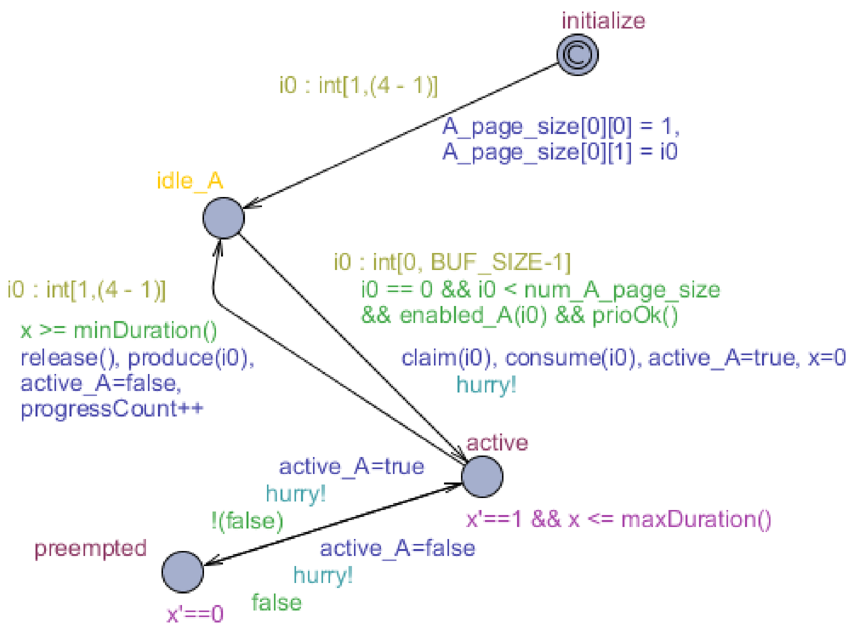


Figure 4.7: Automaton of scan task(task A)

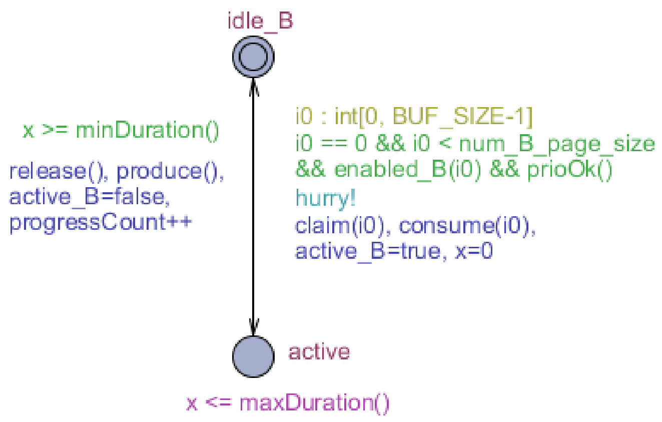


Figure 4.8: Automaton of image processing task(task B)

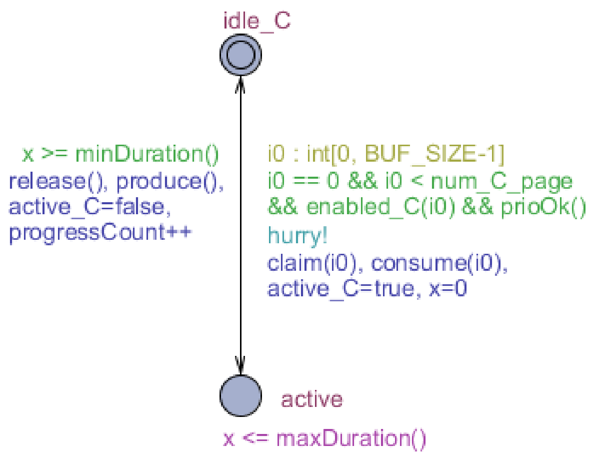


Figure 4.9: Automaton of print task (task C)

Chapter 5

Extending DSEIR with Deadlines

The first goal of the thesis is to make schedulability analysis *possible* in Octopus. Given the description of the current translation as discussed in Chapter 4, we identify the additions that would help us build the schedulability analysis procedure on the existing infrastructure. The necessary additions that we require to the existing DSEIR translations are:

1. The DSEIR language already consists of tasks, loads and resources but lacks *deadline*, an important feature required to perform schedulability analysis.

2. The current implementation considers only execution of tasks and pre-emption, we now need to consider the newly added deadline feature.

Solving the first part is fairly simple and can be done by adding an attribute to the task in the DSEIR language. We then incorporate this change in the DSEIR translation by creating corresponding translation rule for deadlines in the Uppaal syntax. Focusing on the second part, schedulability analysis can be realized by creating a mechanism in Uppaal that monitors each task using their relative deadline and moves to an error state if a deadline is missed. This allows for embedding schedulability analysis into a reachability checking problem. The approach to be taken is divided into two steps:

- We need to initially create possible deadline detection mechanisms in Uppaal. This step involves creating some models, validating and verifying them.

- The next step is to modify the existing DSEIR translation to incorporate this deadline detection mechanism.

The extension of the current translation, to realize the second part along with the designs discussed in detail in Section 5.1. These changes are then also incorporated into the DSEIR translation by making the necessary changes to the existing one. The new translation then considers *deadline* as one of the attributes of the tasks and adds the deadline detection mechanism. The changes that will occur in the work-flow through the modifications made in this section are reflected in the work-flow diagram depicted in Figure 5.1. The changes are, namely, the addition of the deadline feature and the deadline detection mechanism to the translation procedure and the simultaneous creation of a deadline query depending on the model and use it in the verification process. The running example is used to explain the working procedure with respect to the new translation, as well as verification process in Section 5.2. In this chapter, we describe the general framework that would be applied to any DSEIR model performing schedulability analysis. We then compare this framework with another schedulability analysis framework described in [13]. This is done by first describing the scheduling framework in DSEIR terms and applying schedulability analysis on it, is explained in Section 5.3.

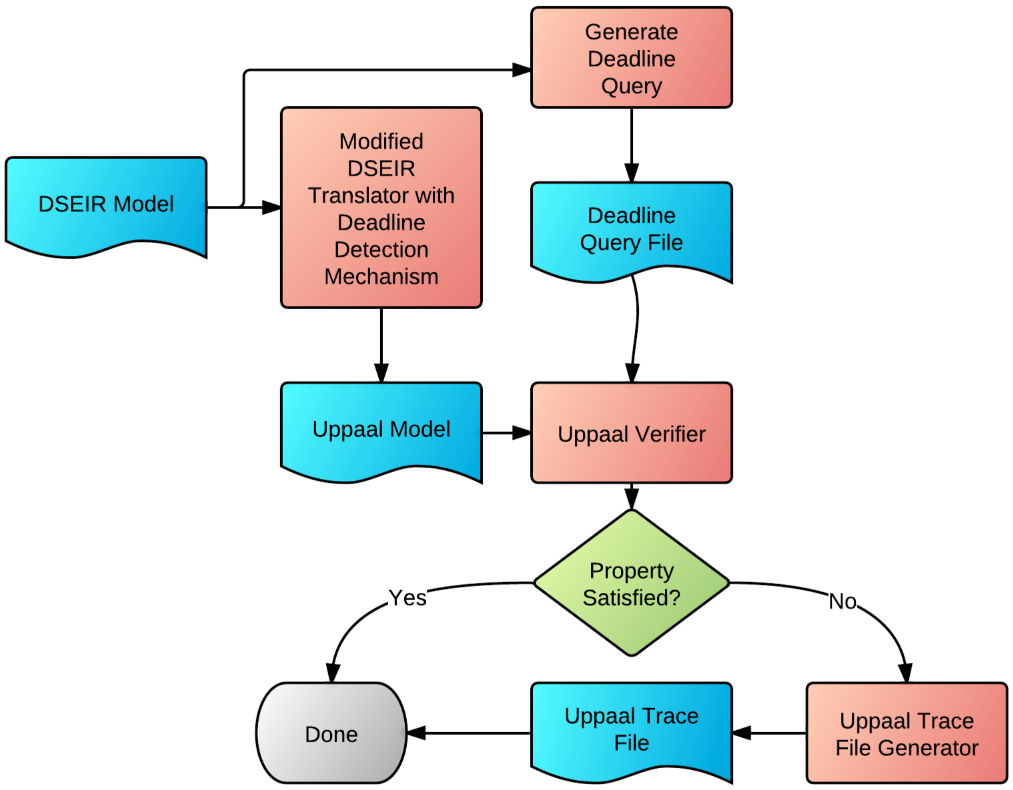


Figure 5.1: Work-flow with the extended DSEIR translation

5.1 Extending Task Automatons

In this section, we explain the designs used for creation of a deadline detection mechanism. Since we are using Uppaal we consider performing schedulability analysis, in the form of reachability analysis. The idea of performing reachability analysis for similar analysis procedures can be found in detail in [13, 3]. We create an extra location called *stop/error* which is reached when a task misses its deadline and still has some execution time left. The schedulability property can then be verified using the formula "E<> *Task.Stop*", which states that "There is a path in the system on which the system eventually moves to the Stop location". The expected result is "false" or "property not satisfied". In case the property is true, the model violates a deadline. In order to verify schedulability by means of a safety property, we specify the query as "A[] *not Task.Stop*", which states "Across all the path the system never goes to the *stop* location".

Firstly we thought of using the clock comparison with relative deadlines by creating a clock that starts when the system starts in and by capturing the starting time and obtaining the difference between the current time and the starting time, thereby detecting deadline violations. But this was not feasible since Uppaal does not allow for capturing clock values during verification. Thus we then decided to use clock guards to capture the deadline violation scenario.

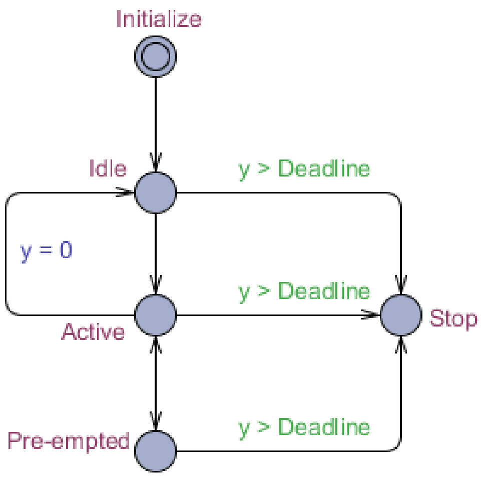


Figure 5.2: Uppaal model with deadline detection mechanism - First approach

In the first model (Figure 5.2) we added a new location *stop* and an edge to this location from each of the 3 other locations *idle*, *active* and *pre-empted*. A guard is placed on these edges, which is used to ensure that the task has exceeded its deadline and thus can reach the *stop* location then. This model uses an additional clock ("y" because the model already has a clock variable "x") which keeps track of the lifetime of a task (time to deadline). In this model, once a task is initialized, a new clock variable “y” starts ticking which is used to check deadline violations; it is reset every time the task goes from *active* to *idle* because this is when the task becomes “ready”.

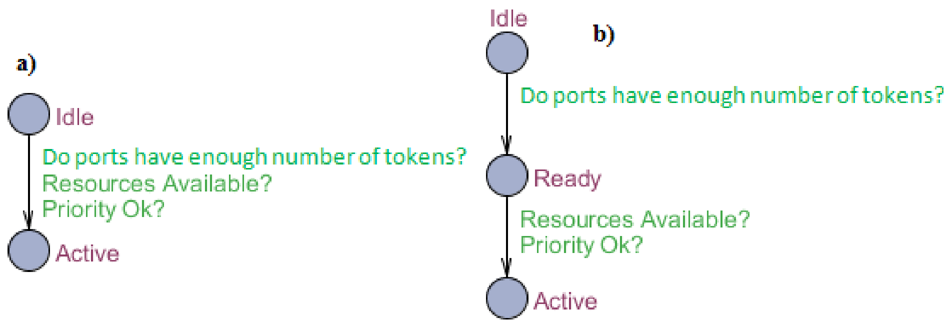


Figure 5.3: a) The *Idle* to *Active* location in the initial approach, b) The addition of *Ready* location in between

In DSEIR, the concept of “phase” in its scheduling theory sense is not present, however, a task is said to be “ready” only when its ports have enough tokens. The clock for the deadline detection, has to start ticking when the task becomes ready. But in the model of Figure 5.2 the task is not “ready” when it is in the *idle* location. As we can see in Figure 5.3 a), we check that “Do Ports have enough tokens?” and directly go to *active* along with checking other conditions required to be “active”. In practice, it is not always the case that a task can go from *idle* to *active* directly, but it might sometimes remain in the ready state until the resources are available and the priority is satisfied. Thus we identified that the model in Figure 5.2 does not faithfully represent the “ready” state of a task. Hence we first add a new location to the model, called *ready.* We then divide the guards on the pervious “*idle* to *active*” edge over two edges, namely the “*idle* to *ready*” and “*ready* to *active*”. We present this second approach in Figure 5.3 b) and Figure 5.4. Although we have added this *ready* location, we still preserve the previous model (without *ready* and *Stop*) for analysis procedures that do not involve deadlines in them because it is consistent and properly represents the models that do not have deadlines. Thus we have 2 options of implementation for DSEIR translation to Uppaal depending on whether the deadline should be considered or not.

5.2 Application of the Extension on the Running Example

We now apply the new modified DSEIR translation to the running example. Here the task A, task B, and task C are corresponding representations of scan, image processing and print tasks respectively. We explain only the new elements that are included in the Uppaal model created using the modified DSEIR translator. The rest of the elements

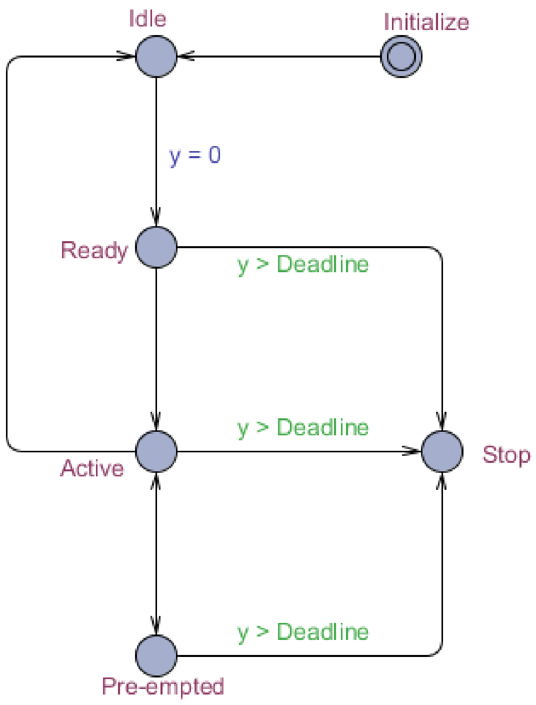


Figure 5.4: Uppaal model with deadline detection mechanism - Second approach

remain the same as discussed previously. As mentioned earlier two new locations are added to the automaton and named as the *stop*location and the *ready* location. The additional clock variable used here is “y”. The attribute for defining the deadline is also used in the model *deadline_A()*, which is a function that returns an integer value on evaluation. Also the variable *progressCount* is eliminated from the translated model, which causes unnecessary *state-space* explosion. The *state-space* explosion is caused because it was initially designed to keep track of the number of task executions in the model, but when we create infinite models through abstraction (with elimination of conditions explained in Chapter 6), this is one number that keeps on increasing due to which the verification never obtains any result.

The templates of the task automata of task A and task B from the example are shown in Figure 5.5 and Figure 5.6. The running of the model remains almost same as previous except of the deadline detection mechanism and with respect to the newly inserted location *ready*. The changes occurring due to the insertion of the location has been described previously. We proceed further with the description of the deadline detection mechanism. The clock variable “y” is started when the task becomes ready and goes on until it is reset again through the edge from “*idle* to *Ready*”. Also note that the clock value keeps on progressing when the system is in idle location but has no effect whatsoever on the model, since it is reset when going from “*idle* to *ready*. Deadline violations can be found by comparing this variable with the current relative deadline.

