Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Optimization of Systems with Dynamic Structures

SCIENTIFIC STUDENTS' ASSOCIATIONS REPORT

Authors:

Dániel László Magyar
Attila Hoangthanh Dinh

Supervisors:

Dr. András Pataricza
László Gönczy

2014.

Contents

Kivonat

Számos rendszer működése leírható folyamatok segítségével, melyek optimalizálására növekvő igény mutatkozik. A valóságban azonban az előre nem tervezhető, környezeti események a végrehajtás közben módosíthatják a folyamat paramétereit (például erőforráshasználati- vagy beszerzési költségeit), illetve struktúráját (például már használt erőforrások meghibásodása, vagy újabbak elérhetővé válása).

A dolgozatunk során célunk egy olyan módszer megalkotása volt, amellyel elősegítjük az ilyen változásoknak kitett folyamatok optimálizálási- és rekonfigurációs problémájának megoldását. Az előbbi esettel ellentétben, a megfelelő rekonfiguráció keresése során figyelemmel kell lennünk a korábbi futások által módosított rendszerállapotokra.

Napjainkban a biztonságkritikus rendszereken túl, mind üzleti-, mind ipari alkalmazásokban egyre inkább kardinális kérdésként jelenik meg a költséghatékonyság mellett a hibatűrő működés biztosítása is. Ez utóbbi szempont fő motivációja, hogy a szolgáltatások, illetve termelési folyamatok csupán átmeneti fennakadása is jellemzően komoly bevételkiesést jelent a vállalatoknak, biztonságkritikus rendszerek esetén pedig gyakran az anyagi vonatkozásokon túlmutató következményekkel is számolnunk kell.

E követelményeket kielégítő, helyreállítási- illetve rekonfigurációs folyamatok megtalálásának fontos része az újraoptimalizálás és átkonfigurálás költségének, erőforrás- és időigényének minimalizálása. Az általunk bemutatott módszer e komplex problémára keres megoldást.

A munkánk alapötlete az optimalizálási folyamat konstrukciós és javítási fázisokra bontásából ered, mellyel célunk az optimalizálási folyamat gyorsítása volt. A konstrukciós fázis első lépéseként a bemeneti modellen strukturális redukciót végzünk a numerikus paraméterek figyelmen kívül hagyásával, felhasználva a folyamatszintézis (PNS) területéről megismert módszereket. Az ily módon redukált bemeneti modellen futtatott optimalizálás az így csökkentett megoldástéren futtatva jelentősen gyorsítható. E megoldás kiegészítését használtuk fel a rekonfigurációs feladat megoldására.

A megvalósítás során definiáltuk a folyamatszintézis (PNS) probléma formalizmusának kiterjesztését, illetve ennek leképzését az Alloy modellkereső eszköz elsőrendű logikai nyelvére. Az így definiált modellt az eszköz elemző motorjának felhasználásával hatékony módon transzformáljuk SAT problémává. Szintén a bemeneti modell szintjén definiáltuk a folyamatot érintő változásokat, melyek szimulációját is megvalósítottuk. Kihasználva az eszköz modellkereső képességeit, az ilyen eseményekre az új megoldási tér elemkészletének és struktúrájának előállításával reagálunk. Ezen leírások előállításánál különös figyelmet

fordítunk az átkonfigurálandó folyamat által már elvégzett feladatok, illetve előállított elemek felhasználására, így minimalizálva a meghibásodás kezelésének költségeit, illetve a folyamatok lehetőség szerint zavartalan működésének biztosítását.

Abstract

There is an increasing demand for the optimization of complex systems that are usually modeled as processes. Although solving this problem unexpected external events during the process execution can alter the parameters of the system (e.g. cost of resource usage or supply), and its structure (e.g. component failures).

The purpose of our method is to help to find a solution to the problems of optimization, confinement of error propagation, and reconfiguration of such dynamically changing processes. Typically the solutions of the three problem classes require different approaches, for example unlike in case of optimization, during the search of a reconfiguration plan, we have to be attentive to the system states modified earlier, by previous process instances.

In addition to safety-critical systems, many areas of business and industrial applications require cost-effective operation even in the presence of resource failures. There are several well-known methods for designing safety-critical systems, nevertheless, the issue of determining an optimized plan for recovering or reconfiguring the process, in response to a component failure or change in the parameters, is crucial. Since, even the slightest interference in the services or the production process could result in immense financial loss. Furthermore, the consequences of a failure in safety-critical systems might be more severe than mere loss in revenue. As a result the time required to perform both the search and the execution of the configuration process should be minimized along with its costs and resource usage. Therefore, the calculation of recovery and reconfiguration processes, including the re-optimization, should be performed with minimal time consumption, resource usage and cost. Thus, the goal of our method is to find a solution to this complex problem.

The underlying principle of our work is to build upon the well-known "construct and improve" two-phase optimization process, so that we can enhance it by means of reducing its computational needs. As the first step of the construction phase, ignoring all the numerical parameters, we perform a structural reduction of the input model, based on the methods of Process Network Synthesis (PNS). Right after this step, the optimization process is performed on the resulting reduced problem space, which makes the problem space significantly easier to compute. We use an extension of this method solve the task of confining error propagation and reconfiguration.

Additionally, we define an extension to the PNS problem, and create translation rules to the first-order logic modeling language of the Alloy model finder tool. This model is then efficiently translated to a SAT problem by using the tool's analyzer component.

Moreover, we define the changes that affect the operation of the process, along with the simulation of these changes. Afterwards, using Alloy's model constructional features, we generate the elements and structure of the new solution space. While determining these elements, we pay particular attention to the finished tasks and already produced elements of the process, in order to minimize the cost of reconfiguration.

Chapter 1

Introduction

1.1 Context

As most complex systems rely upon services which depend on IT infrastructures, the effect of an error at the IT component level is increasingly important. Such errors can lead to system failures which may result in huge financial loss, or in extreme scenarios, they can even endanger human lives.

There are many methods described in the literature to improve the fault tolerance properties of a system, but these typically introduce additional cost in the phases of design, construction, implementation, operation, and maintenance, which naturally results in less income from the business point of view. Hence finding the golden mean between fault tolerance and cost-optimized operation is a cornerstone of the design of a system.

1.2 Problem statement

Our goal was to design a method that can satisfy a process based system's optimization and fault tolerance needs simultaneously. Also we wanted this method being able to support numerous mathematical tools. The technical implementation of monitoring and error detection is out of the scope of our current work, as we assume that such services are available in most IT infrastructures. Therefore our objective is mainly focused on the problem of error isolation and system recovery.

1.3 Case study

In Figure 1.1. the enterprise architecture model of our case study is shown, which will be used to demonstrate the capabilities and main concepts of our method. The case study describes the problem of depositing cash from the bank's point of view, while the business process and its underlying IT infrastructure are exposed to various failures. The example was originally described in [30] for demonstrating a process diagnosis and failure analysis framework. Our method also provides a fault tolerance technique, more precisely speaking, a technique, which embodies the isolation and reconfiguration of a process based system. Hence due to the numerous similarities shared by the two methods' our problem domain,

we found it perfectly suitable for presenting our method as well. The model depicts the process in three layer:

- **Business Processes Layer** describes the `activities` initiated by the cashier, along with their `precedence constraints` (represented by execution path), similar to a BPMN[12] model. The precedence constraints of the activities serve as *hard constraints* i.e. they cannot be violated during the execution of any process instances.
- **Supporting Applications Layer** describes the services and logical components the above layer builds upon. The depicted dependency constraints also serve as *hard constraints*.
- **Physical Resources Layer** describes the physical components, as the lowest level structure of the system.

