Supporting design-space exploration with synchronous data flow graphs in the Octopus toolset

Moily, A.

Award date:
2011

Supporting Design-Space Exploration with Synchronous Data Flow Graphs in the Octopus Toolset

Ashwini Moily

August 23, 2011

Supporting Design-Space Exploration with Synchronous Data Flow Graphs in the Octopus Toolset

Ashwini Moily

August 23, 2011

Master Thesis

Eindhoven University of Technology
Department of Mathematics and Computer Science
Software Engineering and Technology Group
And
Manipal Institute of Technology
Department of Information and Communication Technology

In cooperation with:
Océ Technologies B.V
Venlo, The Netherlands

Embedded Systems Institute
Eindhoven, The Netherlands

Supervisor at TU/e and Océ:
Dr. L.J.A.M. (Lou) Somers
Associate Professor
lou.somers@oce.com

Supervisor at MIT:
Dr. Radhika M. Pai
Professor and Head of Department
radhika.pai@manipal.edu

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

External Supervisor:
Prof. dr. ir. Twan Basten
Professor
a.a.basten@tue.nl

Abstract

During the design of embedded systems, there are a lot of options that need to be considered to provide an optimal solution for the system. There are various software and hardware requirements that need to be satisfied. It is not feasible to manually search through all the design options to choose the best among them. Hence, a tool that performs Design Space Exploration (DSE) and provides an optimal solution could be of great added value. A toolset is being developed as part of the Octopus project run by Océ Technologies B.V., Embedded Systems Institute and several Dutch universities in close collaboration with each other.

This toolset developed for the Octopus project aims at providing support to model, analyze and select appropriate design alternatives during the early phases of system development. The toolset developed uses a DSE Intermediate Representation (DSEIR) modeling language which specifically supports the Y-chart approach, a popular approach used for the design of embedded systems. The architecture of the toolset allows for separation of concerns such as application, platform and the mapping between them.

In the domain of printers, there is, for example, a trade- off between the number of pages printed per minute (throughput) that can be guaranteed in the worst case and the peak memory that is required in the worst case. In the development of the Octopus toolset, the trade off analysis based on worst case system behaviour is done using a tool called SDF3 that analyzes Resource Aware Synchronous Data Flow Graphs(RASDFG) since it is very fast. The initial version of the toolset consisted translation of the application part of DSEIR to RASDFG only and it worked for generic applications implicitly assuming that the application was an RASDFG.

This assignment aims at extending the DSEIR2SDF translation such that it is capable of handling an RASDFG. This implies that the platform and mapping parts of DSEIR need to be covered by the translation. Furthermore, SDF3 assumes that the input provided is an RASDFG. But there is no mechanism to confirm this. This assignment implemented a mechanism to check if the input is an RASDFG and display appropriate error messages when it is not true. It also added methods to detect anomalous behaviour of models that could occur as a result of the translation.

Acknowledgements

This thesis would not have been possible without the support of my three supervisors. I owe a lot to my supervisor at Océ and TU/e, Dr. Lou Somers who guided and supported me throughout the project. He has constantly monitored my progress and has patiently reviewed several versions of my thesis. I would like to express my gratitude to Dr. Nikola Trčka, who has provided me with timely and constructive criticism and helped me shape my thesis into what it is right now. I have to mention Prof.dr.Twan Basten, who guided and motivated me to do better. He showed an interest in my work even though he was not required to and provided me with a lot of support every single time. I owe my deepest gratitude to you.

I am thankful to Dr. Radhika M. Pai for being my project coordinator from Manipal University and for her evaluation. I am grateful to Prof.dr.Mark van den Brand for his valuable guidance at every step of the way, without whom I would not have been doing my internship at Océ Technologies in the first place. I would like to thank all the people involved with the Octopus project at Océ and ESI, Frans Reckers, Klemens Schindler, Martijn Hendricks, Bram in't Groen, Jacques Verriet and all others who made my thesis, an unforgettable experience. I would also like to thank all the people involved with SDF3 tooling, which helped me start my project, with a special mention to Yang Yang and Sander Stuijk. I have to mention my friend Ajith Kumar, with whom I collaborated on various parts of the project. I owe my gratitude to you. This project would not have been possible without the support of Dr. Manohara M. Pai and Prof. dr. J.C.M Baeten, who gave me the opportunity to come to the Netherlands. Last but not the least; I must express my appreciation to my friends and family who stood by me through all the highs and lows of this project.

Contents

Chapter 1

Introduction

1.1 Context

In the modern world, the development of most of the embedded systems such as printers, wafer handlers, medical scanners etc. involves dealing with complex interaction between the hardware and software components. These components have stringent requirements in terms of cost, real time requirements and execution environments. Also, these systems are usually capable of parallel and distributed computation. Hence, the time and resources required for the development is increasing rapidly due to the increase in the complexity of these systems. This further emphasizes the need of developing a system that is cost-effective and error-free.

The challenge during the development of such systems is to consider all the available, sometimes conflicting design options and choosing the optimal solution among them. The design space often includes a wide range of metrics of interest such as timing, resource usage, energy usage, cost, etc. and multiple design options such as the number and type of processing cores, sizes and organization of memories, interconnect, scheduling policies, etc. Deciding to choose the best design alternative among all the available design alternatives by establishing a relation between them is often a tough decision.

Océ Technologies B.V, Embedded Systems Institute (ESI) and several Dutch academic research groups are working in close collaboration with each other in a research project called the Octopus project. This project developed a toolset that is capable of providing support to model, analyze and select appropriate design alternatives in the early phases of product development[1].

1.2 The Octopus toolset

The Octopus toolset provides a model-based framework that supports Design Space Exploration (DSE). Figure 1.1 shows the vision of Octopus as an appropriate modeling and design approach for DSE along with the benefits of the approach. The Octopus project attempts to realize this approach with the help of several modeling techniques by exploiting the benefits of each approach. This framework allows users to specify a formal model of the system without being experts in the domain of formal modeling by making the process highly automated. Its primary features are:

1. high-level modeling based on a clear separation of application, platform, and the application-to-platform mapping,

2. formal analysis of functional correctness and performance, and

3. (semi-)automatic exploration of alternatives and synthesis of optimized designs.

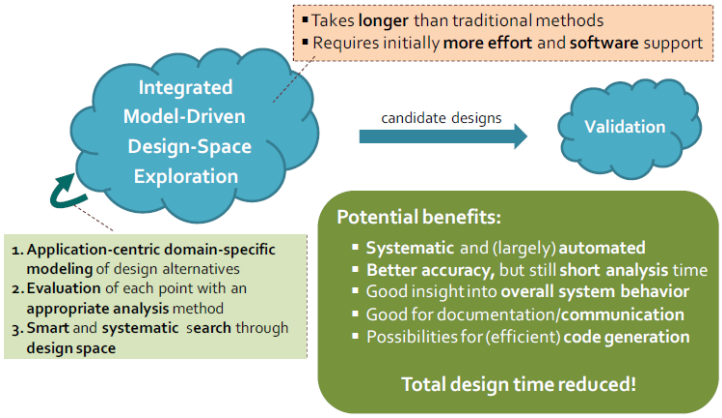

Figure 1.1: High level model driven design space exploration[2]

In the Octopus toolset, the DSE focuses on the context of printer data paths. A printer data path contains various stages such as rendering, resampling, halftoning, zooming etc. that an image or data file goes through before the final printed result is given to the end user. The quality of the printer design is judged by a wide range of metrics such as resource usage, speed, cost to print, scheduling etc. These factors have to be considered during DSE to provide the various design alternatives for the system in terms of timing properties such as throughput(for example, the number of pages printed per minute).

It is not possible to provide a complete optimal solution for a design space using a single modeling tool or analysis approach, since they have their own strengths and weaknesses. The Octopus toolset intends to combine the power of several modeling and analysis methods such as data flow analysis using Synchronous data Flow (SDF) graphs, discrete event simulation using Colored Petri Nets (CPN), and model-checking using Uppaal. A

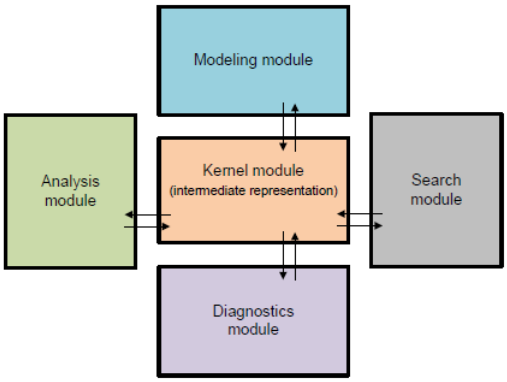

Figure 1.2: Conceptual architecture of the integrated framework of Octopus[2]

design problem can be specified in the framework using an intermediate representation called Design Space Exploration Intermediate Representation (DSEIR). This intermediate representation is used since most of the users of the toolset are not experts in the domain of these modeling tools. It also encourages reuse and consistency of models across different tools [2]. The conceptual architecture of the toolset is shown in Figure 1.2. It shows the different modules used in the toolset. The different intended modules and their functionalities are:

- **Kernel module:**
  It is the core of the toolset where a design problem is specified using an intermediate representation called DSEIR which is used across all analysis methods and tools.

- **Modeling module:**
  This module is used to create the models, for example, using a domain specific editor such as the DSEIR editor and different well known modeling languages like UML, Simulink etc.

- **Search module:**
  This module searches the design space for optimal and feasible design solutions.

- **Diagnostics module:**
  In this module, the diagnostics of the results can be produced using ResVis (Result Visualization), Excel, ProM (Process Mining) which can be used to evaluate the feasibility of the models further.

- **Analysis module:**
  This module contains the different tools such as SDF3[3], CPN[4], Uppaal[5] that are used in the toolset for the analysis of the models along with the translation procedures from the intermediate representation, DSEIR to these analysis methods.

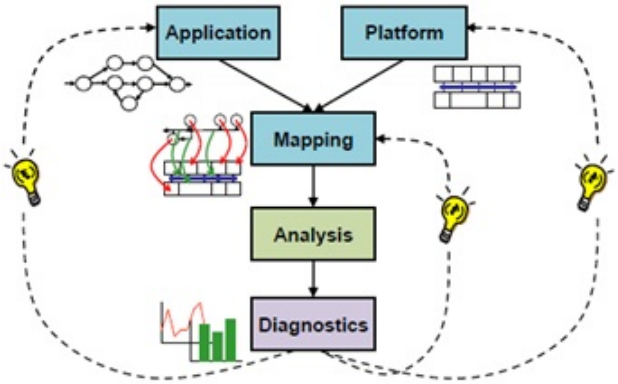

Figure 1.3: The Y-chart[6]

The DSEIR modeling language developed for the Octopus toolset is based on the Y-chart approach as shown in Figure 1.3, which is a popular methodology used in hardware design of embedded systems [6]. In the development of an embedded system, the different components namely, an application, a platform and a mapping between them need to be developed in compliance with each other since these components have an impact on the performance of the system when combined with each other or individually. This separation of concerns, achieved with the help of the Y-chart allows the evaluation of certain aspects of the system while the other aspects remain fixed [6]. This is also the basic principle followed in the Octopus toolset. In the domain of the printer data paths, the components of application domain consist of atomic tasks like rendering, resampling, halftoning etc., which vary less frequently, while the components of the platform and mapping domain need to be explored for design options. Similarly, in other cases, the components of platform might be fixed while the application and mapping need to be varied to explore the design space. These components use the results generated by the analysis tools to make the design space decisions. These results are mostly diagnostic information about the system and can be fed back to the various components to improve their performance.

The different components of the DSEIR editor that has been designed according to the different components of the Y-chart is shown in Figure 1.4 and is used to specify the model. The details of each component will be discussed in Chapter 2.

1.3 Motivation and Problem statement

This thesis concentrates on the translation of a DSEIR model to an appropriate Resource Aware Synchronous Data Flow (RASDF) graph. RASDF graphs are used for data

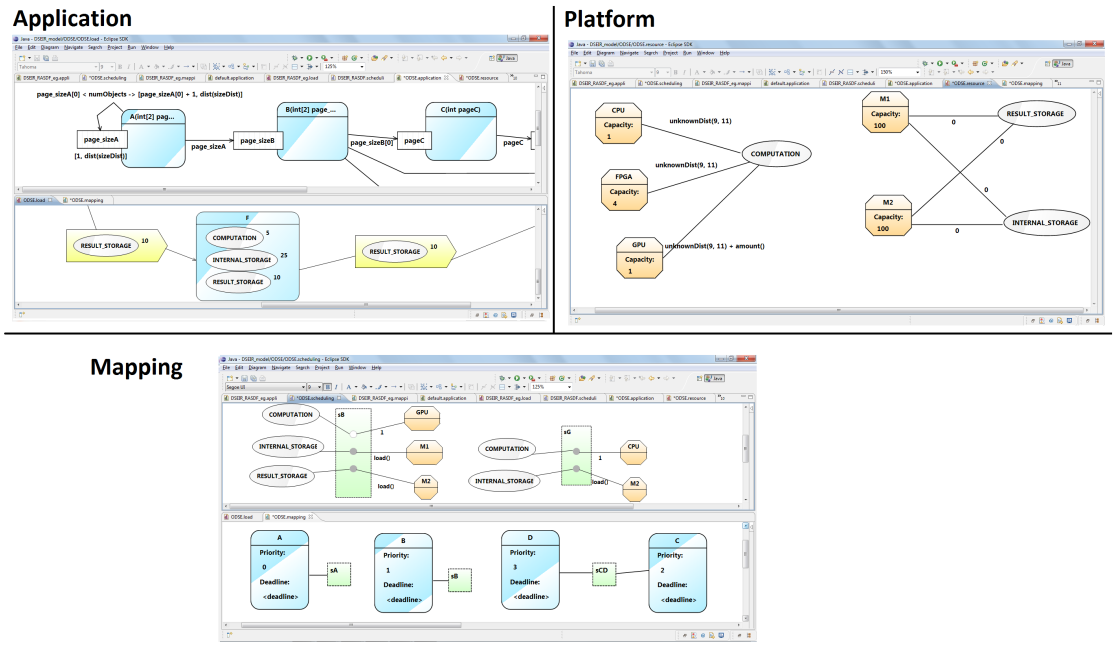

Figure 1.4: Components of DSEIR editor

flow analysis of systems to analyze parallel platforms, pipelined executions and cyclic dependencies between different operations in a model. These graphs are then used for throughput analysis of the models using a tool called SDF3. The advantage of using SDF3 to analyze RASDF graphs in the toolset is that even though they are restrictive as compared to the other analysis tools, they can produce faster analysis results.

The translation that needs to be achieved should be conservative in nature which means that the timing property such as throughput of the model remains the same or is less than the initial model after translation. By achieving this translation, we intend to obtain a worst guaranteed possible throughput of the system. This implies that the throughput obtained is the minimum possible throughput and that it cannot go lower under any circumstances. In the initial version of the toolset, there was a translation from a DSEIR model to a Synchronous Data Flow (SDF) graph that considered a subset of the application part of the DSEIR model. The real challenge is extending this translation to incorporate the platform and mapping components of the model since the addition of the resource and timing constraints pose a restriction on the behaviour of the models. Hence the DSEIR model needs to be translated appropriately. This feature was added as a result of this thesis.

During the course of the assignment, the following research questions were investigated:

1. How can you extend the existing translation procedure to handle an RASDF graph with platform and resources?

2. How can throughput analysis be achieved using SDF3 as a result of this translation?

3. Is the translation procedure capable of translating any arbitrary DSEIR model to an equivalent RASDF graph? If not, how is it handled?

1.4 Proposed solution

The methodologies used to answer the research questions are explained in this section. The main issue during the translation is that the data dependent values of DSEIR cannot be translated to an RASDF graph since an RASDF graph does not contain data dependent values. Therefore, we map a DSEIR specification to more than one RASDF graph (one RASDF graph for each boundary value on the range of the values) in order to preserve the throughput behaviour of the model. The translation of a DSEIR model to a corresponding RASDF graph is realized incrementally in various stages as shown in Figure 1.5. The outermost circle on the left denotes the set of all possible DSEIR models. In the first stage, a subset of DSEIR that corresponds directly to an RASDF graph is identified, and is called DSEIR-RASDF (innermost circle on the left in Figure 1.5) and the translation procedure is implemented.

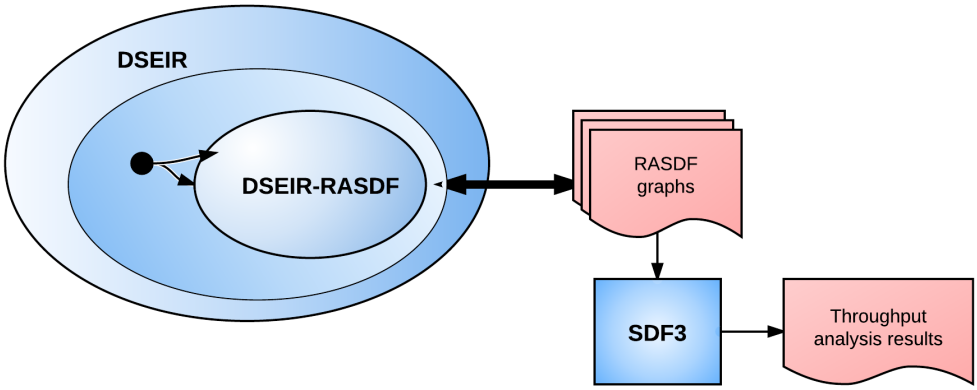

Figure 1.5: Approach used for the translation of a design problem from DSEIR to RASDF

In the next stage, an arbitrary DSEIR model is chosen and the translation procedure is extended to handle this model. As a result of this translation, there could be more than one DSEIR-RASDF graph that is generated. Thus, different issues that could arise in translating an arbitrary DSEIR model are identified and methods are implemented to handle these issues. This leads to an increase in the size of the subset of DSEIR models that can be translated to an RASDF graph (second innermost circle on the left in Figure 1.5). The RASDF graphs that are generated can be checked for correctness by a translation procedure that translates an RASDF graph to a corresponding DSEIR-

RASDF model and is shown by the arrow from RASDF to DSEIR-RASDF. This DSEIR-RASDF model is then translated again into an RASDF graph and the two graphs are compared for correctness by SDF3 and manually. The connection to SDF3 is made in the toolset transparently to allow throughput analysis of models. The RASDF graphs that are generated as a result of the translation are then used by SDF3 to generate the throughput results (right side of Figure 1.5).

1.5 Organization of report

The remainder of the report is organized as follows. Chapter 2 explains the structure of DSEIR. Chapter 3 will then explain the structure and working of RASDF graphs and the need for using it in the Octopus project. The architecture of the implemented translation procedure is explained in Chapter 4. The structure of the intermediate representation, DSEIR-RASDF is explained in Chapter 5. The translation of DSEIR-RASDF to RASDF and the backward translation from RASDF to DSEIR-RASDF is explained in Chapter 6. Chapter 7 contains the translation procedures of the translation from DSEIR to DSEIR-RASDF, along with the issues encountered and the solutions provided. Chapter 8 then contains the throughput analysis methods while Chapter **??** concludes the thesis.

Chapter 2

Design Space Exploration Intermediate Representation(DSEIR)

2.1 Introduction

In the context of the Octopus toolset, Design Space Exploration Intermediate Representation (DSEIR) is the modeling language used to specify every design problem. It follows the Y-chart methodology and allows for the separation of concerns by modeling each component of the system, namely application, platform and mapping separately. DSEIR aims at being a generic representation for any problem that is being specified in

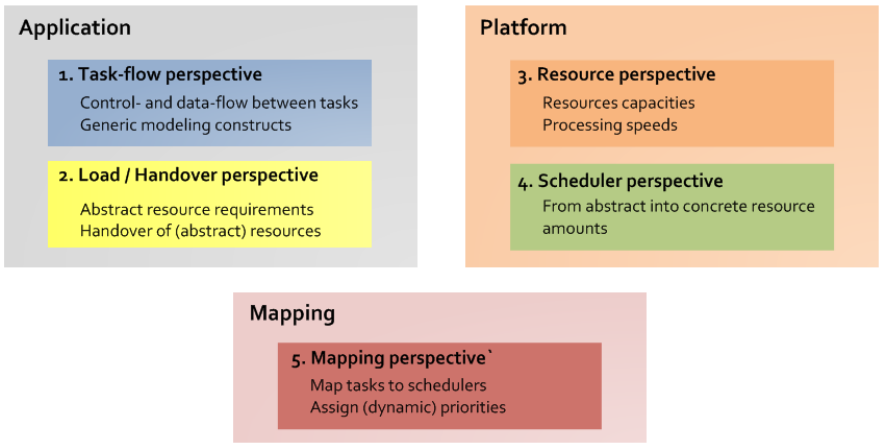

Figure 2.1: Structure of different perspectives of DSEIR with respect to Y-Chart

the toolset with a goal that the Octopus toolset supports domain specific abstractions and reuse of tools across application domains [1].

This chapter is largely based on [7]. All the figures have been drawn with the DSEIR editor that is developed for the toolset (Figure 1.4). In the editor, the different components are specified as different perspectives which is explained in detail in this chapter. The concepts of DSEIR has been developed by other members of the project. There are five perspectives that are specified in the DSEIR editor and they are grouped together to form the three different components of the Y- chart. This is given in Figure 2.1 [8]. Each component of the Y-chart is explained as a separate section in this chapter.

2.2 Application

An application perspective in DSEIR is used to represent different actions of the system and the interactions between them. The actions can be considered atomic at the current level of abstraction. These actions are represented as tasks. The task consists of variables, global variable declarations and a task flow graph.

Integers are the most basic form of variables in DSEIR. Arrays of integers can also be used to construct variables. The variables could also contain expressions with integers, array of integers or booleans that are constructed using the standard operations and relations such as addition, equal to etc. Arbitrary user-defined expressions are not allowed in DSEIR. The variables that are required to be visible to all the tasks in the model are declared as global variables. The data and control dependencies are modeled as a task flow graph in the model. The syntax of the different elements of the application perspective is shown in Figure 2.2.

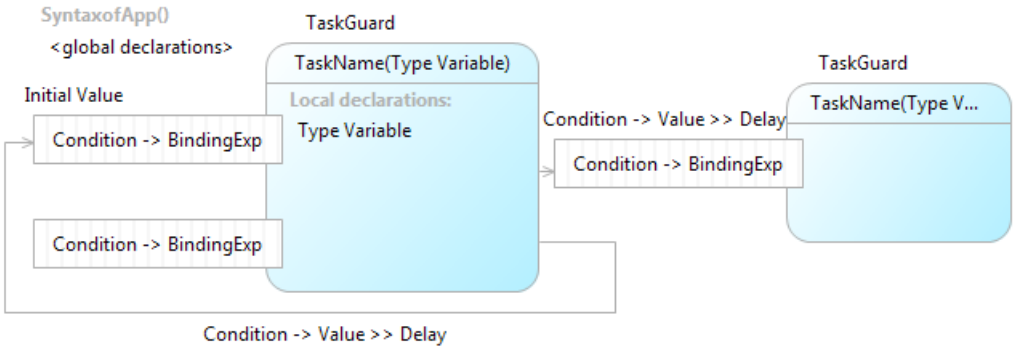

Figure 2.2: Syntax of the task flow graph in DSEIR

Tasks: A task contains zero or more parameters. Based on the values of the parameters that can be derived from the variables, different instances of the tasks are produced.

There is a guard on every task which is a boolean expression that is derived from the global variables or parameters of the task. It is assumed to be true if it is not specified. The tasks will produce results after execution or firing that is used by the succeeding task or global variables. The different components of the application model can be explained using Figure 2.3. The model in the figure consists of two tasks A and B. Task A has two integer parameters x and y. Task B has an integer parameter x along with a guard x+G<10 where G is a global variable. A task can also have local variables in addition to global variables and task parameters. These variables can be used during start or end of the task firing to capture the global state or to simplify complex result expressions [7]. The scope of a local variable is a task instance and ports are used to specify variables that are used at every instance.