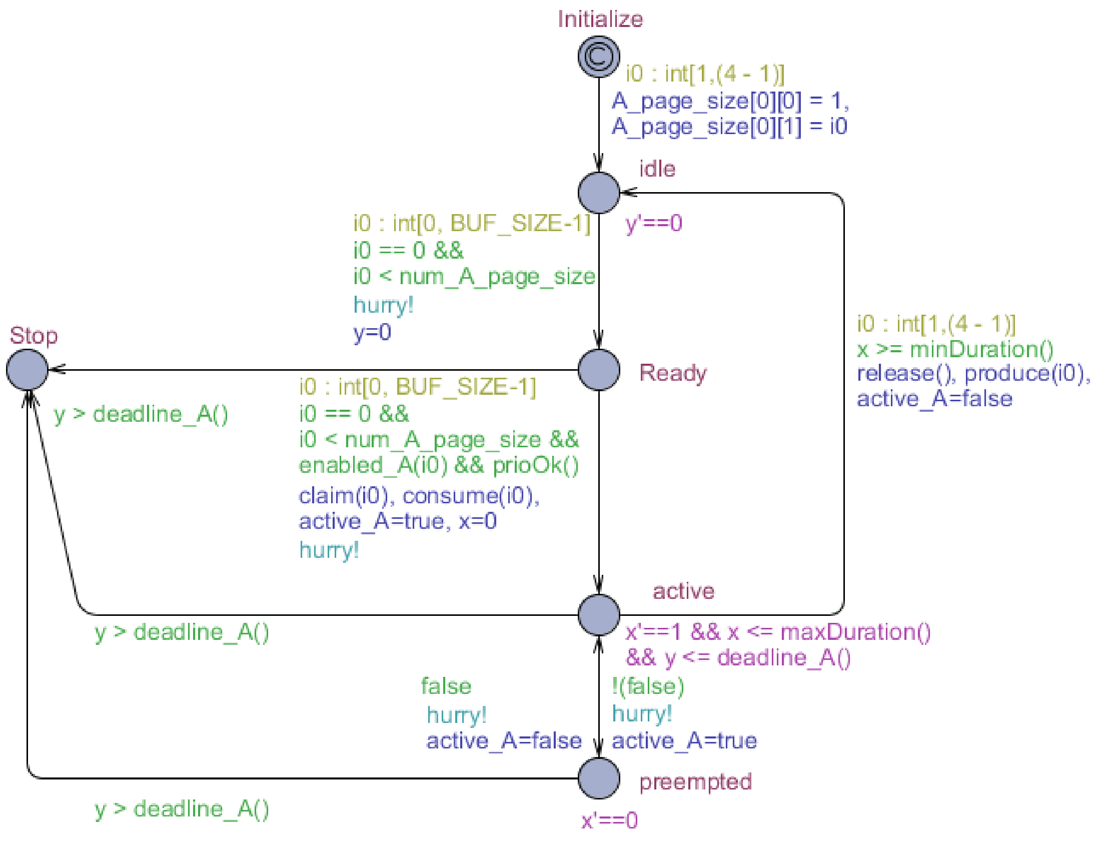


Figure 5.5: Template of the modified automaton of task A

Hence, when the deadline violation takes place in any of the locations namely, *ready*, *active* or *pre-empted*, the corresponding edge leading to the *stop* state is taken. The translation for task C is similar to task B, model changes in variable names.

Verifying the Deadline Violation

We define the models in Uppaal in the form a of a reachability problem, hence we use the following formula, to check whether the Error state (*stop* location) is reached or not.

```
E<> (A.Stop or B.Stop or C.Stop)
```

Checks for the property that if any instance of task A, or task B or task C violates the *deadline* in any state. The expected result for a proper model here is that the property is not satisfied; we obtain this result for the current example and is shown in Figure 5.7. Note that the schedulability analysis implemented is an entirely automated process where given the DSEIR model, the Octopus engine will transparently perform translations, call

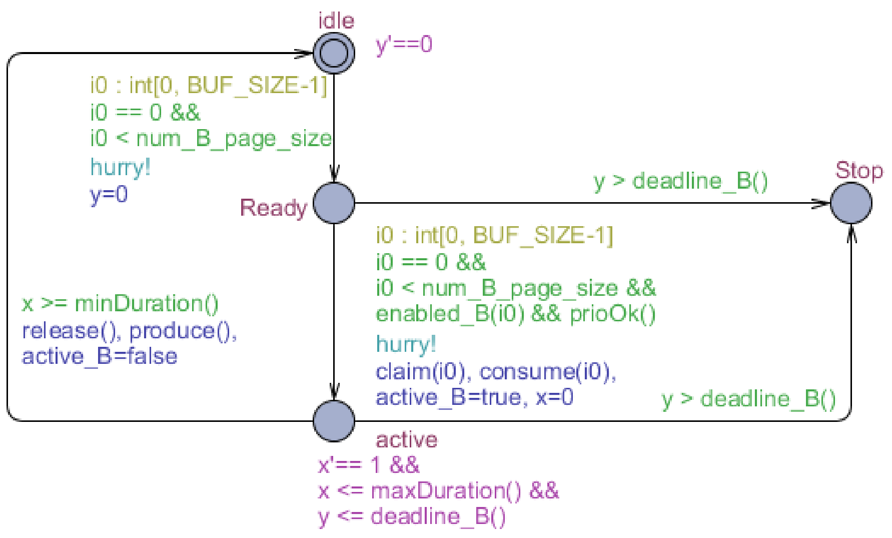


Figure 5.6: Template of modified automaton of Task B

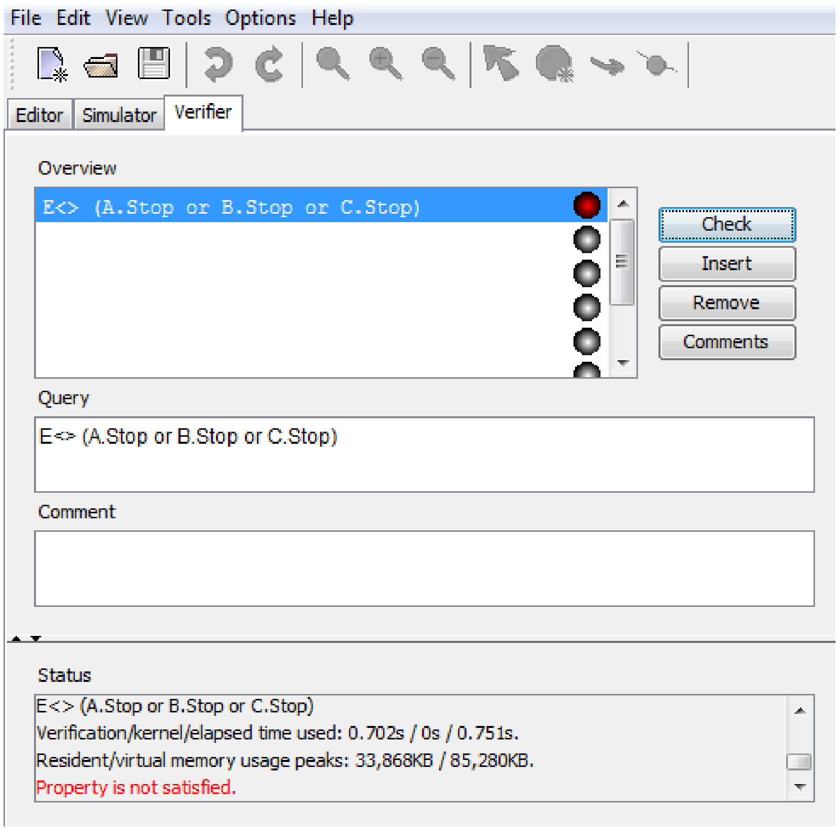
Uppaal, verify, fetch the results and displays then. Figure 5.7 is manually checked and depicted here for understanding purpose.

5.3 Application of Extension

Alexandre et al. explained in [13] the procedure to create a scheduling framework in Uppaal to define a range of classical schedulability problems. We now model this scheduling framework using the DSEIR language. We use this example to compare between the manually created model in Uppaal following the approach in [13] and the automatically generated version by the Octopus toolset.

[13] also includes an example model that consists of 5 tasks, and is explained in detail in this section. The task dependency graph of this task-set is shown in Figure 5.8. This task-set is a general real time task-set with certain attributes set accordingly. These tasks are defined using the following attributes.

1.Best case execution time(BCET)	5.Best case execution time(BCET)
2.Worst case execution time(WCET)	6.Maximum period
3.Deadline	7.Priority
4.Offset	8.Initial Offset



```
E<> (A.Stop or B.Stop or C.Stop)
```

Query: E<> (A.Stop or B.Stop or C.Stop)

```
E<> (A.Stop or B.Stop or C.Stop)
Verification/kernel/elapsed time used: 0.702s / 0s / 0.751s.
Resident/virtual memory usage peaks: 33,868KB / 85,280KB.
Property is not satisfied.
```

Figure 5.7: Model checking results for the running example, displayed in Uppaal verifier

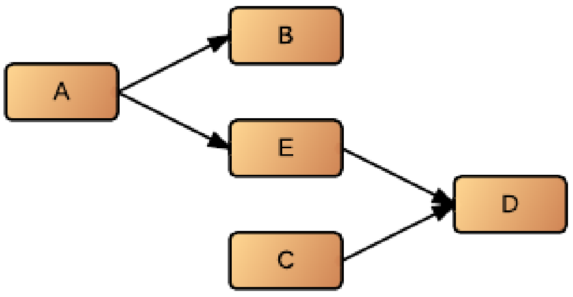


Figure 5.8: Task-flow graph of the scheduling framework example in general scheduling terms

5.3.1 Application

Given the syntax definition of the DSEIR language, this scheduling framework is modeled in the DSEIR language with the help of tasks, ports and tokens. This example requires periodicity in its model since the tasks are periodic in nature, but DSEIR has no concept of periods. Although this could be modeled via self loop on each task with (period - execution time) as the delay on its edge, we model it differently owing to the restriction that delay is not supported, specified in Section 4.1.4. An extra task can be used along with the pre-existing tasks which is used to model period. This special task is the Init task which produces 6 tokens, one for each task including itself. We use a single task for this purpose, since all tasks have equal periods. This token is used to indicate the start of a new period. Figure 5.9 shows the model. The various dependencies between the tasks are modeled via the token and port system so as to make the dependent tasks wait for their predecessors. The basic requirement for the tasks to begin execution is to have at least one token on each of its ports.

The idea of modeling an extra task for imitating periodicity can be applied generically to other models as well. Although for models with different periods for different tasks, an extra task can represent the period for each task, the representation of this period through the delay can be more useful. This concept has to be further investigated since delay is not directly applicable when variable load/execution times are involved.

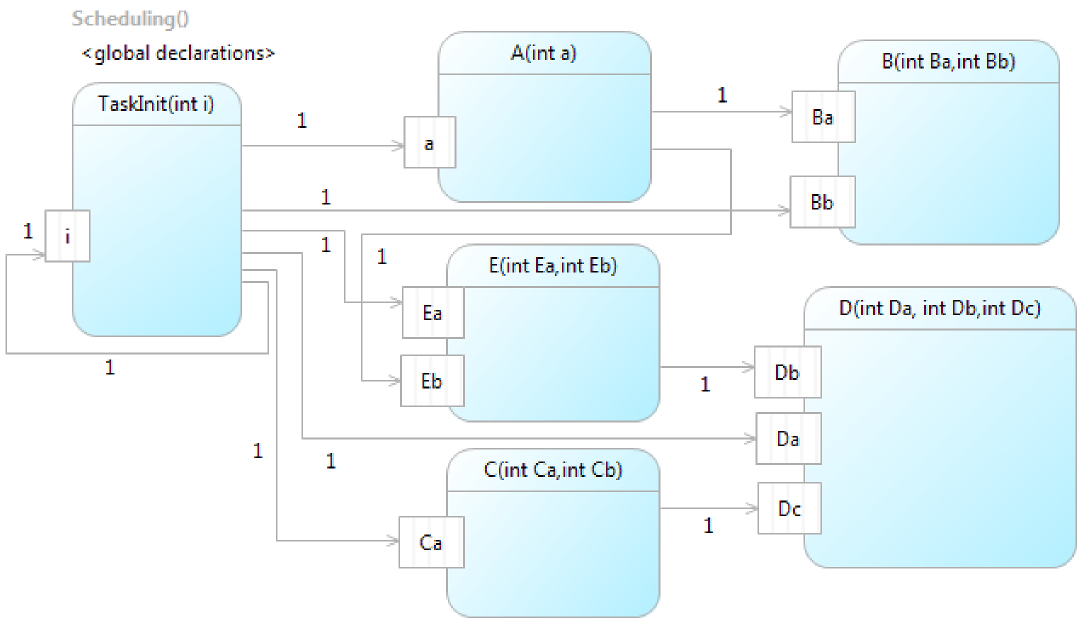


Figure 5.9: Task-flow perspective of the scheduling framework example

5.3.2 Platform

This task-set uses 3 resources providing service_type COMPUTATION and one resource providing the service_type TRANSFER. The service_type COMPUTATION is used by

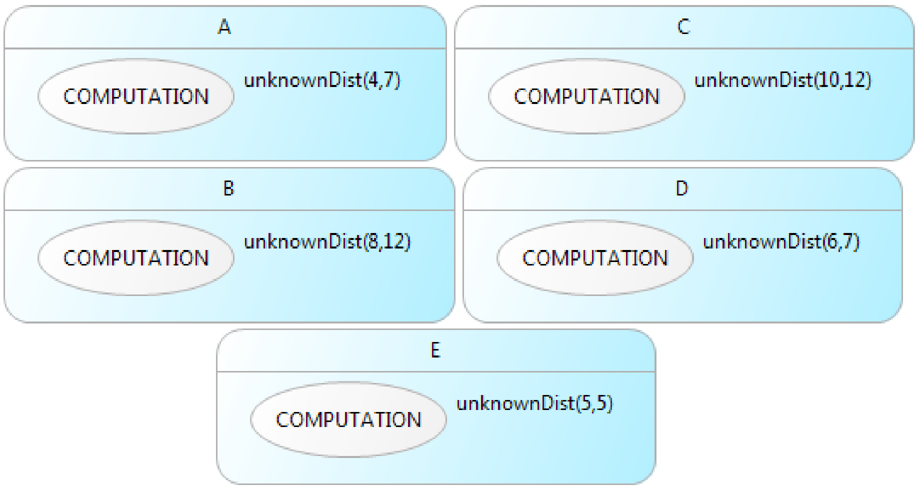


Figure 5.10: Load perspective of the scheduling framework example

the Init task and other tasks A, B, C and D. E uses the service_type TRANSFER provided by the resource bus. The resources used and the scheduling of those resources can be seen in Figure 5.11 and Figure 5.12, respectively.

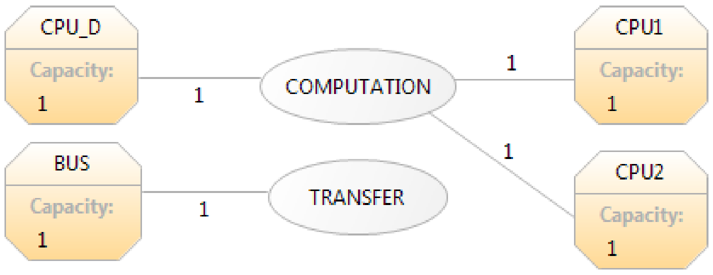


Figure 5.11: Resource perspective of the scheduling framework example

5.3.3 Mapping

The task-set uses 4 schedulers. The mapping perspective diagram is shown in Figure 5.13.

We then successfully translate this DSEIR model to Uppaal model and check for schedulability using the schedulability query.

```
E<> (A.Stop or B.Stop or C.Stop or D.Stop or E.Stop)
OR
A[] not (A.Stop or B.Stop or C.Stop or D.Stop or E.Stop)
```

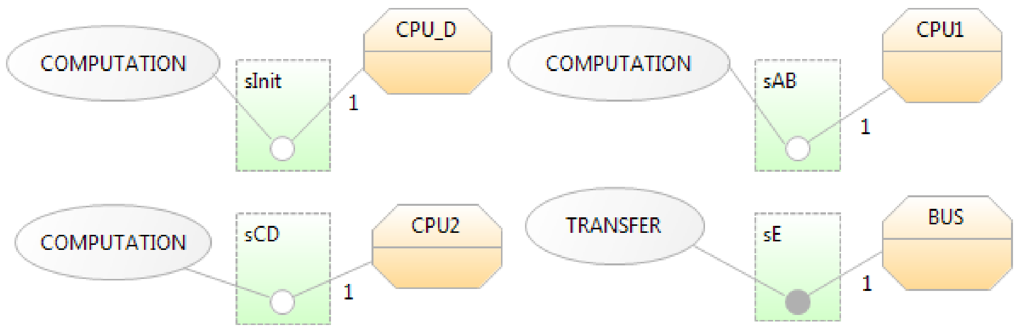


Figure 5.12: Scheduling perspective of the scheduling framework example

.