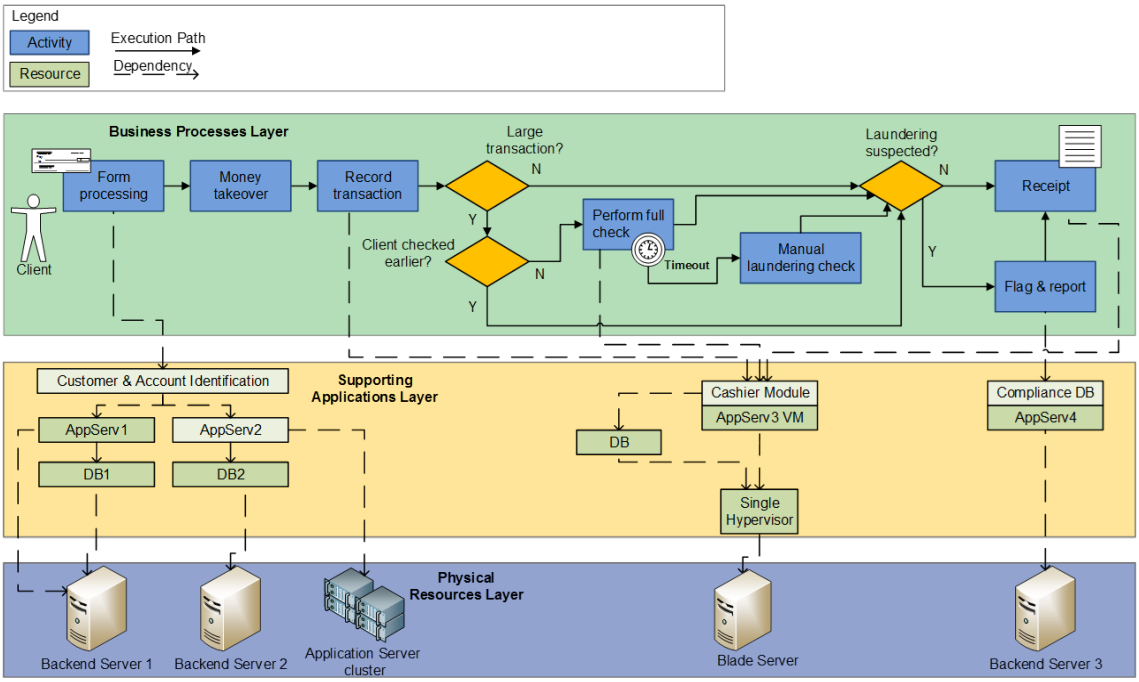


Figure 1.1. Example bank transaction workflow [30]

In [30] the failure modes of this example system were also discussed, which we can use as an initial failed state of a system, from which we aim to recover. A brief overview of the failure modes depicted on Figure 1.2:

- **Single point of failure (1)** A fault in a physical layer (i.e. failure of a resource, such as the `blade server`) can lead to the failure of the whole system through dependency relations. Such vulnerabilities of a system can be discovered during the design phase. In case of this failure mode activates, the typical solution to this recovery problem is to perform failover, i.e. to simply replace the failed resource with a flawless one.

- **Degradation (2)** Assuming, that during the design of a system, the designers paid attention to avoid using components, that would become single point of failure. A fault in a physical layer then may have less sever effects (e.g. failure in the `blade server farm`). In this failure mode, instead of stopping the entire system, the services might slow down, increasing the delay, which ends up with a proper, but more expensive process execution. (e.g. `Perform full check` task might terminate with a `timeout` exception, resulting in a mandatory execution of `manual laundering check` of every transaction). The typical solution to this problem would be to perform a loadbalancing, i.e. to find alternative resources with proper remaining capacity.

- **Backwards propagation (3)** The initial fault is not necessarily propagates from lower logical layers to the higher logical layers of the infrastructure. In this case, a fault in the logical layer, such as an `SQL injection` can cause the `Database server` to become unstable, resulting in the failure of process execution. A possible solution to this problem could be to perform a failover.

It is easy to see that during the recovery of these failures; spare resources and alternative execution possibilities have to be considered. Our method aims to automatically find a solution to this problem.

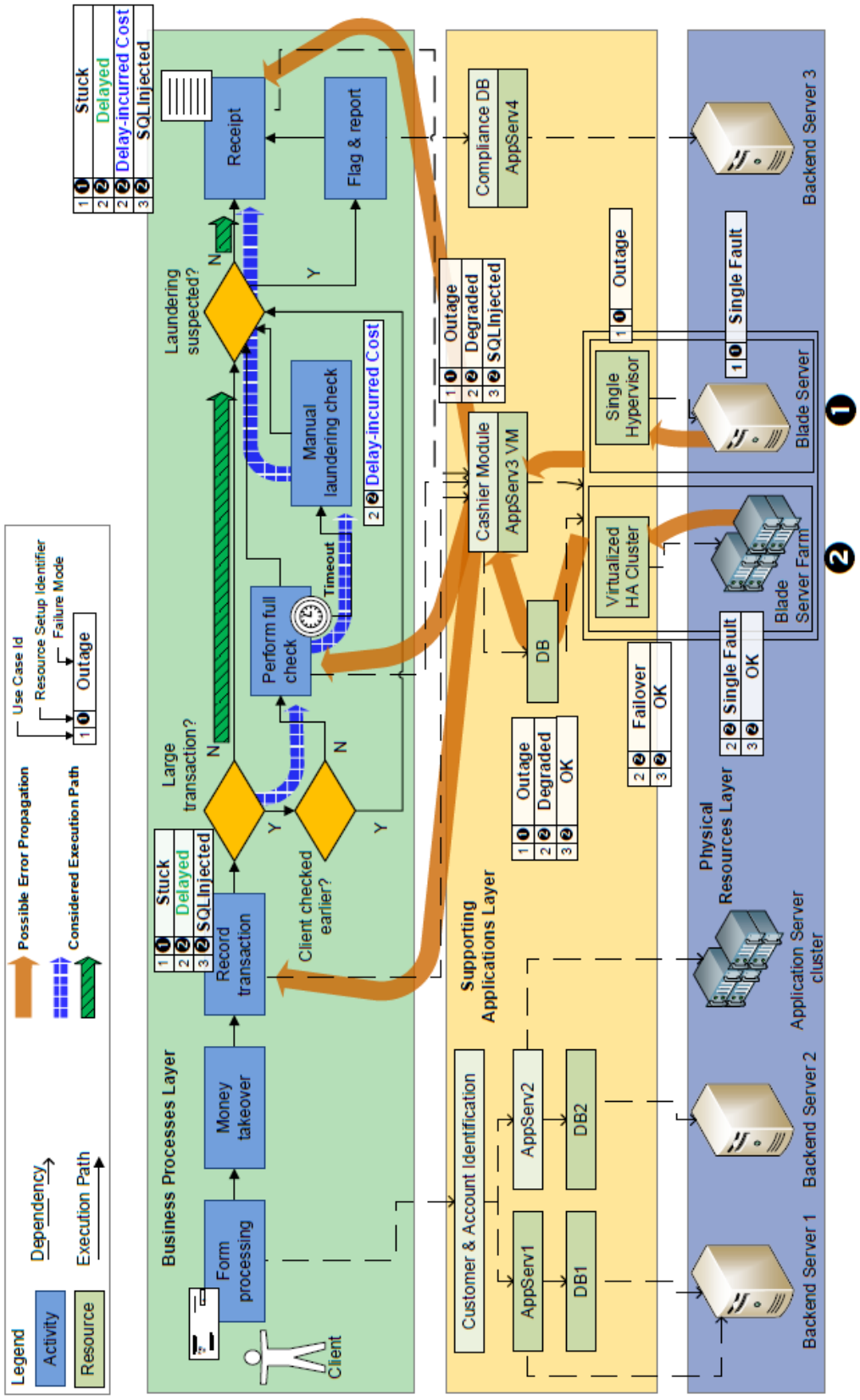


Figure 1.2. Motivational example: failure modes of a bank's enterprise architecture model [30]

1.4 Structure of the paper

The paper is written in the following structure. In chapter 2 we present problems aimed by our method, and explain its main logical steps. In chapter 3 we present the underlying technology of our method, both from the theoretical and practical point of view. In chapter 4 we explains how we used the previously presented technologies to solve the aimed problems. In chapter 5 we presents the main characteristics of our implementation of the method. Finally in 6 we summarize our contributions and the limitation

Chapter 2

Problem description

In this chapter we discuss the general properties of the problem domain aimed by our approach. In section 2.1, we present the general properties of the targeted domain of our method. In section 2.2, we give a formal description of the problems, our method solves, and at last in section 2.3 we introduce the way our approach handles these problems all at once.

2.1 Domain modelling

Our method aims the domain of complex systems, whose structure and execution is both subject to changes due to external events. Since such systems are highly diverse in terms of their purpose, attributes, structures and events that can alter them in certain ways, we need a representation to describe their similarities. Hence in this section we discuss their common properties, in order to design the previously mentioned uniform model.

Behaviour The behaviour of complex systems is commonly described by a business process model (for example a BPMN model[12]), which is an OMG standard, designed to intuitively capture the process as an end to end flow of execution. Hence we mainly use its concepts to initially represent the process' behavioural properties as shown in the topmost layer of Figure 1.1.

- **Activities** of a business process model are the smallest logical representation of a process' execution steps. An activity definition contains its input and output objects (see Structure) and cost functions as well (see Objective). Hence an activity can represent the task of `form processing` or `record transaction`.
- **Execution flow** defines precedence and temporal logical constraints between activities.
  - **Sequences** represent simple precedence constraints between activities and are depicted as arrows. By using sequence flow, in our case study we can express the fact that the `form processing` activity must precede the `money takeover`.

- **Gateways** are depicted as rhombuses and express the temporal logical constraints of a group of activities, hence it can be used to represent basic boolean operations and even more complex predicates. A gateway can express that in case of `suspected laundering` the transaction must be `reported` before creating a `receipt`.

Structure The structural elements of a complex system can be described by an infrastructure model, which - in comparison to BPMN's data object concept - provides a more sophisticated and detailed way to represent the hierarchy of elements while keeping the BPMN's perspective. Structural characteristics of our case study are depicted in the *Supporting Applications Layer* and *Physical Resources Layer* of Figure 1.1.