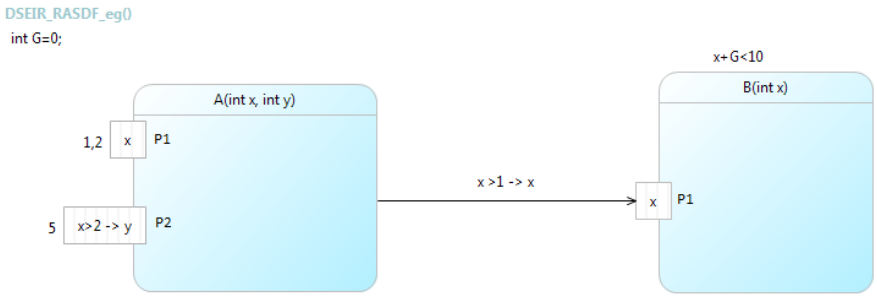

Figure 2.3: A simple task flow graph in DSEIR

Ports: Each task contains zero or more input ports and corresponding bindings to each port. The port has an unbounded capacity since it is capable of storing an arbitrary number of tokens of different types of values. The tokens are capable of carrying values and are used to store data. The ports are used to store data that are received from the other ports. The ports can also be used to specify initial values. A binding expression is derived from the task parameters and global variables of a task. When the binding expression is set to the value of a token on the port, the task parameter is assigned a unique value and is said to be bound to the value. The task A in Figure 2.3 has two input ports, P1 and P2 with two tokens containing values 1 and 2 and a token having a value of 5 respectively. The binding expressions of P1 and P2 are x and y respectively.

To increase the expressiveness of the port, a *port condition* can also be specified. It is a boolean expression derived from the task parameters and global variables. It is assumed to be true if it is not specified. It is used to denote if the tokens of the port can be considered for binding or not. Generally, the port conditions are considered to be true and the binding expressions are the task parameters, reducing the binding procedure to simple assignment of token values to task parameters. In the figure, port P2 of task A has a port condition x>2. The binding creates task instances and when the instance is executed, the corresponding tokens are removed. The order of tokens can be specified at each port. If the order is FIFO, then the token that arrived first should be used for binding. If the order of the tokens is unordered, the task chooses a

token non-deterministically for binding. The initial tokens ensure that a minimum of one binding is possible initially.

Edges: Each task has zero or more incoming or outgoing edges that are used to connect the tasks with each other. The edge is used to connect a task to a port of another task or itself. Each edge has an integer expression associated with it that denotes the value that is sent from the source task to the destination task. The edge may also have a condition, a Boolean expression that denotes whether the value can be sent across the edge or not. It is assumed to be true if it is not present. The edges can also specify the time taken for a token to reach a destination, in the form of an integer expression. It is assumed to be zero if it is not present. In Figure 2.3, the edge between tasks A and B has a condition, x>1. If the condition is true, the value of x is then sent across the edge.

The class diagram of the application perspective is shown in Figure 2.4. It shows the class hierarchy and relationship between different elements of this perspective. These class diagrams have been taken from [9].

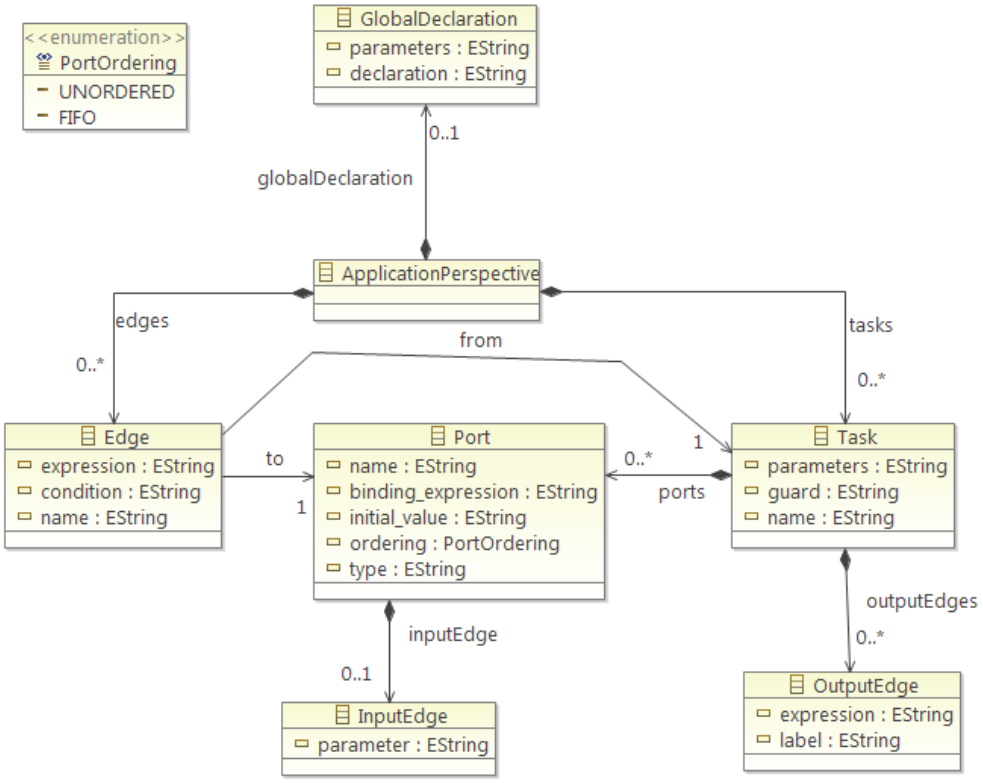

Figure 2.4: The class diagram of application perspective in DSEIR

Type conversions: DSEIR performs certain implicit type conversions for port types during initialization or when values are received at the ports[7].

1. A token with an integer-array value can be implicitly converted to an array of tokens where each token corresponds to a value in the array. The ordering of token values is preserved if the port is of FIFO type. This type conversion is used to bind a token with an array of integer values to an integer port.

2. A type conversion is also possible when the integer port has an integer-array as the binding expression. The integer-array is converted to an array of integer values such that each value in the binding expression corresponds to a token in the port with the same value. The ordering is preserved during conversion if the type of the port is FIFO. But, when the task instance fires, all the tokens that correspond to the binding are removed from the port.

Task Load: In DSEIR, there are a list of services such as *Computation*, *Internal Storage*, *Result Storage* etc., along with the possibility of user-defined services that is used to represent the resources requested by the tasks.

Each task contains a load that is associated with a service, and not with a resource[7] which is used to define the resource requirements and adequate interface with the mapping module. This is specified as a perspective in the DSEIR editor and the syntax of this perspective is shown in Figure 2.5. DSEIR contains various types of services that are provided such as internal storage, computation etc. In Figure 2.6, there is a load of 2 and 2 associated with the services *computation* and *internal storage* respectively for Task C. It is considered to be zero if it is not present. The load is an integer expression that depends on the value of the task parameters, and local or global variables during start of the firing of a task. It is evaluated at every instance of the task. During the execution of a task instance, the value of the load does not change and for every service, the unit of load remains the same. Figure 2.7 shows the class diagram of the load perspective of DSEIR.

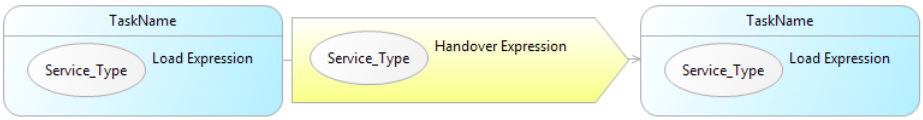

Figure 2.5: Syntax of the load perspective in DSEIR

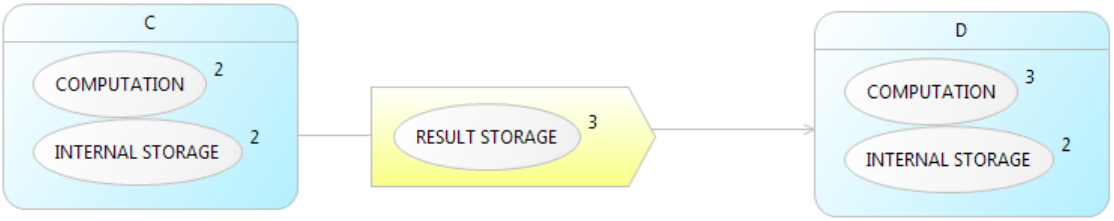

Figure 2.6: Load and handover on a task in DSEIR

Task handover: The concept of handover is used when a task needs to transfer rights

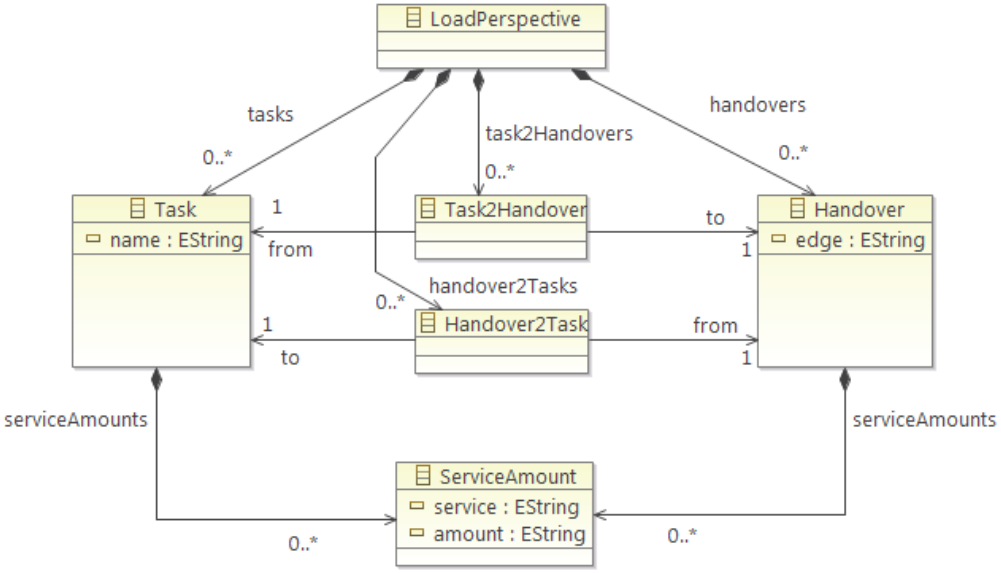

Figure 2.7: The class diagram of load perspective in DSEIR

of some resource to another task during resource reservation, like in the case of buffers. The handover is an integer expression that is evaluated at every task instance during end-firing [7]. Each edge corresponds to a handover, which is based on the parameters, local and global variables. The handover expression may also contain an expression that denotes the amount of resource held by the source task before the completion of the execution. The handover is considered to be zero if it is not present. In Figure 2.6, a value of 3 is handed over from the task C to a subsequent task D.

Task deadline: A task deadline maybe present for each task that denotes the maximum expected relative delay between the start and end of the task instance. It is an integer expression and is essential to model deadline-based scheduling algorithms [7]. It is depicted in Figure 2.15. It is not used in the context of this thesis since throughput analysis does not consider deadlines and hence will not be explained in detail. The remaining components of the figure will be explained in the subsequent sections.

2.3 Platform

In DSEIR, a set of unconnected resources are used to represent the platform component and are represented by the resource perspective in the editor. This is shown in Figure 2.8. The connections between the resources are derived implicitly using the knowledge of the mapping and the application components. There is no explicit distinction between the different types of resources such as computational, storage or communicational resource

[7]. At any time instance, a resource can provide more than one service and more than one resource can provide the same service, mostly with a different speed or capacity.

Each resource has the following components:

- **Name**: It is used to identify the resource.
- **Total capacity**: The total amount of the resource that can be taken at any point of time, and is always a positive integer.
- **Speed:** Based on the current allocation of the task and the service to be performed, it is the time taken to process one unit of load.

Figure 2.8: Syntax of the resource perspective in DSEIR

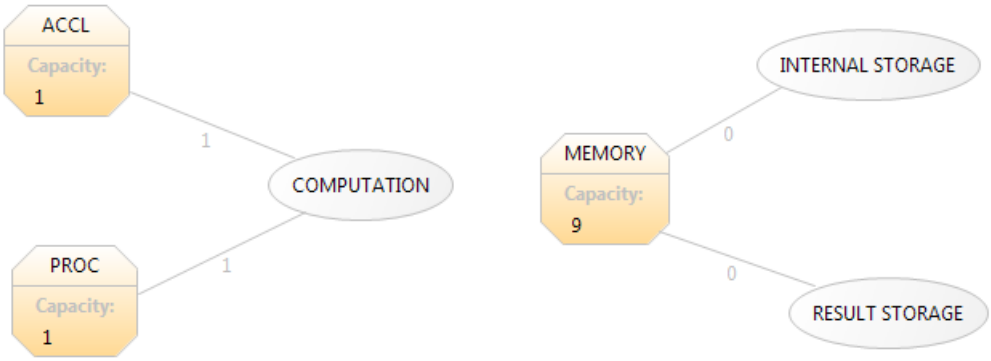

Figure 2.9: Specification of the platform perspective in DSEIR

Figure 2.9 depicts the specification of a memory resource with a capacity of 9, maybe in gigabytes [7]. It provides two types of services *Internal Storage* and *Result Storage*. A processing time of zero on the edge indicates that the speed of the memory is infinite since the processing takes no time for both the services. The figure also contains two processors ACCL and PROC which use the service, *Computation* and has a processing time of one, which is the time taken to process one unit of load. Figure 2.10 shows the class diagram of the resource perspective depicting the different types of resources and the services provided by each resource.

Schedulers: Usually, one or more resources are assigned to a scheduler along with the amount of the resource that is to be provided to every service that requests the resource. The amount of resource delivered to a particular service depends on the load of the task that requests the service, the amount of the resource that is available during scheduling

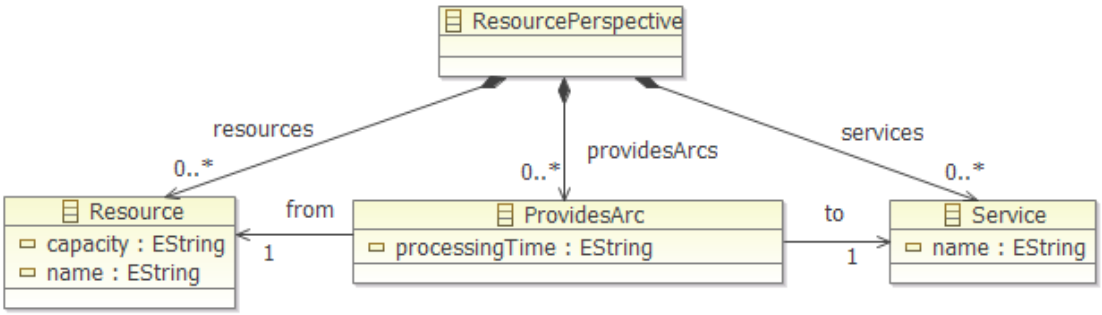

Figure 2.10: The class diagram of resource perspective in DSEIR

and the number of tasks that are waiting for the resource. Schedulers allow modeling preemption of resources. The syntax of the scheduling perspective is shown in Figure 2.11.

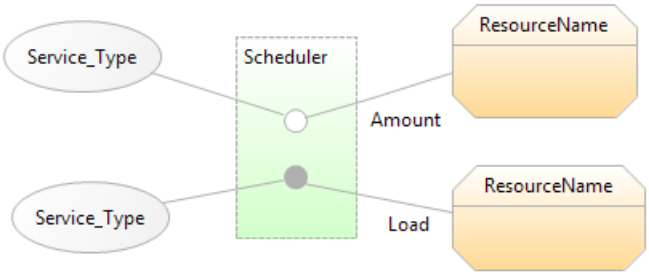

Figure 2.11: Specification of the scheduling perspective in DSEIR

In Figure 2.12, the scheduler Sch_A has two resources *Memory* and *CPU* that provide services *Internal Storage*, *Result Storage* and *Computation*. The amount of services provided for a certain resource is depicted on the edge between the resource and the scheduler. For a *CPU*, it is one which means that the resource can be used by one service at any instance of time. It is either provided to the respective service or not. The amount of service provided to *Memory* depends on the load on the task. The black filled circle in the scheduler is called a knot and is used to depict whether the scheduler is pre-emptive or not. If it is unfilled, it means that the scheduler is preemptive. The class diagram of the scheduling perspective is shown in Figure 2.13 which shows the mapping of the services to different resources by the scheduler.

2.4 Mapping

The mapping component is used to assign tasks to schedulers in DSEIR. The priorities of tasks are also defined in this perspective. The syntax of the mapping perspective is shown in Figure 2.14.

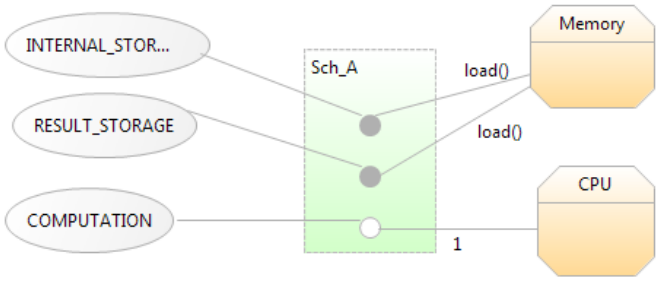

Figure 2.12: Mapping of resources and services to a scheduler in DSEIR

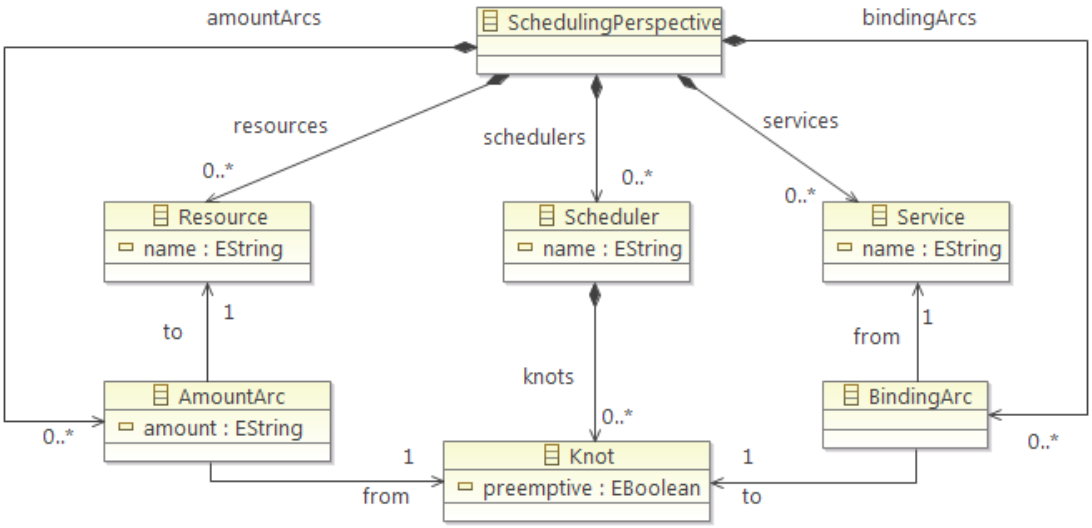

Figure 2.13: The class diagram of scheduling perspective in DSEIR

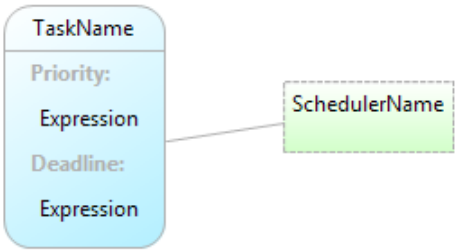

Figure 2.14: Specification of the mapping perspective in DSEIR

The notion of mapping is very simple in DSEIR with every task being mapped to a scheduler. A scheduler can be mapped to more than one task but a task cannot be mapped to more than one scheduler. The mapping between the task A and a scheduler Sch_A is depicted in Figure 2.15. The description of the task also contains information about deadlines and priorities. Figure 2.16 shows the class diagram of the mapping

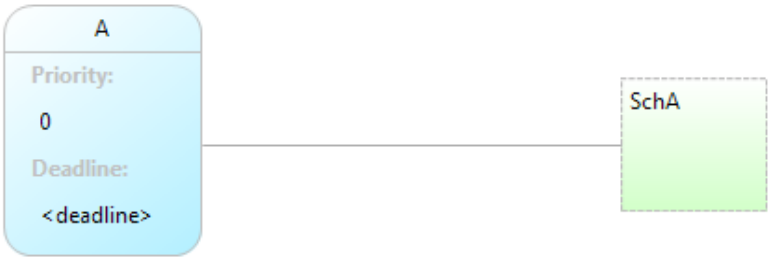

Figure 2.15: Mapping of a task to a scheduler in DSEIR

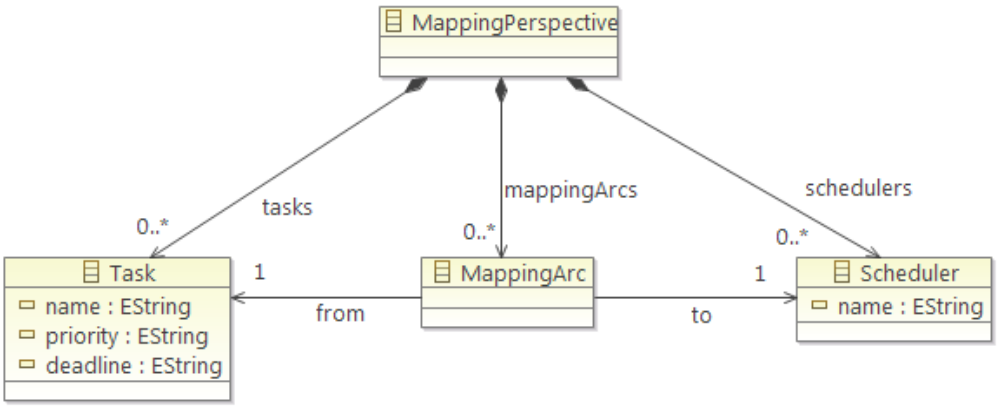

Figure 2.16: The class diagram of mapping perspective in DSEIR

perspective. It contains the attribute *Deadline*, but it is not used in the translation of DSEIR to RASDF.

Task priorities: Mapping in DSEIR also defines priorities between tasks. They are expressions over task parameters, global variables and run-time information [7]. During multiple task instances, a task instance with a higher priority is always scheduled before a task instance with lower priority. The priority on a task is specified in Figure 2.15 where 0 is the highest priority and the priority decreases with the increase in the number specified for the priority in the task. DSEIR allows for dynamic assignment priorities with the help of expressions.

Chapter 3

Resource Aware Synchronous Data Flow Graphs (RASDF)

3.1 Introduction

This chapter explains the concepts and working of an RASDF graph. It is explained in Sections 3.2 and 3.3. Section 3.4 will then emphasize the need of this analysis in the Octopus toolset while Section 3.5 explains anomalous behaviour in RASDF graphs, how they affect the worst case analysis of models and how it is solved.

Synchronous Data Flow(SDF) graphs are used for the design time analysis of synchronous, parallel applications [10]. An SDF graph is comprised of actors and channels, as seen in Figure 3.1. Actors are nodes in the graph and are used to represent a particular action, code or functionality in a system that needs to be performed for the execution of the actor. Each actor has an execution time that denotes the duration for which the actor executes. In the figure, A, B and C are actors with execution times 3, 2 and 1 respectively. Each actor contains input and output ports that denote the number of tokens that are produced or consumed during a firing, which is called the rate of the actor. The input and output ports of the actor B has a rate of 1 each.

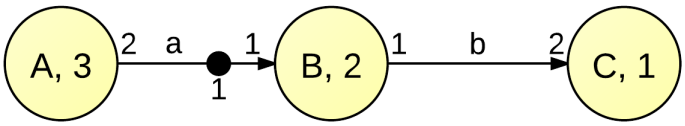

Figure 3.1: An SDF graph

Channels exist between actors in a graph and are used to represent data dependencies or

order of execution of the different actors. In the figure, the graph contains two channels, namely a and b. Each channel can contain an infinite number of tokens that are used for the firing of an actor and it also contains a certain number of initial tokens present on the channels before the start of the system. In the example, the channel a has one initial token. An actor can only fire if the specified number of tokens are present on all its input channels. Each actor will always produce and consume a constant number of tokens in each firing. During the firing of actor B, it will consume always consume one token on its input edge and produce one token on its output edge.

3.2 Resource Aware SDF

When the resource configuration along with requirements and constraints, and the architecture of the system is taken into consideration, the SDF graph is extended and is called a Resource Aware SDF (RASDF) graph. The resources of the system are represented as rectangles in the graph that are connected to different actors.