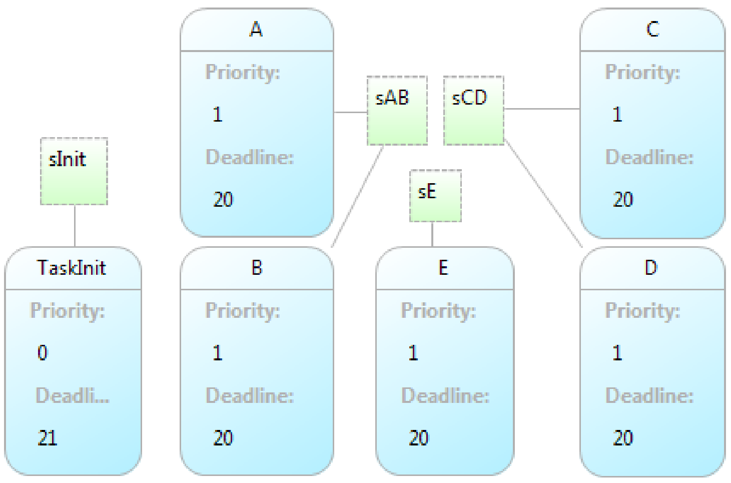


Figure 5.13: Mapping perspective of the scheduling framework example

5.3.4 Statistics

The modeling framework given by Alexandre et al. is a manually built model in Uppaal. In Octopus we manually build the corresponding model using DSEIR and then it is translated to Uppaal syntax using automatic methods. We obtain the statistics by running some tests on both of these models. We consider 3 test cases with 5, 10 and 12 tasks on both models. For the initial test case of 5 tasks, we model it from [13] example. For test case with 10 and 12 tasks, we replicate some of its tasks keeping the number of resources the same and increasing the period value correspondingly. Looking at the results (Table 5.1) we infer that although for small number of task instances the automatic model is slower than the manual model. For larger model (large number of instances) the automatically generated model has a much smaller *state-space*, lower timing and memory requirements, though the reason for this huge difference is not entirely known. Also DSEIR is far more expressive than the manual model since it allows for handover, variable instances for each tasks via condition on edges and passing actual data between tasks. Although we can conclude that the DSEIR based automatic models can be successfully used as modeling framework for schedulability analysis, and do not suffer from performance degradation due to the automation process, it is still an open research area to be investigated. In Table 5.1, a cell for specifying memory requirement of the manual model with 12 tasks is left blank since the value could not be obtained.

Table 5.1: Performance Comparison

Feature	No of tasks	Manual	Automatic Model
States	5	983	56
Time	5	0.08s	0.7s
Memory	5	10MB	50MB
States	10	282996	190
Time	10	116s	2s
Memory	10	364MB	95MB
States	12	3224144	224
Time	12	44mins	2.6s
Memory	12	-	201MB

Chapter 6

Data Abstraction

In Chapter 5, the first goal of the thesis was achieved, i.e., schedulability analysis is made *possible* for DSEIR models. This analysis process includes automatic translation to Uppaal and mechanized verification of this translated model. Although schedulability analysis is now possible its scalability to larger models remains an issue. Applying DSEIR translation and performing model checking on such models results in the *state-space* explosion problem. Thus the next part of the thesis is to make schedulability analysis *scalable* on arbitrary DSEIR models, with focus on the reduction of the state space.

In a DSEIR model, a task can have numerous instances but these instances share part of their behavior. Thus if we are able to identify data parameters of these instances that can be unified and unify them, we obtain abstract models that approximate the behavior of a class of instances. To perform the abstraction, we need to:

- Identify the data parameters that can be abstracted.
- Perform data abstraction on these data parameters.
- Transform the original model to a new abstract model using the obtained abstractions.

Although the behavior of the abstract model approximates the behavior of the concrete instances, it can also comprise new behavior that is not present in any concrete instance. This is the price paid for obtaining reduced *state-space.* The abstraction procedure changes the previous work-flow (Figure 4.1) for schedulability analysis in Octopus. The new work-flow diagram is shown in Figure 6.1. On an abstract note, the change is that after a DSEIR model is created, the next step now is to create an abstract model using the new abstractor function in DSEIR. After an abstract model is created, the work-flow uses the same old flow of actions, omitted in Figure 6.1 for the sake of brevity. The

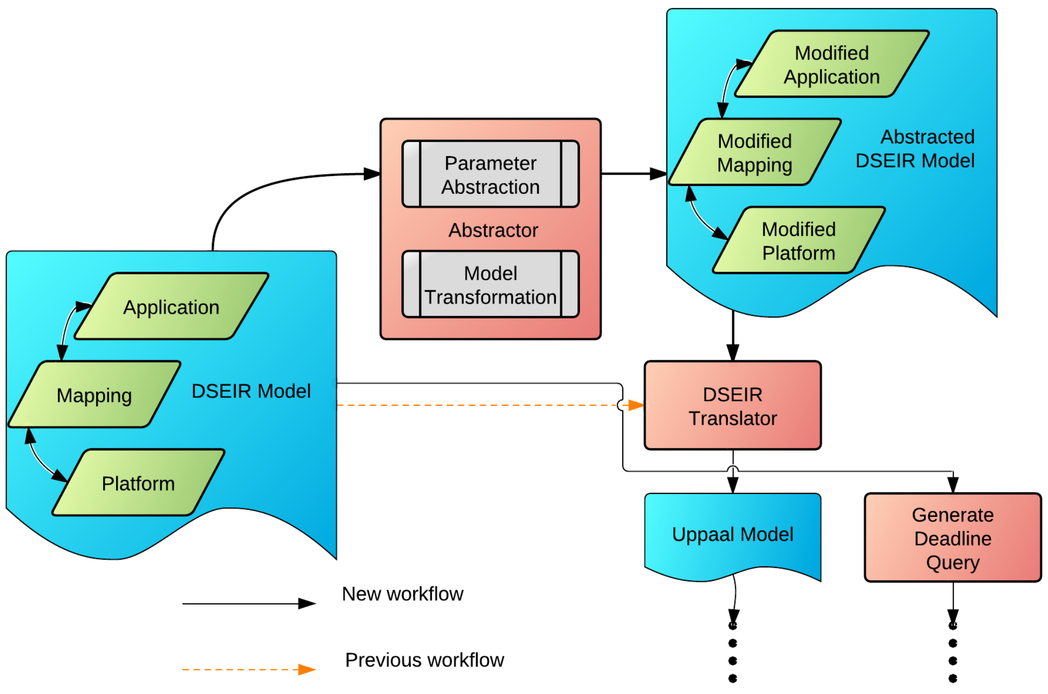


Figure 6.1: New work-flow diagram with abstraction procedure incorporated

work-flow will be further explained in detail in Section **??**

In this chapter we first describe the motivation to perform abstraction in detail, with an example in Section 6.1. Also in this section we present the design approaches that can be used to extract the data from the user model and to create an abstraction from it. The abstraction algorithm is described in Section 6.2. The necessary helper functions required by the algorithm are explained in Section 6.4. An example is used to illustrate the working of the abstraction algorithm in Section 6.5.

6.1 Motivation and Approach

State-Space Explosion: Models used in Octopus can contain tasks with a large number of instances resulting in numerous combinations of elements/attributes and hence a huge *state-space*. The *state-space* grows exponentially with the increase in number of attributes of an arbitrary model. Thus the model checking procedure becomes time consuming and the verification tool may reach its resource limits, abruptly stopping the procedure and giving no results.

Consider an example task model with 3 tasks A, B and C, whose possible task execution

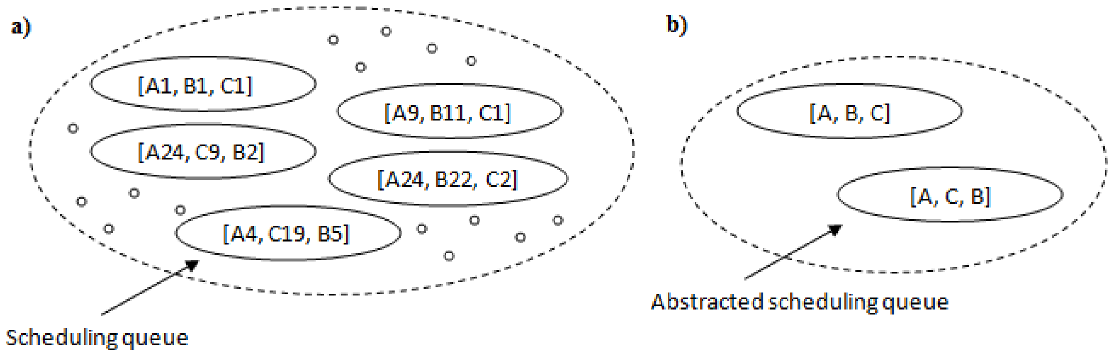


Figure 6.2: a) *State-space* explosion b) *State-space* of abstracted model

flows are A− >B− >C or A− >C− >B. Assume all of these tasks have "n" instances. In such a case the *state-space* of the model created using the current DSEIR translation will be as the one shown in Figure 6.2 a), the right hand side shows the *state-space* of the abstracted model. Along the execution sequence each task might have a lot of repetitions in terms of its task instances. Assume that A1 and A3 might be same in terms of the value all its attributes; they are still considered as different instances in the state space and represented twice. There can also be a scenario where after 2 instances of each task, this one execution pattern ($Ax = A(x + 2) = A(x + 4) \cdots$ where $x = 1 \cdots n - 2$ same for B and C) is repeated "(n/2)-1" times (-1 for first execution) when the total task instances are "n" for each task. Note that both A(x+2) and Ax represent instances, the brackets are to show that 2 is added on to value of x. Although repeating the execution sequence has no effect whatsoever on the model-checking results because the behavior does not vary when the instances and the execution sequence is same, it is still verified for all instances since they have different attribute values. For example, A1 and A10 are equal in terms of their execution time and other important factors but have different instance number which is sufficient to differentiate them in *state-space*. Thus ideally we need to check only one execution sequence of the cycle to verify the properties. The effect of this abstraction procedure on the load is shown in Figure 6.3. We eliminate the instances by capturing the load of all instances and replacing with an instance that represents all possible loads with a range of load specified by its minimum and maximum ([minA,maxA]). Note: In DSEIR each instance could have a range specified as its load value. The bounds of this range are calculated using the following set of formula:

$$min_{Ai} = minimum(\text{minMaxExpEvaluator}(LoadAi)), i = 1 \cdots n$$

$$max_{Ai} = maximum(\text{minMaxExpEvaluator}(LoadAi)), i = 1 \cdots n$$

$$minA = lower\_bound(min_{A1}(LoadA1)...min_{An}(LoadAn))$$

$$maxA = upper\_bound(max_{A1}(LoadA1)...max_{An}(LoadAn))$$

```
//lower_bound and upper_bound is used because
```

```
their input is a range with two bounds.
The function MinMaxExpEvaluator used here is explained later.
Given an input expression it gives the minimum and
maximum possible values for that expression.
```

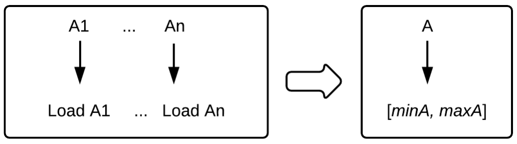


Figure 6.3: The effect of abstraction on load values

Also in a case where the task instances repeat but not the same execution sequence we still have considerable reduction in *state-space* by eliminating these instances. To create an abstract model we need to know all possible values, for all the attributes through all the instances of the task. Thus we require an approach to capture all possible values and record them into a new model that approximates the original model.

Approach

The data that can be abstracted in the DSEIR model is the parameter values of that task. Currently it can take a range of values considering all the possible instances of a task. The procedure of abstraction has two steps as shown in Figure 6.1. The steps are listed below:

- We perform *parameter (data) abstraction* by mapping parameter values into a range of values, by finding the minimum and maximum value that the tasks can take during the execution of the input model. We create a map in which each parameter is mapped onto its respective range specified by minimum and maximum values. The effect of parameter abstraction on the parameters is shown in Table 6.1. We discuss the parameter abstraction in detail in Section 6.2. Note: here {} represents a set and [] represents range.

Table 6.1: Effect on parameter values

Parameter	Previous Values	New Value
a1	$\{a1_1...a1_m\}$	[min($\{a1_1...a1_m\}$),max($\{a1_1...a1_m\}$)]
.	.	.
.	.	.
an	$\{an_1...an_k\}$	[min($\{an_1...an_k\}$),max($\{an_1...a1_k\}$)]

- In the next step we use this parameter map and certain functions to transform the DSEIR model into an abstract model. This step is further subdivided into two parts, listed below:

– Abstracting the load and deadline expressions. The effect of parameter abstractions on the load value and deadline value is shown in Figure 6.4 and Figure 6.5, respectively. The priority value is also affected in the transformation which is shown in the mapping perspective along with deadline value. The new priority values are constants and hence dynamic priority is not supported like in original DSEIR.

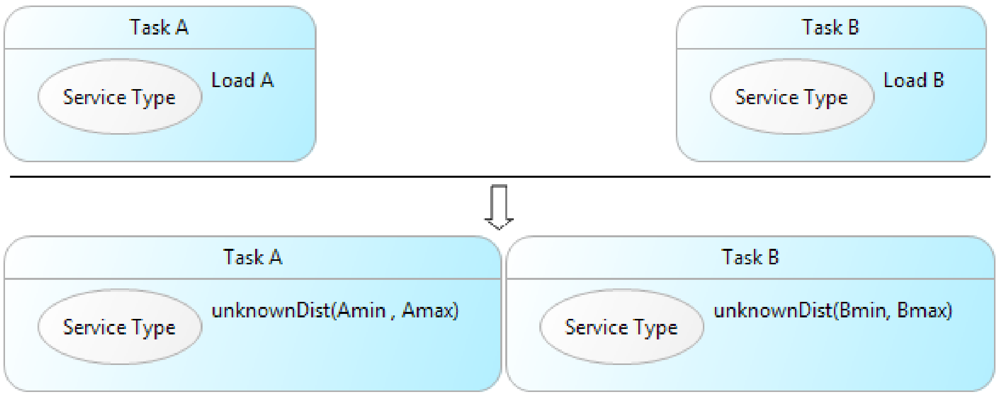


Figure 6.4: a)Affect of abstraction on load value

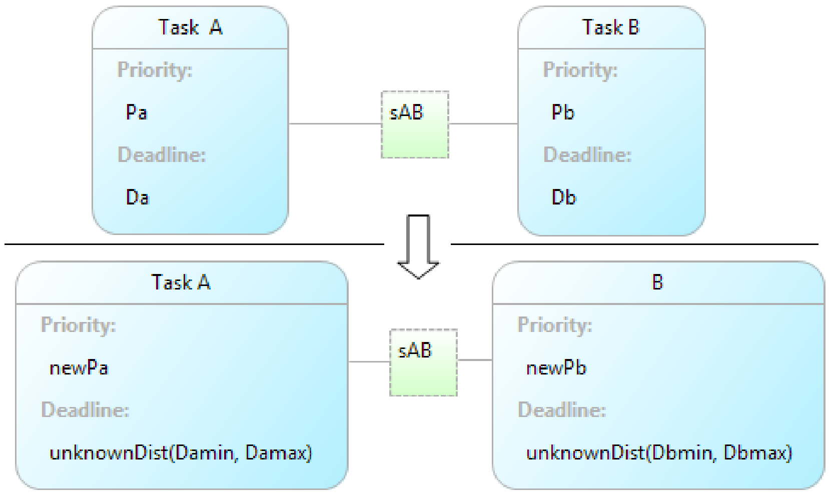


Figure 6.5: Affect of abstraction on deadline value and priority

– Transformation of the model using the new values for load and deadline, value expressions over the edges are changed to a constant (1) since they are no longer required for calculations but are preserved to maintain the task-flow and the conditions on the edges are eliminated, this change can be seen in Figure 6.6.

This transformation step is further explained in detail in Section 6.3

These two steps, parameter (data) abstraction and model transformation can be seen in the new work-flow in Figure 6.1. Also the change in the 3 parts of the DSEIR model namely, application, platform and mapping is explicitly specified.

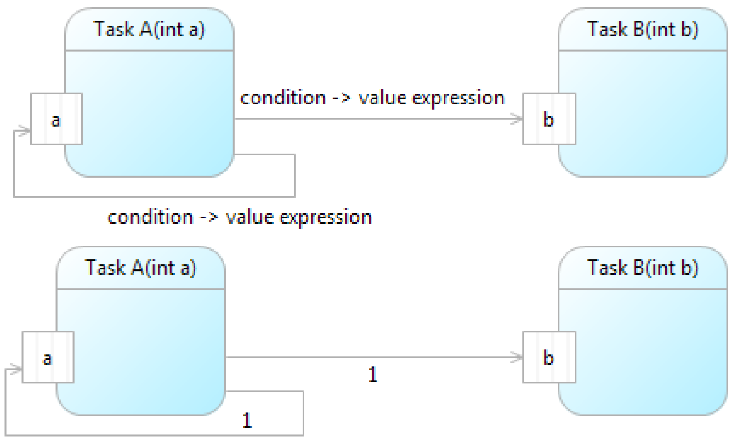


Figure 6.6: Affect of abstraction on value expression and condition, on the edges

6.2 Parameter Abstraction

The parameter (data) abstraction from the DSEIR model can be done in different ways. We have identified two possible ways in which this can be done, namely, using simulation and syntax analysis. The basic function of both methods is to identify the range of values a particular parameter can take and to record it. In the following sections we discuss these possible solutions in detail along with their advantages and disadvantages.