- **Objects** represent the inputs and outputs of activities and may have different properties in terms of usage and role.
  - **Resources** are initial or immediate inputs and outputs of activities from the process perspective. Resources may have two different ways they react to the operation of activity they are allocated to.
    - **Renewable** resources can be reserved, used and then released by an activity, hence they cannot be modified. Hence they can represent logical services like `application server`s or physical components like `backend server`s.
    - **Consumable** resources are used by an activity if they serve as its inputs, and usually have cumulative properties such as capacity constraints. Hence they can represent immediate items that can be used by activities for communicating, such as an `incoming form` or a `processed form` which can be processed (i.e. consumed) by another task. (Note that in figure 1.3 immediate consumable resources (i.e. they serve as both inputs and outputs, like `processed form`) are not depicted for the sake of clear presentation, but they are innate parts of any processes).
  - **Products** can only serve as outputs of activities, as they represent the result of a flawless process execution. A product can stand as a `receipt` or a `report` produced by different activities.
- **Dependencies** represent requirement relations between two objects or an activity and an object. In other words, dependencies express that activities cannot be executed and objects cannot be used without all of their dependent objects being usable. Hence we can use dependency relation to represent that an `application server` service requires both the `database server`'s service and the physical `backend server` itself to be usable.

Objective Behavioural properties describe the logic of processes, structural properties describe their underlying infrastructure, therefore together they provide the execution policy (i.e. *hard constraints*) of a system. On the other hand, the following elements are needed to describe the objective or purpose (i.e. *soft constraints*) of a system and its processes in order to optimize it (i.e. to distinguish the effective process executions from the space of valid executions).

- **Cost functions** are used to express the various costs of activity executions and resource usage. Cost functions can describe either the time-requirements (i.e. duration) of an activity execution, or the prices of using a resource like the `compliance DB` service. (Note, that cost functions have no standard notation, hence they are not shown in Figure 1.1)

- **Objective functions** stand as the fundamental subjects of optimization, as they directly express the purpose of the system processes. Objective functions can represent the intent to minimize the total execution time (makespan) of all the process instances that can help to achieve maximized throughput of the system.

Dynamic structures: system changes In our approach, system changes are the effects of external events, which can either be an observed failure of a used resource, or a new one becoming available to use. It's important to emphasize that according to our assumption, an activity has only one possible execution mode in terms of its resource usage. In other words, the change of a resource will affect its consumer and producer activities naturally, hence we do not have to take dynamic changes of activity execution modes into account.
We make the assumption of activities with single-mode behaviour based on experimental observations of problems from the given domain. The concept of multi-mode resource-constrained project scheduling problem is discussed in [1]. The concept of multi-mode resource usage can further simplify the complexity of the optimization for some particular problems, but on the other hand it makes the reconfiguration more complex. Our current work focuses on the single-mode resource-constrained project scheduling problem, while activities with multi-mode executions can still be translated to our problem in case of finite number of given execution modes. In conclusion, our concept of system changes is suitable for modelling typical failures of the components of an IT infrastructure, e.g. a faulty `backend server`'s error can propagate to the corresponding `application server` and `DB` services, making the activity of `form processing` unavailable, resulting in a system failure.

2.2 Cost effective reconfiguration

Given the definition of a process along with its underlying infrastructure, our goal with the appliance of our method is to not only optimize the execution of the process, but to enhance its fault tolerance by automatic resource reconfigurations and activity rescheduling. In other words, our purpose is to minimize the costs of process execution and reconfiguration together.

The problem consists of two well differentiated subproblems, which have known methods of solving separately, but neither of them is directly applicable for solving the problems together while scaling well with the size of the system model. Below we briefly describe the properties of these subproblems.

Scheduling In our case, the scheduling problem can be formulated as $P_{scheduling} = \langle \alpha_w, \beta, \gamma, \delta \rangle$ [26, 21, 1], where

- $\alpha_w = \{a_1, a_2, ..., a_n\}$ is the set of *workflow activities* given in the description, where each activity $j \in \alpha_w$ determines its duration d_j.
- $\beta = \{R, C, P\}$ is the given set of *objects*, where R is the set of renewable resources, C set of consumables and P set of products. For each $k \in \beta$ object, its *capacity* g_k is also defined, which represents
  - the maximum number of simultaneous usage of g at any point of time, if $g \in R$ renewable, or
  - the summed number of usage of g throughout the process execution, if $g \in C$ consumable.
- γ is the set of hard-constraints, i.e. for all $j \in \alpha_w$ activities their given set of inputs $In(j)$, outputs $Out(j)$, predecessors $Pred(j)$, and furthermore for all $k \in \beta$ objects their given set of g_k capacity constraints.
- δ is the set of soft-constraints, i.e. *objective functions*.

The task is to find a feasible, non-dominated *schedule* $S = \{A, T\}$, which is a pair of *activity* and *start time* vectors with the size of $m \leq n$, representing the scheduled activities and their start time. An S_1 schedule is

- *feasible*, if none of the set of hard-constraints denoted by γ is violated. In other words all precedence, resource and capacity constraints are satisfied in S_1
- *non-dominated*[4], if there is no objective function $f \in \delta$ and S_2 schedule, for which $f(S_2) \leq f(S_1)$ assuming minimization problem.
  Note, that a maximization problem can easily converted to a minimization problem, e.g.: for each $f_x|x \leq i$ objective function that have to be maximized, the $f_x(S) = -f_x(S)$ transformation should be applied.

Informally we have to allocate start times for a subset of activities, while this allocation does not violate any hard constraints and according to the objective functions it is the best solution amongst the found ones.

Recovery Fault tolerance is carried out via error detection and system recovery. Our method provides the latter, hence after the detection of an error (e.g. by proper monitoring mechanics) the recovery task will be executed. Its goal is to transform a system state that contains one or more errors and possible faults into a state without detected errors and faults that can be activated again[2, 5, 16]. The recovery problem can be formulated as $P_{recovery} = \langle \alpha_w, \alpha_r, \beta \rangle$, where

- $\alpha_w = \langle a_{failed}, a_{completed}, a_{incompleted} \rangle = \{a_1, a_2, ..., a_n\}$ is the same set of *workflow activities* that is described in the previous paragraph, but this time its partitioning to three subsets is also defined:
  - a_{failed} is the set of failed activities of the process workflow, i.e. the tasks, that failed during their execution as a consequence of an object failure. Hence failed activities are the ones, that did not terminated as expected.
    (Note, that besides the monitoring of objects, these activity failures can signal the presence of errors in the system, but proper detection mechanisms are still required to identify the exact active errors.)
  - $a_{completed}$ is the set of successfully finished activities of the process workflow, i.e. the tasks, that terminated properly before the occurrence of process failure, hence the effects of their completion could possibly modified the system's state.
  - $a_{incompleted}$ is the set of incompleted activities, of the process workflow, i.e. the tasks, that haven't even started operating before the occurrence of failure.
- α_r is the set of *recovery activities* which can be an empty set. In comparison of the process activities, instead of being a part of the business logic, recovery activities represent repairing tasks assigned to set of objects. This assignment is expressed the same way as the inputs and outputs of process workflow activities, i.e. all $j \in \alpha_r$ determines its duration d_j, a set of failed objects to be repaired as its input $In(j)$, and the set of fixed objects as its output $Out(j)$.
- $\beta = \langle \beta_{up}, \beta_{down} \rangle = \{R, C, P\}$ is the same set of objects, defined in the Scheduling paragraph, but again, with a partitioning of it is defined as well:
  - β_{up} is the set of objects, that are usable to use and flawless according to our observations of the system, i.e. objects that can be possibly used by activities, even without free capacity.
  - β_{down} is the set of failed objects, that can not be used by any $j_w \in \alpha_w$ workflow activity. Failed objects can only be allocated to $j_r \in \alpha_r$ recovery activities for repairing purposes.

Isolation The first part of recovering the system is the isolation step, since the dependency and control flow mechanics can enable the 'chain of threats' to be completed, i.e. by propagation, several errors can be generated before a failure occurs. Hence adherent to the detection of an error, the isolation step must be executed, which performs logical exclusion of the faulty objects from further participation in service delivery, i.e. makes the fault dormant.

Besides the separation of α_{failed} failed activities and the β_{down} failed objects, the isolation step also determine their $\sigma = \langle \alpha_{failed}, \beta_{down}, \alpha_{blocked}, \beta_{blocked} \rangle$ **error confinement region**, which is a logical environment of activities and objects around the detected errors. The error confinement region consists of:

- α_{failed} is the set of activities that are detected as failed (see Recovery paragraph).
- β_{down} is the set of faulty resources (see Recovery paragraph). It is noteworthy, that these resources stand as the *root causes* of system failures.
- $\alpha_{blocked}$ is the set of *blocked activities*, where $j \in \alpha_{blocked}$ can either be
  - (i) a $j \in \alpha_w$ activity, which *dedicatedly enables* a $k \in \{R \bigcap \beta_{down}\}$ failed renewable resource. In other words, $k \in Out(j)$ and there is no other $j' \in \alpha_w : k \in Out(j')$ activity, which enables the usage of k.
    The motivation of this rule is to eliminate activity executions, which could enable the usage of failed resources, since its usage could lead to further propagation of the detected error, e.g. an activity of `starting up` a faulty `Database server` can result in numerous erroneous transaction, hence the isolation of this activity is mandatory.