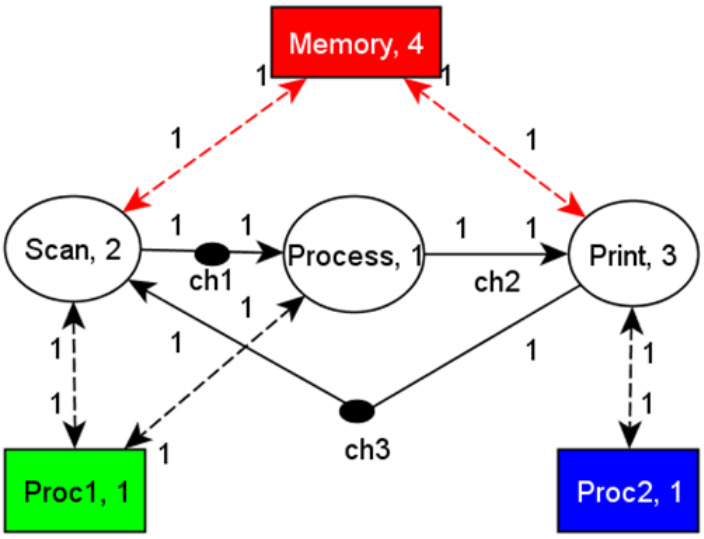

Figure 3.2: An RASDF graph

Each resource in an RASDF graph has a certain capacity. During the start of the firing, the actor claims a certain amount of resource and it releases resources at the end of the firing. Figure 3.2 shows a simple RASDF graph with actors, channels and resources. It has three actors; *Scan, Process* and *Print*, three channels; ch1, ch2 and ch3 and three resources; *Memory, Proc1* and *Proc2* with capacities 4, 1 and 1 respectively. The actor *Scan* claims one unit of Proc1 during the start of the firing and releases one unit after the firing.

3.3 Formal definitions and terminologies in an RASDF graph

This section defines all the concepts and terminologies used to explain semantics of the RASDF graph in this thesis. It is largely borrowed from [10], [11] and [12] and tries to give an idea of all the different concepts necessary to understand an RASDF graph.

- **Ports and Rate of ports**
  In an RASDF graph, there is a finite set *Ports*, used to define the ports in the graph, and with each p $\in$ *Ports*, there exist a positive finite rate, *Rate*(p) such that *Rate* : *Ports* $\mapsto \mathbb{N}\backslash\{0\}$. In Figure 3.2, the actor *Scan* has an input port and an output port with a rate of one each. Similarly, the other actors have input and output ports too.

- **Actor of a graph**
  There is a finite set A of *actors* and each actor a $\in A$ is a two-tuple (*In, Out*; denoted by *I(a)* and *O(a)* respectively) such that the sets *In*, *Out* $\subseteq$ *Ports* represent the input and output ports of the actor 'a' respectively with *In* $\cap$ *Out* $= \emptyset$. In Figure 3.2, *Scan, Print* and *Process* are the actors in the graph.

- **SDFG**
  A Synchronous Data Flow Graph (SDFG) can be defined as the tuple (A, C, τ), where A is a finite set of *actors*, a finite set C $\subseteq$ $Ports^2$ is a set of *channels*, and a mapping τ : $A \mapsto \mathbb{N}$, that denotes the amount of time taken by each actor 'a' to complete the firing of a single instance. The source of every channel is an output port of some actor and the destination is the input port of some actor. All ports of all actors are always connected to exactly one channel. τ can also be called as the *execution time* of each actor 'a' to complete its firing. In Figure 3.2, for the actor *Print*, we have $\tau(Print) = 3$.

- **RASDFG**
  A Resource Aware SDFG (RASDFG) can then be defined using the tuple (*A, C,* τ, *R, Req,* ρ), such that it consists of an SDF graph (A, C, τ), finite set of resources *R, Req* $\subseteq A \times R$ requirement edges, and a resource configuration ρ : $R \mapsto \mathbb{N}$, which denotes the available amount of resources for every resource r $\in R$. When an actor a starts firing, it consumes *Rate(q)* tokens from all its input ports *q* $\in$*In(a).* After a period of time τ(a), the actor has completed its firing and then produces *Rate(p)* on all of its output ports *p*$\in$*Out(a).* Since we assume that resources are claimed and released at the start and end of each actor firing respectively, we say that for each resource requirement edge *(a,r)*$\in$*Req; Clm(a,r)* gives the amount of *'r'* claimed at the firing start of *'a'* and *Rel(a,r)* gives the amount of *'r'* released at the firing end of *'a'.*

The example in Figure 3.2 can then be formally defined as,
G= (A, C, τ, R, Req, ρ),
A = $\{Scan, Process, Print\}$,
C = $\{ch1, ch2, ch3\}$,
$\tau = \{(Scan, 2), (Process, 1), (Print, 3)\}$
R = $\{Memory, Proc1, Proc2\}$,
Req = $\{(Scan, Mem), (Scan, Proc1), (Process, Proc1), (Print, Mem), (Print, Proc2)\}$,
$\rho = \{(Memory, 4), (Proc1, 1), (Proc2, 1)\}$.

- **Repetition vector and consistency**
  A repetition vector γ of an RASDFG $(A, C, \tau, R, Req, \rho)$ is a function $\gamma : A \mapsto \mathbb{N}$ such that for each channel with its output and input port *(o,i)*, denoted by *(o,i)* $\in$ C from an actor $a \in A$ to another actor $b \in A$,

  $$Rate(o).\gamma(a) = Rate(i).\gamma(b);$$

  also called the balance equation[11]. A repetition vector γ is non-trivial iff γ(a)>0, $\forall a \in A$.

  An RASDF graph (RASDFG) is called consistent iff it has a non-trivial repetition vector. For every consistent graph, there is a unique smallest non-trivial repetition vector, which is then called the repetition vector of the RASDFG. Any graph that is not consistent will enter into a deadlock or will need unbounded memory to execute[12].

- **Homogenous SDFG**
  It is an SDFG such that $\forall p \in Ports$, $Rate(p) = 1$. The repetition vector will then be equal to one for all its actors since the rate of all ports is equal to one. It is observed that any SDFG can be converted to an equivalent HSDFG[13][14]. But this conversion could lead to an exponential increase in the number of states of the resulting graph. In Figure 3.2, since it can be observed that the $Rate(p) = 1$, $\forall p \in Ports$, the repetition vector $\gamma = < 1, 1, 1 >$ and the graph is a homogenous graph.

There are certain additional properties of the graph that are not explicitly specified in the semantics of the graph such as priority between tasks, preemptive behavior of resources etc. In this example (Figure 3.2), a static priority has been defined between actors such that the task *Scan* is of highest priority and task *Print* is of lowest priority. It has also been specified that the resources are non-preemptive which adds a certain restriction on the behavior of the system. It can be observed that the firing of actors in the graph is sequential since resource contention, priorities and number of tokens in the graph restrict the concurrent execution of tasks.

- **Throughput**
  Every RASDFG can typically have more than one firing.[15] observes that the

highest achievable throughput, *T(a)* is mostly observed if the actor is fired as soon as it is enabled when all the required resources are available. The throughput analysis is mostly calculated on a strongly connected SDFG[12].

The throughput of each actor $a \in$ A of a strongly connected RASDFG G, is defined as the average number of firings of *'a'* per unit time. The throughput of the graph G is then defined as

$$T(G) = \frac{T(a)}{\gamma(a)}$$

where γ(a) is the repetition vector of an arbitrary actor *'a'*. The throughput of G denotes the average number of iterations of the graph per unit time [12].

- **Monotonicity**
  Let T be the observed throughput for a graph G = *(A, C,* τ*, R, Req,* ρ*)*, denoted by *T(G)*. If an actor a$\in$A has variable execution times, say $\tau(a)$ and $\tau'(a)$ such that it is represented by G'= (*A, C,* τ'*,R, Req,* ρ) for each of the set of the execution times, i.e.$\forall\tau'(a)$. We can then say the model represented by the graph G is monotonous iff it holds that $\tau'(a) \leq \tau(a) \Rightarrow$ $T(G') \geq T(G)$.

  Non-monotonicity of a graph can then be defined as the graph in which there exists an actor, $a \in A$, such that if $\tau'(a) < \tau(a)$, it can be observed that $T(G') > T(G)$

If one of the tasks in the graph in Figure 3.2 has variable execution tasks i.e., if different firings of a certain actor has different execution times. Let us assume that the task *Process* has variable execution times, x such that $1 \leq x \leq 4$ but the execution times of the remaining tasks remain constant. Table 3.1 shows the throughput of the graph when the execution times of the task *Process* is varied but the execution times of the other tasks remain the same.

Table 3.1: Throughput results of the graph with variable execution time

Execution time of *Process*	**Throughput**
1	0.33
2	0.25
3	0.20
4	0.1667

From the definition of monotonicity, it can be observed that the graph is monotone w.r.t. task *Process* since the throughput is always decreasing when the execution time of this task is increasing. The throughput results are obtained by executing the graph specification in SDF3 as separate graphs for each value of the execution time for the task *Process*.

3.4 Need of RASDF analysis in Octopus

In the context of the Octopus project, large and complex models are generally used for Design Space Exploration. These models have stringent requirements in terms of timing, resources etc. The toolset needs to be capable of performing trade-off analysis for such models within an acceptable period of time. Hence the toolset contains the analysis chain using SDF3 for RASDF graphs. Also, there was a need to extend the existing translation to capture all the components of the Y-chart in order to have an effective translation of a DSEIR model.

Among the analysis tools available in the Octopus toolset, it has been observed that the worst case throughput analysis using RASDF graphs provided by SDF3 is the fastest. But there is a tradeoff with using SDF3 analysis. The RASDF graphs are not capable of expressing the models in the same level of detail as the DSEIR models and hence there is a need for an abstraction from DSEIR to RASDF by preserving the properties such as time, resources etc. Preserving the different properties would mean that the behavior of the model before and after translation would remain the same after the abstraction with respect to the properties under consideration. Let us consider a property X for a model M. The resulting model after translation is called M'. We can then say that the property X such as throughput is conservatively preserved during translation iff $X(M) \geq X(M')$. It can be $X(M) \leq X(M')$ for other properties that are not considered in this thesis.

In the context of the Octopus project, if a meaningful translation can be made from a DSEIR model to an RASDF graph, then a quicker worst case analysis of the design space can be obtained. When this worst case analysis is satisfactory, the other analysis tools may be used for further analysis of the model. This would save the user a lot of time and effort during design space exploration and hence seems a beneficial addition to the toolset.

3.5 Anomalous behavior of RASDF models

In RASDF graphs, it has been observed that the maximum execution time of the actors does not always guarantee the worst case throughput of the system and identifying this non-monotone behaviour is very essential for the worst case throughput of the system.

Consider the example from Figure 3.2, Table 3.2 shows the throughput of the example when the actors have variable execution times. The task *Print* has variable execution time in this example and it can be observed that the throughput of the graph is increasing when the execution time changes from 2 to 3. Hence the model is said to exhibit non-monotone behaviour.

Table 3.2: Throughput results of the graph with variable execution time

Execution time of *Scan*	**Execution time of *Process***	**Execution time of *Print***	**Throughput**
2	1	1	0.33
2	1	2	0.285
2	1	3	0.33
2	1	4	0.25

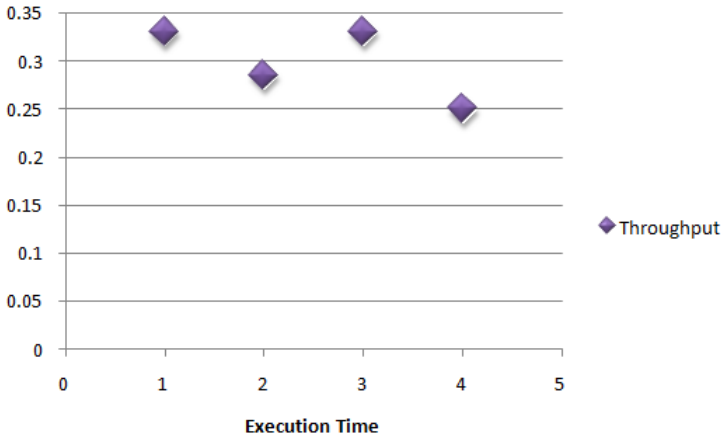

Figure 3.3: Graph showing non-monotone behaviour

Figure 3.3 shows the graph with the throughput of the model plotted against the execution time of the actor, *Print*. According to the definition of non-monotonicity, for execution times 2 and 3 such that $\tau(a) < \tau'(a)$, it can be observed that $T(G') > T(G)$ and hence the model is non-monotone. Additional checks will be required to detect non-monotone behaviour during the throughput analysis of the system. This is required in order to guarantee the worst case throughput of the system as non-monotonicity could lead to inaccurate results and thus identifying this behaviour becomes very important.

3.6 RASDF analysis using SDF3

SDF3 is a tool used for generating and analyzing SDF/ RASDF graphs [16]. The tool consists of many analysis and transformation algorithms, throughput analysis being one of them. The tool also provides functionalities to visualize these results. The RASDF graph can be specified in the XML format where the actors, ports, nodes, resources etc. are specified in the form of XML tags. For the specific configuration, the tool generates all possible execution traces and calculates the throughput for the graph, which is used in the toolset.

The tool SDF3 has been used to detect non-monotone behaviour in the RASDF graphs that are generated. Since RASDF graphs cannot model variable execution times like a DSEIR model, each value of the execution time has been modeled as a separate RASDF graph in this thesis. The throughput analysis is done for the entire set of graphs and non monotone behaviour is detected in the model. The methods used for the detection of non monotone behaviour will be explained in Section 8.3.2.

Chapter 4

Architecture of DSEIRtoSDF

4.1 Introduction

This chapter contains an overview of the approach used for the translation of a DSEIR model to an RASDF graph in the toolset. The DSEIR model is translated into various intermediate models using different methods to obtain the RASDF graph. This graph is then analyzed for throughput results of the DSEIR model. The overall architecture has been explained in this chapter.

4.2 Architecture

Figure 4.1 shows the architecture of the translation procedure that is implemented in this thesis to achieve the throughput of the model. It consists of two main components: DSEIRtoSDF and RASDFAnalyzer. The DSEIRtoSDF component reads a DSEIR model and performs various translation procedures internally to produce more than one RASDF graph. The details of the translation will be discussed in Section 4.2.1. The RASDF graphs are then read by the RASDFAnalyzer to perform throughput analysis and produce the worst guaranteed throughput behaviour of the initial DSEIR model. The details of the analysis procedures is explained in Section 4.2.2. The RASDF graphs can also be read by the DSEIRtoSDF component to perform different translations and produce RASDF graphs that are same as the input graphs. This is used to verify the translation procedures.

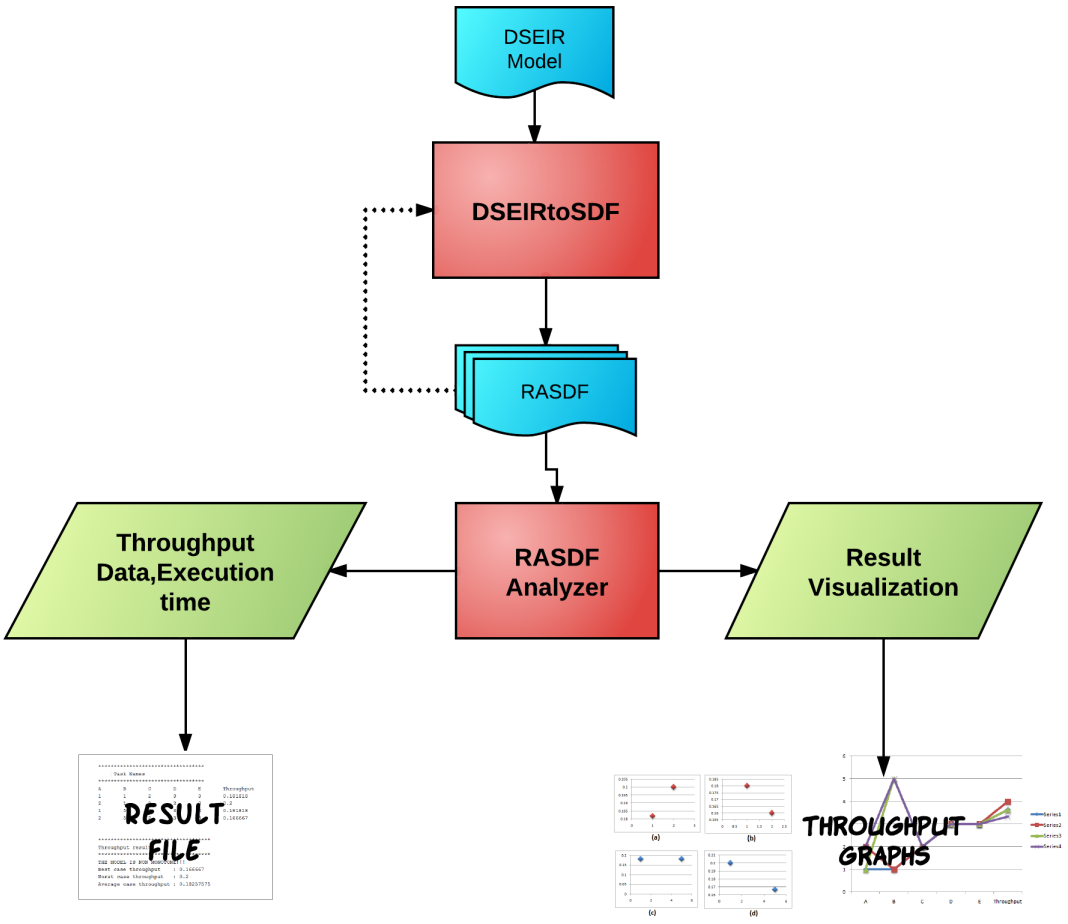

Figure 4.1: DSEIR2SDF Translation

4.2.1 DSEIRtoSDF

The DSEIRtoSDF component from Figure 4.1 is expanded and the internal structure is shown in Figure 4.2. This component takes a DSEIR model as an input and produces more than one RASDF graph. This component contains many JAVA methods. Each method is explained in detail as a separate chapter. This section emphasizes the importance and motivation behind each method. The different methods and their functionalities are:

1. **ConformanceChecker:** This method is used to check if the DSEIR model can be translated into an RASDF graph. Since there are a large subset of DSEIR models cannot be translated to an RASDF graph, a method is implemented to check if the given DSEIR model can be translated such that it can be handled by the tool SDF3 for throughput results. The different issues that can arise during

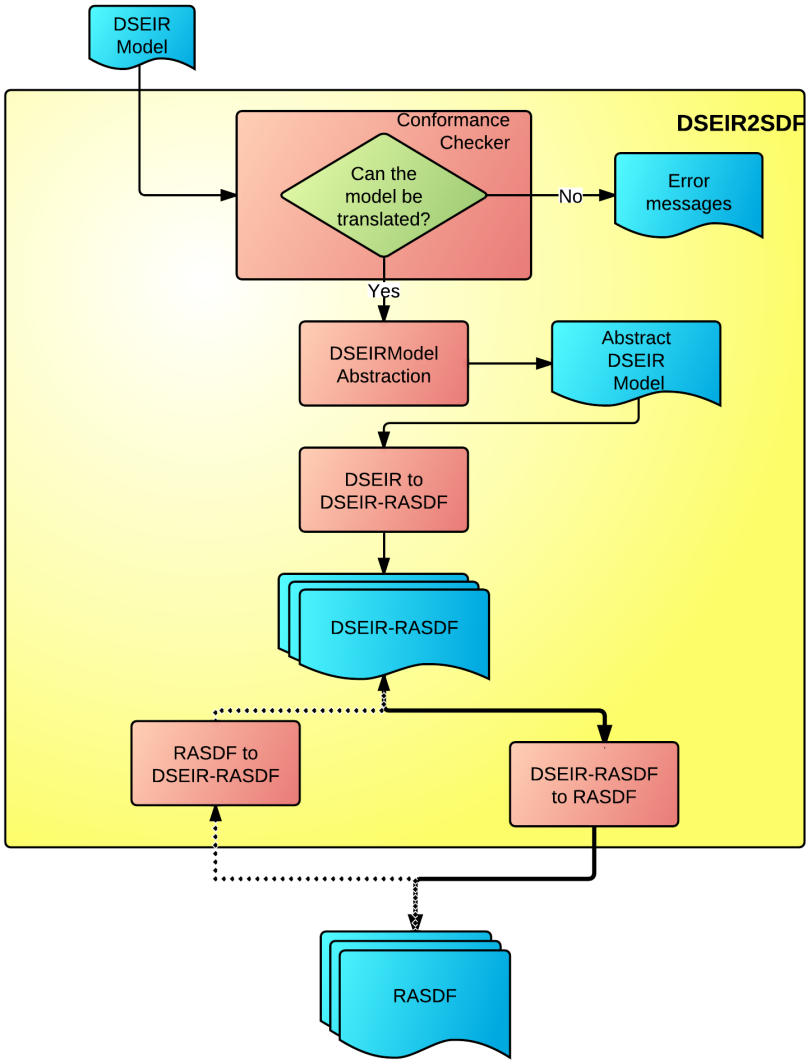

Figure 4.2: DSEIRtoSDF Translation

the translation and how they are solved with the help of this method is explained in Chapter 7. This is one of the important problems addressed in this thesis since this will help to increase the size of the subset of the DSEIR models that can be translated.

2. **DSEIRModelAbstraction:** This method addresses a specific issue in the translation of the DSEIR model to its corresponding RASDF graph. DSEIR models can contain task parameters which depend on data and cannot be handled by an RASDF graph. This method is used to create an abstraction that translates the DSEIR model into an intermediate abstract DSEIR model that does not contain

task parameters. Section 7.2.1 explains the abstraction procedure for the evaluation of task parameters along with evaluating the various expressions in the DSEIR model. The method is implemented by evaluating the different expressions in the variables that form the task parameters. All the variables are translated to integer constants in this method.

3. **DSEIRtoDSEIR-RASDF:** The intermediate abstract DSEIR model obtained after the abstraction using the method, *DSEIRModelAbstraction* contains the possible range of values for the different parameters of the different tasks in the model. Using this range of values, a DSEIR-RASDF model is generated for all possible combinations of the boundary values. This method is explained in detail in Section 7.2.2. The models that are generated are converted to the intermediate models called DSEIR-RASDF models. These models are DSEIR models which is similar to an RASDF graph in structure.

4. **DSEIR-RASDFtoRASDF**: This method is used to generate an RASDF graph from the DSEIR-RASDF model obtained after the DSEIRtoDSEIR-RASDF translation. This translation is trivial since the model generated is comparable to the structure of an RASDF graph. Any DSEIR model can be translated to an RASDF graph iff the DSEIR model can be translated into a DSEIR-RASDF model. The translation procedure for each component of the Y-chart along with the restrictions of the translation is explained in Section 6.2.

5. **RASDFtoDSEIR-RASDF:** This is the translation procedure used to verify the DSEIR-RASDFtoRASDF translation. An arbitrary RASDF graph can be converted into a DSEIR-RASDF model using this method. This DSEIR-RASDF model is then translated into an RASDF graph with the help of the method DSEIR-RASDFtoRASDF. The results of the translation are manually verified against the initial RASDF graphs and using SDF3 to check the correctness of the translation. The translation procedure along with the benefits is explained in Section 6.3.

As a result of the translation using this component, one or more RASDF graphs will be generated. They are then analyzed for throughput results using the tool, SDF3 in the component RASDFAnalyzer.

4.2.2 RASDFAnalyzer

This component is responsible for analyzing the model that is generated as a result of the translation from DSEIR to RASDF graphs. The DSEIR model needs to be analyzed for throughput behaviour.

The RASDFAnalyzer component comprises of:

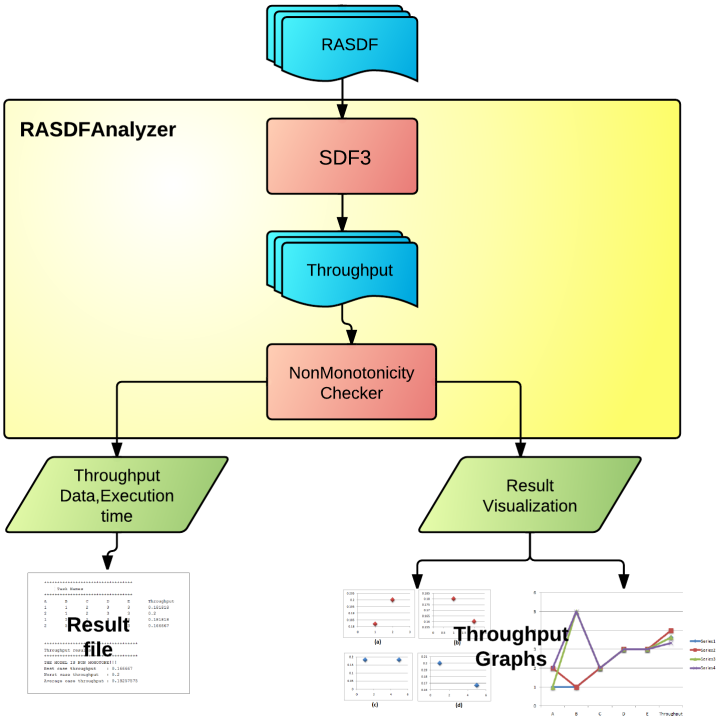

Figure 4.3: RASDF Analyzer

1. **SDF3:** This is a state-space based exploration tool used to obtain the throughput results of an RASDF graph as explained in Section 3.6. It is capable of producing throughput analysis results and visualization capabilities [16] for a given set of actors and resources configuration.