6.2.1 Simulation

One of the solutions to create an abstract model is by simulation. In this approach, we run the DSEIR model and record all the data through this process. Using this recorded data the possible values for each and every parameter in the DSEIR model can be obtained. The data is recorded in a trace file. We then parse this trace file to get the range of parameter values. The range is used to create a parameter map. This parameter map is used in further abstraction procedure steps. Although this is an easy way to obtain the parameter values, it has the following disadvantages:

- Is time consuming when models become large.

- Considers only one execution sequence at a time, due to which it is not exhaustive.

Although we do-not directly use this method for the purpose of abstraction, we use it as a verification method to check the results obtained from the syntax analysis method of data abstraction; this is explained in Chapter 8.

6.2.2 Syntax analysis

The other possible solution for parameter abstraction is syntax analysis. In this approach we syntactically analyze the DSEIR model parsing the expressions and conditions across the tasks, ports and edges to obtain, the range of values that the parameters in the input DSEIR model can take. This approach is better than simulation since it is not as time consuming and it also exhaustively goes through the model until a fixpoint is reached for the parameter values. Although syntax analysis imposes some restrictions on the DSEIR model in order to be feasible to apply the algorithm. These restrictions are based on the restrictions on the DSEIR translation to Uppaal. The trade-off here is that this method is typically is pessimistic and conservative. The algorithm that performs abstraction through syntax analysis and the restrictions involved are explained in detail in the following section.

6.2.3 Algorithm

In this section we explain the parameter abstraction algorithm in detail. The input model for the abstractor algorithm is the DSEIR model. The algorithm considers the task-flow perspective, load perspective and mapping perspective. The output of this algorithm is a mapping of parameters to their respective range, which is specified by its minimum and maximum values.

Input:

The input of the algorithm is a user defined model in DSEIR with restrictions. The restrictions in the translation to Uppaal discussed in Section 4.1.4 are also considered here. The restrictions are listed below:

- The number of ports should be equal to the number of parameters, with each of those ports having their binding expression as one of those parameters. The task cannot have extra ports as in the actual DSEIR construct.
- The local declarations, start and end-statements are not considered.
- There should be one task that has initial values set for all of its ports.

- The conditional operators are now limited to $\{<, >, \leq, \geq, =\}$.
- The maximum number of iterations should be known.
- Resource allocation does not depend on deadline values.

Prior to describing the algorithm, we introduce the generic DSEIR model used as input here, which is a subset of the actual DSEIR model. Note: That for one of the requirements like there should be one task that has initial values for all of its ports, in case of more tasks with initial values, the task with maximum outgoing edges and has at least one self loop is chosen. The task-flow perspective of this model, with the imposed restrictions is shown in Figure 6.7. The load and the mapping perspectives are shown in Figure 6.8. The description of the generic task model can be found in Chapter 4. The symbols and notations used here are derived from the previous description. The change caused due to the restriction affects only the task-flow perspective; the restriction "number of ports = number of parameters" allows "n" ports for A and "m" ports for B.

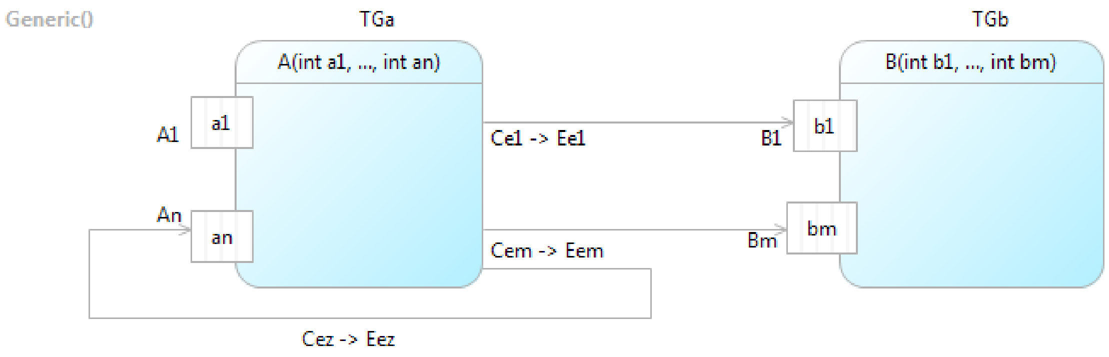


Figure 6.7: Generic input model(Task-flow perspective)

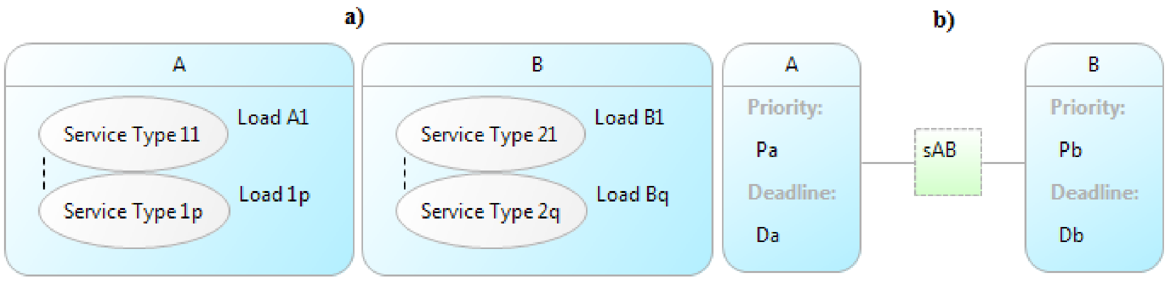


Figure 6.8: Load perspective of the generic input model

We use an algorithm to perform the data abstraction, where we obtain the range of values for the parameters. The following algorithm performs this task:

```
EXTRACTDATA(DSEIR_Model DSEIR, Integer maxIterations)
 1 Parameter↦List<Integer> parameterMap
 2 Integer Current_parameter_min, Current_parameter_max
 3 Integer negativeInfinity ← −999999
 4 Integer positiveInfinity ← 999999
 5 Set<Task> setOfTasks ← GETTASKS(dseir)
 6 Task task
 7 Parameter parameter
 8 for (Every task ∈ setOfTasks)
 9 do Set<Parameter> parameterSet ← GETPARAMETERS(task)
10     for (Every parameter ∈ parameterSet)
11     do Port p ← FINDPORT(parameter)
12         Set<Integer> initial_values ← INITIALVALUE(p)
13         if (initial_values ≠ ∅ )
14            then Current_parameter_min ← min(initial_values)
15                 Current_parameter_max ← max(initial_values)
16            else
17                 Current_parameter_min ← negativeInfinity
18                 Current_parameter_max ← positiveInfinity
19                 end if
20         parameter ↦ [Current_parameter_min, Current_parameter_max]
21         end for
22     end for
23 //The parameterMap for the generic model can be seen in Table 6.2
24 Task initTask ← FINDINITIALTASK(dseir)
25 Parameter↦List<Integer> parameterMap(previous), parameterMap(current)
26 parameterMap(current) ← parameterMap
27 Integer i
28 while ((parameterMap(previous) ≠ parameterMap(current)) ∧ i < maxIterations)
29 do List<Task> taskList
30     taskList ← ADDTOLIST(taskList, initTask)
31     parameterMap(previous) ← parameterMap(current)
32     parameterMap(current) ←
33           FINDMINMAX(parameterMap(current), init_task, taskList)
34     i ← i + 1
35     end while
36 Output parameterMap(current)
```

Note: We have also used several functions through this algorithm which are explained briefly but not defined, since it is not important. Those functions are listed below:

Table 6.2: The parameter table

Parameter	Min-Max Value
a_1	$[a_1 min, a_1 max]$
..	..
a_n	$[a_n min, a_n max]$
b_1	$[b_1 min, b_1 max]$
..	..
b_m	$[b_m min, b_m max]$

- **getTasks**: Input: *DSEIR model*, Output: All the *tasks* present in the model in the form of a **set**.
- **getParameter**: Input: *task*, Output: All the *parameters* of the task.
- **findPort**: Input: *parameter*, Output: The *port* in which the *parameter* forms the binding expression.
- **initialValue**: Input: *port*, Output: The *initial values* of the port in the form of a set.
- **findInitialTask**: Input:*DSEIR model*, Output: The *initial task* of the task-set, which is always first in the execution sequence.
- **addToList**: Input: *List* and the *element* to be added, Output: The input *List* with the added *element*.

The algorithm/pseudo-code **extractData** is the main algorithm that takes DSEIR model and maximum possible iterations as input and provides the data parameter values (range) as output. It has 2 integer constants *negativeInfinity* and *positiveInfinity* which specifies extreme values a parameter could attain. The next step is to check for each parameter in each task for the existence of initial values on the port they reside (A1 to An w.r.t the Figure 6.7). Then a mapping of parameter to their intimal minimum and maximum values (constructing a range) is created. Its declaration is shown in line 1 of the algorithm where parameter maps to a list of integer (with 2 values). The parameters which reside on ports with initial values set have proper minimum and maximum values, the rest have *negativeInfinity* and *positiveInfinity* as minimum and maximum, respectively. This parameter map is shown in Table 6.2. Each parameter is mapped to its possible range specified as a list with 2 values minimum and maximum. Then we start from initial task of the task-set and traverse through the entire task-flow perspective of the DSEIR model to obtain the minimum and maximum values. The while loop stops when 2 consecutive iterations end up having same parameter map (indicating that a fixed point is reached) or when *maxIterations* is reached.

The notion of maxIterations is used to here to terminate the algorithm in cases where it goes on infinitely. The algorithm can run infinitely in conditions where the model has a loop in which the edges do not have any guard conditions. An example model that could run infinitely is shown in Figure 6.9. In this example the edge from task D to task E and task E to D does not have any condition, hence the algorithm does not terminate on this loop, this is one case where maxIterations comes into play. Although, the value of maxIterations needs to be an input for this algorithm, the method to determine this is unknown yet and is predicted to be dependent on the model.

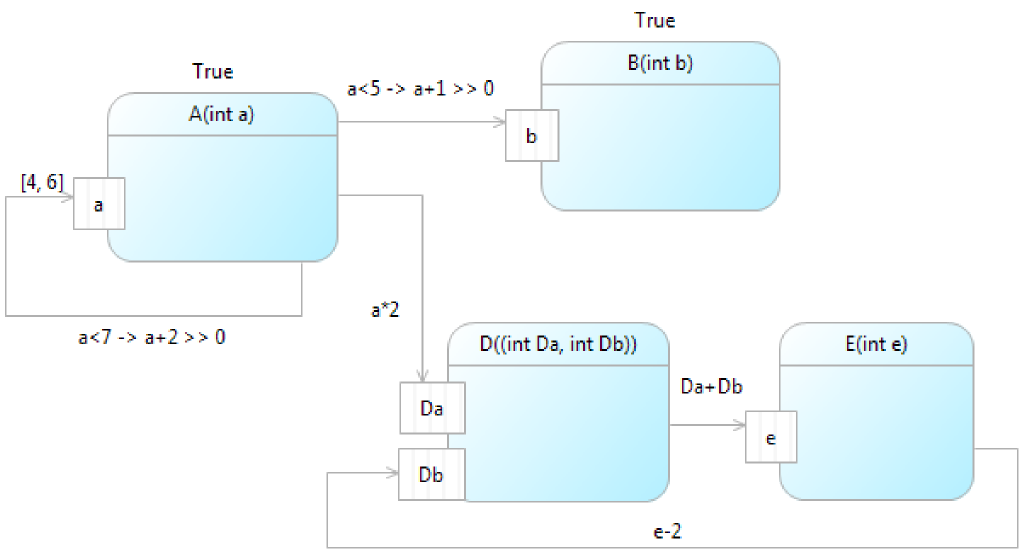


Figure 6.9: An example model where *maxIterations* is required to terminate

This algorithm calls another function **findMinMax** which performs the traversing procedure through the application, is defined as:

```
FINDMINMAX(Parameter ↦ List<Integer> parameterMap,Task task,List<Task> taskList)
 1  List<Task> processedTaskList;
 2  Parameter↦List<Integer> backupMap
 3  if (HASOUTGOINGEDGES(task))
 4    then return parameterMap
 5         end if
 6  Set<Edge> outGoingEdges ← GETEDGESFROM(task)
 7  Edge edge
 8  for (Every edge ∈ outGoingEdges)
 9  do backupMap ← parameterMap
10      Task taskNew ← DESTINATIONTASK(task, edge)
11      //Over the edge
12      Condition Ce ← CONDITIONOF(edge)
13      //"Ce" is the condition on the edge considered can be seen in Figure 6.7.
14      List<Expression> expressionList ← GUARDEVALUATOR(Ce, parameterMap);
```

```
//Guard Evaluvator is a helper function and is further explained in 6.4
Parameter $\mapsto$ List<Integer> *bufferParameterMap*
*bufferParameterMap* $\leftarrow$ SOLVE(*expressionList*, *parameterMap*)
//Solve is a helper function and is further explained in 6.4
*expressionList* $\leftarrow$ CLEAR(*expressionList*)
Expression $Ee$ $\leftarrow$ VALUEEXPRESSION(*edge*)
List<Expression> *minMaxList*
*minMaxList* $\leftarrow$ MINMAXEXPEVALUAVATOR($Ee$, *bufferParameterMap*)
//MinMaxExpEvaluvator is a helper function here and is
further explained in 6.4
*expressionList* $\leftarrow$ ADDLIST(*expressionList*, *minMaxList*)
//At the port
Port $p$ $\leftarrow$ DESTINATIONPORT(*task*, *edge*)
Parameter $Pa$ $\leftarrow$ PARAMETERINPORT($p$)
// Pa $\in \{a1 \cdots an, b1 \cdots bm\}$
*expressionList* $\leftarrow$ ADDTOLIST(*expressionList*, $Pa$)
*parameterMap* $\leftarrow$ SOLVE(*expressionList*, *parameterMap*)
if (processedTaskList has taskNew)
  then continue //skips the new edge
  else
      *processedTaskList* $\leftarrow$ ADDTOLIST(*processedTaskList*, *initTask*)
      if (CHECK($TGp$, *parameterMap*))
        then *parameterMap* $\leftarrow$
                FINDMINMAX(*parameterMap*, *taskNew*, *taskList*);
        else
            *parameterMap* $\leftarrow$
                FINDMINMAX(*backupMap*, *taskNew*, *taskList*);
            end if
      end if
end for
```

The simple function calls used in this algorithm/pseudo-code are listed below:

- **clear**: Input: *List*, Output: Given the input *list* clears the elements of the list and outputs an empty list.
- **addToList**: Similar to the one explained previously for the first algorithm.
- **getEdgesFrom**: Input: *task*, Output: The outgoing edges of the *task*.
- **destinationTask**: Input: *task* and one of its *edge*'s, Output: The *task* at the destination of the considered edge.

- **conditionOf**: Input: *edge*, Output: The *condition* (conditional expression) over the edge.
- **valueExpression**: Input: *edge*, Output: The *value expression* over the edge.
- **destinationPort**: Input: *task* and one of its *edge*'s, Output: The *port* at the destination of the edge.
- **parameterInPort**: Input: *port*, Output: The *parameter* present as the binding expression in the port.

The **findMinMax** algorithm traverses through the entire task-flow perspective starting from the initial task and covering all the tasks, edges to syntactically analyze the expressions and conditions that lie on them. This is a recursive algorithm which calls itself until tasks with no outgoing edges are reached as the stopping criteria. Until the stopping criterion is reached, starting from the initial task each edge is traversed. This traversal includes checking the effect of the guard condition on the parameter (if any) in the conditional (Guard) expression. This is done using the helper functions **GuardEvaluator** and **Solve**. A guard can restrict the value of a parameter range, for example if a = [4,5] and the guard condition is a $<$ 5 then the temporary range is stored in the *bufferParameterMap* will be a = [4,4]. The *bufferParameterMap* is a temporary map variable which is used in only in calculations of a particular edge. This temporary map would be used in the calculation of the minimum and maximum of the value expression (Ee) over the edge.