  It is noteworthy, that in case of a consumable resource, its failure can be masked by simply producing this item again, i.e. re-executing its j producer activity. For example in case of an erroneous `processed form` it can be reproduced by rerunning its producer `form processing` activity (as seen on Figure 1.3). If the error does not occur again, it was an intermittent fault of a corresponding component (e.g. `Customer and Account Identification` service), which does not necessitate reconfiguration. If the error recurs, then an error remained undiscovered.

  - (ii) a $j \in \alpha_w$, which uses at least one $k \in \beta_{down}$faulty resource. In other words, $k \in In(j)$.
    The motivation of this rule is to eliminate activity executions, which could use failed resources, regardless of its type, i.e. this rule applies to the activities, which uses failed either consumable or renewable resources, since using a faulty erroneously `processed form` (consumable) or the services of a faulty `Database server` may result in further error propagations.

- $\beta_{blocked}$ is the set of blocked resources, where $k \in \beta_{blocked}$ is a resource whose all producer activity is either failed or blocked. The motivation of this rule is to disable the usage of possibly erroneous resources by blocking them. For example if a `transaction form` could be processed by several activities and they are all erroneous, blocking the `transaction form` will signal in the business logic, that a request for this resource cannot be satisfied.

Informally speaking, we have to isolate activities and objects whose execution or usage could further propagate the erroneous state. The isolation step does not only protects the yet flawless components of a system by the confinement of the possibly faulty region, but it also reduces the number of possible process execution scenarios, i.e. the there can be no dead-end executions of a process. For example, if an error of `cashier module` gets detected, blocking only its dependent `record transaction` activity may still let the `form processing` and `money takeover` activities to be executed. The error confinement region will block these activities as well, avoiding the wasting and unnecessary executions of them.

Reconfiguration Adherent to the isolation step of recovery the reconfiguration step is to be executed. Generally speaking the goal of this step is to either switch in spare resources or reassign tasks among non-failed resources for two reasons, (i) to secure the interruption-free service of the system, and (ii) to recover the elements of error confinement region. This problem is approached from the cost minimization point of view, i.e. the step aims to reduce the amount of loss of work already done before the error detection and the price of actual reconfiguration as well.
The reconfiguration problem can be formulated as $P_{reconfiguration} = \langle \alpha_w, \alpha_r, \beta, \sigma \rangle$, whose elements are defined above. We have to emphasize, that α_r set of recovery activities can be an empty set, that represents the absence of activity definitions, which could repair failed resources. In this case, the reconfiguration step can only use the diversity and redundancy, built in the infrastructure and process workflow design, since removing errors of components is cannot be carried out automatically, manual assistance is needed. On the other hand, if α_r is not empty, this step can mix the activities that implements recovery and business logic (i.e. process workflow) tasks. In other words, the output of this reconfiguration is a set of trajectories (i.e. workflow) of recovery and business logic activities to carry out parallel recovering and service execution. This set of trajectories will be *scheduled* to obtain the cost minimized execution plan of this extended process.

Hence our job is to find a new resource allocation of resources to activities that satisfies the following criteria:

$$\forall\ k_l, j_i, j_m : k_l \in In(j_i) \bigcup Out(j_m) \iff (i)\ k_l \in \beta_{up} \setminus \beta_{blocked},\ and$$
$$(ii)\ j_i, j_m \in \alpha_r \bigcup (\alpha_w \setminus (\alpha_{blocked} \bigcup \alpha_{failed}))$$

Informally speaking, in the new allocation a k_l resource can be an input of j_i or output of j_m if and only if k_l is a usable (up), not blocked, and j_i, j_m can either be recovery activities, or workflow (business logic) activities, that are not failed or blocked. The exact method we used to determine such an allocation is described in the Optimization and Reconfiguration method in Alloy chapter.

2.3 Suggested method

Overview Generally speaking, our method is based on the principle of the *two-phase optimization* methodology, that is in order to enhance scaling and reduce computational complexity, initially a *construction phase* is executed, which generates a partial solution to the given problem. Adherently to this pre-optimization phase, the *improvement phase* is performed on the resulting reduced state space, which makes the problem space significantly easier to compute. Our current work focuses on the *construction phase*, which provides both a *partial solution* and a *reconfiguration state space* as well, that can be elaborated through an *optimization solver interface* such as the Minizinc constraint modelling language [19] or the JSR-331 constraint programming API [6] to find a complete scheduling. The usage of such solver interfaces may also allows us to run the improvement phase on different solver engines, based on the characteristics of the construction phase's result, making it possible to exploit the different strengths of the supported solvers.

Our method's workflow is shown on Figure 2.1. as a BPMN workflow, where the *activities* represent the logical steps of our method, and the *data objects* represent the data (such as models and problem descriptions) used or produced by the activities. In this workflow, the direction of dependency arcs (dashed arrows) play an important role, that is the incoming dependencies of an activity represent its inputs, while the outgoing ones represent its outputs. The color of data objects aims to capture the similar usage modes of data objects, such as:

- Orange: workflow input data objects, they cannot be produced by any activities of our method.
- Green: internal data objects, they are only produced and used by activities of our method.
- Blue: data objects that is the mixture of above, they can be workflow inputs and also produced (simulated) by activities
- Red: workflow output data objects, these are the data that our method provide as service.

Problem relaxation steps

Transform problem to PNS model This step stands as a cornerstone of our method, where we perform an abstraction to express only the desired properties of the system and its process description, while eliminating undesired ones [20]. The translation rules of this abstraction function is defined in the IT Process to PNS translation section, that are applied to the input `Problem model`, which removes its numerical parameters (such as cost functions, capacity constraints, etc.). As a result of this function, the output `PNS model` will only express the structural constraints of the process and the underlying infrastructure. The structural constraints refer to the previously described *hard-constraints*, i.e.: the precedence and dependency constraints of the system.

The result of this step is then used by the *maximal structure generation* to find the solution space of structurally (combinatorially) feasible solution. It is also noteworthy, that even though we used BPMN `Problem model` to represent our case study, we could also use other methods for this purpose, such as the ArchiMate[25], which is an Open Group Standard and modelling language, that is capable of describing, analyzing and visualizing relationships among business domains. In conclusion, this step provides the relaxed problem description, which is elaborated by later steps.

Transform process instances to PNS model This step extends the previous one activity's function of abstracting (i.e. relaxing) the input problem, hence the usage of and-gateways in the workflow description, which semantically represents simultaneous execution of the two steps. While the `Transform problem to PNS model` step created an abstraction from the problem model, this task applies the same translation rules to the currently active instances of the problem model's process. Thus the process instances describe the current resource allocations of each instances, i.e. the current state of the process execution. The output of this step is a set of solution structures (see chapter Background). The output of this step is the input of the recovery step to calculate the $\alpha_w = \langle \alpha_{failed}, \alpha_{completed}, \alpha_{incompleted} \rangle$ partitioning of the process workflow activities, in order to minimize the loss of work already done before the error detection. If there are no active process instances (i.e. the system is idle), there are no allocated resources and we do not have to calculate with the effects of previous executions. Errors can still be detected by external monitoring services or can be simulated. In conclusion, this step provides the snapshot of the system at the time of error detection, projected to the relaxed problem.

Steps of introducing errors

Get component changes This step embodies the function of reading the data provided by an error detection service. Hence its output is the set of erroneous elements.

Simulate component changes This step comes into play if there are no erroneous elements detected by an error detection service, if there is any. Thus this activity represents our implementation of error simulation, which randomly flags a subset of resources as faulty. Note, that the exclusive-gateway is used to represent the relationship of the task of change simulation and change reading, i.e. only one of these can be executed and produce the `change model`.

Inject changes This step simply set faulty the elements of `PNS model` which correspond to the faulty components listed in the `change model`, thus producing the `changed PNS model` output.

Solution space reduction and Recovery steps

Process state space reduction Generally speaking, this step produces the solution space of the previously relaxed problem elements, which is carried out via the execution of MSG algorithm, described in chapter Background. The output of this algorithm is the reduced solution space, `Maximal structure` and the `impact estimation` of changes, i.e. the *error confinement region* and the new elements introduced to the process for example by a repairing activity. It is noteworthy, that the elements of error confinement region and the maximal structure form disjunct sets, i.e. blocked components cannot be parts of the solution space.