2. **Throughput results:** The throughput results for the different RASDF graphs obtained are stored in a class. It contains the best case, worst case and average case throughput results of the model. The method is explained in detail in Chapter 8.

3. **Non-monotonicity checker:** As discussed in Section 3.5, the issue of non-monotone behaviour of models is an important issue that needs to be solved during the translation of the model. It is solved in this method such that the worst case guaranteed throughput can be provided for any DSEIR model. This is explained in Section 8.3.

4.2.3 Translation results

The results of the translation is always in terms of throughput of the model, along with a few visualization results. There are two components in the results of the model, the details of which is discussed in Section 8.4. The components are:

1. **Throughput result file:** This is the result file that contains information about the throughput of the model for all the RASDF graphs that are generated. It also contains information about best, worst and average case throughput of the model.

2. **Result visualization:** The throughput results of the model is visualized in the form of parallel coordinates and two dimensional graphs. It gives a better understanding of the throughput behaviour of the model based on execution times of each actor in the RASDF graph.

Chapter 5

A subset of DSEIR: DSEIR-RASDF

5.1 Introduction

In the Octopus toolset, there are certain translation procedures that are implemented to translate a model specified in the DSEIR language into a form that can be understood by the different analysis tools like SDF3, CPN, Uppaal etc. Since DSEIR modeling language is capable of specifying a design problem with data parameters, a model specified in the DSEIR language requires more than one translation before it can be used for analysis in SDF3. We use a subset of DSEIR in this work to help the translation of the DSEIR model to an RASDF graph. This subset is called DSEIR-RASDF and is explained in this chapter. The class diagrams used in this chapter have been drawn to emphasize the differences between DSEIR and DSEIR-RASDF.

5.2 Motivation

The translation of a problem specification from DSEIR to an RASDF graph is not performed directly. There is an intermediate translation that is carried out invisibly in the tool set to make the translation simpler. This resulting representation is called DSEIR-RASDF. It is a restricted proper subset of DSEIR since all the features of DSEIR-RASDF are present in DSEIR but not vice-versa. The advantage of this subset of DSEIR is that it is similar to RASDF in structure and can be mapped to an RASDF graph by a simple 1:1 mapping as depicted in Figure 1.5. The second advantage of DSEIR-RASDF is that it reduces the dependency of the model on RASDF by keeping all the

translation procedures within DSEIR. The translation from DSEIR to DSEIR-RASDF requires a method that is capable of producing a conservative translation that preserves all the information necessary to construct a RASDF graph. If a suitable abstraction can be achieved (Chapter 7), translation from DSEIR-RASDF to RASDF graph will produce the same throughput results before and after translation since the structure and parameters for DSEIR-RASDF resemble the structure of RASDF graph, making the translation rather trivial.

5.3 Application

The basic notion of tasks, ports and edges remain in the same in DSEIR-RASDF. But the major difference between the two is the absence of task parameters, global variables, guard conditions, port conditions and edge conditions. All the conditions are assumed to be true. In DSEIR-RASDF, each port is bound to tokens, eliminating the need for task parameters or global variables. The tokens carry integer values similar to DSEIR, but the value of the tokens is always equal to one.

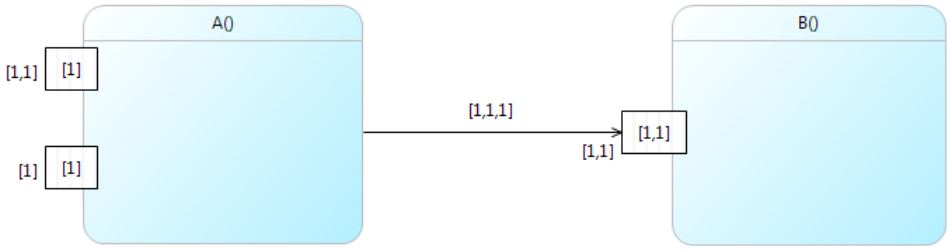

Figure 5.1: A simple application model in DSEIR-RASDF

In an RASDF graph, the execution is based on the flow and number of tokens and the tokens do not carry any value rendering the existence of values for tokens in DSEIR-RASDF unnecessary. Figure 5.1 contains an example used to explain the application component. The port of the task B has binding to two tokens with a value of one each since bindings are not allowed for tokens with values greater than one.

It can be observed that the edges, ports and tasks remain the same. The difference being that the conditions on the edge and port are missing and the ports are not bound to any parameter, but to constant values, eliminating the need of task parameters in DSEIR-RASDF. Figure 5.2 shows the class diagram of the application perspective of DSEIR-RASDF. It can be seen that it is more restrictive than DSEIR.

The load perspective is also a part of the application component in Y-chart similar to DSEIR. In DSEIR-RASDF, the load on each service type is always an integer constant and cannot be an expression, which is depicted in Figure 5.3. This is the restriction

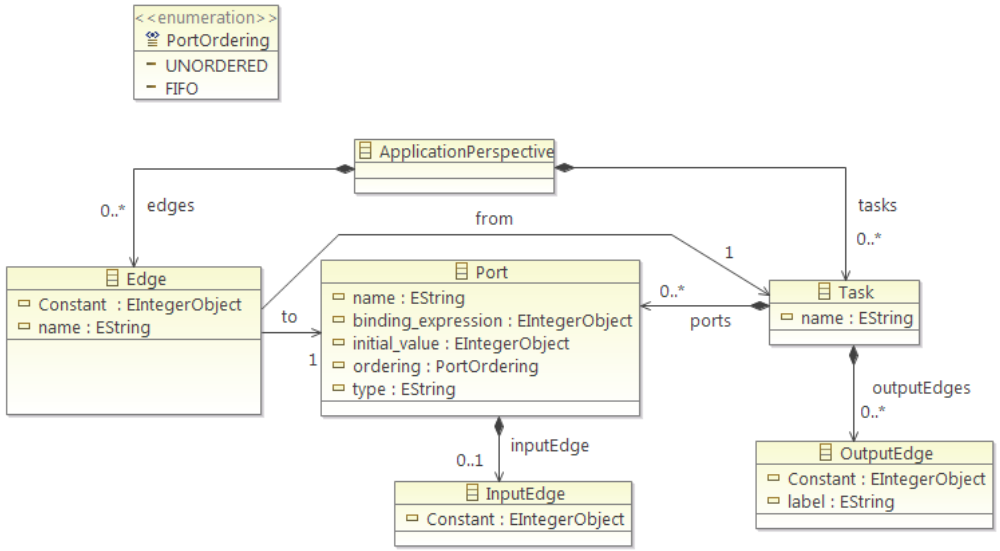

Figure 5.2: The class diagram of application perspective in DSEIR-RASDF

added on the load perspective in DSEIR-RASDF. Figure 5.4 shows the load perspective of DSEIR-RASDF with the restriction that the amount is an integer or an array of integers.

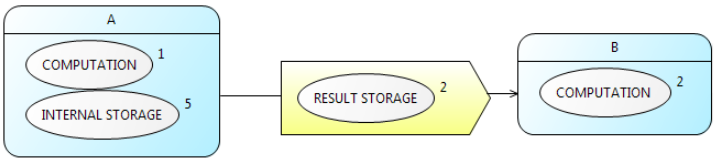

Figure 5.3: The load perspective in DSEIR-RASDF

5.4 Platform

For the platform component of DSEIR-RASDF, there is a restriction with respect to DSEIR. In the resource perspective, the processing speed for a service type *Computation* is always one and it cannot be more than that since the speed of the processor is not considered in an RASDF graph. It is shown in Figure 5.5. Figure 5.6 shows the resource perspective of DSEIR-RASDF.

In the scheduling perspective, all the schedulers are always non-preemptive in DSEIR-RASDF and cannot be preemptive due to the execution pattern of RASDF graphs. This is an additional restriction and is depicted in Figure 5.7. The class diagram of the

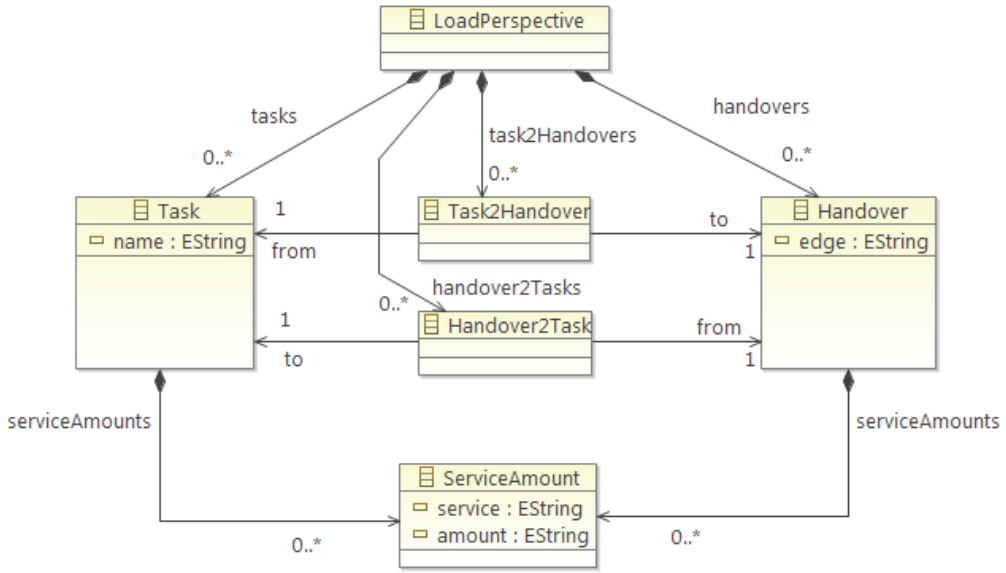

Figure 5.4: The class diagram of Load perspective in DSEIR-RASDF

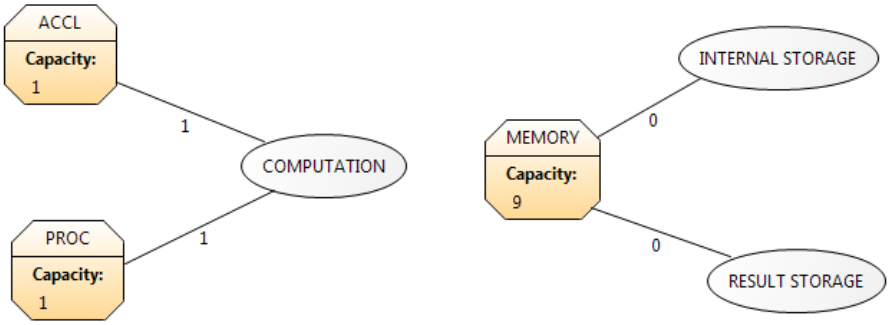

Figure 5.5: The resource perspective in DSEIR-RASDF

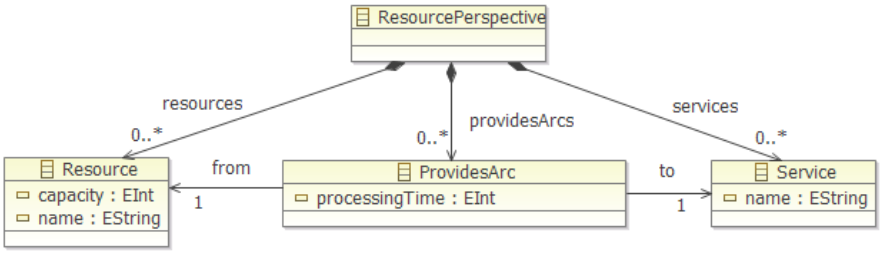

Figure 5.6: The class diagram of resource perspective in DSEIR-RASDF

scheduling perspective is shown in Figure 5.8 with the restriction that the scheduler is always non-preemptive only.

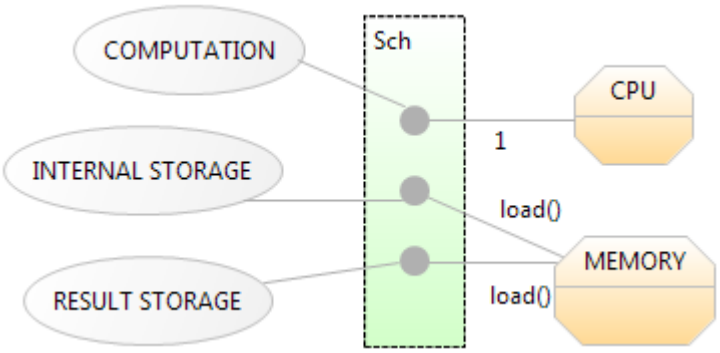

Figure 5.7: The scheduling perspective in DSEIR-RASDF

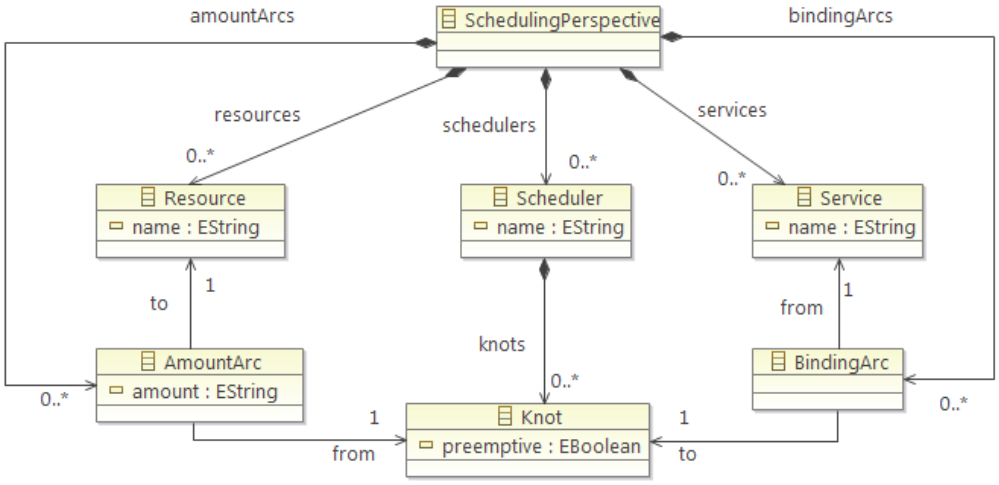

Figure 5.8: The class diagram of scheduling perspective in DSEIR-RASDF

5.5 Mapping

In the mapping perspective of DSEIR-RASDF, the task deadline is an integer, but is not considered in the translation similar to the task deadline in DSEIR. The major difference in this perspective with respect to DSEIR is the task priority. The task priority is always static in DSEIR-RASDF unlike DSEIR where the task priority can be dynamic. The mapping perspective is shown in Figure 5.9. The class diagram of the mapping perspective is shown in Figure 5.10.

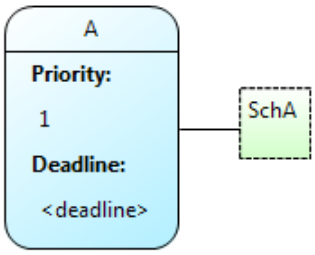

Figure 5.9: The mapping perspective in DSEIR-RASDF

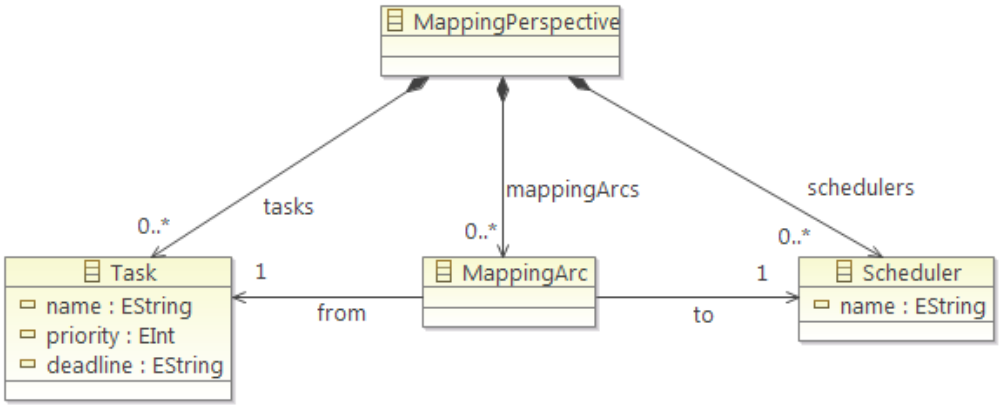

Figure 5.10: The class diagram of mapping perspective in DSEIR-RASDF

5.6 Summary

The restrictions on DSEIR-RASDF is summarized below showing that it is a proper subset of DSEIR.

- No task parameters.
- No global variables.
- No guard is specified on the task; it is considered to be true always.
- The initial value on the port is always specified as an integer with a value one or as an array of integers that have a value of one.
- The binding expressions are also based on the integer or integer array constants and cannot be expressions.
- No condition is specified on the port; it is considered to be true always.
- The value sent across the edge is always one or an array of one's.

- No condition is specified on the edge; it is considered to be true always.
- The load on a task is always a constant value and cannot be an expression.
- The speed of the computation resource is always one.
- The schedulers are always non-preemptive and follows a priority based scheduling policy.
- The task priority is always an integer constant.
- The task deadlines are ignored in DSEIR-RASDF similar to DSEIR.

Chapter 6

Translations between DSEIR-RASDF and RASDF

6.1 Introduction

In this work, the notion of DSEIR-RASDF has been introduced for an effective translation from DSEIR to RASDF. The motivation behind using this intermediate representation for the DSEIR to RASDF translation is to enable a 1:1 mapping between the components of DSEIR-RASDF and RASDF, thus leading to a good translation procedure. This chapter explains the translation procedures for the translation of a model in DSEIR-RASDF to an RASDF graph and from an RASDF graph to DSEIR-RASDF model respectively. This chapter is introduced before the translation of the DSEIR model to DSEIR-RASDF model since the translation of a DSEIR-RASDF model to an RASDF graph is easier and simpler than the translation of DSEIR to DSEIR-RASDF.

6.2 Translation from DSEIR-RASDF to RASDF

This section explains the translation procedure from a DSEIR-RASDF model to an RASDF graph with the restrictions and methodologies used for the translation, with respect to each component of the Y-chart.

6.2.1 Application

In the application component of the DSEIR-RASDF model, the different tasks that are atomic for the given level of abstraction are depicted with the appropriate edges between them. These tasks are converted into actors in their corresponding RASDF graphs and the data dependencies in the form of edges are converted to form channels between the actors. The translation procedure is conceptualized in different stages as shown in Figure 6.1.

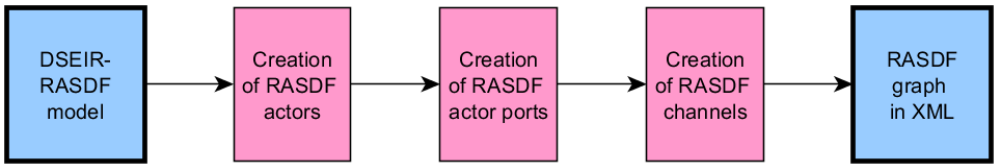

Figure 6.1: The different stages of translation of the application component

During the translation, the DSEIR-RASDF specification is read and the various parameters of the RASDF graph are extracted from the specification using the following steps:

- **Creation of actors:**

  Each task in the DSEIR-RASDF specification is converted to form an actor in the corresponding RASDF graph.

  For each edge on the DSEIR-RASDF specification, the source task of the edge is mapped to a source actor and the actor that contains the destination port is mapped to the destination actor. This will be used later to identify channels in the RASDF graph.

- **Creation of ports:**

  Input ports:
  All the ports on the DSEIR-RASDF specification are mapped as input ports on the RASDF graph.

  Output ports:
  The specification in DSEIR-RASDF does not specify output ports. But a RASDF graph requires output ports as one of its parameters. Hence it becomes vital to create output ports in the RASDF graph based on the information derived form the input ports and edges of a DSEIR-RASDF specification. The information regarding the tokens on the ports and channels in the RASDF graph is also obtained based on the token information in the ports and edges of the DSEIR-RASDF

specification. It is achieved using the following steps:

- All the tasks containing an outgoing edge in the DSEIR-RASDF specification are mapped to form a corresponding output port on the RASDF graph.
- The number of tokens on the input port of a RASDF graph is obtained based on the number of tokens binding to the port in the DSEIR-RASDF specification.
- The number of tokens on the output port of a RASDF graph is based on the value on the edge in the DSEIR-RASDF specification.

- **Creation of channels:**

  In this stage, the channels between the actors are created. The number of tokens in the channel is derived based on the information in the edges and ports. The number of tokens on the output port is equal to the value sent over the edge and the number of tokens on the input ports is equal to the number of tokens that bind to the port in the task. The initial number of tokens existing on each channel, if any is equal to the initial values of the ports in the DSEIR-RASDF specification. The source actor and the destination actor, with their respective ports and rates are then mapped onto the RASDF graph to form channels with the appropriate number of initial tokens.

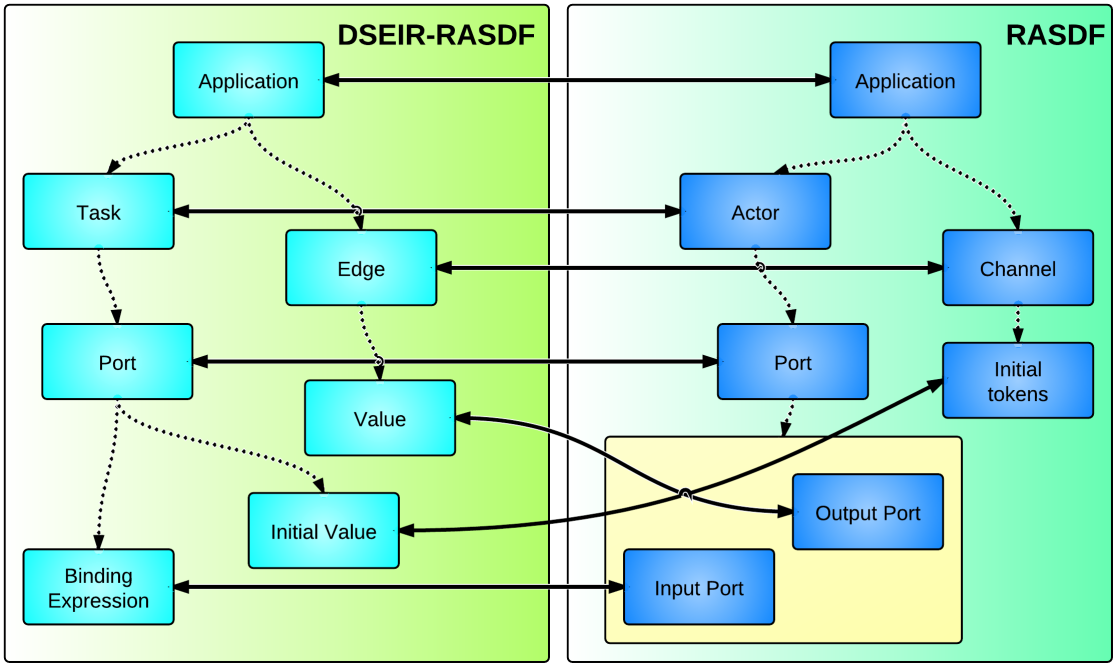

Figure 6.2: An overview of translation of application component

An overview of the components of the DSEIR-RASDF model and its counterparts in the RASDF graph is shown in Figure 6.2. The dotted lines are used to denote the hierarchy of the different components in both DSEIR-RASDF and RASDF while the thick solid arrows indicate the translation of a component in DSEIR-RASDF to its corresponding component in RASDF. The component 'value' in DSEIR-RASDF corresponds to 'constant' atrribute of edge in the class diagram of the application perspective of DSEIR(Figure 2.4). All the other components remain the same.