The helper function **MinMaxExpEvaluator** is used for this purpose, resulting in the minimum and maximum value for the input expression. We perform the binding to the parameter on the destination port of the considered edge; the range of the value expression (Ee) is used for this purpose. New values for the parameter are obtained via **Solve** helper function. The task guard TGp (p $\in \{A, B\}$ task-set) is checked using the new *parameterMap* values via an extra function **Check** which if satisfied, the modified *parameterMap* will be used, otherwise the *backupMap*, backup parameter map is used. A *processedTaskList* is maintained which stores the tasks that are processed already to prevent repetitions of the procedure on the same task. After an edge is processed we go to the destination task of the edge on a recursive call, which is processed next. The output of this step consisting of the algorithm is an updated parameter map shown in Table 6.2 with all parameters mapped to their minimum and maximum value that they can take during the execution of the input model.

6.3 Model Transformation

Once we obtain the minimum and maximum values of the parameters in the first step through the parameter abstraction, we then use these values in the next step of abstraction to translate the DSEIR model into an abstract model. During this translation, we use the obtained parameter map to find the range of load and deadline for each task, since the load and deadline values are dependent on their respective task parameters, the priority values are also affected. The effect on priority is that the function **MinMaxExpEvaluator** is applied on to it and the maximum value of the range obtained is used as the final priority value. They are specified as mathematical expressions, hence we need to find the minimum and maximum for the load and deadline expression. This can be done using the helper function **MinMaxExpEvaluator**. The input for this function is the load/deadline expression and the parameter map obtained in the first step, output is a range specified using 2 values (minimum and maximum). The resulting values from the helper function are then specified as an unknown distribution in the abstract model ranging from the minimum to the maximum value. Note that, the min and max stored in this load expression inside the unknown Distribution are not always the min and max of the load expression but depends on the parameters if any present in the expression. With this change in the load, deadline expressions and elimination of condition on edges and equating value expression as constant integer 1 on all the edges, we obtain the abstract DSEIR model. The final mapping perspective can be seen in Figure 6.11. The final task-flow perspective of the model is depicted in Figure 6.10.

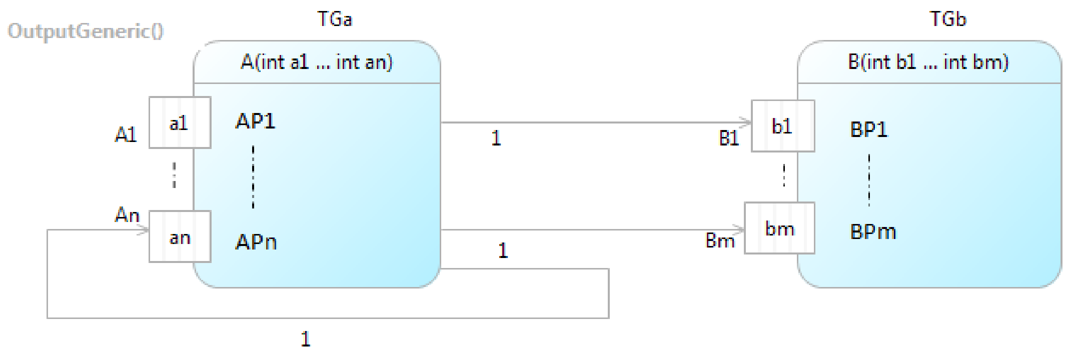


Figure 6.10: Task-flow perspective of the output

6.4 Necessary Functions

The algorithm requires some helper functions for its execution. In this section we explain these necessary helper functions in detail.

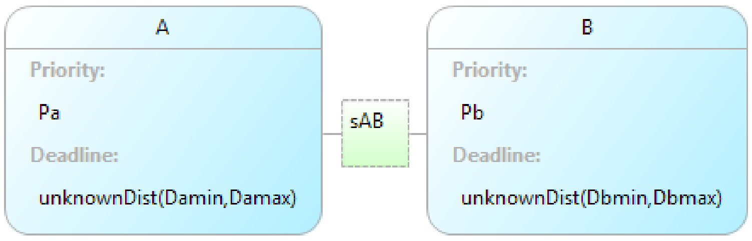


Figure 6.11: Mapping perspective of the output

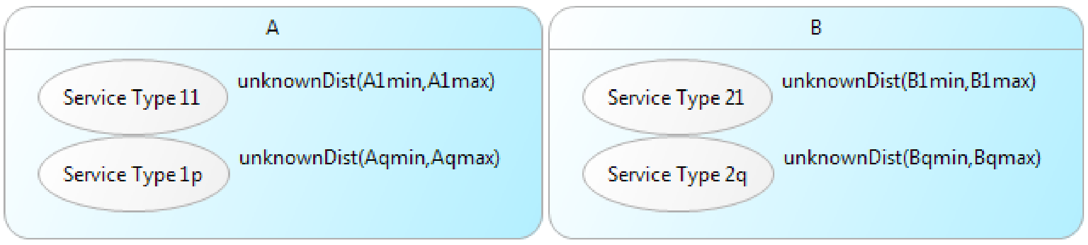


Figure 6.12: Load perspective of the output

Equation Solver

The function **solve** referred to previously in the abstractor algorithm is an equation solver. Given an equation and the parameter table it solves the equation, for one parameter at a time until all parameters are solved. For example given an equation a + 2 = 10, then it solves for “a” as “a = [10,10] - [2,2] = [8,8]”. This new obtained range is compared with the old range to obtain a new range. This is done using the **rangeCalculations** function defined in 6.4.

Range Calculations

In the main algorithm we convert the data parameters to a range of values specified through the minimum and the maximum of the range. Through the algorithm execution, the range of the parameters changes, i.e., it is replaced with a new range if found. This is called inside the *solve* function where the parameters are given values, then range calculations are done to obtain actual new values from possible values and old values (of the parameter). It is different for the different type of expressions (conditional expressions and normal mathematical expressions). The algorithm of the range calculations function is shown.

```
CALCULATERANGE($Old\_min, Old\_max, New\_min, New\_max$)
1  Result_min = Old_min
2  Result_max = Old_max
```

```
if (Normal Expression)
  then //Min part
       if (Old_min == negativeInfinity)
         then
              Result_min = New_min

         else if (New_min < Old_min)
                then
                     Result_min = New_min
                     end if
              end if
       // Max Part
       if (Old_max == positiveInfinity)
         then
              Result_max = New_max

         else if (New_max > Old_max)
                then
                     Result_max = New_max
                     end if
              end if

  else if (Conditional Expression)
         then //When the ranges are disjoint
              if ((New_min > Old_max) || (New_max < Old_min))
                then
                     return [Result_min and Result_max]
                     end if
              //Min Part
              if (New_min == negativeInfinity)
                then
                     Result_min = Old_min

                else if New_min > Old_min
                       then
                            Result_min = Old_min
                            end if
                     end if
              //Max Part
              if (Old_max == positiveInfinity)
                then
                     Result_max = New_max

```

```
                    else  if New_max > Old_max
                             then
                                    Result_max = New_max
                                    end if
                          end if
                 end if
        end if
return [Result_min and Result_max];
```

An example of range calculations: old list [3,5] new list [2,5] - resulting list [2,5].

Guard Evaluator

This section explains how the guards are evaluated to solve for the parameters in the guard, in order to obtain the range to which the parameters are reduced by the guard conditions. This function takes a conditional expression (guard) and the parameter map as input and produces a new temporary parameter map as output. For example, Guard evaluations are done as:

- (Mathematical Expression logicalOperator constant). E.g. $(a + b) < \text{Constant}$
- $a = \text{newConstant dualof(operator) } y$
  - Here newConstant depends on the logical operator, E.g. for $a+1 < 5$ newConstant will be $a = 4 - 1$ or in $a + 1 > 5$ then $a = 6 - 1$.
  - For the set that we consider $\{<, >, >=, <=, =\}$.
    - $<$, newConstant = constant $-$ 1;
    - $>$, newConstant = constant + 1;
    - $>=$,$<=$, $=$, newConstant = constant
- Dualof(operator) are defined for mathematical operators +,- (dual-set),/,*(dual-set)
- Example: $a + [5,6] < 10$
  - $a = [9,9] - [5,6]$ ([9,9] is obtained via GuardEvaluator)
  - $a = [3,4]$ (done via MinMaxExprEvaluator) This is the new possible temporary range of a. In the actual calculations this will be compared with the existing range of A, consider it to be $a = [2,6]$ the resulting range will be [2,4]

values above 4 clipped because of the guard condition.

Expression Evaluator

This helper function evaluates a given expression provided that the parameter ranges for the parameters in the expression are provided, gives the minimum and maximum possible values for a particular expression. It parses and solves both mathematical expressions and logical expressions. For logical expressions it uses a special 3-valued logic with values true, false and unknown. More information about this function is provided in [14]. For example, consider an expression (a + b) with a = [2,4] and b =[3,6] , The range of the expression is [5,10].

6.5 Abstraction Example

In this section we show the application of the abstraction algorithm to an example DSEIR model. The task-flow perspective of this DSEIR model can be seen in Figure 6.13. The load perspective of the DSEIR model can be seen in Figure 6.15. We perform

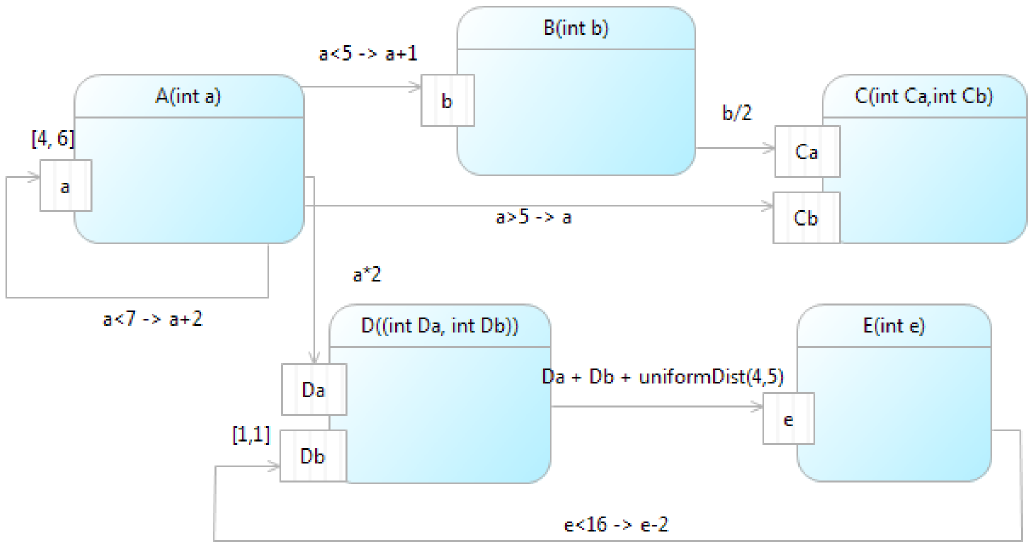


Figure 6.13: Task-flow perspective of the example problem

data abstraction on this model using the abstraction algorithm, to first extract the value of the parameters. This is achieved in the following steps:

- The range of the parameters is calculated based on the initial values on the port

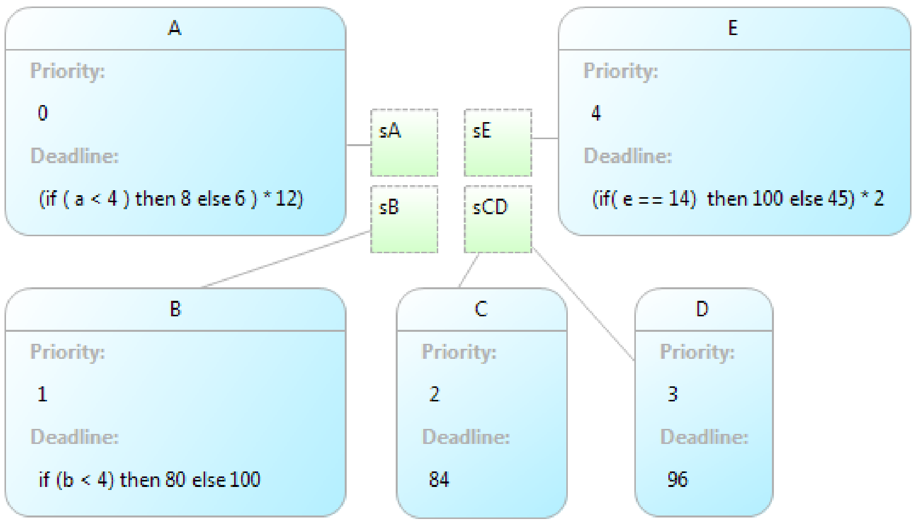


Figure 6.14: Mapping perspective of the example.

of the tasks. Note that initial value [4,6] on the port means two tokens and not range as we will use the same notation for a range as well.

- A mapping of the parameter to their respective ranges is created and can be seen in the form of a table in Table 6.3.

Table 6.3: Initial parameter values

Parameter	Min-Max Value
a	[4,6]
b	[negativeInfinity,positiveInfinity]
Ca	[negativeInfinity,positiveInfinity]
Cb	[negativeInfinity,positiveInfinity]
Da	[negativeInfinity,positiveInfinity]
Db	[1,1]
e	[negativeInfinity,positiveInfinity]

- Task to edge mapping is created, to be used in the function
- **getEdgesFrom** in *task*.**findMinMax**.
- Initial task of the model is obtained using the funtion **findInitialTask**. The result obtained is task A in this case.
- We iterate over the all the tasks until a fixed point is found for the parameter values or the maximum number of iterations are passed.

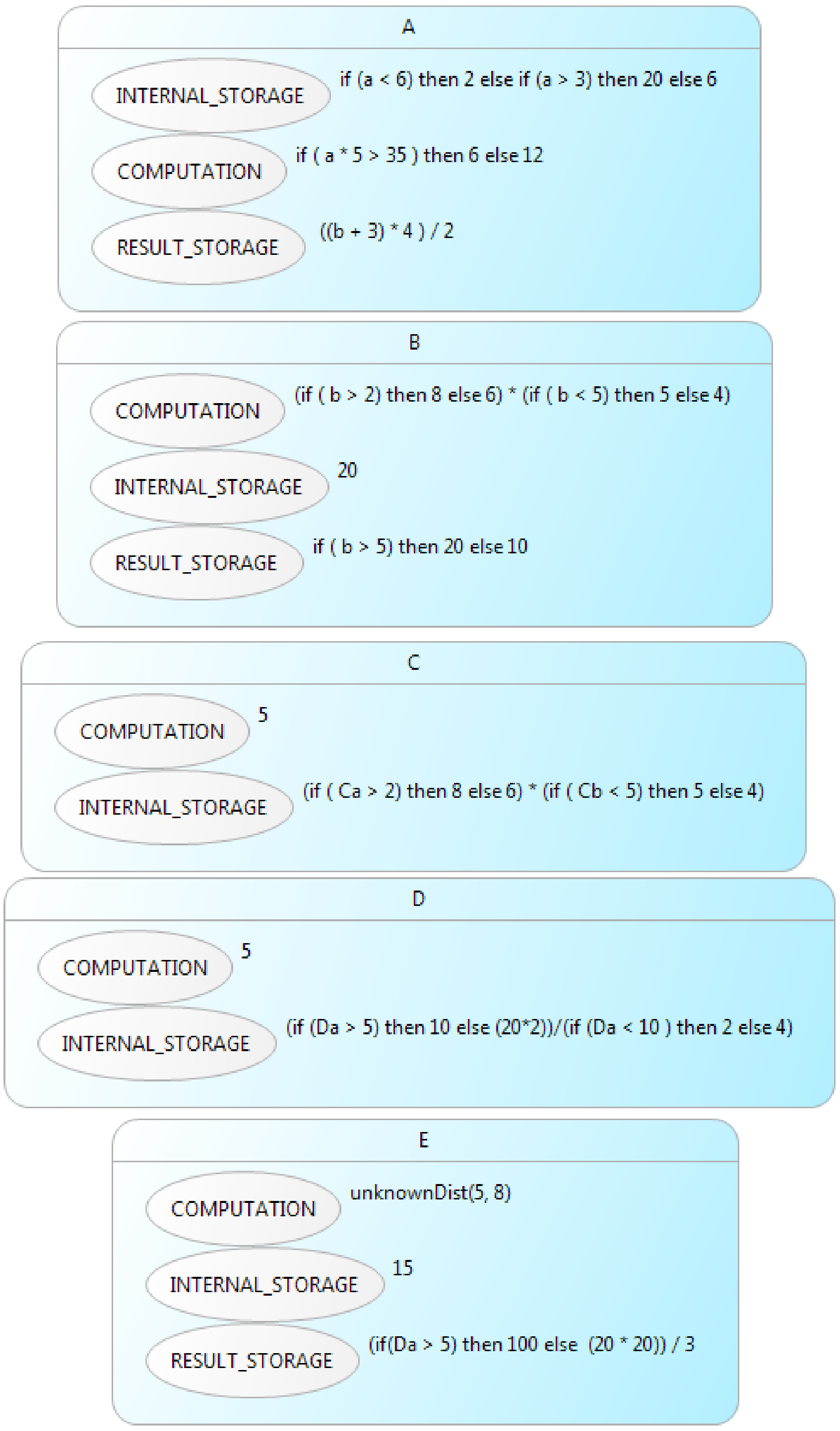

A
INTERNAL_STORAGE
if (a < 6) then 2 else if (a > 3) then 20 else 6
COMPUTATION
if (a * 5 > 35) then 6 else 12
RESULT_STORAGE
((b + 3) * 4) / 2
B
COMPUTATION
(if (b > 2) then 8 else 6) * (if (b < 5) then 5 else 4)
INTERNAL_STORAGE
20
RESULT_STORAGE
if (b > 5) then 20 else 10
C
COMPUTATION
5
INTERNAL_STORAGE
(if (Ca > 2) then 8 else 6) * (if (Cb < 5) then 5 else 4)
D
COMPUTATION
5
INTERNAL_STORAGE
(if (Da > 5) then 10 else (20*2))/(if (Da < 10) then 2 else 4)
E
COMPUTATION
unknownDist(5, 8)
INTERNAL_STORAGE
15
RESULT_STORAGE
(if(Da > 5) then 100 else (20 * 20)) / 3

Figure 6.15: Load perspective of the example.