Calculate reconfiguration trajectory space This function produces that space of all possible reconfiguration trajectories of the system. Generally speaking, this step takes $\alpha_r$recovery activities into account as well, and it adds them to `maximal structure` to its corresponding resources. Then it removes the execution paths that cannot be used in any reconfiguration trajectory, which is carried out via the above mentioned MSG algorithm. The result is the space of possible reconfiguration trajectories, which will be used by the reconfiguration step.

Numerical optimization

Scheduling This step embodies the function of finding the numerically feasible and non-dominated scheduling to the problem, as described previously, which is carried out via the usage of an also previously mentioned solver interface, and by exploiting the generated reduced solution space.

Reconfiguration and scheduling This task is an extended version of the `Scheduling` step, i.e. the scheduling is not only executed over the reduced set of α_w workflow activities, but the reduced set of α_r reconfiguration activities are taken into account as well. Note, that the numerically feasible and cost minimized reconfiguration trajectory cannot be generated separately from the scheduling, since the $j_i \in \alpha_w, j_l \in \alpha_r$ activities can use the same resources, thus they can either conflict, and joining the two non-dominated solutions could result in a non-feasible, or dominated solution.

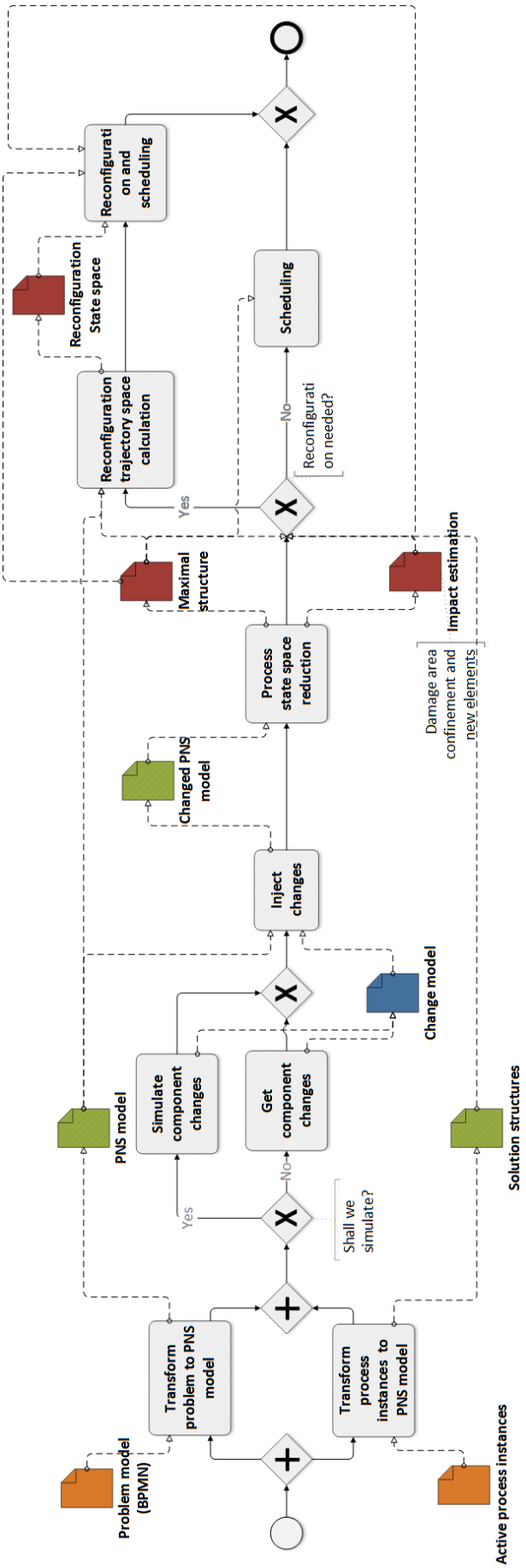


Figure 2.1. Method workflow

Chapter 3

Background

In this chapter, we briefly introduce optimization problems from a more generalized perspective based on [21], along with the main modelling and optimization paradigms(i.e. *Constraint Programming*, *Boolean Satisfiability* and *Mixed Integer Linear Programming*), to provide a further understanding of the motivation of our method. Adherently we present the main properties of the problem representation we used to simplify the problem on the theoretical field (i.e. *PNS*). Finally we present the basic characteristics of the modelling language we used to practically solve exact problems (i.e. *Alloy*).

3.1 Optimization problems

Generally, optimization problems can be formulated as

$$z = minf(x) \tag{3.1}$$

$$C \tag{3.2}$$

$$x \in D \tag{3.3}$$

where z is the value of *objective function* $f(x)$ of the variable x to be minimized, D is the *domain* of x, while C is the finite set of constraints to be satisfied. Then we call x a *solution*, which is *feasible* iff it satisfies all the constraints of set C. The x^* solution is called *optimal*, if $f(x^*) \leq f(x)$ for all feasible x.

Note, that in section Scheduling we described a specific part of optimization problems, for which $f(x)$ is a composition of objective functions denoted by δ, the C constraints is represented by γ and D domain was denoted by $\alpha \bigcup \beta$ set of activities and objects respectively.

The *Constraint Programming* (CP) paradigm is used to declarative model and effectively solve large optimization problems, and solves *Constraint Satisfaction Problems* (CSP), which can be formulated as a triple $\langle x, D, C \rangle$, where

- $x = \{x_1, x_2, ...x_n\}$ represents the set of *variables*
- $D = D_1 \times D_2 \times ... \times D_n$ represents the *domain* of the above mentioned variables
- C represents the set of *hard-constraints* restricting the values that can be assigned to the variables.

The task of solving CSP defined by the above properties is to find one feasible solution, all feasible solutions or prove that there is no feasible solution. If an f objective function is given, then we may want to find optimal solution to the problem as well. The CSP includes satisfiability problems as special case, hence CSP is $\mathcal{NP}$-hard.

The *Boolean Satisfiability Problem* (SAT) is a specific case of CSP, where the problem is to determine whether the variables of a given formula of propositional logic can be assigned so, that the variables satisfy the formula, or prove that no such assignment exists. The formula in propositional logic is expressed *clauses*, where a clause is a disjunction of *literals*, that are atomic propositions or their negations. SAT problems may seem simple compared to the other discussed problems, they are also $\mathcal{NP}$-hard, which was proven first to be such.

The *Mixed Integer Linear Programming* (MILP) are problems, that has *linear* objective functions and constraints, while being *mixed integer* because noninteger values are also allowed in the problem[13]. MILP can be formulated as

$$z = minc^T x \tag{3.4}$$

$$Ax \leq b \tag{3.5}$$

$$l \leq x \leq u \tag{3.6}$$

$$x \in \mathbb{R} \tag{3.7}$$

$$x_j \in \mathbb{Z} \forall j \in I \tag{3.8}$$

where $c^T\ x$ is the *objective function*, $Ax \leq b$ represent the set of linear constraints, l and u stand for the *lower* and *upper bounds* of variable $x$respectively and I denotes, that the variables must be integers. A special case of the above model when the integer values are bound as $0 \leq x_j \leq 1$, i.e. variables are *binary*. Since SAT is a special case of such a *Binary Program* (BP), BP and most MILP are $\mathcal{NP}$-hard. Finding a solution to a MILP problem is typically carried out via the execution of *branch and bound* (BB) algorithms.

Exceptions are *Linear Programs* (LP), for which $I = \emptyset$ holds. LPs were proven to be polynomially solvable, although in practice the *simplex* non-polynomial algorithm is used to solve them. Also its noteworthy, that MILPs are mixed, since noninteger variables

The recent solution methodologies for resource-constrained scheduling problems are typically use the concept of MILP, which is shown to be $\mathcal{NP}$-hard, hence the same applies to the scheduling subproblem, that our method partially focuses on.

3.2 The PNS problem

The *Process Network Synthesis* (PNS) problem is a MILP problem, which originates from the domain of chemistry. Informally speaking PNS problems are production problems, where the desired products of the given process must be produced, which is carried out by the consecutive and simultaneous executions of *operating unit* actions, that transform their input *materials* into output materials. Operational units may have attributes, that describe their cost of execution, such as fix cost and proportional cost.

3.2.1 P-graph representation

A process graph (P-graph)[8] is a simple and intuitive way to represent a PNS problem. A P-graph $\langle Mat, Op \rangle$ is a directed bipartite graph where the two disjoint, sets of nodes are:

- Mat which represents the finite set of materials, symbolized by circles. Materials are further classified by a partition to 3 disjoint set of materials:
  - $Raw \subset Mat$ raw materials are the ones that cannot be outputs of operating units
  - $Inter \subset Mat$ intermediate materials are the ones that can serve as an input and output of any operating units as well
  - $Prod \subset Mat$ products are the ones that can only be produced, i.e. no operating unit can use use them as inputs.
- Op which represents the finite set of operating units, symbolized by horizontal bars. For the set of operating units $Op \subseteq Mat \times Mat$ and $Mat \bigcap Op = \emptyset$ hold.