- **Example:**

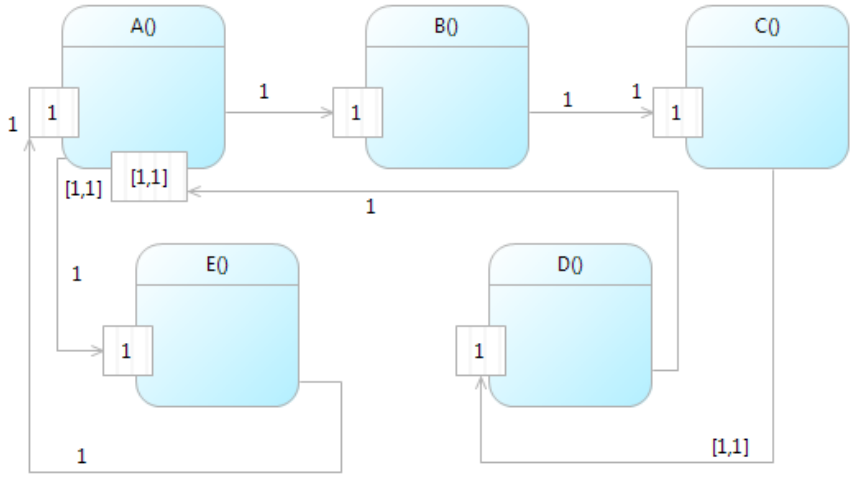

Figure 6.3: The application perspective of a DSEIR-RASDF model

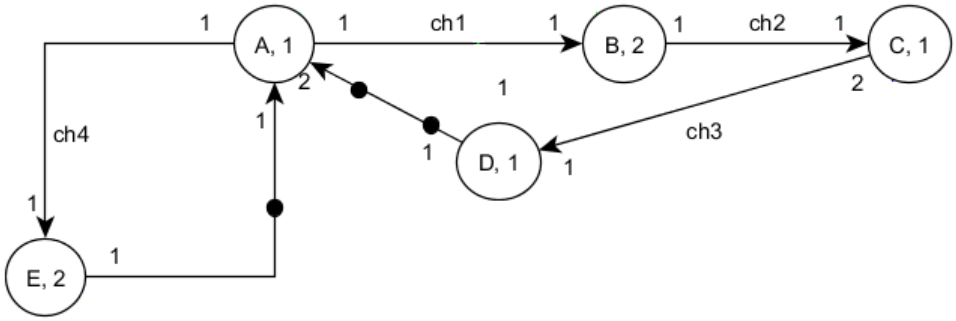

Figure 6.4: The SDF graph as a result of the translation

The Figure 6.3 shows the application perspective of a DSEIR-RASDF model. The binding expressions are values that are expressed inside the ports which are mapped as the rates of the input ports in the SDF graph in Figure 6.4. The term SDF graph is used since the resources are not added yet. The values sent over the edge form the rate of the output ports on the SDF graph, which will then bind to the

values on the port. The execution times of the actor is derived from the load on the task.

6.2.2 Platform

In the DSEIR-RASDF specification, the various resources such as memory, processors etc. are specified using the platform component. This section explains the translation of the specification of resources in DSEIR-RASDF to a RASDF graph. Since all the components required to specify the resources in a RASDF graph already exists in the DSEIR-RASDF specification, this translation is achieved by a simple 1:1 mapping technique. In the current translation, it is assumed that the RASDF graph can handle only two types of resources, namely *Memory* and *Processor* and hence cannot handle arbitrary resources. All other resources belong to one among these resources. Hence the service types, *Internal Storage, Result Storage* and *Storage* are all mapped to form the resource *Memory* in the corresponding RASDF graph. The other restriction on the translation is that the speed of the resource is always equal to one since the speed of the resource cannot be specified in an RASDF graph.

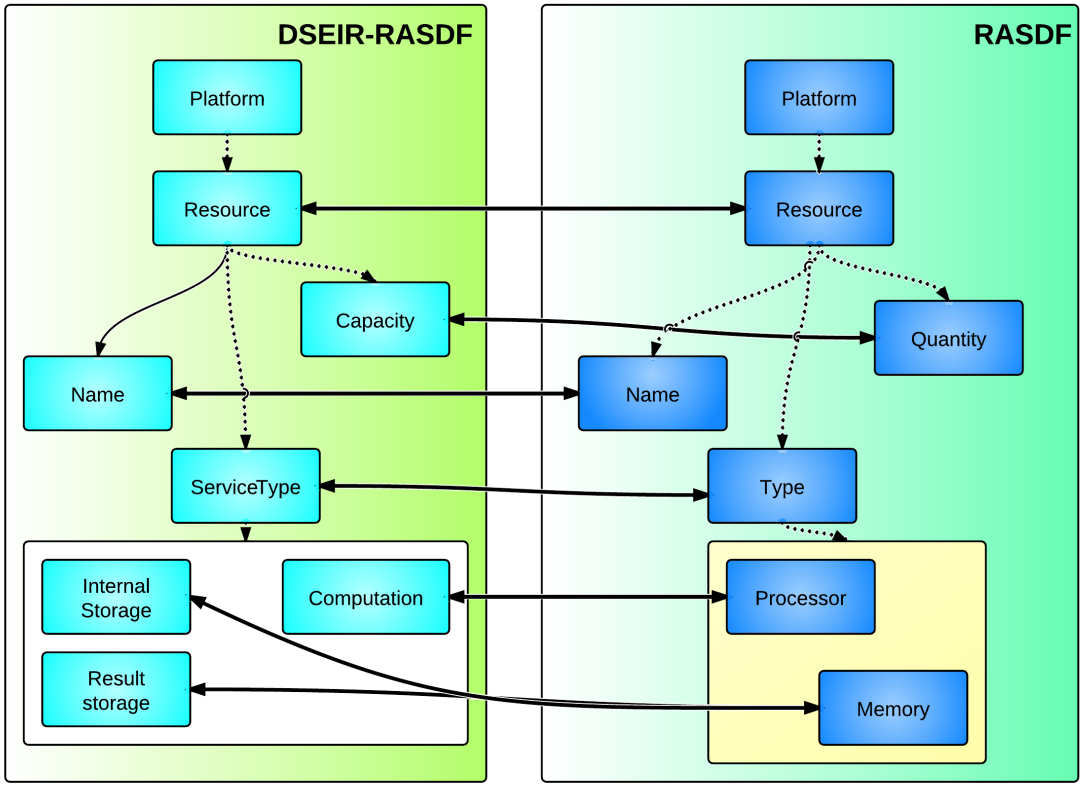

Figure 6.5: An overview of translation of platform component

The simple translation procedure adapted can be explained with the help of Figure 6.5. The platform part of the DSEIR-RASDF specification is scanned and the different properties of the resources are extracted. The important properties are name of the resource, amount of resources available for the model etc. which are then mapped to generate the different resources required for a RASDF graph. Similar to Figure 6.2, the solid lines indicate the translation of the elements in DSEIR-RASDF to elements in RASDF, while the dotted lines indicates the composition of elements in both DSEIR-RASDF and RASDF. The class diagram from Figure 2.10. The different service types that are visible in the class diagram are instantiated to show the mapping between DSEIR-RASDF and RASDF clearly.

Example:

The resource perspective of a DSEIR-RASDF model is shown in Figure 6.6. It consists of three resources, *ACCL, PROC* which provides services of Service type *Computation* and another resource *Memory* which provides services of Service Type *Internal Storage* and *Result Storage* with capacities 1, 1 and 9 respectively. These resources are translated into resources in RASDF as shown in Figure 6.7. In an RASDF graph, the resources are represented as either *Processor* or *Memory*. *ACCL* and *PROC* is of type *Processor* and the resource *Memory* is of type *Memory* in this example.

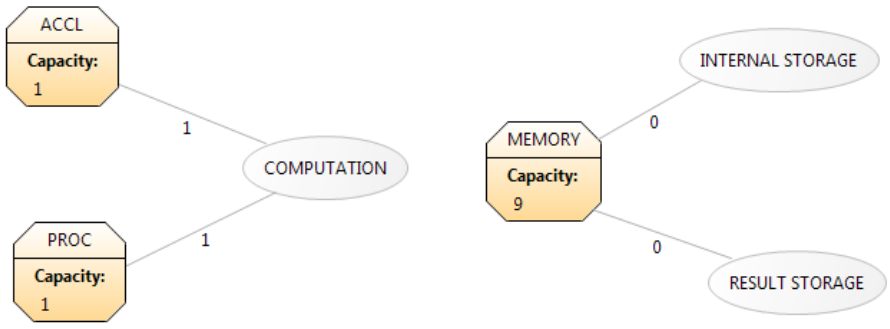

Figure 6.6: Resource perspective of the DSEIR-RASDF model

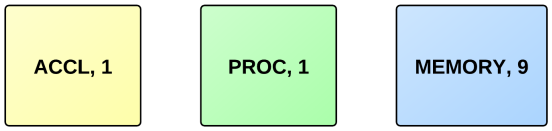

Figure 6.7: Resources in RASDF

6.2.3 Mapping

The mapping component of the Y-chart contains information about the mapping between the different tasks and resources based on schedulers and services. The translation of the mapping component of the DSEIR-RASDF specification to the corresponding RASDF graph is more complicated than the translation of other components to their corresponding counterparts in the RASDF graph. The specification of the mapping component of the RASDF graph is different in structure than the mapping component of the DSEIR-RASDF specification, mainly due to the absence of schedulers in the RASDF graph. In the mapping component DSEIR, the schedulers are specified explicitly but in RASDF graphs, the schedulers are internal to the system and cannot be defined by the user. To accommodate this restriction, we have assumed that the model contains a scheduler for every processor in the system. In the mapping component of RASDF, the mapping between actors and the amount of resources allocated to each actor is specified which implies that both the mapping and scheduling perspective of DSEIR need to be considered for the translation.

The translation procedures are explained in the following steps:

- **Actors and mapping of resources to actors:** In a typical DSEIR specification, the mapping between the tasks and resources is defined in terms of schedulers. These schedulers are assigned to some tasks along with the resources for a particular requested service such as *Computation*, *Internal Storage* or *Result Storage*. But in a RASDF graph, there are no schedulers. The mapping component of the RASDF graph needs information about the amount of resources used by a certain actor in terms of amounts that is claimed and released at the start and end of the firing. Apart from this, the execution time and the priority of every actor needs to be retrieved. The methodologies used to obtain every element of the mapping component is explained in this section.

  Each actor has a specific amount of resource such as the processor or memory associated with it. This information has to be derived from the schedulers in the DSEIR-RASDF specification. The resource assigned to each actor in the RASDF graph is obtained based on the tasks assigned to each scheduler in the mapping component and the resource assigned to each scheduler in the scheduling perspective (platform component) of the DSEIR model. Based on the assigned resources for each scheduler, the amount of resources requested by an actor can be obtained.

  The execution time of a particular actor is derived from the load on the service, *Computation* on each task and the speed of the corresponding resource in the DSEIR-RASDF specification. Since the speed is always assumed to be one, the execution time of the actor is always the load on the service type *Computation* on each task in the DSEIR model. If the speed of the resource is greater than one,

then the execution time is the product of load and speed.

- **Claim and release of resources:**

  The amount of services claimed and released for a particular task depends on the amount of load on the different services in the DSEIR-RASDF model. It also depends on the handovers that exists between tasks.

  We define three sets called *Computation, Internal Storage* and *Result Storage* for the corresponding services, *Computation, Internal Storage* and *Result Storage*. Since it is possible to define user-defined services in the DSEIR language, all the other services in the toolset and the user-defined services belong to these sets. A handover can exist only for the service, *Result Storage*. These restrictions are added since the claim and release exists for a particular service in DSEIR-RASDF whereas it is for a particular resource in the RASDF graph. It was also added based on the assumption that one resource can provide service of only one service type.

  Handover also plays an important role in the claim and release of resources. If there is a handover between two consecutive tasks, then the translation method considers that the preceding task claims the resource and the succeeding task releases the resource. But if the handover is not between two consecutive tasks, an edge is added between the two tasks with a token that is sent along the edge. This extra edge is added to prevent the resource from being claimed by a third actor when it is supposed to be claimed by the succeeding task in the handover. Hence, to summarize the claim and release of resources in an RASDF graph, the claim of a particular memory resource depends on the load on the services *Internal Storage* and *Result storage* and if there is a handover to this actor from any preceding tasks. The release of the memory resource also depends on the load on the services *Internal storage* and *Result storage* and if there is a handover to a subsequent actor. The claim and release of the computation resource is usually of the same amount and depends on the load on the service, *Computation*.

- **Priority of actors:**

  In the RASDF specification, the priority of all the actors is specified by naming the actors in an ordered list that denotes decreasing order of priority, without any indication in terms of numbers for the priorities. Hence during the translation, the priorities of the actors are retrieved from the mapping component and the actors are sorted according to their priorities from the actor of the highest priority to the actor of the lowest priority and specified in the graph accordingly.

An overview of the translation is presented in Figure 6.8 where the solid lines depict the translation and dotted lines depict the structure of the different elements of DSEIR-

RASDF and RASDF.

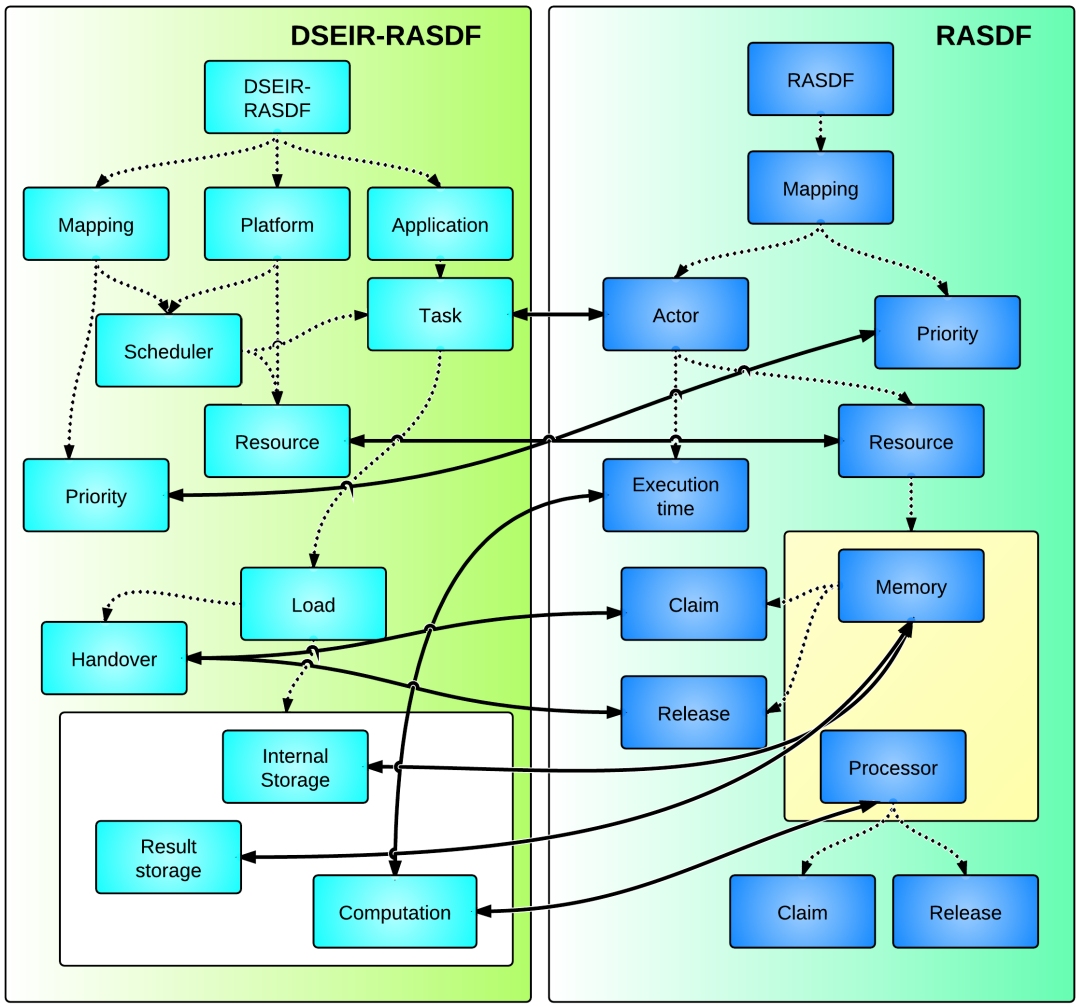

Figure 6.8: An overview of translation of Mapping component

- **Example:**

  We consider an example and show the translation from the DSEIR-RASDF model to an RASDF graph. For the sake of simplicity, we have skipped the application perspective of the DSEIR-RASDF model.Figures 6.9, 6.10, 6.11 and 6.12 show the DSEIR-RASDF model using the different perspectives of the DSEIR editor. There is a handover between tasks C and E (Figure 6.9)that needs to be translated. The resulting RASDF graph is shown in Figure 6.13. The claim and release of the resources is denoted by the dotted arrows between the actors and resources. The arrow head at the resource end denotes the amount of resource claimed and the

arrow head at the end of the actor denotes the amount of the resource released by the actor. We can observe that all the actors are mapped to the appropriate resources based on analysis of the scheduling and mapping perspective in DSEIR-RASDF.

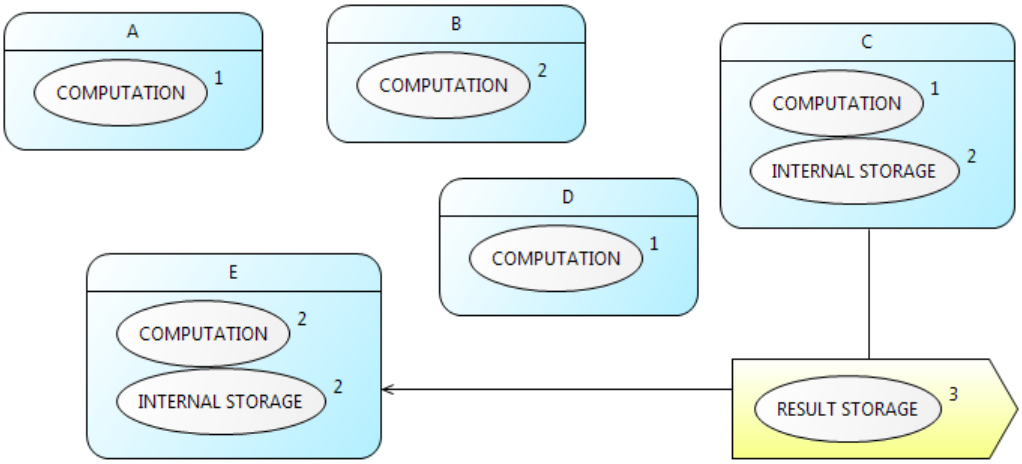

Figure 6.9: Load perspective

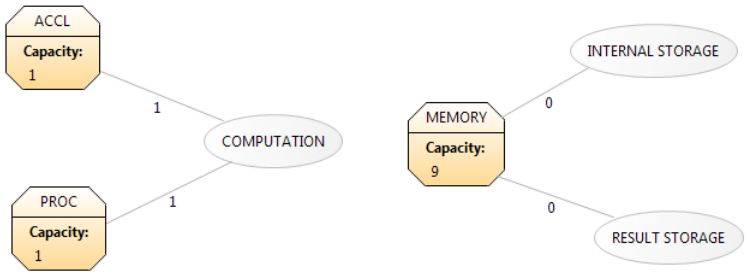

Figure 6.10: Resource perspective

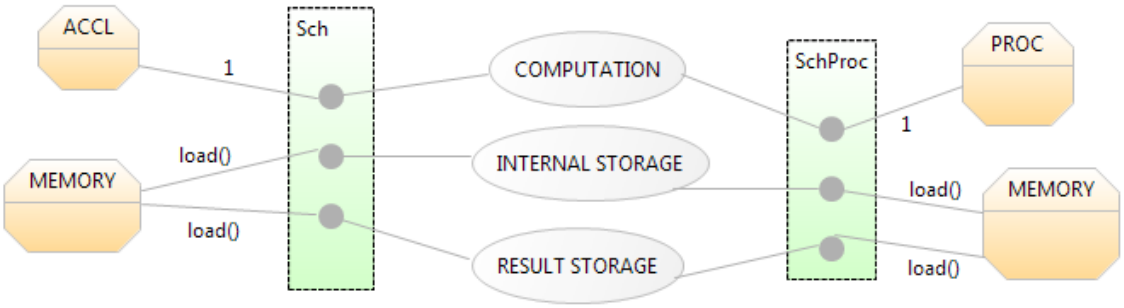

Figure 6.11: Scheduling perspective

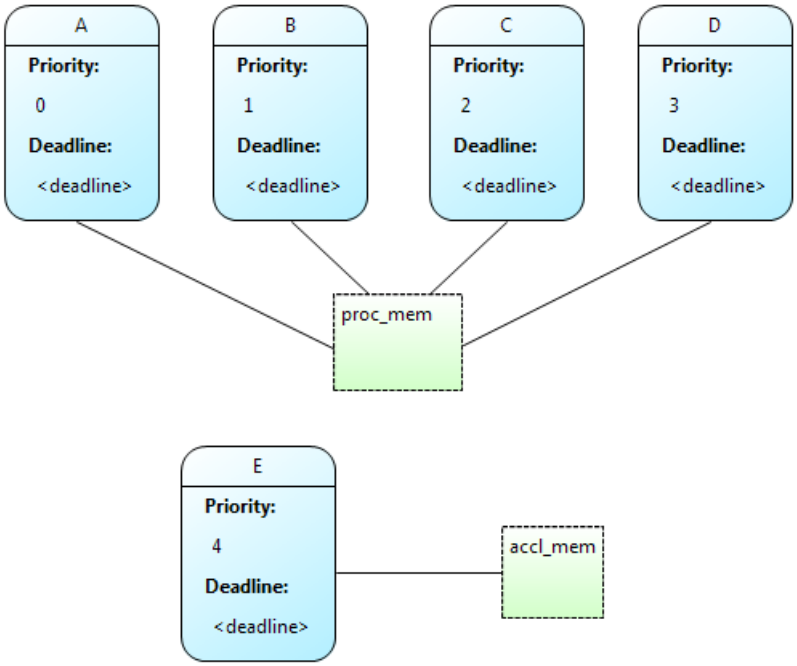

Figure 6.12: Mapping perspective

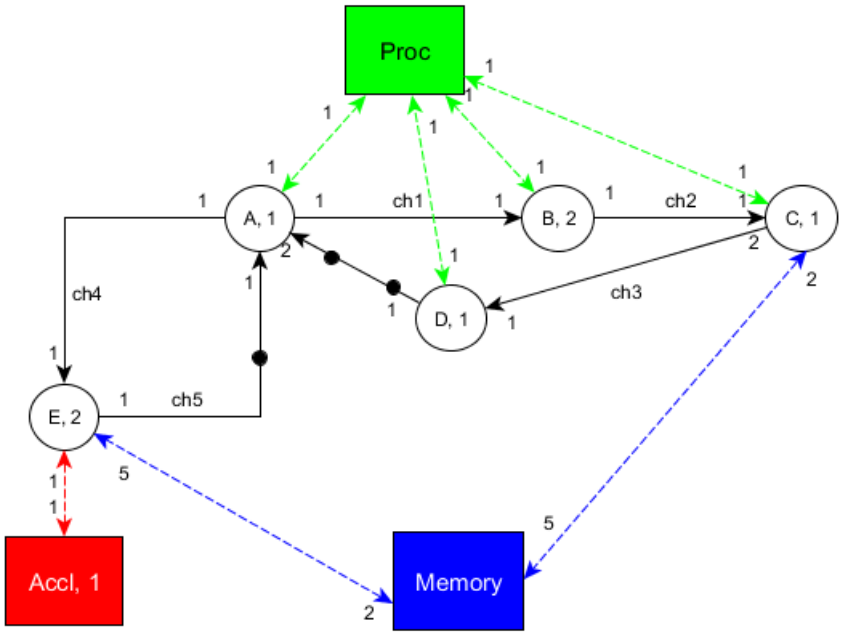

Figure 6.13: Resultant RASDF

6.3 Translation from RASDF to DSEIR-RASDF

In this section, a new translation procedure is introduced. This method of translation translates an RASDF graph to a DSEIR-RASDF specification. The primary reason behind the implementation of this translation procedure is to verify the correctness of the translation procedure of a DSEIR-RASDF specification to a RASDF graph. With the help of this translation procedure, any RASDF graph is converted to a DSEIR-RASDF specification which is then converted back to a RASDF graph using the translation procedure from the previous section. This RASDF graph is then compared manually with the initial graph for the correctness of the translation. The translation procedure of an RASDF graph to the corresponding DSEIR-RASDF model consists of the following steps as shown in Figure 6.14:

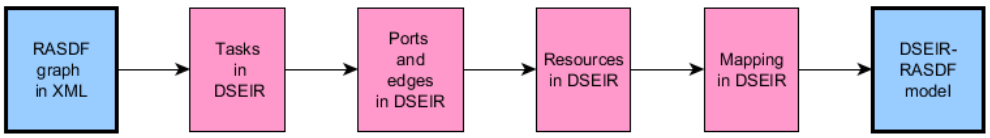

Figure 6.14: The different stages of translation of an RASDF graph to DSEIR-RASDF

6.3.1 Application

Creation of tasks:

Each actor in the RASDF graph is mapped to form a task in the DSEIR-RASDF model. These tasks does not contain parameters.