Table 6.4: Task-edge map

Task	Edges
A	aToA, aToB, aToC, aToD
B	bToC
C	-
D	dToE
E	eToD

- The iteration includes calculating *parameterMap(current)* using **findMinMax**. Figure 6.16 shows the execution of the algorithm, for one iteration. There are total 7 edges in the model. The algorithm starts from A (initial task) and goes through these edges iteratively starting with the edge to B. For parameter "a" of task A and "Db" of task D the ranges (using initial values) are set to [4, 6] and [1, 1] respectively as shown in Table 6.3. We use these values and the rest from the table in the **findMinMax** function. When guards are present on the edges, parameters present in the guard conditions obtain temporary values, which is used in the value expression of that edge. We explicitly specify in the explanation when temporary values are used.

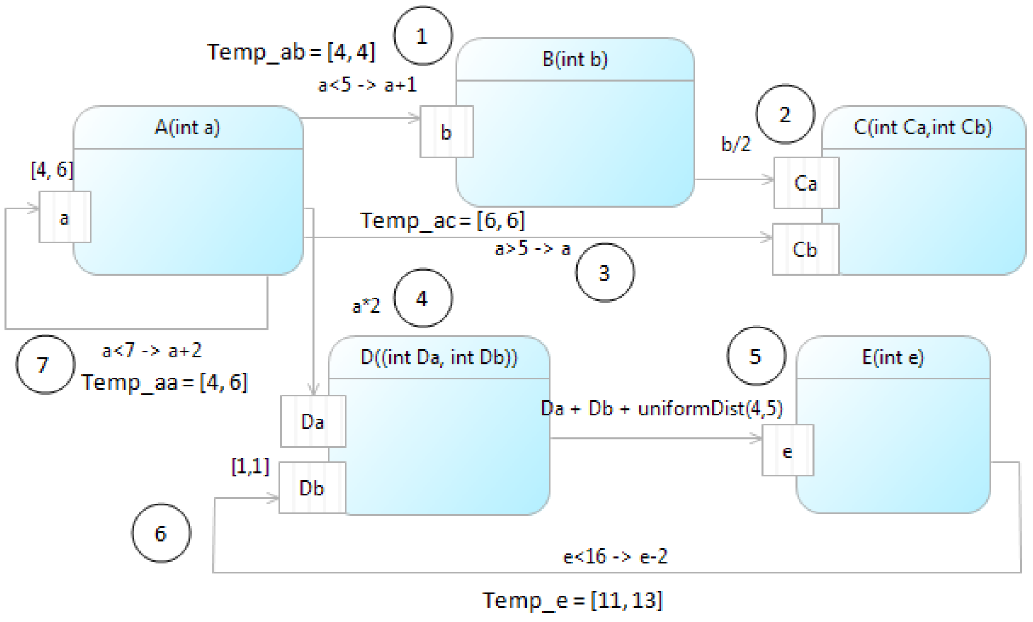


Figure 6.16: One iteration of the parameter abstraction algorithm on the working example

We explain each of the steps in the iteration, below:

– 1) aToB: old_b = [negativeInfinity,positiveInfinity]

b = temp_ab + 1
new_b = [5,5]

– 2) bToC: old_Ca = [negativeInfinity,positiveInfinity]
Ca = [5,5] / 2
new_Ca =[2,2]

– 3) aToC: old_Cb = [negativeInfinity,positiveInfinity]
Cb = temp_ac
new_Cb = [6,6]

– 4) aToD: old_Da = [negativeInfinity,positiveInfinity]
Da = [4,6] * 2
new_Da = [8,12]

– 5) dToE: old_e = [negativeInfinity,positiveInfinity]
e = [8,12] + [1,1] + [4,5]
new_e = [13,18]

– 6) eToD: old_Db = [1,1]
Db = temp_e - 2
new_Db = [1,13]

– 7) aToA: old_a = [4,6]
a = temp_aa + 2
new_a = [4,8]

After these 7 steps the cycle is repeated again with the new values becoming old values to the next iteration, also temporary values are refreshed depending on the new ranges. The process is continued until two iterations have same parameter values.

- After the parameter map is obtained we apply the model transformation as explained previously, to obtain new task-flow perspective with conditions on edges eliminated and their value expressions set to constant 1. The output task-flow perspective diagram obtained through the model translation can be seen in Figure 6.17.

- During the model translation, we also calculate the load of each task on each service type they require, using the parameter map obtained during data abstraction. The final load diagram of this example model is shown in Figure 6.18, in this figure every load value is displayed as an unknown distribution in the range. The mapping perspective is shown in Figure 6.19, with deadline values changed to the range specified via an unknown distribution similar to the load values. There is no effect on priority since it is a constant.

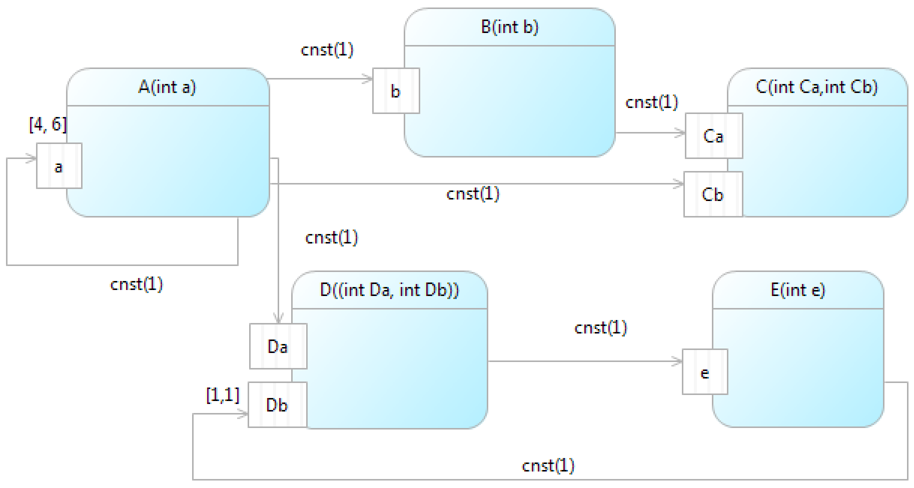


Figure 6.17: Task-flow perspective of the output model

Table 6.5: Performance Comparison

Feature	Original model	Abstract model
States	28,842	1057
Time	2.22s	0.25s
Memory	21MB	14MB

Results of Abstraction

In this section we discuss the performance gain in abstracting the original model. We manually performed some performance comparisons using the Uppaal tool, similar to the one in Chapter 5. The resulting abstract model shows reduction in *state-space* and better performance when compared to the original model. The result is backed by recorded statistics that can be seen in table 6.5. The correctness of this parameter abstraction method is also verified and applied on the running example later in Chapter 8.

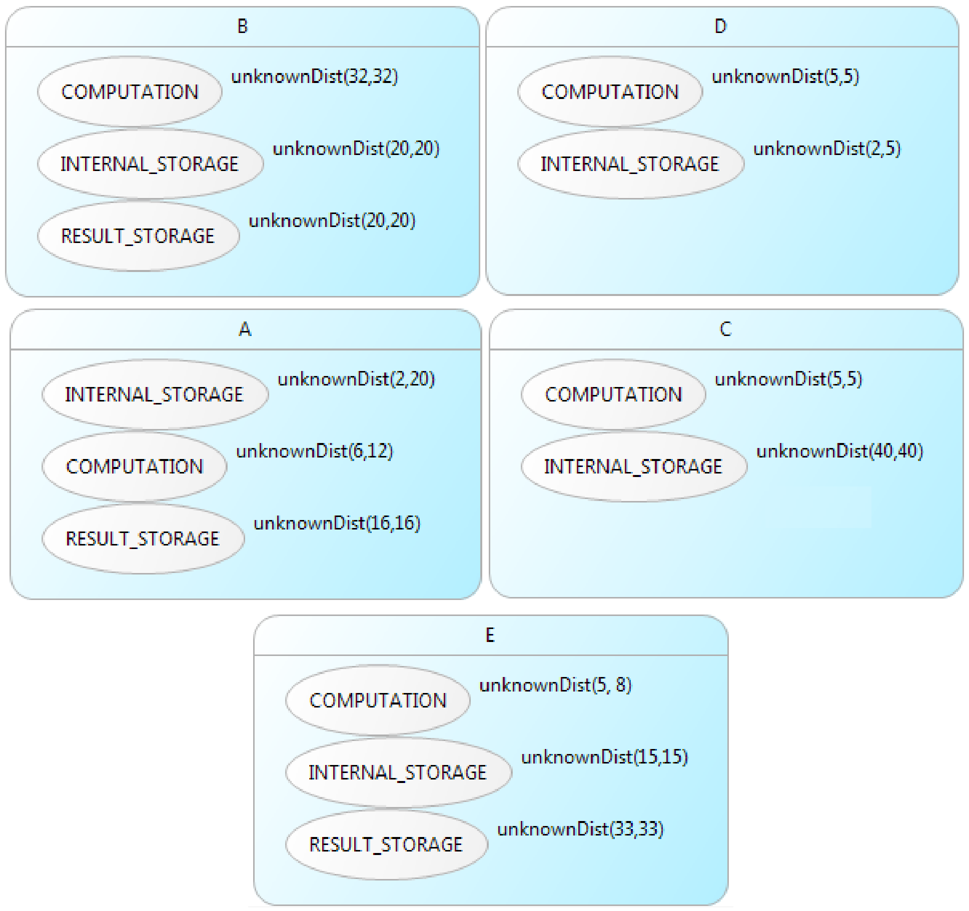

B
COMPUTATION unknownDist(32,32)
INTERNAL_STORAGE unknownDist(20,20)
RESULT_STORAGE unknownDist(20,20)
D
COMPUTATION unknownDist(5,5)
INTERNAL_STORAGE unknownDist(2,5)
A
INTERNAL_STORAGE unknownDist(2,20)
COMPUTATION unknownDist(6,12)
RESULT_STORAGE unknownDist(16,16)
C
COMPUTATION unknownDist(5,5)
INTERNAL_STORAGE unknownDist(40,40)
E
COMPUTATION unknownDist(5, 8)
INTERNAL_STORAGE unknownDist(15,15)
RESULT_STORAGE unknownDist(33,33)

Figure 6.18: Load perspective of the output model

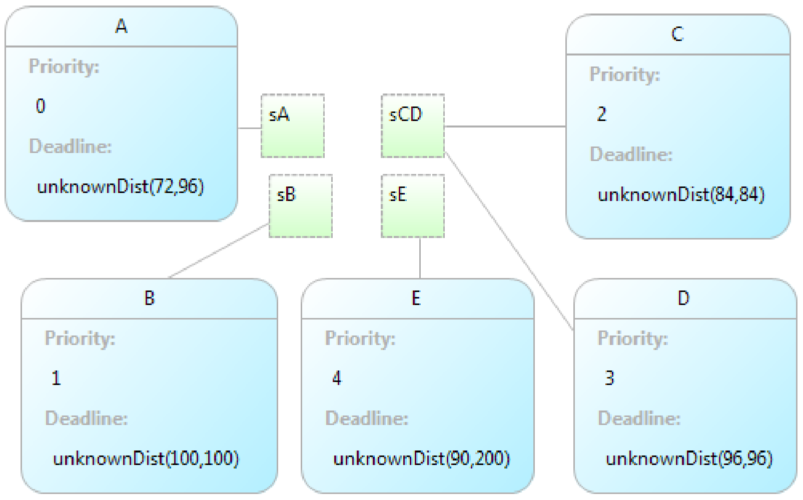


Figure 6.19: Mapping perspective of the output model

Chapter 7

Verification

The new model obtained from the abstraction process is an abstraction of the original model. This new model is an over-approximation that can contain behavior not existing in the original system. This makes the current model checking scheme inapplicable to this scenario producing false negatives ("not-schedulable") for schedulable systems, because of non-existing behavior. Thus we make an attempt to eliminate such false negatives, and consider only those results that could help in making the design decisions for the Octopus toolset.

In this chapter we discuss a revised scheme for model checking in Section 7.1. We apply this revised model checking scheme to the running example along with the abstraction procedure to perform schedulability analysis on the running example and is explained in Section 7.2.

7.1 Revised Model Checking Scheme

We investigate possible approaches to overcome this problem. In the proposed solution we apply new model checking formulas to get more accurate results considering the over-approximation. These formulas are different from the ones used in Chapter 4. Given the current output model (abstract model), the load and deadline are specified as independent unknown distributions of values in a given range. Considering all the combinations of load and deadline values is tedious and time consuming. Hence we decided to consider some specific cases here, which are enough to capture possible schedulability information, remaining cases which could result in false negatives are omitted. These cases are listed below;

- In the first case we check the model with respect to the combination of minimum

Table 7.1: Model checking results

Case No		[*maxDuration()*, *minDuration()*]	Result
1	*maxDeadline*	No	Not-Schedulable
2	*maxDeadline*	Yes	Non-Conclusive
3	*minDeadline*	Yes	Schedulable
4	*minDeadline*	No	Non-Conclusive

possible deadline (minDeadline) and range of execution time. The formula used for this is: A[] (A.minDeadline) imply not(A.Stop).

- In the second case we check the model with respect to the combination of maximum possible deadline and range of execution time. The formula used for this is: A[] (A.maxDeadline) imply not(A.Stop).

Note: Although we mention the load and deadline combinations, we actually consider the execution time in the verification process. As explained earlier, in DSEIR the execution time is defined as the product of load and processing time of the service type it executes on. Since the processing time is a constant and load is the varying term, we here mention load in our explanation.

We use the above 4 cases in combination of 2 parts, where one component is constant in both, thus obtaining 4 cases, to form an analysis of the system in terms of schedulability. The 4 different cases are shown in Table 7.1. In the table, *maxDeadline* and *minDeadline* refer to maximum and minimum possible deadline respectively and *maxDuration()* and *minDuration()* refer to maximum and minimum possible duration of execution respectively. We discuss these 4 cases resulting in 3 possibilities shown below:

- *Schedulable*: This result suggests that the system is schedulable for all possible combinations of load and deadline, since the range of execution satisfy the minimum deadline here (The minimum possibility for the all the instances of the task). This is an ideal case that rarely happens.

- *Not schedulable*: This result suggests that for the current setting the system is not schedulable. This case is resulted when one of the execution values in the range does not meet the maximum deadline, which is the maximum the task can have for all its instances. This possibility could still result in false negatives which cannot be prevented. This is the trade of we make to obtain reduced *state-space*. This case can be further refined by considering only *minDuration()* or *maxDuration()* along with *maxDeadline* (2 queries) resulting in an answer “No” to declare the system as non-schedulable. This would not have any false negatives since both these values are always part of the original system.

- *Non-conclusive*: This result suggests that the verification results for the system considered is *non-conclusive*, since we cannot confirm either that it is schedulable or not. If the results of these conditions were to be considered either false positives or false negatives could be the result. Two cases result in this value and are listed below:

- When the range of execution is able to meet the maximum deadline. Given the consideration of maximum value for deadline in combination with minimum duration (is included in the range) we can neither conclude the system is "schedulable" nor as "not schedulable" hence we describe it as a *non-conclusive* case.

- When the range of execution is not able to meet the minimum deadline. This is similar to the previous case in a different way since we cannot expect all the execution duration values to have the minimum deadline as their deadline and hence we can only say it is *non-conclusive* case.