Arcs originate from a material with potential loss towards an operating unit (i.e. a consumer), or from an operating unit (i.e. a producer) towards a material with a potential gain. This property indicates that the directions of arcs are usually identical to the direction of material flows in a process. Such representation was originally dedicated to describe the material flow of chemical processes, but it's also suitable for describing the data flow of an IT infrastructure, thus the potential error propagations as well. Such representation is described in section IT Process to PNS translation. An example P-graph is shown in Figure 3.1.

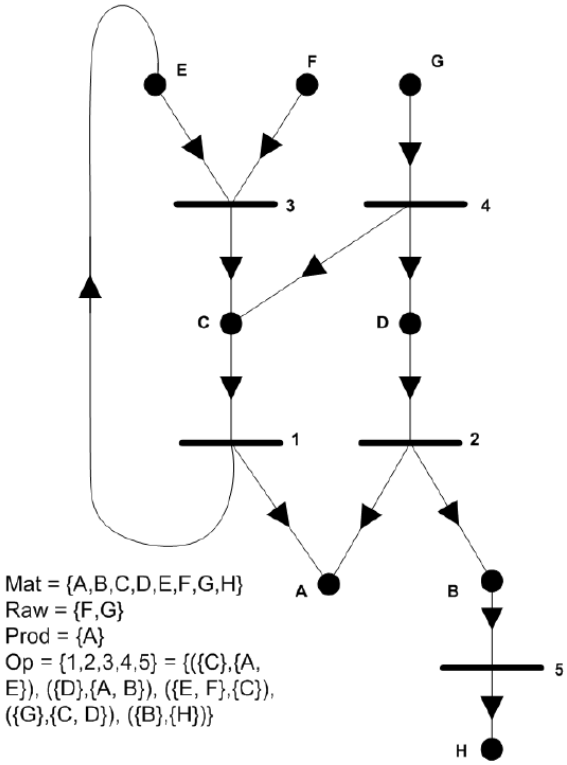


Figure 3.1. An example P-graph representing a simple PNS problem [22]

3.2.2 Process structure

The structure of a process can be defined as a P-graph of a PNS problem as described previously. The possible solutions on the other hand, are represented by *solution structures* (SS), that are P-graphs which conform the following necessary and sufficient combinatorial properties:

1. Every product of the PNS problem is represented in the P-graph.
2. A material has no input operating unit (i.e. producer) iff the material is raw.
3. Every operating unit in the P-graph represents an operating unit in the PNS problem.
4. Every operating unit has at least one path leading to a product of the PNS problem, thus there are no isolated (and non-functional) operating units in the P-graph.
5. Every material is an input or an output (or both) of an operating unit, thus there are no isolated materials in the P-graph.

Hence all the solutions of a given PNS problem can be represented by corresponding solution structures, i.e. subgraphs which conform the above listed axioms. We use this interpretation to describe the instances of a given IT process, detailed in section IT Process to PNS translation. The set of solution structures of the previously shown example PNS problem can be seen on Figure 3.2.

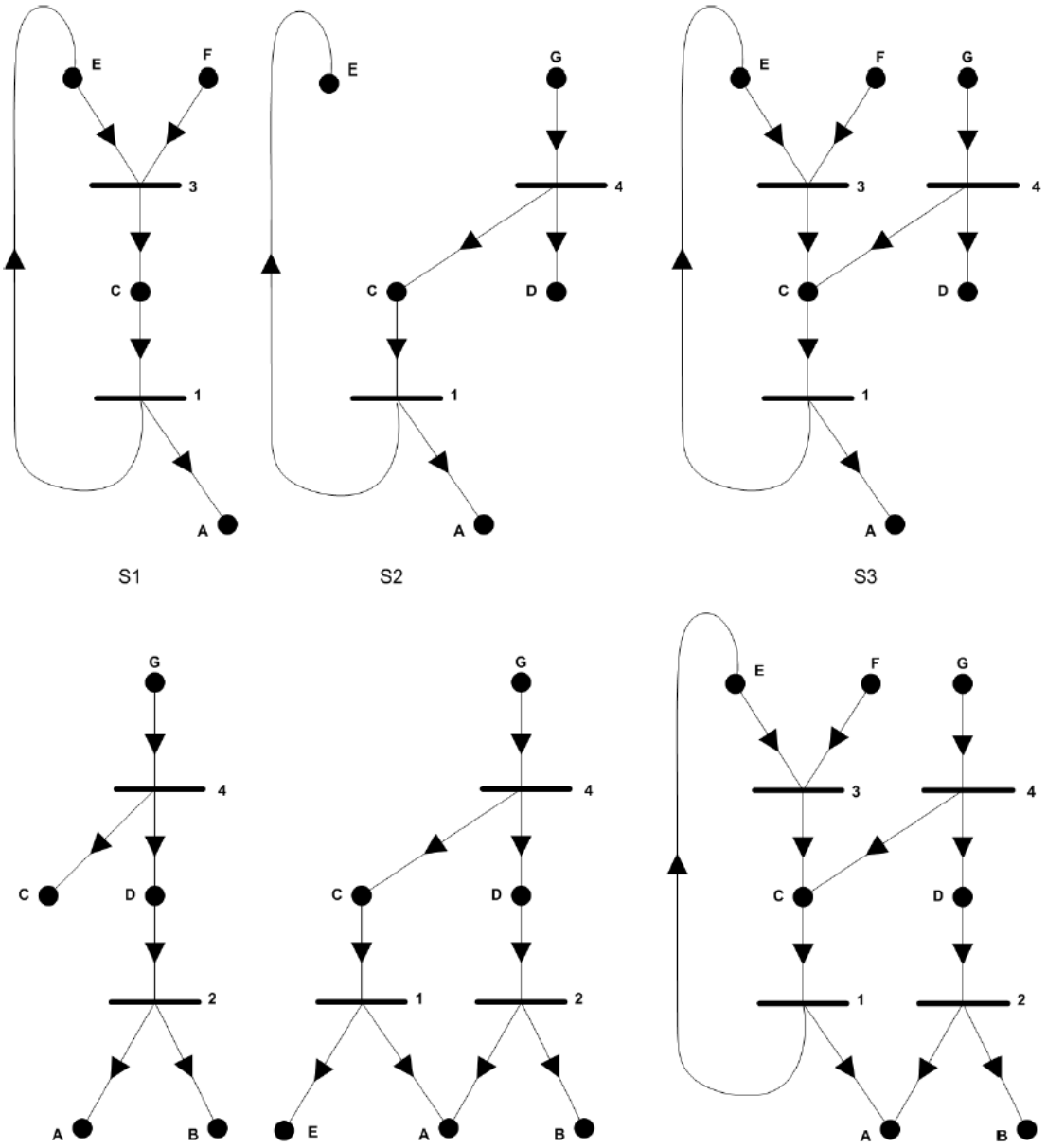


Figure 3.2. P-graphs denoting all the solution structures of the PNS problem shown in Figure 3.1 [22]

Another artefact of the PNS concept of process structure description is the *maximal structure* (MS). Since the set of combinatorially feasible solution structures is closed under union, the superstructure of these solution structures form the maximal structure, i.e. a solution structure which contains all the solution structures. In other words, a node is present in the P-graph of the maximal structure if and only if that node is present in at least one of the P-graphs representing the solution structures of the PNS problem. Hence we can use this concept of MS to represent the possible execution paths of an IT infrastructure.

A *maximal structure generation* (MSG) algorithm is proposed in [9], which generates the superstructure of the combinatorially feasible solution structures by eliminating the un-

necessary materials and operating units in polynomial time. We exploit the characteristics of the MSG to find all the possible execution paths of a given system process.

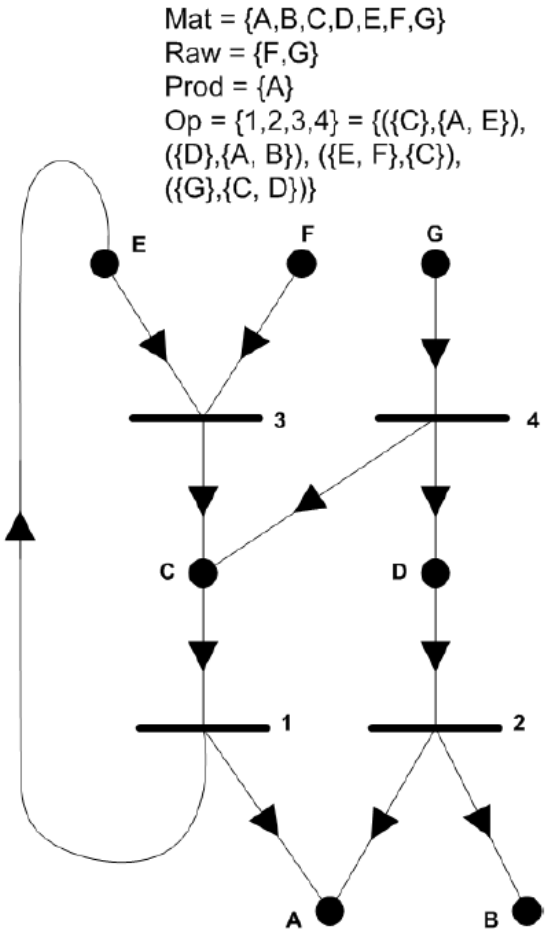


Figure 3.3. P-graph denoting the maximal structure of the PNS problem shown in Figure 3.1 [22]

3.3 Alloy

3.3.1 Overview

Alloy is a language and a tool for describing structures and analyzing them, developed by the Software Design Group at MIT [14] [15] . It has an expressive and flexible first-order logic based declarative language with relational algebra, that is used to define complex structure models. The structures that satisfy the constraint of the model in Alloy can be analyzed by the Alloy Analyzer tool. It can both explore or check certain properties of the model, by generating sample structures or a counterexample structure respectively. The Alloy model is efficiently translated into a SAT problem and can be solved by a variety of powerful SAT-solvers. The output of the SAT-solver, which contains the structures that satisfy the model, is translated back into Alloy, and can be viewed and evaluated by queries. In addition, Alloy has a customizable built-in visualizer feature as well, that can visualize the solution structures found by the Analyzer.