Load on tasks:

There is also a certain load defined on the service, *Computation* for each task in the DSEIR-RASDF model. This information is derived from the execution time of the actor and the speed of the resource (which is assumed to be always equal to one). The load on other Service types, *Internal Storage* and *Result Storage* need to be derived too. The load on the service type *Result Storage* also depends on handovers. In this thesis, we have assumed that the handover will always exist for *Result Storage* only. This is taken care by the method, *ConformanceChecker*. In this translation, the handover between tasks is always detected based on the amount of resources claimed and released by the resource *Memory*. But since an RASDF specification does not mention handovers explicitly, it becomes difficult to detect and translate handovers.

In order to solve this issue, an algorithm that detects handover in RASDF is created. This algorithm calculates the amount of handover required based on its neighbour's

information. The handover is detected at each actor. The difference between the amount of resources claimed and released is calculated and stored for every actor. It is called value of the actor. It is illustrated using the example from Figure 6.15. In the figure, there are three actors using the resource *Memory*, and the amount of resources claimed and released are depicted by the arrows between them and the resource. The algorithm is given as shown in Algorithm detectHandover.

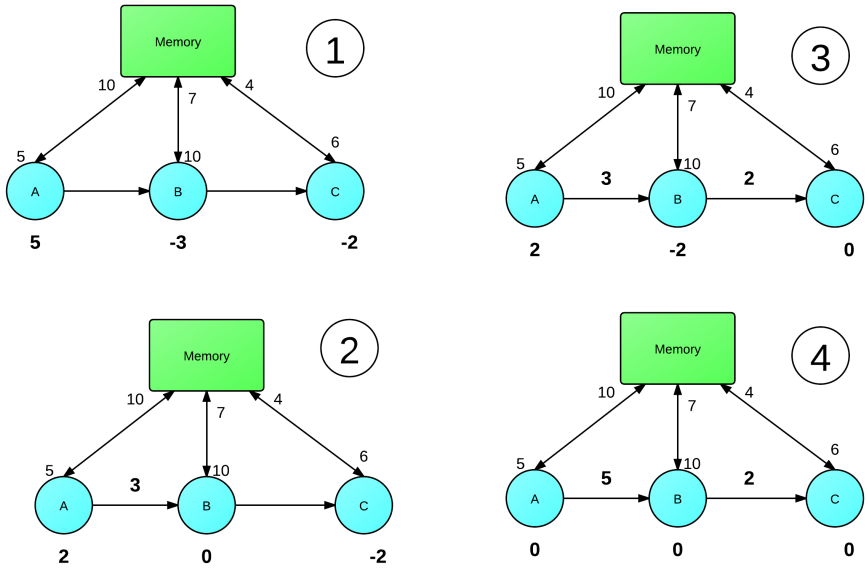

Figure 6.15: Detection of handover in RASDF

```
detectHandover(RASDFModel model)
 1  Set<Actor> actorSet ← getActors(model)
 2  for (All actor ∈ actorSet)
 3  do Actor ↦List<Integer> actorMap
 4      Integer value ← claim(memory) − release(memory)
 5      actor ↦ [value, 0]
 6      end for
 7  while (checkActorValue(actorSet))
 8  do Actor preActor ← actor
 9      actor ← getNext(actorSet, actor)
10      Integer handover ← −(getValue(actor))
11      value ← (handover + getValue(actor))
12      actor ↦ [value, handover]
13      Integer valuePreActor ← getValue(preActor) − handover
14      precActor ↦ [valuePreActor, handover(preActor)]
15      end while
```

The values are stored in a table and is constantly updated until the value of all actors is zero and is shown in Table 6.1. If the value of any actor is negative, the preceding actor is checked for available handover and there is a handover from the preceding actor to the succeeding actor. The handover value is subtracted from the node giving the value of the handover to the task and this procedure is repeated until the value on all the nodes is greater or equal to zero.

Table 6.1: Different steps of the method

Actor	**value**	**handover**
A	5	0
B	-3	0
C	-2	0
A	5	0
B	-5	0
C	0	2
A	0	0
B	0	3
C	0	2

Adding ports and edges:

Translating the ports and channels to form ports and edges on the DSEIR-RASDF model is relatively easy. All the input ports in RASDF are translated to form the corresponding ports on the tasks. The rates on the ports are translated to form the corresponding tokens binding to the port. The output ports are not translated since DSEIR-RASDF models do not contain output ports on their tasks. The channels in the RASDF graph are used to create edges in the DSEIR-RASDF model. The edges are created from the the source task and the destination port as the edges are between tasks and ports in the DSEIR-RASDF model. The initial tokens on the channels are translated to form initial values of the ports. These values are always constant integer values and are not expressions.

The overview of the translation of the application perspective of the DSEIR-RASDF is the same as Figure 6.2 if the entire block of RASDF and DSEIR-RASDF is swapped and hence is not shown again.

6.3.2 Platform

From the specification of the RASDF graph, the information about the resources can be derived directly. This is obtained from the platform part of the RASDF graph along

with all the necessary parameters such as name, quantity, type of resource etc. to create the platform component of the DSEIR-RASDF model.

The overview of the translation of the resource perspective is also the same as Figure 6.5 when the entire block of RASDF and DSEIR-RASDF is swapped and hence is not shown again.

The scheduling component of the DSEIR-RASDF model contains a mapping from resources to definite amounts of service types. Since there is no external indication of the schedulers, we have assumed that each processor has a scheduler during the translation. The translation of the *Processor* type of resource is translated to form a resource that provides the Service type *Computation*. All the *Memory* type of resource is translated to form *Internal Storage*. If there is a handover in the graph, then the resource will also provide the Service type *Result Storage*. But detecting handover is not always trivial as discussed.

6.3.3 Mapping

In the DSEIR-RASDF model, the mapping component contains a mapping from different schedulers to their corresponding tasks. Since the RASDF graph contains no information about the mapping to the schedulers, the scheduler that has been created for each processor is assigned to the task that requests the processor. The creation of schedulers is possible based on the assumption that every actor requests the service of a processor but might not always use a memory. The priorities between the actors are retrieved in the order that they are specified in the graph to assign priorities of tasks in decreasing order.

This RASDFtoDSEIR-RASDF translation procedure is used for testing purposes. This is achieved by translating the resultant DSEIR-RASDF model back into an RASDF graph through the translation procedure, DSEIR-RASDFtoRASDF and verifying yhe graphs manually.

Example: The example shows the same example used in Section 6.2. The resultant RASDF graph (Figure 6.16) is used an example and translated to a DSEIR-RASDF model. The resultant DSEIR-RASDF model (Figures 6.17, 6.18, 6.19, 6.20, 6.21) is compared with the initial model (Figures 6.3, 6.9, 6.10, 6.11, 6.12). We can see that there is a difference in the application perspective (Figures 6.17 and 6.3) of the model. An extra edge is added as a result of the handover between tasks C and E. This does not affect the behaviour of the model since the succeeding task is always enabled as soon the preceding task finished execution.

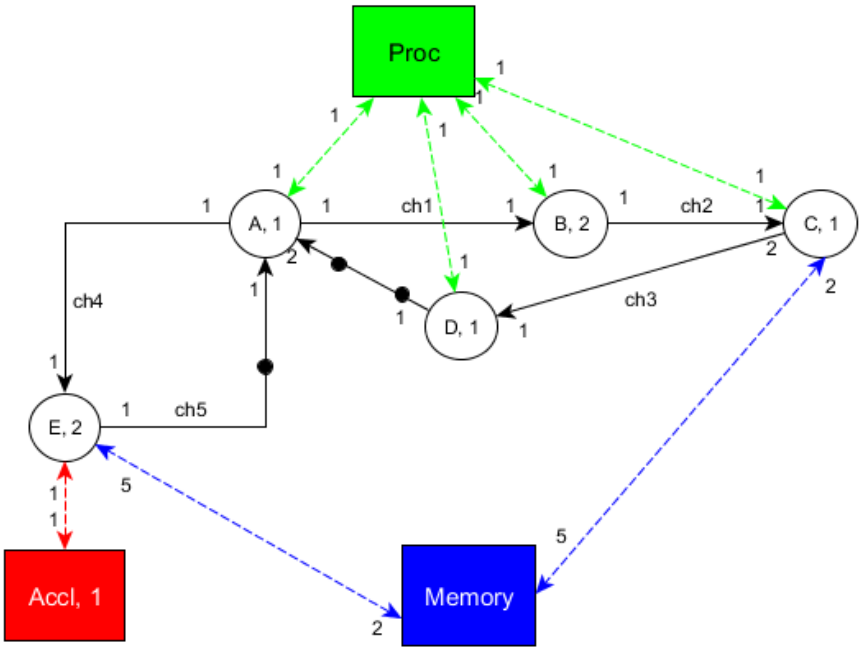

Figure 6.16: The RASDF graph

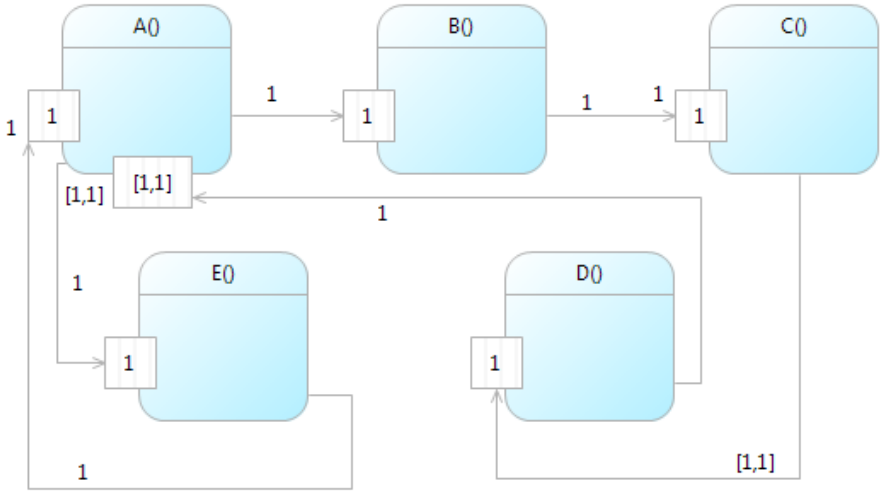

Figure 6.17: Resulting application perspective

6.4 Summary

The translation of an RASDF graph to a DSEIR-RASDF model is accurate as seen from the examples and the translation of DSEIR-RASDF to RASDF graph using the model from the previous translation results in the same RASDF graph in which the

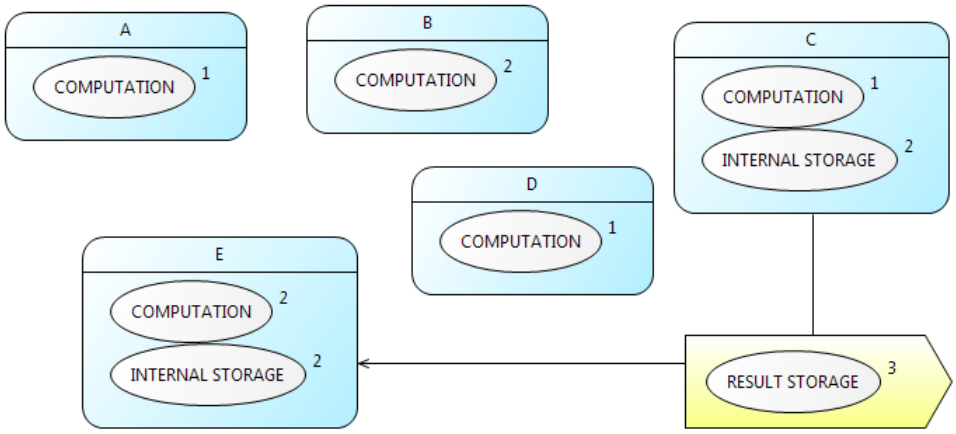

Figure 6.18: Resulting load perspective

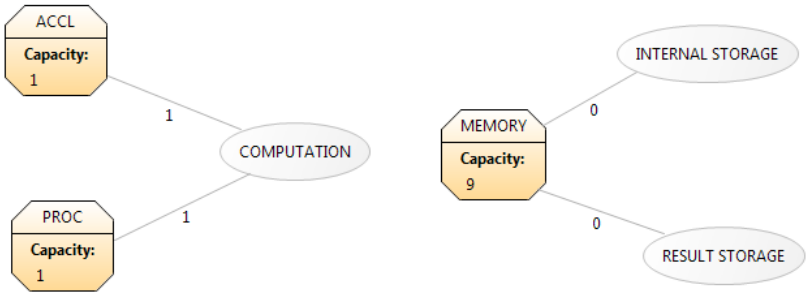

Figure 6.19: Resulting resource perspective

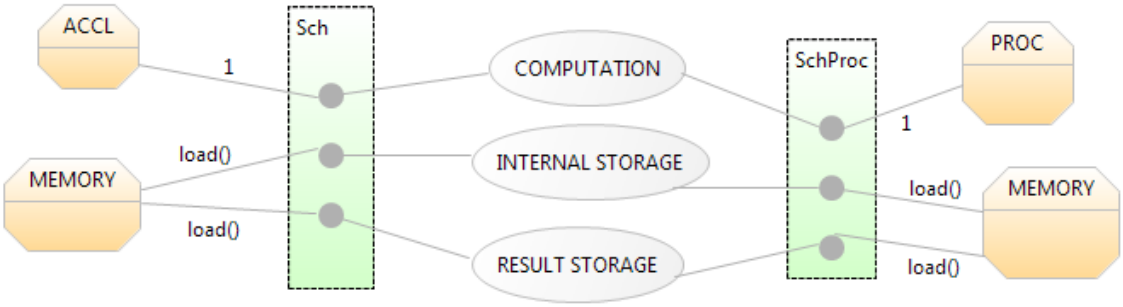

Figure 6.20: Resulting scheduling perspective

throughput remains the same. The only drawback is that the RASDF graph contains no information about the schedulers which is an essential part of the DSEIR-RASDF model. So the translation from DSEIR-RASDF to RASDF is based on the assumption that each processor has a unique scheduler which may not be true in all the models. But since there is no way to obtain any information about the schedulers from the RASDF

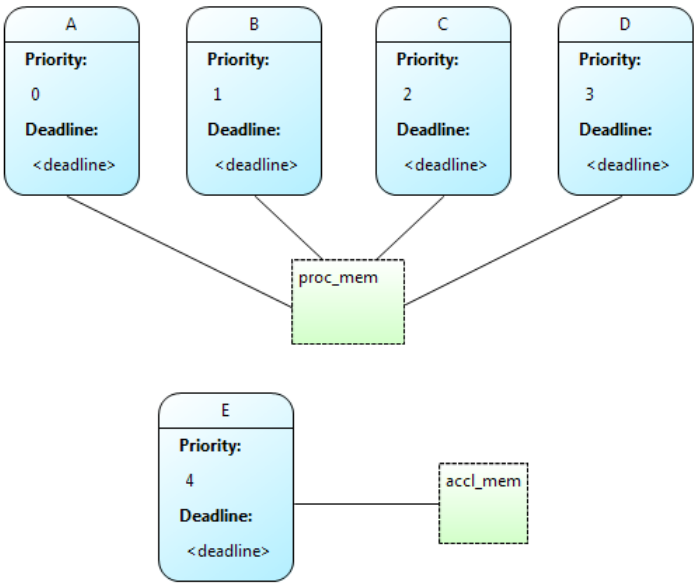

Figure 6.21: Resulting mapping perspective

graph, this would be the most appropriate translation possible based on the information available. The schedulers along with their associated resources may be further analyzed to combine two or more schedulers such that the overall behavior of the system remains the same w.r.t. throughput. This issue is not addressed in the current translation.

Chapter 7

Translation from DSEIR to DSEIR-RASDF

7.1 Introduction

DSEIR-RASDF is a restrictive model that does not capture all the features of the DSEIR model but is capable of representing an RASDF graph without any loss. In this section, we try to address the various issues in the translation of the DSEIR model to the corresponding DSEIR-RASDF graph and provide methodologies that have been used to solve the issues. By solving the issues, we attempt to increase the size of the subset of DSEIR that can be translated into an RASDF graph such that throughput properties hold before and after translation. This is achieved by an abstraction of the different components of the DSEIR model to an appropriate DSEIR-RASDF model, thus increasing the number of models that can be translated.

7.2 Translation procedures and issues encountered

This section addresses the different issues that need to be solved during the translation of a DSEIR model to an RASDF graph. Since DSEIR is a highly expressive language as compared to an RASDF graph, the RASDF graph obtained as a result of the translation may not contain all the features that have modeled in the DSEIR model. This research aims at a conservative translation of the model that tries to preserve the timing properties of the model w.r.to throughput. The term *conservative translation* implies that the model exhibits the same behavior in terms of the throughput before and after translation. If TD(M) is the throughput of the model before translation, and TR(M) is the

throughput of the model after translation, then the translation of the model should be that TD(M) $\geq$ TR(M). This is called conservative translation of a model w.r.to timing parameter such as throughput.

The different issues that need to be solved are:

- Data dependent parameters
- Variable load/ actor execution times
- Data dependent choices
- Data dependent loops
- Static and dynamic scheduling

The remainder of this section gives a detailed explanation of why the different issues need to be addressed and how they are solved within the context of this research.

7.2.1 Data dependent parameters

The DSEIR models may contain task parameters. But the RASDF graph cannot handle these parameters. Hence the possible range of the parameters need to be extracted for an efficient translation. Each task in the DSEIR model can contain one or more parameters that can receive different values for every token. This occurs at every instance of the task based on the binding expression at every port. The parameter may also be mapped to the values sent over the edge between ports and may thus change the pattern of execution of a certain instance of the model. This has been explained in Chapter 2. But from the definition of an RASDF graph in Chapter 3, it is evident that an actor cannot handle parameters. Hence for the proper modeling of the system and to achieve the goal of translation, it becomes necessary to represent the RASDF graph with a proper abstraction and representation of these parameters in the actors of the graph.

In this research, the abstraction of the parameters is achieved by extracting the possible range of the parameters and is shown in Figure 7.1. It contains two components *DataAbstraction* and *ExpressionEvaluator*. The range of the parameters are affected by various other parameters such as guard conditions at the ports, edges and tasks, binding expressions etc. A method *DataAbstraction* that evaluates all the conditions that affect the parameters has been developed and this method returns the range of the parameters for the instance of the model. The Figures 7.2 and 7.3 shows the DSEIR model and the resulting DSEIR model after the abstraction of the parameters using the method *DataAbstraction*. It has been developed by a colleague working in the Octopus project [17] and hence will not be discussed in the context of this thesis. The structure of the

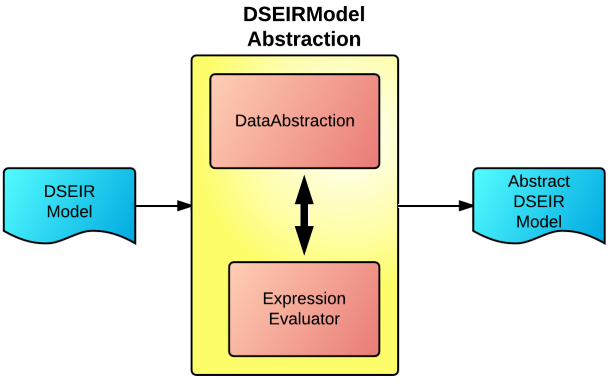

Figure 7.1: DSEIRModelAbstraction

resulting DSEIR model after abstraction is shown in Figure 7.4. In the DSEIR model, the parameters in a task can also affect other components of the model such as the load on the task. The possible values of these components need to be extracted based on the range of the parameters and it has been achieved using a method called *ExpressionEvaluator*. These two methods form the method *DSEIRModelAbstraction* in the component DSEIRtoRASDF (Figure 4.2).

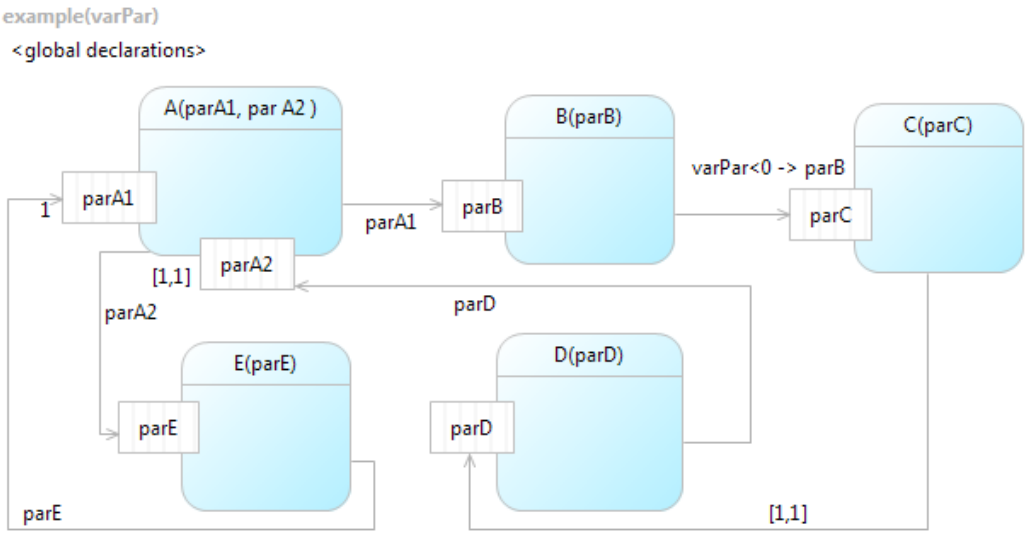

Figure 7.2: DSEIR Model with task parameters

In this section, we will discuss the working of the method *ExpressionEvaluator* and how the method has been used to solve the different types of expressions that are possible in the DSEIR language.

The different form of expressions that are specified in the DSEIR modeling language need

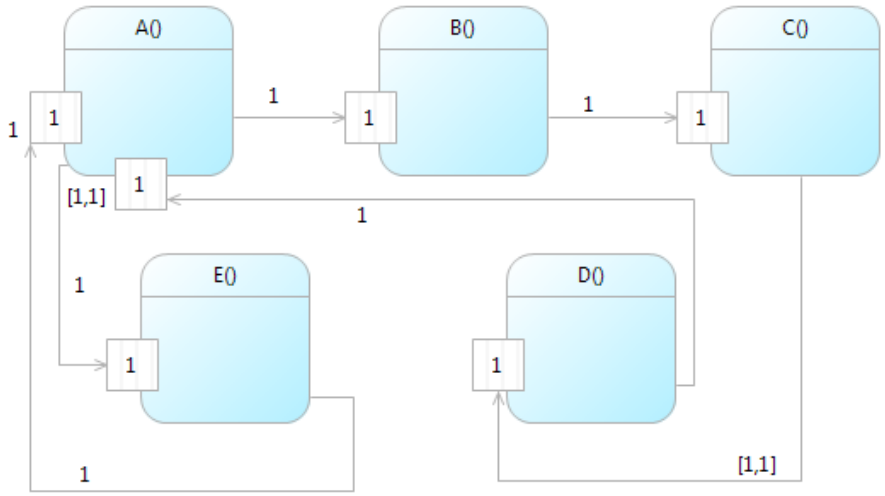

Figure 7.3: DSEIR Model without task parameters

to be evaluated since the load of a task could depend on the results of these expressions. This section will explain the different types of expressions and how they are handled in the translation. In the DSEIR modeling language, the type of expressions that can be handled are mostly pre-defined and user-defined expressions cannot be handled in the current version. If there is a user-defined expression that cannot be handled by the translation method, the method *Conformance Checker* will generate an error message. The different types of expressions that are allowed in DSEIR are:

- Integer expressions
    1. Addition
    2. Subtraction
    3. Multiplication
    4. Division
    5. Modular division
- Comparison operations
    1. Equal to
    2. Not equal to
    3. Greater than

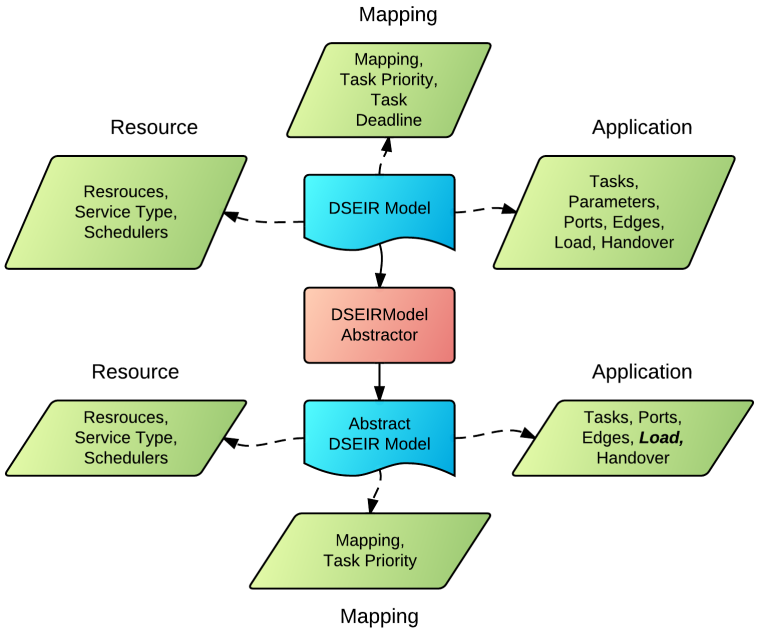

Figure 7.4: DSEIR models before and after translation

4. Greater than or equal to
5. Less than
6. Less than or equal to

- Boolean operations
  1. not
  2. and
  3. or

The key constraint in realizing this method was evaluating the expressions over intervals since the extraction of parameter ranges will always result in intervals since we consider only the boundary values in this thesis, which is an overapproximation. These range of values imply that the boundary values are the only values that are considered and hence the range of values is merely a set of two values. Evaluation of expressions over intervals which overlap with each other cannot be easily evaluated. To facilitate this evaluation better, we developed the method using three-valued logic that evaluates any expression and returns binary values True (1), False (0) and a third indeterminate value in-between, say Unknown (0.5) when the expression cannot be evaluated due to reasons

such as overlapping intervals. The results of the evaluation of different types of expressions are explained below:

Integer expressions:
The different integer operations that are defined in the DSEIR modeling language are handled efficiently to retrieve the range of the results. The basic arithmetic operations can be handled without any issues since the operations for an interval are extensively defined in various literatures. The actual difficulty is in handling the comparison operators such as greater than, less than etc. as they cannot be defined for overlapping intervals.