We particularly focus on the result that system is *non-conclusive*, which is obtained in two cases here because it is not a usual result. We do not consider model checking the cases which result in this possibility. Further in this section we describe this case justifying the reason to omit it. We obtain this result because the approximation also considers values that do not include in the original model and hence schedulability analysis can fail for those values not included in the actual system. Consider a system with three tasks A, B and C. The DSEIR model of such a system is shown in Figure 7.1. From this model we obtain an abstracted form. In Figure 7.2 we show only the load and mapping perspective of the abstract model, since we concentrate on the execution time and deadline only. Execution time is the combination of load and processing time and since processing time is a constant namely 1, it is entirely dependent on the load in this example.

When this model is translated into the Uppaal syntax and verified, we obtain a *non-conclusive* result for the case number 4 for schedulability check from the table 7.1. This along with the option to generate the trace allows us to see where the system fails. In the original model, task A has two possible load values 2, 4 and the deadline ranges from 3 to 9. B and C have constant load values of 5 and 6 since their minimum and maximum values are equal. Given the processing time as constant 1, the load values can be considered as the execution time. Considering the application of case number 4 (or 3) we consider minimum deadline of 3 for A and maximum duration of execution time 4, which is originally impractical to consider and declare not schedulable considering the results. Thus we omit these false negatives by declaring the system as *non-conclusive.*

With the new model checking scheme defined we verify the running example after abstraction procedure is applied on it, explained in Section 7.2.

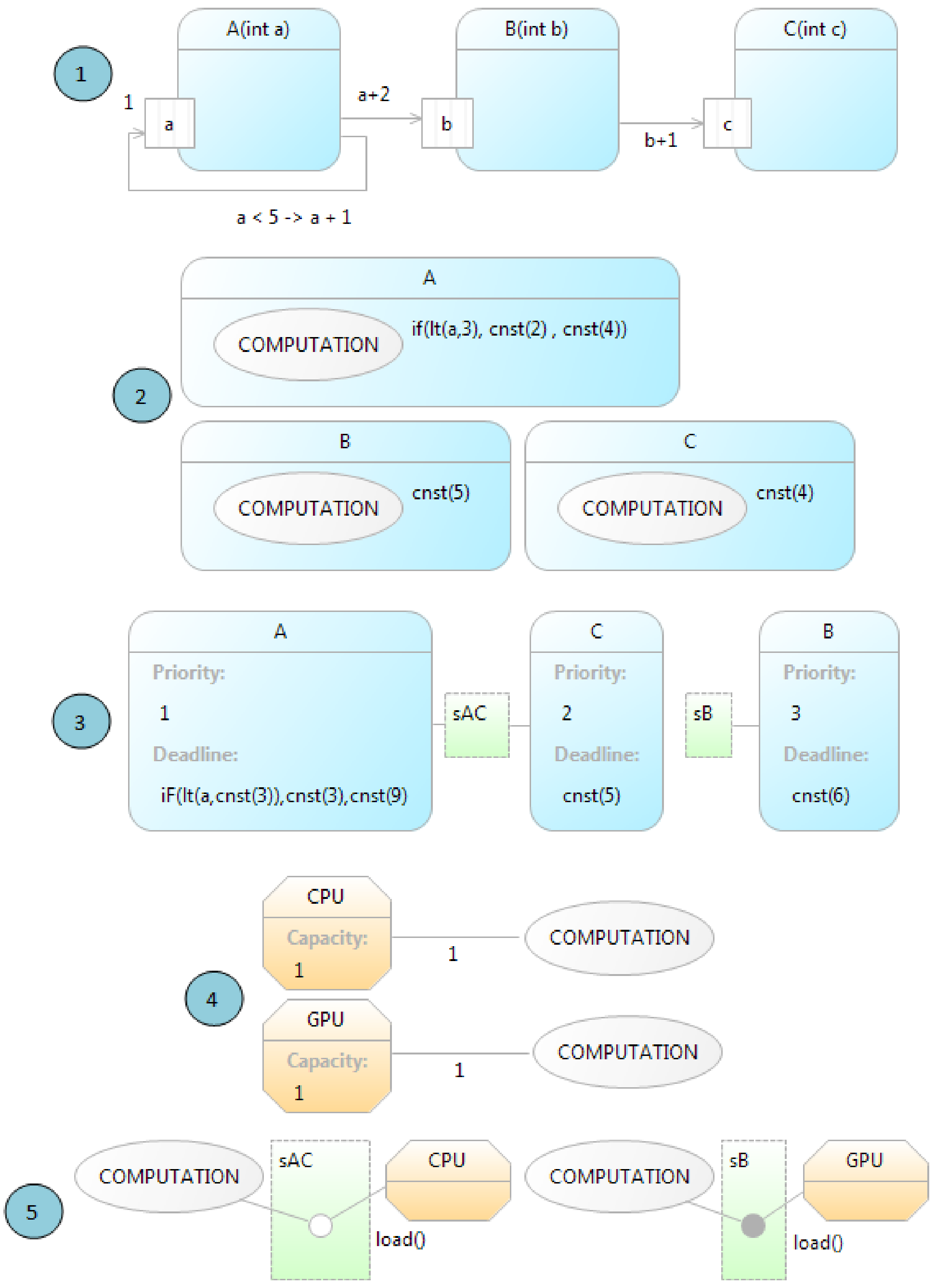


Figure 7.1: The perspective diagrams of the example model. 1) Task-flow perspective 2) Load perspective 3) Mapping perspective 4) Resource perspective 5) Scheduling perspective

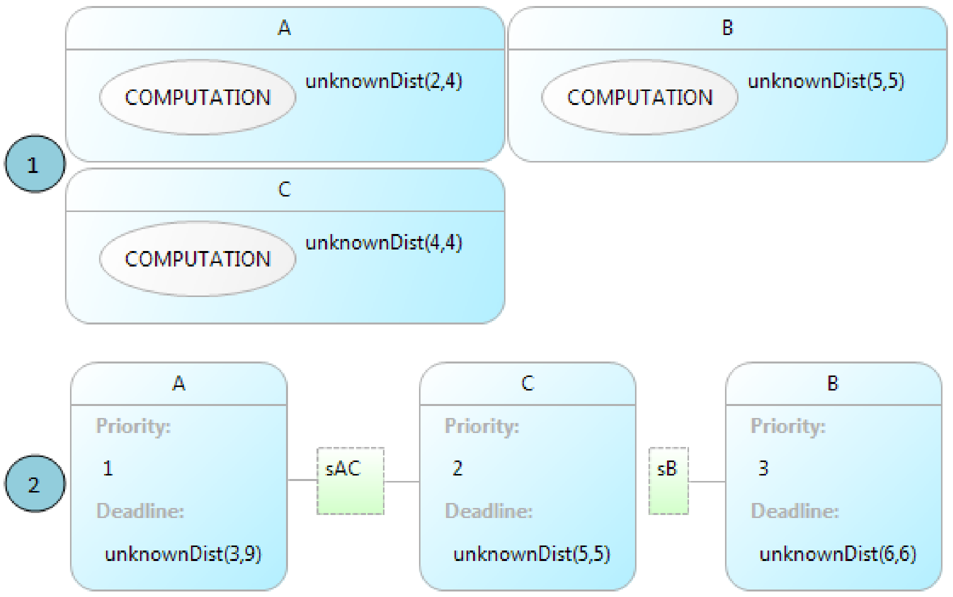


Figure 7.2: The final load and mapping perspectives

7.2 Abstraction and Verification of Running Example

The resulting final task-flow perspective will be as shown in Figure 7.3. Resulting load perspective will be as shown in Figure 7.4. Also the mapping perspective is depicted in 7.5. The figures are shown just to depict the effect of abstraction on the running example, with eliminated condition and value expressions over the edge, modified load and deadline values. Further in this section we also apply the new model checking schemes to verify the running example for schedulability property.

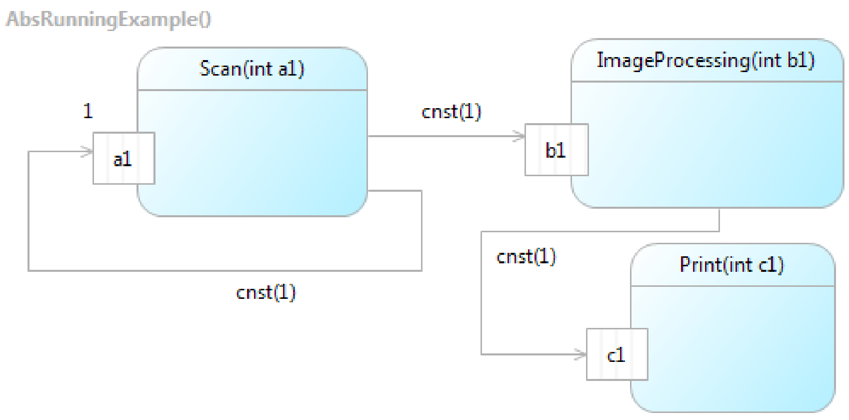


Figure 7.3: Task-flow perspective of the output model

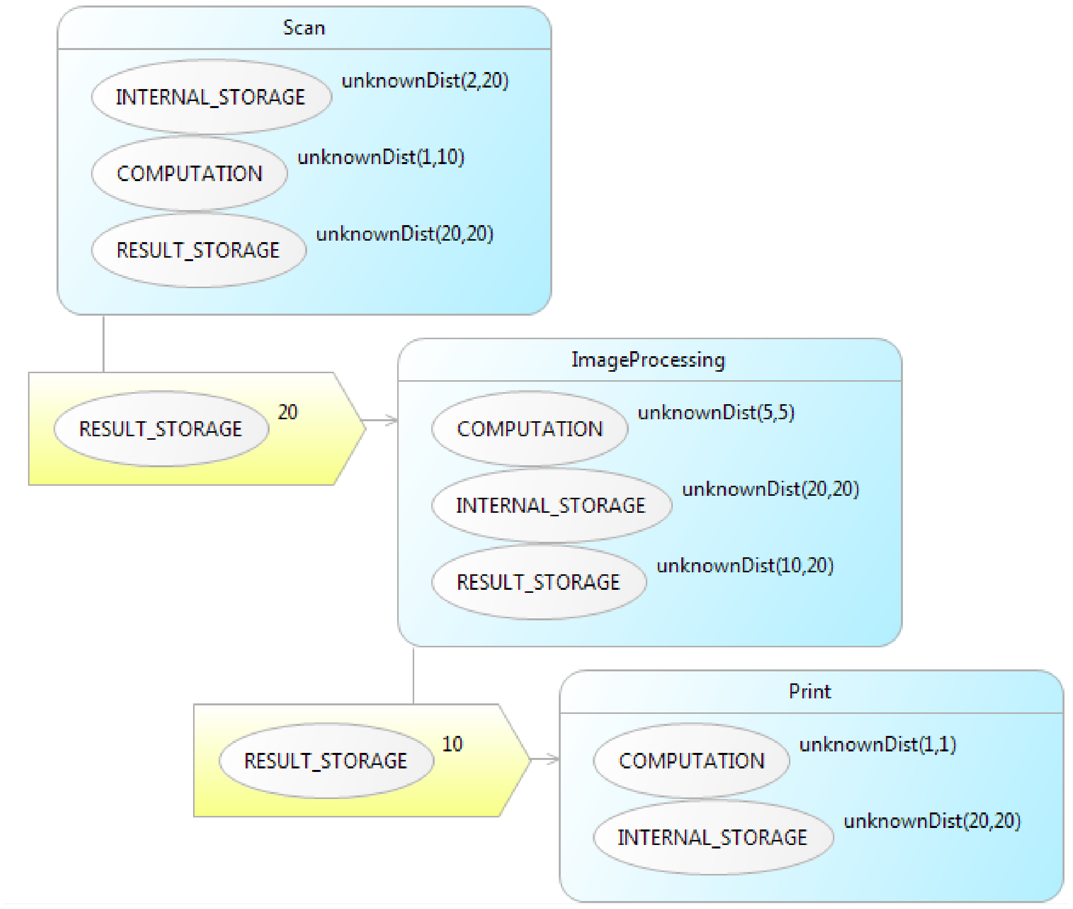


Figure 7.4: Load perspective of the output model

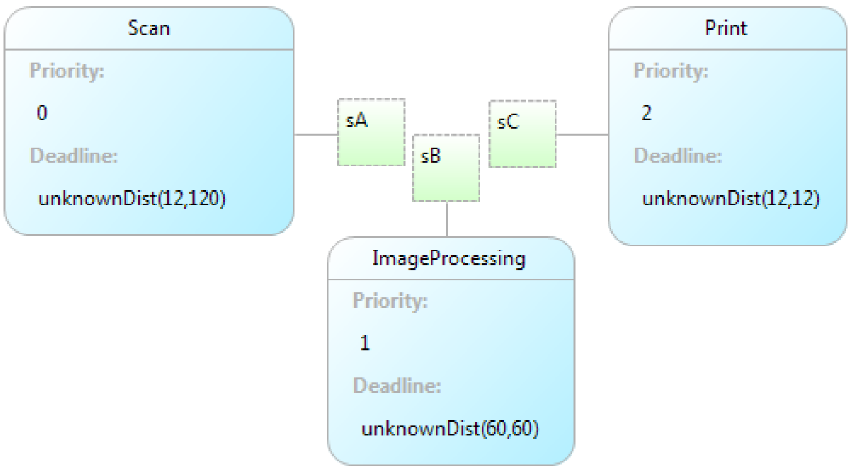
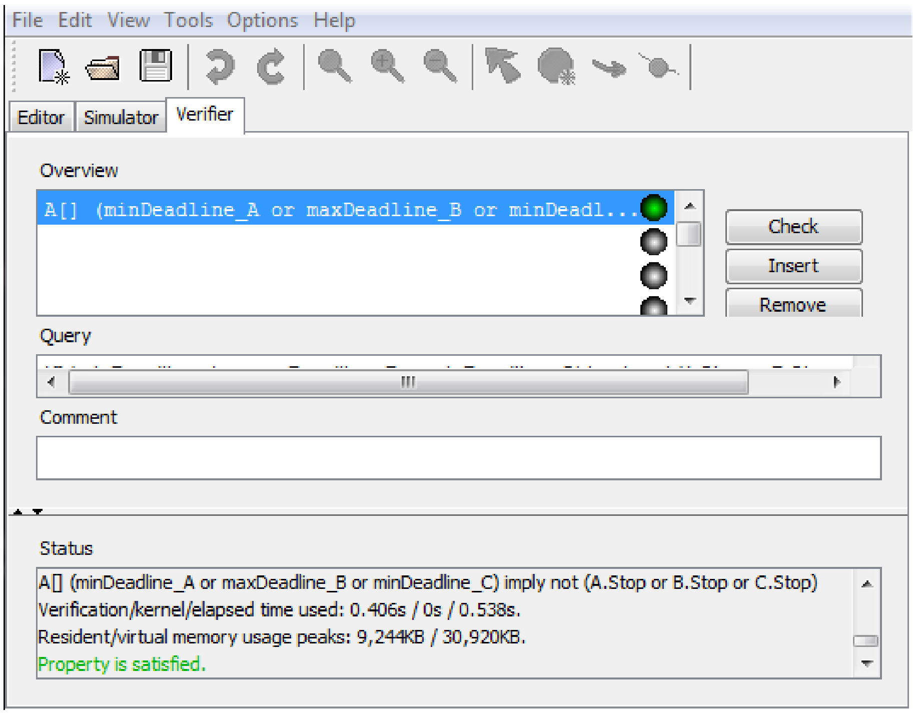


Figure 7.5: Mapping perspective of the output model



```
A[] (minDeadline_A or maxDeadline_B or minDeadl...
```

```
A[] (minDeadline_A or maxDeadline_B or minDeadline_C) imply not (A.Stop or B.Stop or C.Stop)
Verification/kernel/elapsed time used: 0.406s / 0s / 0.538s.
Resident/virtual memory usage peaks: 9,244KB / 30,920KB.
Property is satisfied.
```

Figure 7.6: Model checking results for the running example

After the abstraction procedure applied on the running example, to complete the schedulability analysis on this model, we verify it using the new model checking scheme. The result obtained is as shown in Figure 7.6 shows that this system is schedulable since it satisfies case number 3, here A corresponds to scan, B to image processing and C to print.

Chapter 8

Abstraction Fitness

In the view of this project, this chapter explains a process to assist the users in measuring the effectiveness of the proposed solutions. In this thesis we have proposed an algorithm to abstract the DSEIR model in order to reduce the *state-space*. We have performed data abstraction on the DSEIR models to achieve this. Thus now we intend to provide a method to measure the fitness of the results obtained through the abstraction algorithm which follows syntax analysis method for performing abstraction. The chapter is mainly concerned with the parameter abstraction part discussed in Section 6.2. The model transformation part is not treated, since it meagerly performs transformation of the model based on range of the parameters obtained via data abstraction. The method in which we measure the fitness/precision is discussed in Section 8.2, in abstract terms we perform a comparison of parameter values obtained using other known methods versus the parameter abstraction method. Since we deal with parameter values, we can use simulation to obtain the parameter values of the original model. We explain this simulation based comparison technique in the following section.