3.3.2 Modelling in Alloy

Core building blocks At its core, Alloy's language has predicate logic and relational calculus, i.e. relational logic. Hence, it has multiple ways of expressing constraints and structures, from using quantifiers known in predicate logic, to using natural join known in relational algebra. Moreover, Alloy uses atoms as primitive entities (which are indivisible, immutable, uninterpreted) and relations which associate atoms. A relation is a set of ordered tuples of atoms, i.e. a table, where atoms in a specific tuple means they are related. Furthermore, their arity is the number of atoms in each tuple, i.e. number of columns, and their size is the number of tuples, i.e. number of rows. In fact everything that can be described and used in the language itself is a relation. Additionally, there are three main ways a user can view and read the syntax of an Alloy model, from the highest abstraction level to the lowest:

- as an Object-oriented programming model
- using set theory (sets, elements, relations)
- as atoms and relations, i.e. the real formalism behind the language

The most intuitive way is to use the second option, set theory. Although there are two types of expressions, relation-valued and boolean-valued, the grammar of Alloy does not distinguish one from the other. Nevertheless, they are disjoint.

Basic elements Listing 3.1 shows the basic signature element, that can be used to describe structures declared using the `sig` keyword. A signature is a unary relation, i.e. table with one column, that can also be viewed as a simple set. For example, `sig Person` represents a set that contains Person instance elements. The fields `driver` and `passenger` in the `sig Taxi` are binary relations, i.e. tables with two columns, that are containing binary tuples between concrete elements in the form of $(Taxi, Person)$. For the `driver`

Listing 3.1. Signature examples

```
sig Person {}

one sig Truck {
       driver: Person
}

sig Taxi {
       driver: Person,
       passenger: set Person
}
```

in `sig Truck` it is in the form of $(Truck, Person)$. These relations can also be viewed as mappings from one set's element to another set's element.

Moreover, the multiplicity of each relations and sets are defined as the following: for signatures, if nothing is written explicitly, it is by default a `set` multiplicity, i.e. there can be any number of `Taxi`. For relations, `one` multiplicity is the default, as a Taxi has exactly one driver, i.e. exactly one `driver` tuple exists with each specific `Taxi` instances. Whereas a Taxi can have multiple passengers, even zero, denoted by the `set` keyword, i.e. there can be multiple `passenger` tuples with each specific `Taxi` instances. As we can see, there is exactly one element in the set `Truck`. The remaining multiplicity keywords are `lone`, that is zero or one, `some` which means at least one, and `no` means zero elements.

As Alloy contains first-order logic, the relations cannot contain other relations in their tuples, only signatures. Most importantly, Alloy creates a solution structure by instantiating the defined signatures and relations, i.e. creating set elements in the signature sets and relations between these concrete set elements, along with satisfying all other constraints and scopes.

Furthermore, Alloy has three important set constants.

- `none`, which is the empty set
- `univ`, which is a set, that contains all concrete signature elements
- `iden`, which is the identity relation, i.e. every concrete signature element is in a tuple with exactly itself

Set operators The basic set operators work between relations that have the same arity, i.e. between two arbitrary signatures, or two arbitrary binary relations.
Operators that return a set, relation:

- union: `a + b` returns a set, relation that contains all elements of `a` and `b`
- intersection: `a & b` returns a set, relation that contains elements that are both in `a` and `b`
- difference: `a - b` returns a set, relation that contains elements that are in `a` but not in `b`

Operators that evaluates to a boolean value:

- subset: `a in b` evaluates to true if `a` is a subset of `b`, i.e. all elements from `a` is in `b` as well
- equality: `a = b` evaluates to true if `a` and `b` contains the same elements

The number of tuples in a set, i.e. the size of the relation, the number of instances can be counted with the `#` operator, for example `#Taxi` is an integer number that equals to the number of `Taxi` set elements. Note, that `none in a` returns true for every set `a`.

Listing 3.2. Example of inheritance

```
sig Person {}

abstract sig Vehicle {
       driver: Person
}

sig Truck extends Vehicle{}

sig Taxi extends Vehicle {
       passenger: set Person
}
```

Inheritance With the `extends` keyword, a signature can extend another signature, i.e. inherit its fields and constraints as Listing 3.2 shows. `Taxi` and `Truck` are disjoint subsets of `Vehicle`, i.e. they partition the parent set, and inherit the `driver` relation, which is in the form of $(Vehicle, Person)$. Therefore, the set of `Vehicle` and the left-side of the `driver` tuples contain elements that are `Taxi`, `Truck`, or simply `Vehicle`. Additionally, we can use the keyword `abstract` to prevent `Vehicle` from having instances that are not from its subsets, i.e. the set of `Vehicle` contains elements that are either from `Taxi` or `Truck`.

Navigation In Alloy, navigation can be achieved by using the relational join operator, unary operators and the set operators already discussed. Due to the fact that everything is a relation in Alloy, and relations can be addressed outside the signature they were defined in, we can use relational join between any signatures or relations by utilizing the `.` dot join operator. The resulting product is also a set, but as the natural join is not commutative, switching the order of the operands yields a completely different product, and might not even be meaningful, i.e. an empty set. In addition, the box operator `[]` can also be used for relational join, in the form of `b[a]`, which is the same as `a.b` with dot join. Listing 3.3 shows a few examples in Alloy.

Furthermore, Alloy's unary operators are the following:

- transpose: `~driver` is the transposed `driver` relation, i.e. in the inverted form of $(Person, Vehicle)$
- transitive closure: `a.^b`, the set of all elements that can be reached from `a` by repeatedly joining with relation `b`, but where `a` is not included

Listing 3.3. Relational join examples

```
//if two subsets have the same body, they can be defined on the same line
sig Man, Woman extends Person{}

               ...

       //return a set of all persons who drives a vehicle
       Vehicle.driver

       //return a set of all persons who are travelling in a taxi
       //NOTE: the union of driver and passenger binary relations
       //is a binary relation with every tuples from driver and passenger
       Taxi.(driver + passenger)

       //return a set of all vehicles where the driver is a woman
       //REMINDER: driver contains (Vehicle, Person) tuples!
       driver.Woman //or with box join: Woman[driver]

               ...
```

- reflexive transitive closure: `a.*b`, the same as above, but `a` is included as well

Note, that in order to have a meaningful transitive closure in the case of `a.^b`, `b` must be in the form of (a, a) or any subset of `a`. Listing 3.4 shows an example usage of transitive closure for the famous Erdos number.

Listing 3.4. Transitive closure example

```
sig Mathematician{
       coAuthor: set Mathematician
}

one sig ErdosPal extends Mathematician {}

               ...

       //return the set of all mathematicians
       //who has a finite (i.e. defined) Erdos number,
       //including Paul Erdos himself
       ErdosPal.*coAuthor

               ...
```

Ternary relations The cross product operator `->` can be used to define ternary relations, i.e. tables with three columns. Moreover, we can set the multiplicity as in the case of binary relation, but the default is not `one`, but `set` multiplicity. Listing 3.5 shows an example, where `sig Project` contains a ternary relation `superior` in the form of $(Project, Person, Person)$. For each element in the set `Project`, there is a set of binary relations between `Person`. A specific tuple in the `superior` relation shows that in a `Project` a `Person` has a superior other `Person`. Furthermore, the multiplicity implies that in a specific `Project` a `Person` can have zero or one superior, and zero or more subordinates.

Domain and range restriction The operators domain restriction (`<:`) and range restriction (`:>`) are used for getting a set of relations where the left, or right tuple element

Listing 3.5. Ternary relation example

```
sig Person {}

sig Project {
       superior: Person set -> lone Person
}
```

is from a specific set. For a signature `a` and a binary relation `b`, `a <: b` returns the set of tuples from `b`, where the left tuple element is from `a`. Whereas `b :> a` returns the set of relations from `b`, where the right tuple element is from `a`.