- **Basic Arithmetic operations:**

  Addition: Finding the range of the addition of two intervals is rather easy. The minimum of the operation is the minimum of the two intervals to be added while the maximum value is given by the maximum value of both the intervals respectively. The addition of two intervals (minL, maxL) and (minR, maxR) is given by the formula:

$$[minL, maxL] + [minR, maxR] = [minL + minR, maxL + maxR]$$

  Subtraction: Deducing the interval of subtraction between two range of values is not as straightforward as addition but can be deduced easily. It is given by the formula:

$$[minL, maxL] - [minR, maxR] = [minL - maxR, maxL - minR]$$

  Multiplication: The range of multiplication can be easily calculated just like the addition operations. The maximum of the multiplication is the product of the maximum of both the intervals and the minimum is the product of the minimum of the two intervals respectively. It is given by the formula:

$$[minL, maxL] \times [minR, maxR] = [minL \times minR, maxL \times maxR]$$

  Division: This is similar to subtraction and can be given by the formula:

$$[minL, maxL]/[minR, maxR] = [minL/maxR, maxL/minR]$$

  Modular Division: This is a bit more complex than the rest of the arithmetic operations and hence to avoid any complications, we always take the minimum and maximum of all the possibilities on the intervals. The formula is given by:

$$[minL, maxL]\%[minR, maxR]$$
$$= [max(minL\%minR, minL\%maxR, maxL\%minR, maxL\%maxR),$$
$$min(minL\%minR, minL\%maxR, maxL\%minR, maxL\%maxR)]$$

Examples for the evaluation of Arithmetic expressions:

```
Addition:
[1,2] + [3,4] =  [4,6]

Subtraction:
[1,2] - [3,4] = [-3,-1]

Multiplication:
[1,2] * [3,4] = [3, 8]

Division:
[1,2] / [3,4] = [0, 0]

Modular division:
[1,2] % [3,4] = [1,2]
```

- **Comparison operations:**:
  The different comparison operators that are expressed in the expressions of DSEIR are greater than, greater than or equal to, less than, less than or equal to, equal to and not equal to. It is simple to draw a conclusion in the form of a boolean value about these operations when the intervals under consideration are not overlapping with each other. When the intervals overlap with each other, we use the three-valued logic as discussed earlier to deduce the possible range of the interval after the corresponding operation. Since the results of these operations is always three-valued, this section will now explain how the conclusion for different operations are derived based on the various possibilities of overlapping between the two intervals.

  The same procedure is used to for all the evaluation of all operations in this section. The results of the operations are always either of the three values, True, False or Unknown. The results of the operation is said to be True or False based on the conditions when the intervals are non overlapping as shown in Figure 7.5 where the grey and red coloured boxes are two non-overlapping intervals (for example, [2,5] and [8,10]), since a clear conclusion can be derived in that case. When the interval ranges to be evaluated overlap each other as shown in Figure 7.6 (for example, [2,8] and [5,12]) and Figure 7.7 (for example, [2,12] and [4,8]), a judgment about the result cannot be made and hence it is always set to the value, Unknown.

Figure 7.5: Non overlapping intervals

– **Greater than:** There are three cases that have to be considered and conclusions for the results are derived.

When the intervals do not overlap with each other:
As seen from Figure 7.5, when the intervals do not overlap, a clear judgment can be made about whether a given interval (grey) is greater than the other interval (red) or not. If it is greater, then the value true is returned. Else, a false is returned. This is ensured by comparing the maximum value of the interval in the left (grey) with the minimum of the interval in the right(red).
When the minimum of an interval overlaps with the maximum of another interval:
As seen from Figure 7.6, if the maximum value of one interval is greater than the minimum of the other interval, a conclusion about whether either of the two intervals is greater than the other cannot be made and hence the value is always set to Unknown.
When the range of an interval is within the range of another interval:
As seen from Figure 7.7, if the range of an interval is within the range of another interval, then a conclusion about whether the interval is greater than the other interval cannot be derived and hence the value Unknown is returned.

Figure 7.6: Overlapping intervals

Figure 7.7: Overlapping intervals

– **Greater than or equal to:** A reasoning similar to the *greater than* operation is made to derive results of the greater than or equal to operation. The additional condition is that if the intervals overlap with each other completely, i.e. are equal to each other, then the result is also set to True, along with the

results derived by reasoning using the three cases from Figures 7.5, 7.6 and 7.7.

– **Less than:** The reasoning for deriving the results for *less than* is the same as *greater than* operation with the three different cases.

– **Less than or equal to:** A reasoning similar to the *less than* operation is made to derive results of the *less than or equal to* operation. The additional condition is that if the intervals overlap with each other completely, i.e. are equal to each other, then the result is also set to True, along with the results derived by reasoning using the three cases from Figures 7.5, 7.6 and 7.7.

Figure 7.8: Equal intervals

– **Equal to:** The reasoning for *equal to* is slightly different than the previous operations like greater than, less than etc. The results require a few additional cases to derive a conclusion about whether two intervals are equal or not.

When the intervals do not overlap with each other:
It is clear from Figure 7.5 that the two intervals are not equal to each other and hence the value False is always returned in this case.
When the intervals overlap each other:
As seen from Figure 7.8 (for example [2,5] and [2,5]), this is the case when the two intervals can be equal to each other. The value Unknown is always returned in this case, since it might not be true if all the actual values within the range are covered.
When the minimum of an interval overlaps with the maximum of another interval:
The situation from Figure 7.6 clearly indicates that a conclusion cannot be made about whether the two intervals are equal to each other or not. The value Unknown is returned in this case.
When a boundary of the two intervals are equal to each other, but the other boundary is not:
This is an extended version of the previous case. In this case, either of the minimum or the maximum of both the intervals are equal to each other but the maximum or minimum are not equal respectively (for example, [2,5] and [2,8] or [6,9] and [5,9]) as seen from Figure 7.9. In this case, the value Unknown is always returned since it cannot be said whether the two intervals are equal to each other or not.

When the range of an interval is within the range of another interval:
This is the case seen from Figure 7.7 and it is quite evident that a conclusion about whether the two intervals are equal cannot be made and hence the value Unknown is always returned in this case.

Figure 7.9: Overlapping intervals for *equal to*

– **Not equal to:** This operation has a reasoning similar to *equal to* operation with five different cases. When the intervals are non-overlapping (Figure 7.5), the value True is returned since it can be clearly argued that the two intervals are not equal to each other. For all the other cases, the value Unknown is returned similar to *equal to* operation since a conclusion cannot be derived.

The results of all the operations can be consolidated into a table. It is shown in Table 7.1.

Operation	**Non Overlapping intervals**	**Minimum of one less than maximum of the other**	**Interval within another interval**	**Equal minimum or maximum but not both**	**Equal ranges**
Greater than	True/ False	Unknown	Unknown	Unknown	Unknown
Greater than or equal to	True/ False	Unknown	Unknown	Unknown	True
Less than	True/ False	Unknown	Unknown	Unknown	Unknown
Less than or equal to	True/ False	Unknown	Unknown	Unknown	True
Equal to	False	Unknown	Unknown	Unknown	Unknown
Not equal to	True	Unknown	Unknown	Unknown	Unknown

Table 7.1: All comparison operations and their results

Examples for the evaluation of a few cases of the comparison operators:

```
Equal to:

[1,2] == [1,2] = True
[1,2] == [2,5] = False

Greater than
[1,5] > [2,4] = Unknown
[1,5] > [6,9] = False
```

Boolean operations:

The Boolean operations are also evaluated using the three-valued logic to maintain consistency during evaluation of the expressions since they could contain results of a comparison operation. The different types of Boolean operations that are allowed in DSEIR are *not*, *and* and *or*. The truth tables of evaluations of these operations for three-valued logic are well-defined in literatures and the same results have been used here. This section will now explain how the different operations are evaluated in the method *ExpressionEvaluator*.

1. **Not:** The truth table of the boolean operator *not* for the three- valued logic is simple. It is given by:

$$not(\text{True}) = \text{False}$$

$$not(\text{False}) = \text{True}$$

$$not(\text{Unknown}) = \text{Unknown}$$

2. **And:** The truth table of the *and* operation is given by Table 7.2. The total number of possible cases has been reduced due to symmetry of the Boolean operator. The Boolean logic *and* operation has been extended to accommodate three-valued logic *and* operation. The *and* operator returns the minimum of the result for every combination for the values of True (1), False (0) and Unknown (0.5).

3. **Or:** The results of the *or* operation is given in Table 7.3. It is an extended version of the *or* operation in Boolean logic. The *or* operation returns the maximum for every combination of the different values in the truth table.

Example:

The examples in Listings 7.1 and 7.2 contain two examples that depict the approach used to solve the expression in a parameter *P*. In the first example, the *IF* condition

A	B	And (A, B)
True	True	True
True	False	False
True	Unknown	Unknown
False	False	False
False	Unknown	False
Unknown	Unknown	Unknown

Table 7.2: Results of *and* operation

A	B	Or (A, B)
True	True	True
True	False	True
True	Unknown	True
False	False	False
False	Unknown	Unknown
Unknown	Unknown	Unknown

Table 7.3: Results of *or* operation

results to Unknown since a conclusion cannot be derived about whether the range [1,8] is less than 5 or not. Hence both the conditions are chosen and evaluated returning a range of [2, 48] is derived for the ranges [6, 16] and [2, 48].

In the second example, since the *IF* condition returns true only the *TRUE* statement is evaluated and the range of [6,16] is returned.

```
if(P<5)
    then P + [5,8]
    else P * [2,6]


Assume P = [1,8]

if([1,8]<5)
    then [1,8] + [5,8]
    else [1,8] * [2,6]


(Unknown)
    then [6,16]
    else [2,48]


FINAL RESULT = [2, 48]
```

Listing 7.1: Example of an expression (1)

```
if(P<10)
    then P + [5,8]
    else P * [2,6]

Assume P = [1,8]

if([1,8]<10)
    then [1,8] + [5,8]
    else [1,8] * [2,6]

(True)
    then [6,16]
    else [2,48]

FINAL RESULT = [6, 16]
```

Listing 7.2: Example of an expression (2)

7.2.2 Variable load/actor execution times

The DSEIR models are capable of handling variable loads on their tasks. The value of these loads could depend on parameters of a task or could be based on different expressions that can be specified in the DSEIR model. But the limitation is that the RASDF graphs cannot handle variable loads as seen in Chapter 3 and effective methods need to be developed to extract these values for the guaranteed worst case throughput of the system after the translation of a DSEIR model. This problem has been handled in the method *DSEIRtoDSEIR-RASDF* in the *DSEIRtoSDF* component of the architecture (Figure 4.2). This method extracts the range of the different services using the method *DSEIRModelAbstraction* to determine the load on each task. This range is then used to generate different models for each value of the service using a recursive procedure to generate all possible combinations for the different values of the services for each task. The procedure to extract the range of the different services and generate different models for the same is explained in this section.

There are a number of services that can be provided to a task in the DSEIR modeling language. In this thesis, we have restricted that the model can consist of three services only, namely *Internal Storage, Result Storage* and *Computation*. The method *Conformance Checker* will assure that all the other services that are available in the DSEIR language along with the user-defined services will belong to one of the sets of the three services. This decision was made due to the fact that the other services may not be represented in an RASDF graph or these services may be represented as one of these

services. The choice of assignment of the different services to one of the assigned sets is defined by the user specifying the model.

The method to handle variable execution times is realized in five stages:

1. Abstraction of the model:
   In this stage, the DSEIR model is abstracted using the method *DSEIRModelAbstraction* and a resulting model that is independent of any data dependent parameters is obtained.

2. Retrieval of range for each service:
   In this stage, the range of all the services, *Internal Storage, Result Storage* and *Computation* is retrieved for every task and stored in a variable. It is assigned to zero if a particular service is not used by the task. An error message is generated by *ConformanceChecker* if the *Computation* is not available since it is the service that is used to derive the execution time of an actor in an RASDF graph.

3. Creating a list of all permutations for all the services for the task:
   In this stage, we will retrieve the range of values for each service. The boundary values are taken as the only possible values as all the possible values within this range of the execution times can never be determined. This is the tradeoff that we had to make during the extraction of the range of the values.

   There were two design options in this stage. In the first design option, we can assume that all the values that fall within the retrieved range can be a possible value of the service and generate models for each of them. But this is not accurate since this is not true in every case. The amount of service provided to a certain task is always a set of distinct values and the range of values retrieved during the abstraction of the model does not provide this set of distinct values. It will instead produce a range within which these set of distinct values will always appear. Hence we lose a few values in our abstraction. The possible number of combinations for each task in this case is given by $x \times y \times z$ where x, y and z are the possible values for each service in each task. In the second design option, since we are certain that the boundary values for the range are always two of the possible values of the services, we will always consider these two values. The possible number of models generated in this case is a maximum of $(2 \times 2 \times 2) = 8$, since each service will always have a maximum of two values. Based on the explosive nature of the number of models that could be generated and the accuracy of the abstraction in the first case, we decided to choose the second option.

4. Creating a list recursively that contains the combinations for all values of the execution times of each task:
   Based on the design strategy chosen in the previous step, a recursive procedure has been written to generate multiple combination of models based on the values of

each service. This recursive procedure creates a two dimensional list that contains the execution times of all actors for each combination. If 't' is the total number of tasks in the model, then the maximum number of models generated in this case is given by:

$$2^3 \times \underbrace{2^2 \times 2^2 \times \ldots}_{t-1} = 2 \times 2^{2^t}$$

5. Generating an RASDF graph for each model:
   For each combination that is generated, a DSEIR-RASDF model is created with the new value of load on each service for every task and a new RASDF graph is produced for each DSEIR-RASDF model. Multiple RASDF graphs are produced as a result of this method.

The RASDF graphs are produced as the output of the component *DSEIR2SDF* and are given as the input for the component *RASDFAnalyzer*. They are used for analysis of the worst case throughput along with other analysis procedures and is discussed in Chapter 8. The variable execution time of the DSEIR model could lead to non-monotone behaviour of the models. This issue is addressed in *RASDFAnalyzer* and is explained in Section 8.3.

Example:

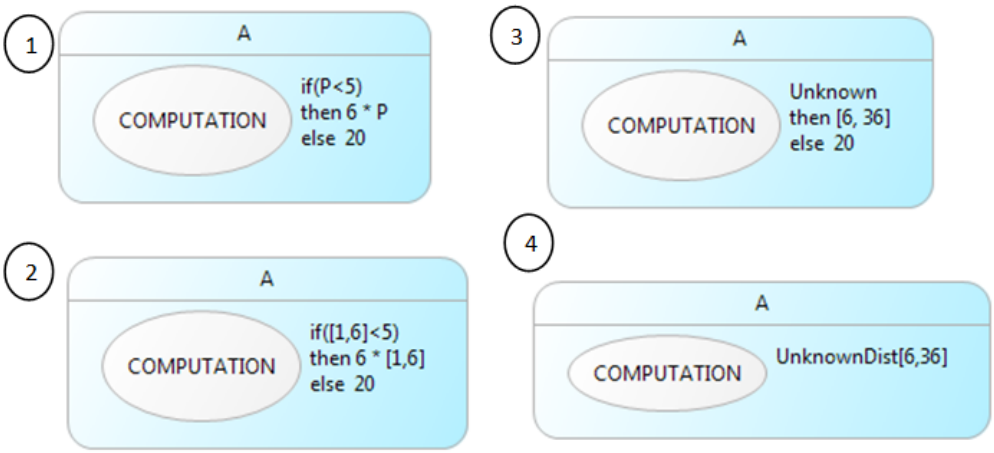

Figure 7.10: Example for the evaluation of an expression

The Figure 7.10 shows an example of a complex expression. The expression is evaluated in several steps. In Figure 7.10 (1), the expression is shown as a load on the computation. The expression is shown as a pseudo code and not as DSEIR syntax for the sake of simplicity. There is an *IF* condition that needs to be evaluated based on the parameter *P*. We assume that the range of the parameter to be [1,6] (Figure 7.10 (2)) and hence the value Unknown will be returned and is shown in Figure 7.10 (3). As a result of the

evaluation, the load on the task will be in the range [6, 36]. But the actual possible values in the range are only [6, 20, 36] and not all the values in the range. It is a rough approximation of the range and deriving the possible values of execution is not considered in this thesis. The boundary values are always considered for the evaluation. An example of the load perspective of a DSEIR model is shown in Figure 7.11. It contains five tasks with two tasks, A and B having variable execution times. For each value of the minimum and maximum of the range of the variable execution times, we generate a DSEIR-RASDF model. It is clear in Figure 7.12, where the load on the actors A and B is shown as a result of this abstraction. unknownDist(1,2) means that the range of the load on the service type varies between 1 and 2. The rest of the details of the model has been skipped for the sake of simplicity.

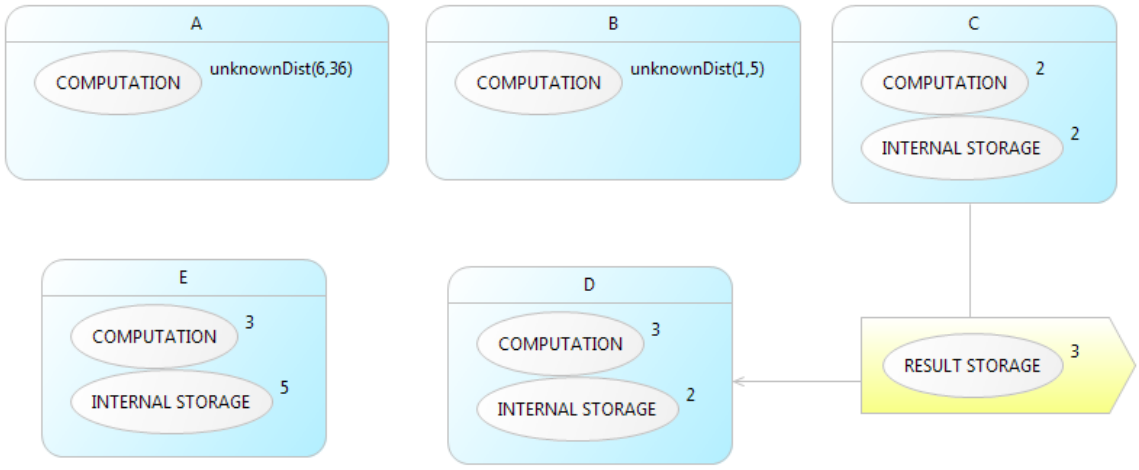

Figure 7.11: Example for handling variable execution times

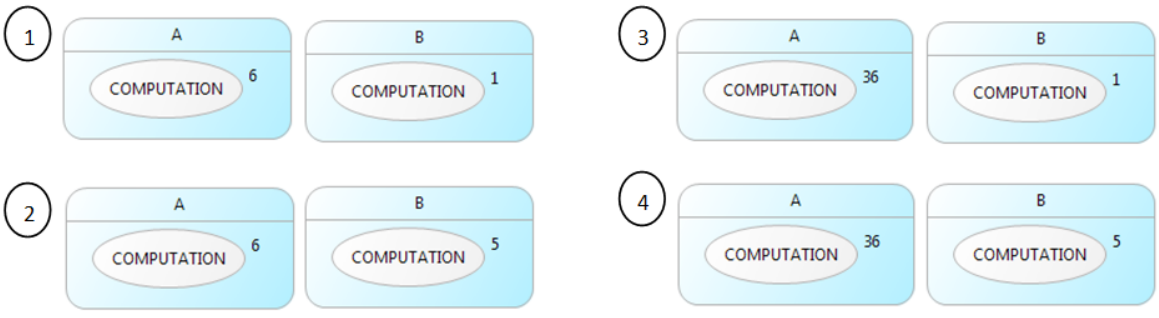

Figure 7.12: Result of handling variable execution times

7.2.3 Data dependent choices

DSEIR modeling language allows for data dependent choices on the edges of the tasks. But in RASDF models, these data dependent choices cannot be handled since an actor will always fire along all the output edges. This issue needs to be handled efficiently. It can be done in two ways. In the first method, a model has to be created along each choice and the throughput analysis has to be calculated for each of these models. But creating unique model for each data dependent choice is often difficult since the number of models that are generated could grow exponentially as the number of choices increase. In the second method, the actor will fire along all the output edges without making any choice. The throughput analysis is done for this model, but it is not an accurate translation due to the over approximation of the number of output edges along which the actor will fire.

We choose the second method and use the method *ConformanceChecker* to generate a warning message that both the choices have been considered in the translation. It is not an accurate translation but has been chosen in the context of this thesis due to its conservative nature. Extending the translation to handle data dependent choices is one of the future topics of this assignment.

7.2.4 Data dependent loops

The DSEIR modeling language supports data dependent loops on the tasks in the model. These data dependent loops are self-edges on the tasks or loops for a group of tasks that executes for a certain number of times based on the condition on the edges. But in an RASDF graph, data dependent loops cannot be modeled as the firing of a token by an actor along an output edge will always happen infinitely and cannot be restricted to occur for a certain number of times. It can be controlled by modeling the number of times of execution of the loops as the number of tokens along the output edge, but it is not accurate as this firing can occur infinitely often as well. In this translation, the method *ConformanceChecker* will generate an error message if there are data dependent loops in the model.

7.2.5 Static and dynamic scheduling

RASDF graphs support static priority based scheduling whereas DSEIR language can support static and dynamic scheduling. Translating a dynamic scheduling strategy to a static strategy is not trivial. Handling this issue is currently out of scope of this project. Currently, the method *ConformanceChecker* will generate an error message if a dynamic scheduling policy exists in the system. It is considered as a future topic of the assignment.

7.3 Overview of Conformance Checker

This section contains an overview of the method *ConformanceChecker* and is shown in Figure 7.13. This method is used to check if the model can be translated to a RASDF graph such that the graph can be used for throughput analysis. The method reads the DSEIR model that is to be translated and checks the model for different properties that can be handled by the translation procedure *DSEIR2SDF*. The different properties that cannot be handled by the translation procedure is shown in the figure and an error message is generated if the model contains properties that cannot be handled by the translation procedure.