8.1 Simulation Based Comparison

By simulating DSEIR models we can obtain the minimum and maximum values each parameter can take in the model. This process is applied on the DSEIR models automatically via simulator in the Octopus toolset that simulates and provides us an execution trace. This execution trace is parsed and the parameter values are obtained. The workflow of this comparison task can be seen in Figure 8.1. A DSEIR model taken as input for two methods namely , parameter abstraction and simulation which result in a parameter map. This parameter map is compared for abstraction fitness and further use of parameter abstraction method on a considered model is dependent on the result of this measurement. This precision measure is a relative measure, measured using the results

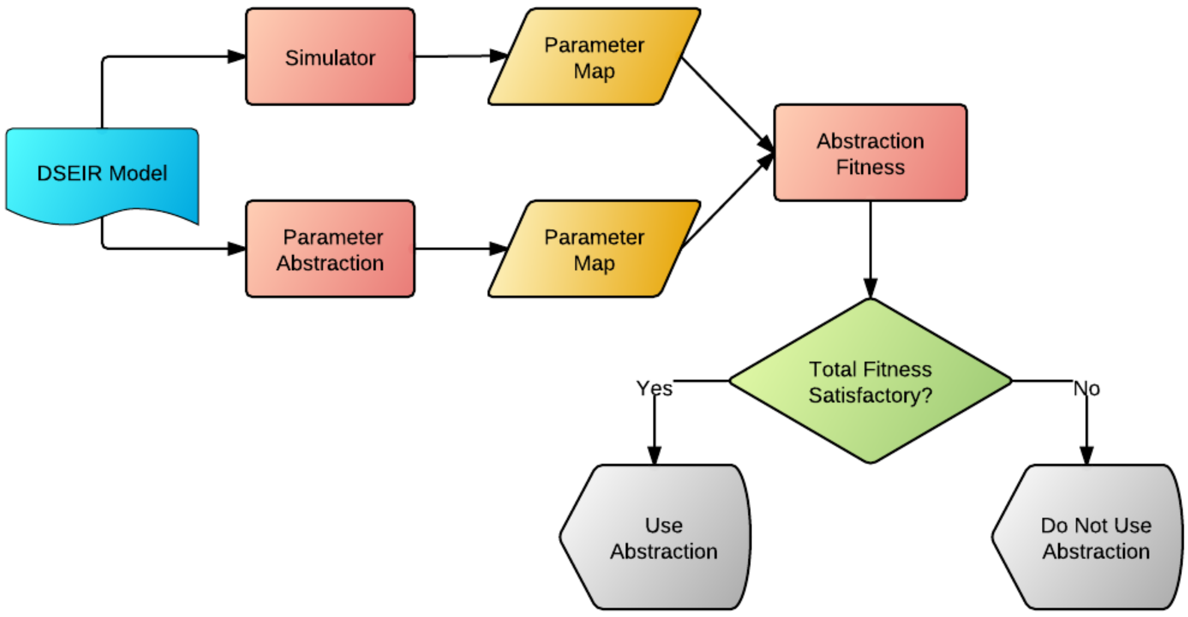


Figure 8.1: Work-flow showing the use of abstraction fitness

of simulation. We illustrate this by treating the running example.

The task-flow perspective, load perspective and the mapping perspective diagrams of the running example model can be seen in Section 2. We show the results from the application of data abstraction using both approaches, i.e., the simulation method and the syntax analysis method.

8.1.1 Syntax Analysis

In this section, we show the results of the application of the abstraction procedure on the running example discussed in Section 2.2. The parameter map obtained for the model using the syntax analysis method for abstraction is shown in the Table 8.1.

Table 8.1: Parameter Map obtained for the Running example via syntax analysis.

Task	Parameter	Min-Max Value
Scan	pageSize[0]	[3,5]
Scan	pageSize[1]	[1,3]
Image Processing	pageSize[0]	[3,5]
Image Processing	pageSize[1]	[1,3]
Print	pageSize	[3,5]

Table 8.2: Parameter Map obtained for the Running example via simulation

Task	Parameter	Min-Max Value
Scan	pageSize[0]	[3,5]
Scan	pageSize[1]	[1,2]
Image Processing	pageSize[0]	[3,5]
Image Processing	pageSize[1]	[1,3]
Print	pageSize	[3,5]

8.2 Results

We validate the performed data abstraction by comparing the parameter values obtained in syntax analysis method with the parameter values obtained from simulation as shown in the simulation work-flow in Figure 8.1. The obtained values can be seen in Table 8.2. The goal of this method is to provide a measure of fitness of the parameter abstraction method to the user. By construction of the algorithm, we know that the abstraction is conservative for the data parameter values. Thus being conservative we have:

- P_O set of parameters in the original model and P_A set of parameters in the abstracted model
- $\forall p \in P_O$ we have corresponding $p_a \in P_A$
- Also if $p = \{min_O \ldots max_O\}$ and $p_a = \{min_A \ldots max_A\}$ then

  $min_A \leq min_O$ & $max_A \geq max_O$.

  Thus $p \subseteq p_a$

We can evidently see that here the parameter abstraction procedure is conservative for the running example, from the data presented in Table 8.1 and Table 8.2. The method applied for abstraction performs over-approximation, we can have parameter values that are not in the set of original parameter values, hence being conservative is not always useful to the user. In cases where the results are conservative but the difference between the actual data and obtained data is large, the fitness is reduced like the example depicted in Figure 8.2, alternatively a useful case can be seen in Figure 8.3. Figure 8.4 shows the actual data range and the data range obtained from the abstraction procedure.

The variables *diff_min* specifies the difference between the minimum(*lower_bound*) of these two ranges and *diff_max* specifies the difference between the maximum(*upper_bound*) of the two ranges. The idea is to keep this difference minimum. The fitness(F) of the parameter abstraction method can be measured either using these variables or using the length of abstracted data and actual data. This measures the precision in which the

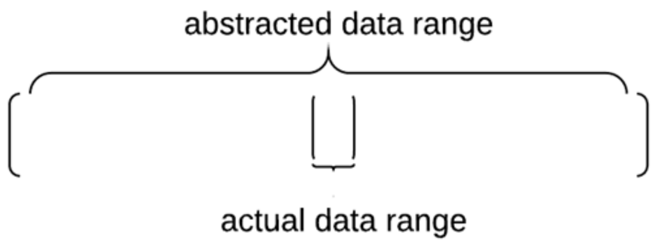


Figure 8.2: The ranges obtained with large difference

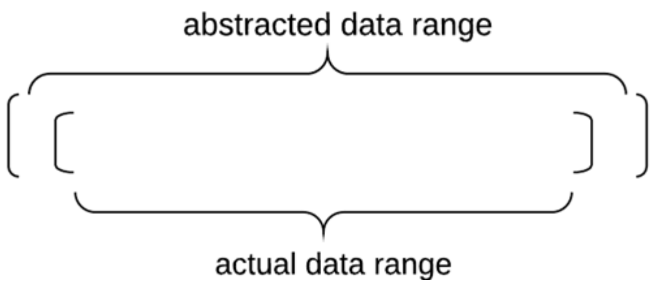


Figure 8.3: The ranges obtained with small difference

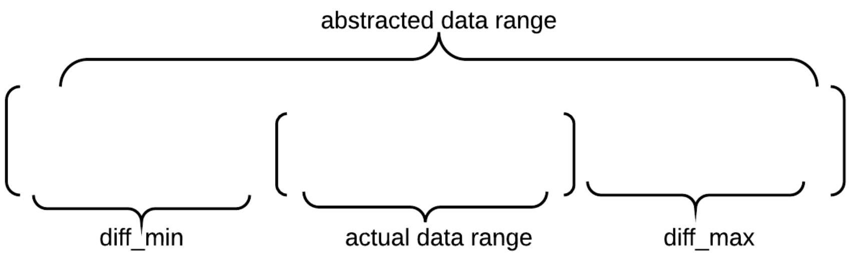


Figure 8.4: The ranges obtained using the different methods, the actual data range - simulation method and abstracted data range - parameter abstraction method, with differences depicted

parameter abstraction method reproduces the original values. We display fitness value (F) for a single parameter range mathematically as:

$$abstracted\_length = upper\_bound(abstracted\ data\ range) - lower\_bound(abstracted\ data\ range)$$

$$actual\_length = upper\_bound(actual\ data\ range) - lower\_bound(actual\ data\ range)$$

Assuming that no other factors affect the fitness, we can say:

$$F = \frac{actual\_length}{abstracted\_length}$$

We then define total fitness(F_T) of a task model with “n” parameters, as:

$$F_T = \frac{\sum_{i=1}^{n} F_i}{n}$$

Table 8.3: Fitness values of the parameter

Task	Parameter	abstracted_length	actual_length	Fitness
Scan	pageSize[0]	3	3	1
Scan	pageSize[1]	3	2	0.67
Image Processing	pageSize[0]	3	3	1
Image Processing	pageSize[1]	3	3	1
Print	pageSize	3	3	1

For the current example we write this difference and measure the total fitness. First we tabulate the fitness of each parameter in Table 8.3. Looking at the table for the example with 5 tasks (n=5) we have;

$$F_T = \frac{4.67}{5} = 0.934$$

We can also specify it in terms of accuracy as 93.4% accurate. Note that, for a parameter type of array, each array element is considered a separate parameter for calculation. We conclude that higher the value of the total fitness, more effective is the abstraction procedure on the considered model, for ideal case where the abstracted range and actual range are equal we obtain result as 1. Since the parameter abstraction is conservative, abstracted range can never be smaller than the actual range hence we can say that each $F_i \in [0,1]$, for $i = 1 \cdots n$ and “n” being the number of parameters, hence also $F_T \in [0,1]$ for any considered model.

Chapter 9

Conclusions

The primary goal of this thesis was to incorporate schedulability analysis to the Octopus toolset. During the design phase we found that the DSEIR models are tuned to be simulation models and hence are less applicable for exhaustive analysis approaches since it causes extensive delay or crashing of the analysis tool. Hence we planned to achieve this goal into two phases, namely to make schedulability analysis *possible* in the Octopus toolset, and then to make it *scalable* for arbitrary DSEIR models, applicable to exhaustive analysis approaches.

The approach in the first phase included making some design decisions to add *deadline* as a modeling primitive to the DSEIR language, also making it usable by the scheduler. After adding this feature, we investigated the possibility of re-using the existing infrastructure in the Octopus toolset to build schedulability analysis. We extend the existing DSEIR translation to Uppaal to incorporate deadline detection mechanism to support schedulability analysis. Hence now schedulability analysis is *possible* in the Octopus toolset. We have also identified an obstacle for our method, in the existing translation to Uppaal syntax during this thesis, where a state "ready" was not precisely defined, and rectified this problem. Note that this is an obstacle to perform schedulability analysis and not a problem otherwise. We have then compared this schedulability framework built in DSEIR against the schedulability analysis framework proposed by Alexandre et al. [13] for performance measures. Although for smaller set of tasks (5 tasks), there was no phenomenal difference in the time, memory usage and *state-space*, for greater task-sets (10 and 12 tasks) the automatic model (DSEIR to Uppaal converted models) had better performance, proving that these models do not suffer from performance degradation due to the automation process.

To make schedulability analysis *scalable* to DSEIR models with focus on the reduction of the *state-space*, we required to perform data abstraction on the model. We considered two approaches, namely, simulation and syntax analysis to make this possible. Owing to the limitations of simulation approach we chose the syntax analysis method and designed an algorithm to perform abstraction using syntax analysis method. This algorithm performs data abstraction and model conversion on input DSEIR models, resulting in an abstract model with reduced *state-space*. The abstraction procedure commonly creates an over-approximation of the input model, along with specifying load and deadline values as ranges. The model checking scheme gets affected, by this over-approximation producing false negatives and by the need to check all combinations of load and deadline values during verification. We solve this by successfully creating a revised model checking scheme, with an attempt to eliminate false negatives and reducing the *state-space* required to be searched by checking appropriate combinations of load and deadline values, enough to conclude with proper results. Yet some false negatives could be produced, which can be eliminated by considering the second refinement. Some queries that could result in non-conclusive or false negative results are omitted. This scalability feature is applicable only on a restricted subset of DSEIR owing to the limitations of the parameter abstraction algorithm and the DSEIR translation to Uppaal.

By construction the parameter abstraction procedure is conservative, but being conservative does not always imply usefulness/fitness. Thus we also provide a method to measure the fitness of the parameter abstraction algorithm by comparing parameter values with the one's obtained by simulating the input model. We also define the total fitness of the model, which is equal to 1 in an ideal case when the parameter ranges obtained using parameter abstraction and simulation are equal, otherwise lies in the range [0,1].

Chapter 10

Future Scope

Through this project phase, considering the work done we also describe some recommendations to follow up on the current work:

- *Scale* the abstraction procedure: Currently the abstraction procedure can be applied on models based on a subset of the DSEIR syntax. Scaling this to the entire DSEIR syntax would make abstraction procedure applicable to arbitrary DSEIR models.

- *Improve* the abstraction procedure: Currently the abstraction procedure sometimes induces new values into the abstracted model, which is not present in the original model. By improving the abstraction procedure to capture only the concrete values we, could omit the *Non conclusive* case in the schedulability analysis results giving proper results. This will also increase the fitness of the procedure.

- *Periodic* Tasks: A suitable method, similar to the extra task for representing periodic tasks have to be investigated. The delay over the edges is also an option to represent periodicity.

- *Infinite* models: The effect of abstraction procedure on the models that require *maxIterations* to terminate have to be investigated further in detail.

- *Manual models v/s automatic models*: To investigate reasons for the difference in the expected results and the obtained results for this comparison is an area for future work.

- *Probabilistic schedulability analysis*: A computing systems always includes interrupts which is not part of a normal application and hence cannot be specified with the system. These appear as sporadic tasks appearing in between the execution of normal tasks and are of usually higher priority than other tasks in the system.

There is a need to consider these in the schedulability analysis, but being external to the system they are not explicitly specified. A probabilistic analysis for schedulability can be done based on their expected interference with the regular tasks.

References

[1] T. Basten, E. Van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. de Smet, L. Somers, E. Teeselink, N. Trcka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang, "Model-Driven Design-Space Exploration for Embedded Systems: The Octopus toolset," in *Leveraging Applications of Formal Methods, Verification, and Validation* (T. Margaria and B. Steffen, eds.), vol. 6415 of *Lecture Notes in Computer Science*, pp. 90–105, Springer Berlin / Heidelberg, 2010.

[2] A. Fredette and R. Cleaveland, "RTSL: A Language for Real-Time Schedulability Analysis," in *Real-Time Systems Symposium, 1993., Proceedings.*, pp. 274 –283, IEEE, dec 1993.

[3] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu, "Schedulability Analysis of Global Fixed-Priority or EDF Multiprocessor Scheduling with Symbolic Model-Checking," in *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pp. 556–560, IEEE Computer Society, 2008.

[4] "www.uppaal.org." Uppaal website.

[5] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, "A Methodology to Design Programmable Embedded Systems," in *Embedded Processor Design Challenges* (E. Deprettere, J. Teich, and S. Vassiliadis, eds.), vol. 2268 of *Lecture Notes in Computer Science*, pp. 321–324, Springer Berlin / Heidelberg, 2002.

[6] N. Trčka and B. in 't Groen, "DSEIR - The Modeling Language of Octopus." User manual, 2011.

[7] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems," 1996.

[8] R. Alur and D. Dill, "Automata for modeling real-time systems," in *Automata, Languages and Programming* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*, pp. 322–335, Springer Berlin / Heidelberg, 1990.

[9] G. Behrmann, A. David, and K. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems* (M. Bernardo and F. Corradini, eds.), vol. 3185 of *Lecture Notes in Computer Science*, pp. 33–35, Springer Berlin / Heidelberg, 2004.

[10] "http://www.comp.nus.edu.sg/ cs5270/notes/chapt6a.pdf." Coffee machine example.

[11] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium one*, pp. 414 –425, IEEE, 1990.

[12] J. Madsen, M. R. Hansen, and A. W. Brekling, *Model-Based Design for Embedded Systems*, ch. Modeling and Analysis Framework for Embedded Systems, pp. 121–141. CRC Press, 2010.

[13] A. David, J. Illum, K. G. Larsen, and A. Skou, *Model-Based Design for Embedded Systems*, ch. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pp. 93–119. CRC Press, 2010.

[14] A. Moily, "Model Transformations for Throughput Analysis using Synchronous Data Flow Graphs in Octopus," Master's thesis, Technical University of Eindhoven, August 2011.