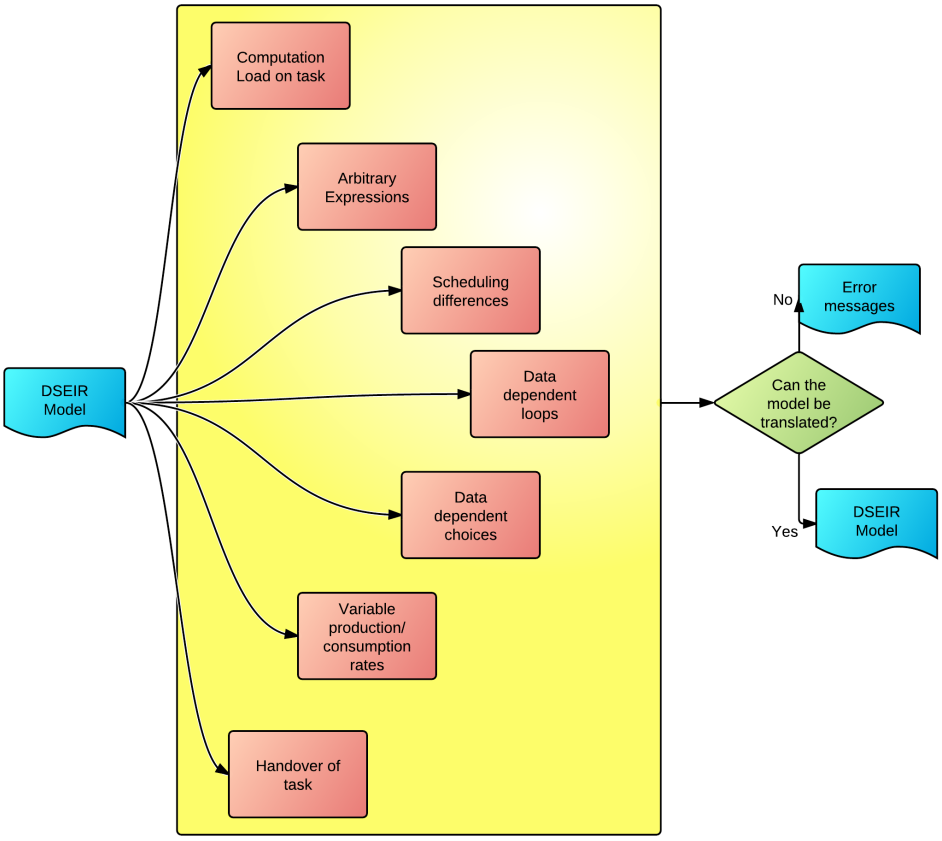

Figure 7.13: The method ConformanceChecker

If the DSEIR model contains each of the following properties, an error or warning message is generated:

- Arbitrary expressions: Since the method *ExpressionEvaluator* can be used to evaluate a pre-defined set of expressions, this method prints an error message when an arbitrary expression is used in the DSEIR model.

- For all the other properties such as scheduling differences between the DSEIR and RASDF model, data dependent loops, choices and variable production and consumption of data, the method will generate an error message and will not be handled during the translation.

Chapter 8

RASDFAnalyzer

8.1 Introduction

In this chapter, we address the different methods implemented in *RASDFAnalyzer* to analyze the RADSF graphs produced for the worst case behaviour. Figure 8.1 shows the *RASDFAnalyzer* method that was discussed in Section 4.2.2. We will also provide the best case and average case throughput results for the model. The tool SDF3 is used to analyze the model and provide the throughput results. It is explained in Section 3.6. A method called *NonMonotonicityChecker* has been used to detect non-monotone behaviour in the models and is explained in Section 8.3. The throughput results that are produced for the model is explained in Section 8.4. Section 8.5 then explains the throughput results of the model used as an example in Chapter 7.

8.2 SDF3 and its results

SDF3 is the tool used in the project for throughput analysis of a model as discussed in Section 3.6. We have made the connection to the toolset transparently such that the throughput results are generated automatically after the translation of the model from DSEIR to RASDF. After the translation of the model is done, one or more RASDF graphs are generated and they are sent to SDF3 for throughput results. The SDF3 tool runs transparently without the knowledge of the user and generates the throughput results for each graph. These results are written into a file. This throughput result file is analyzed in the method *NonMonotonicityChecker* for non-monotone behaviour and the throughput results for the model are generated.

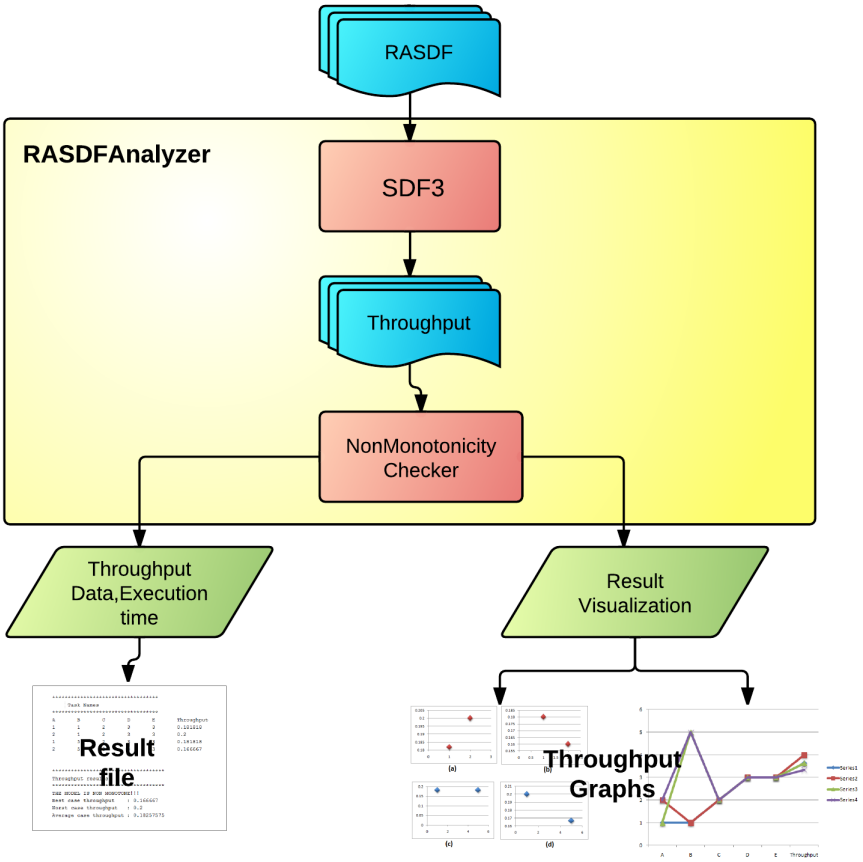

Figure 8.1: RASDF Analyzer (same as Figure 4.3)

8.3 Non monotonicity Checker

This section explains the throughput analysis methodologies for the the RASDF graphs that are generated when the DSEIR model has variable execution times. If the model does not have variable execution time, then this method would have not been necessary and the best, worst and average case throughput results are the same for the model. The tool SDF3 generates a set of throughput values for all the RASDF graphs that are given as input to SDF3. It is denoted by the light blue coloured group of boxes called *ThroughputResultFile* in *RASDFAnalyzer* in Figure 8.1. These files are analyzed for non monotone behaviour.

8.3.1 Detection of non-monotone behaviour

We have used different techniques to identify non-monotone behaviour in the models after the translation. Figure 8.2 shows the throughput behaviour of a model that contains two tasks with variable execution times that are plotted along the x and y-axes respectively. The throughput of the model is plotted along the z-axis. It can be observed that the throughput is always decreasing with the increase in the execution time of each actor. Hence the model does not exhibit non-monotone behaviour as discussed from Section 3.5. This method is used to detect the situation when a model does not exhibit this behaviour. It can be identified by plotting graphs for different values of throughput and execution times and recognizing outliers in the graph that deviate from the expected behaviour.

There are different ways to visualize the throughput behavior of models graphically. We have tried a few ways in this thesis and selected one among them based on the advantages and disadvantages of each of them.

- The execution times of actors can be plotted in each axis and the throughput of the model is plotted in an additional axis for each configuration of execution times of different actors. This works well when a maximum of two actors have variable execution times since the plot will be either 2D or 3D. But when the number of actors with variable execution times increase, these plots will not suffice since it will be difficult to represent a graph greater than three dimensions visually.

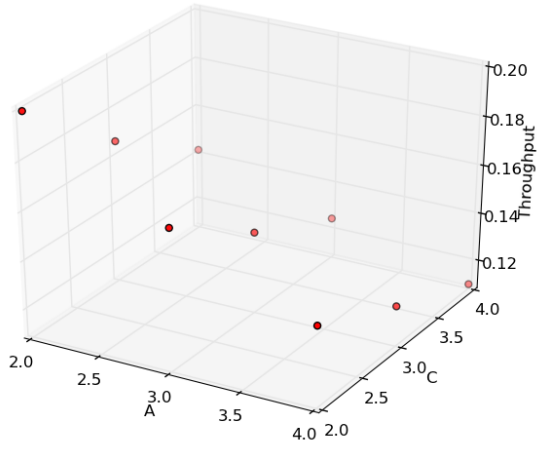

Figure 8.2: Model exhibiting monotone behaviour

A	C	Throughput
2	2	0.2
2	3	0.167
2	4	0.142
3	2	0.167
3	3	0.142
3	4	0.125
4	2	0.142
4	3	0.125
4	4	0.11

Table 8.1: Throughput results of monotone model

Figure 8.2 is an example of this representation for the model given in Table 8.1. It is a complex model with more than two actors, but the details of the model has been skipped for the sake of explanation. The model used in this figure exhibits monotone behaviour. We can observe that the throughput of the model decreases with the increase in the execution times of the model without any outliers or any unexpected behaviour in between. Hence the model is said to exhibit monotone behavior and the best, worst and average case throughputs can be calculated directly.

- To overcome the problem of plotting along multiple dimensions for variable execution times, parallel coordinates[18] can be used. Parallel coordinates is a common method used to visualize multi-dimensional data on a two-dimensional plane. But the non-monotone throughput behaviour might not be very easy to understand with this representation since a lot of information will be represented visually and the user should be able to abstract the necessary information from the graph to analyze the model, which is not the case with every user. Figure 8.3 shows the graphical representation of a model using parallel coordinates. The model is depicted in Table 8.2 with the different execution times and the throughput results for each combination.

	A	B	C	D	E	Throughput
Case1	1	1	2	3	3	0.1818
Case2	2	1	2	3	3	0.2
Case3	1	5	2	3	3	0.1818
Case4	2	5	2	3	3	0.167

Table 8.2: Throughput results of a model

In Figure 8.3, the execution times of different actors and the throughput is plotted along each vertical axis of the graph with each case as a vertical axis. The model

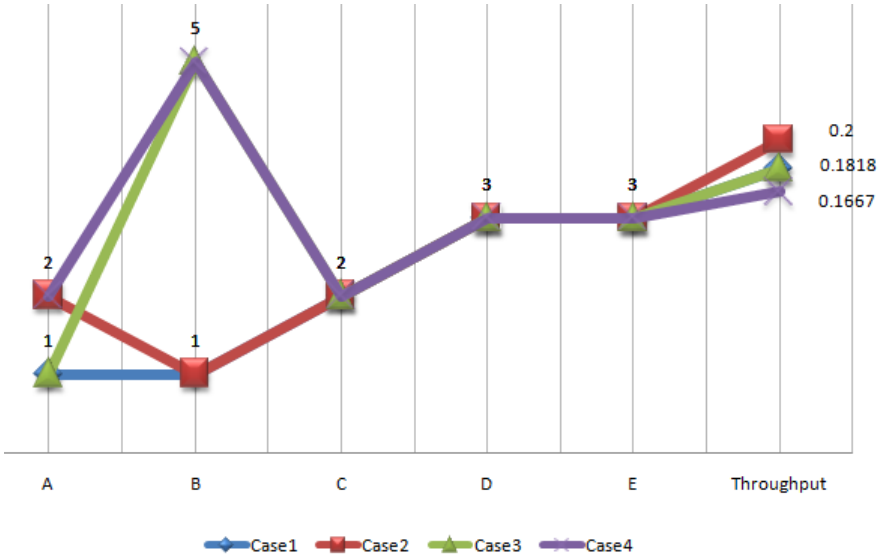

Figure 8.3: Non Monotone behaviour analysis using Parallel coordinates

represents non-monotone behaviour when the execution time of Task A is changed from 1 to 2. The throughput of the model increases from 0.1818 to 0.2 when the execution time of the actor A increases from 1 to 2. This can be observed on the graph by looking at the entry *Case2*. When a model is showing monotone behaviour, the best throughput of the model is obtained from the model with minimum execution times and it is the largest entry on the throughput axis. Similarly, the worst case throughput is obtained for the model when the maximum of all the execution times for all the actors are considered and it is the smallest entry on the throughput axis and every other throughput value should be within this range to exhibit monotone behaviour, which is not the case in Figure 8.3. Hence the model exhibits non-monotone behaviour for the given value of execution times.

- The last approach used to understand this behaviour is by plotting only the actor with variable execution time and the throughput of the model in a two dimensional graph and repeating this procedure for every actor with variable execution times. This will help in identifying the non monotone behaviour of the system by analyzing each actor at a time.

  Figure 8.4 shows the graphs for the same model represented in Table 8.2. The table is written in Table 8.3 with the actors A and B only to understand the throughput behaviour of models using the approach mentioned in this section. The throughput results are plotted for the actors A and B which is represented by the figures and tables (a), (b) and (c), (d) in Figure 8.4 and Table 8.3 respectively. The graphs in (a) and (b) represents the throughput behaviour of the model when the execution time of actor A is varying. In (a), the execution time of actor B is fixed (it is

(a)			(b)		
A	**B**	**Throughput**	**A**	**B**	**Throughput**
1	1	0.1818	1	5	0.1818
2	1	0.2	2	5	0.167
(c)			(d)		
A	**B**	**Throughput**	**A**	**B**	**Throughput**
1	1	0.1818	2	1	0.2
1	5	0.1818	2	5	0.167

Table 8.3: Throughput results of a model from table 8.2

assumed to be constant for these set of execution times of A even if it is a variable value) as 1 and the execution time of actor B is assumed to be constant at 5 in (b). This procedure is repeated in (c) and (d) assuming that the execution times of A is constant at 1 and 2 for (c) and (d) respectively. From the figure, it can be observed that in (a), the throughput of the model improves when the execution time is increasing and this throughput behaviour is non-monotone in nature.

But the drawback of this method is that the number of graphs to be generated can increase exponentially since a graph is generated for each varying value of the execution time of each actor. This can increase the time of throughput analysis of the model.

8.3.2 Implementation of RASDFAnalyzer

A method was implemented to analyze the non monotone behaviour of models. The method exploits the advantages of all the methods mentioned in the previous section. The method searches exhaustively through all the throughput result files and detects non-monotonicity using parallel coordinates and two dimensional graphs.

Non-monotonicity in the graph is detected using the logic used for 2D graphs. The throughput values are checked for each case when the execution time of only one actor is varying and the other actors remain constant. If there is non-monotone behaviour in one of the cases, then the model is said to be non-monotone. Since the minimum and maximum is considered always, the 2D graph for each actor will have only 2 values and the model is said to be non-monotone if the throughput of the model with the smaller execution time is less than the throughput of the model with larger execution time. Algorithm *detectNonMonotonicity* is used to detect non-monotone behaviour by detecting non-monotone behaviour per actor.

DETECTNONMONOTONICITY(File *model*)

1 Integer *flag* ← 0

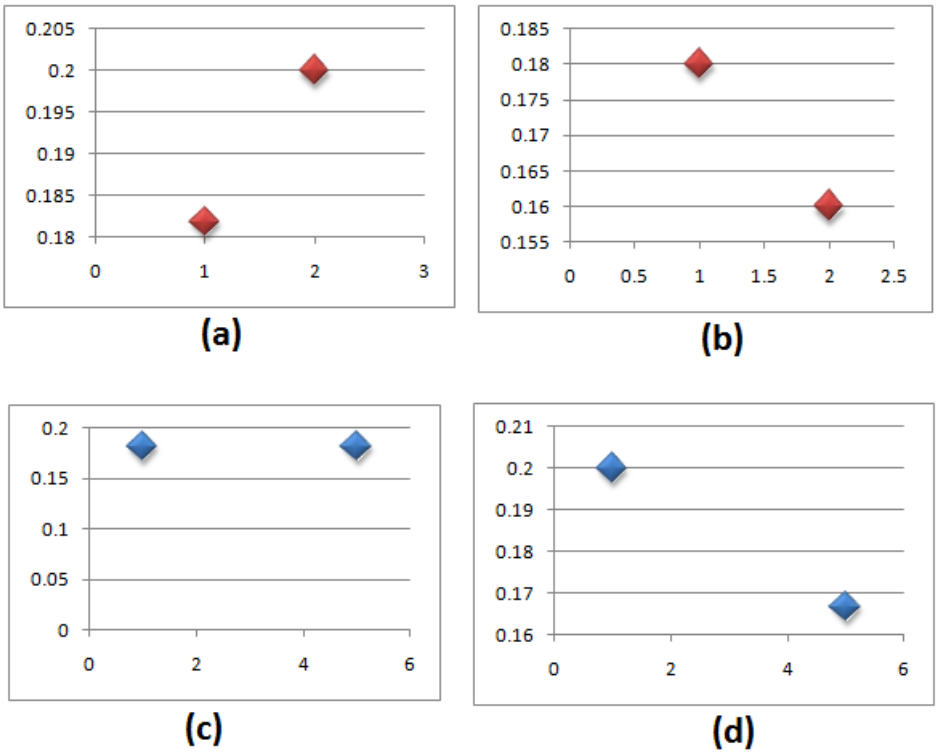

Figure 8.4: Non Monotone behaviour analysis using 2D-graphs

```
2  Set<Actor> actorSet ← getActorsWithVarExecutionTime(model)
3  while (flag == 0)
4  do for (All actor ∈ actorSet)
5      do Set<Integer> executionSet = getExecutionTime(actor)
6          Set<Real> throughputSet ← getThroughput(executionSet)
7          if (getFirst(thoughputSet) < getSecond(thoughputSet))
8            then flag ← 1
9                 Range range ← executionTime
10                String name ← actorName
11                CALLprint("MODEL IS NON-MONOTONE for actor name at range")
12           else print("MODEL IS MONOTONE")
13                end if
14         end for
15     end while
```

8.4 Results of RASDFAnalyzer

In this section, we will describe the two results that are produced after the throughput analysis of the model. In the first form, the different throughput results of the model are retrieved from the class and stored in the form of a text file. In the second form, the throughput results of the model is also visualized in the form of parallel coordinates.

8.4.1 Throughput result file

The results of the throughput analysis is returned in the form of a text file. It contains the execution times of all actors along with the throughput for each value of the execution times. The result file also contains the best, worst and average case throughput of the model along with information about the non-monotonicity of the model.

8.4.2 Throughput visualization

The visualization of the throughput is done in either of the ways discussed previously such as parallel coordinates and using 2D graphs for each execution time of the actor, as shown in Figures 8.3 and 8.4. A text file containing the execution time of actors and the throughput model as coordinates is returned which can be used to visualize the throughput results.

8.5 Example

The example discussed in Chapter 6 is run through SDF3 to obtain throughput results. The throughput result of the file is shown along with the detection of non-monotonicity.

```
**********************************
     Task Names
**********************************
A   B    C     D    E   Throughput
1   1    2     3    3   0.181818
2   1    2     3    3   0.2
1   5    2     3    3   0.181818
2   5    2     3    3   0.166667

**********************************
```

```
Throughput results
**********************************
THE MODEL IS NON MONOTONE when execution time for A[1,2] !!!
Best case throughput : 0.2
Worst case throughput : 0.16667
Average case throughput : 0.18257575
```

Chapter 9

Conclusion

In the initial version of the Octopus toolset, there was a translation procedure from DSEIR models to RASDF graphs to enable a quicker throughput analysis of the models. But this translation procedure could handle only the application part of the DSEIR model during translation. In this thesis, we have extended the translation procedure to accommodate the platform and mapping components of the DSEIR model and obtained throughput results for the same by making a connection to SDF3 invisibly. To achieve this translation, we have identified a subset of DSEIR and called it DSEIR-RASDF. This proper subset of DSEIR corresponds to an RASDF graph directly after translation. There are translation procedures for translation of a DSEIR-RASDF to RASDF and RASDF to DSEIR-RASDF. These translation procedures were initially assumed to be trivial due to the 1:1 mapping between DSEIR-RASDF and RASDF. But it was observed that the absence of schedulers, absence of explicit declaration of handovers and mapping of resources to resource amounts in RASDF as opposed to mapping of service to resource amounts in DSEIR-RASDF posed additional difficult problems during the translation, which have been solved as part of the translation.

In the second stage of the translation, we have chosen an arbitrary DSEIR model and tried to translate it into a DSEIR-RASDF model so that it can be translated to an RASDF graph once the former translation is achieved. But this translation is not trivial and hence the different issues that need to be addressed during the translation were identified. The most complex issues such as data dependent parameter and load extraction have been solved. Since all the issues could not be solved in this thesis, they have been identified as future work in this assignment. In the current translation procedure, they are dealt with in the method *Conformance Checker*. But the issues handled in this method can be reduced by extending the current translation procedure to tackle more issues and hence increase the subset of DSEIR models that can be translated.

The different issues that can be solved as a future scope to this assignment is listed along

with a possible solution for each of them. These ideas could be used as a starting point to solve these issues.

1. **Data dependent choices:**
   In the current translation, all the edges are considered during the translation. An extension to this would be that a model is constructed for each case and the throughput analysis can be done for these set of models. But the number of models that are generated could increase exponentially and this issue needs to be considered.

2. **Data dependent loops:**
   This is a very difficult issue and might not be solved due to the difference in execution of the models. The DSEIR model can be restricted to execute a certain number of times, but an RASDF graph will execute infinite number of times and addition of a counter could restrict this behaviour in some way.

3. **Scheduling differences:**
   RASDF does not provide support for dynamic scheduling while DSEIR models can be scheduled dynamically. This can probably be solved by considering the priorities in a single cycle of execution to be a static scheduling.

4. **Evaluation of arbitrary expressions:**
   In DSEIR, only a set of pre-defined expressions is allowed and any arbitrary expression such as a quadratic function is not supported by the current version of DSEIR. Due to this restriction, the function, *ExpressionEvaluator* can always solve only the allowed set of expressions. If the DSEIR model allows for arbitrary expressions, then the function has to be extended too to accommodate this feature. Addition of this feature would require the function to evaluate the expression dynamically based on the expression, which might be a difficult addition to the function.

Evaluation of expression on the load component produces a range of possible values in the order of a maximum of 2^n for n actors since two values are considered for every actor. In the current version, only the boundary values of the range is considered due to the lack of an algorithm that detects non-monotonicity without executing the entire state space. The current translation procedure can be extended so that the detection of non-monotonicity can be done faster. During the evaluation of the expression, all the possible values cannot be retrieved. Extending the function to retrieve all the possible values could be a good extension to this work too. This would require evaluation of expressions over a range whereas the current method considers the boundary values in the range only. Throughput analysis is performed on the RASDF graphs that are generated as a result of the translation.

The time taken by the different analysis tools in the Octopus toolset was compared to validate the claim that the throughput analysis was very fast in the toolset. We used

the results generated by Uppaal to validate the results. We performed translation on a range of models and observed that while SDF3 produced results in the order of 10-20 ms for the throughput analysis of all the models put together, Uppaal took 70-100ms to do the throughput analysis for a single model. Even though the difference in time is not a large value, Uppaal requires manual translation of each model to enable throughput analysis.

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. Trčka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang, "Model-driven design-space exploration for embedded systems: the octopus toolset," in *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I*, ISoLA'10, (Berlin, Heidelberg), pp. 90–105, Springer-Verlag, 2010.

[2] N. Trčka, M. Hendriks, M. Geilen, T. Basten, and L. Somers, "Integrated model-driven design-space exploration for embedded systems," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS XI,*, July 2011.

[3] "Synchronous dataflow." `http://www.es.ele.tue.nl/sdf3/manuals/moc/sdf/`, 2001.

[4] K. Jensen, "Coloured petri nets," in *Petri Nets: Central Models and Their Properties* (W. Brauer, W. Reisig, and G. Rozenberg, eds.), vol. 254 of *Lecture Notes in Computer Science*, pp. 248–299, Springer Berlin / Heidelberg, 1987. 10.1007/BFb0046842.

[5] "Uppaal." `http://www.uppaal.com/`, 2009.

[6] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *Embedded Processor Design Challenges*, vol. 2268 of *Lecture Notes in Computer Science*, pp. 321–324, Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45874-3_2.

[7] N. Trčka and B. in 't Groen, "DSEIR - The modeling language of Octopus." User manual, 2011.

[8] N. Trčka, "The Octopus toolset." Presentation, 08-11-2011.

[9] B. in 't Groen, "Vdseir, a visual addition to dseir," Master's thesis, Technische Universiteit, Eindhoven, August 2011.

[10] S. Bhattacharyya, *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1994.

[11] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Automated bottleneck-driven design-space exploration of media processing systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 1041 –1046, march 2010.

[12] S. Stuijk, *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2007.

[13] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, 2000.

[14] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, pp. 355–383, July 2003.

[15] A. Ghamarian, M. Geilen, T. Basten, and S. Stuijk, "Parametric throughput analysis of synchronous data flow graphs," in *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 116 –121, march 2008.

[16] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pp. 276–278, IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[17] A. Kumar, "Adding Schedulability Analysis to the Octopus Tool-set," Master's thesis, Technische Universiteit, Eindhoven, August 2011.

[18] A. Inselberg, "The plane with parallel coordinates," *The Visual Computer*, vol. 1, pp. 69–91, 1985. 10.1007/BF01898350.