Boolean operators and quantifiers We can use boolean operators between expressions that return boolean values. There are two ways of writing each of these operators, except the alternative operator `else`.

- negation: `!`, `not`
- conjunction: `&&`, `and`
- disjunction: `||`, `or`
- implication: `=>`, `implies`
- bi-implication, i.e if and only if: `<=>`, `iff`
- alternative: `else`

Alloy's quantifiers can be used in the form of `Q x: s | E`, where `Q` is a quantifier, `x: s` is a variable from the set `s`, and `E` is a boolean expression of `x` that holds true based on `Q`. The quantifiers are the following, expression `E` is true for

- `all`: every
- `some`: at least one
- `one`: exactly one
- `lone`: zero or one
- `no`: zero

element `x` from set `s`. In addition, quantified expressions can be created by using the `some, one, lone, no` quantifiers before a relation-valued expression, similarly as the multiplicity in front of a signature definition. For example, `some (a & b)` means the intersection of sets `a` and `b` must not be empty.

Facts We can define constraints in the body of a `fact` with the keyword `fact`, or as an appended fact in a signature definition after its field definitions. These constraints must always hold for every valid model instance. Listing 3.6 shows examples of facts. In an appended fact's body, we can always assume an implicit `this.` relational join in front of every relation that was defined in the signature of the appended fact. Therefore, every relation that the signatures has can be used as sets instead of binary relation in appended facts.

Listing 3.6. Examples of facts

```
abstract sig Person {}

sig Man, Woman extends Person{}

sig Taxi {
       driver: one Person,
       passenger: set Person
}{
       //appended constraints of the signature Taxi

       //The driver cannot be the passenger of the Taxi!
       //Note the implicit (this.) before "driver" and "passenger" relations
       //i.e. here the "driver" is the driver of this signature
       //"passenger" is the set of passengers of this signature
       driver & passenger = none

}

//some fabricated constraints
fact someConstraints {
       //there must be some taxi, where the driver is a woman
       some p: Person, t: Taxi | (t.driver = p) and (p in Woman)

       //every person must be a passenger or a driver of a taxi
       all p: Person | some (passenger.p + driver.p)

       //every existing taxi passenger must be a man
       Taxi.passenger in Man
}
```

Predicates and functions Predicates (`pred`) and functions (`fun`) are language elements that can make an Alloy model more compact. They can be viewed as templates, that can be instantiated multiple ways in the model. Both work on relational parameters, the former returns a boolean value, while the latter returns a relational value. Listing 3.7 shows a few examples.

3.3.3 Using the Analyzer

Assertion and check After defining structures and constraints, we can define assertions with an `assert` block, which is similar to a `fact` block. However, the constraints defined inside an assertion are properties of the model that we want to investigate, i.e. whether they hold and implied by the the model itself, or there is a counterexample that refutes them. Furthermore, assertions can be checked by the Analyzer, using the `check` command. In the finite scope defined, Alloy negates the assertion constraints and tries to find a model instance, i.e. a counterexample.

Listing 3.7. Predicates and functions

```
              ...

//returns true, if the given Taxi and Person instance
//satisfies the body of the predicate
//i.e. the driver is a woman
pred womanDriver [tax: Taxi, pers: Person] {
       (tax.driver = pers) and (pers in Woman)
}

//return all taxis of a person
//set Taxi is the return value here
fun taxiOfPerson [pers: Person] : set Taxi {
       passenger.pers + driver.pers
}

//using the predicate and function
fact examples{

       //there must be some taxi, where the driver is a woman
       some p: Person, t: Taxi | womanDriver[t, p]

       //every person must be a passenger or a driver of a taxi
       all p: Person | some taxiOfPerson[p]

}
              ...
```

Predicate, function simulation In addition to checking for counterexamples of an assertion, we can also simulate functions and predicates by using the `run` command. In the finite scope defined, Alloy generates example model instances that satisfy the given predicate or function. If Alloy finds no instance, the model is likely to be inconsistent.

Executing search Alloy uses bounded exhaustive search to find a counterexample, or a sample model instance. First, the structures and constraints of the model in Alloy is translated to the language of Kodkod, which is a model finding engine that optimizes the reduction of the model to its likewise relational logic language [28] [29]. Kodkod uses efficient finite SAT-based constraint solving for finding a model or a minimal unsatisfiable core [27]. From Kodkod, the model is translated to a SAT problem in CNF form, which is then solved by a SAT-solver the user specified.

Alloy uses finite scope check, because if an assertion is wrong, we can usually find small counterexamples for it, and hence there is usually no need to search larger spaces. Therefore, Alloy checks all cases within a finite (small) bound with SAT-solvers, which have become very effective throughout the years for solving hard problems despite the fact that SAT problems themselves are NP-complete [3] [11] . Furthermore, the solution model instances are translated back to Alloy, and we can enumerate on the sample model instances or counterexamples, and therefore can return all non symmetrical (by utilizing symmetry-breaking) model instances, counterexamples of the specified model one-by-one.

Evaluator and visualizer The model instances returned by the SAT-solver can be queried by the same expressions that can be used to define a model in Alloy. However, a model instance has concrete signature, relation elements (for example, a concrete element

of the set `Person` would be `Person$0`, and `#Person` would return the concrete number of elements in `Person`). Therefore we can create expressions using the concrete instance elements as well.

The visualizer of Alloy visualizes the model instance as a graph, where nodes are signature elements, arcs are relations between them. We can also project the model instance on any signature, and customize the colors and shapes of the elements.

Chapter 4

Modelling approach

4.1 Overview

Our motivation is to handle all structural models and tasks of our method workflow shown in Figure 2.1 in one place, with one tool. Thus, we can take advantage of Alloy's flexible modelling language when constructing our models, and structural modelling capabilities when modelling changes in our process system. As already discussed in Chapter 2 our method is based on the principle of the two-phase optimization methodology, and focuses on a construction phase, where we reduce the search-space by considering structural feasibility.

First, we create a PNS representation of the process instance and problem definition, and extend the PNS definition with state properties of availability, and error for Materials. Subsequently, we translate the resulting P-graph model into an Alloy model with appropriate structures and constraints for solving the task of our method. The structural model instance that Alloy finds contain the structural models of our method workflow. We performed the following steps in Alloy:

- the definition of translational rules from PNS to Alloy's modelling language,
- extension and implementation of basic PNS algorithms and axioms,
- creation and calculation of initial MS and SS
- specification and simulation of arbitrary Resource configurations, i.e. errors and availability,
- in case of recovery, the calculation of current MS and a reconfiguration alternative for the original SS.

4.2 IT Process to PNS translation

In this section, we present the translation rules we used for the translation of BPMN model to the PNS problem. As an example, the model of Figure 4.1 is the resulting model after the application of the rules listed below to the model of Figure 1.1. It is noteworthy, that there is a tool [24], which can make this translation automatically as well.

- An *activity* is represented by an *Operation* of the PGraph. (e.g. `Process Form`)
- If an execution flow element is a:
  - BPMN sequence (i.e. arc), it will be translated to a material, representing the execution token flow between the activities (e.g. `Processed form`)
  - BPMN AND gateway will be translated to a single operation. Then the outgoing sequences of the gateway can be mapped by using the previous rule, i.e. by materials representing the outgoing execution token.
  - BPMN XOR gateway is translated to the composition of one operation and one material, which serves as a semaphore. Then the output sequences of the gateway can be translated according to the above rules. It is noteworthy, that the operation-material pair representation can be eliminated, as they do not express additional business logic. Hence in the case of `Large transaction` gateway, its input `transaction recorded` execution token can immediately transferred to any of the following operations, thus we can reduce the number of elements of the P-graph.

  In case of objects we have to consider their role and type as well. For this problem two separate sets of translation rules are given, and to properly translate an object to a PNS element, one corresponding rules from both sets must be applied.
- Considering its role, if an object is an:
  - *input* resource of the process, then it will be translated to a corresponding *Raw* material. (e.g. `transaction request form`)
  - output, i.e. *product* of the process, then it will be translated to a corresponding *Product* of the P-graph(e.g. `receipt`)
  - *input and output resource* of the process activities as well, then it is translated to an *Intermediate material* (e.g. `Transaction recorded`)
- Considering its type, if an object is an:
  - renewable resource, then an arc has to be defined in both direction between the object and its consumer operation, hence the execution of the operation will enable the material again. Note, that arcs marked with '*****' symbol represents such loops, i.e. to model renewable resources in the P-graph. E.g. there is an arc pointing from `backend server1 online` to `backend server1` marked with '*****'.
  - consumable resource, then an arc has to be defined pointing from the resource to its consumer operation. (e.g. `Transaction recorded`).
- The dependency relations' translation rule is similar to the sequence's translation rule, i.e. an operation is defined between the two objects, which represents the usage of that object. If the object cannot be used, then in the BPMN model will cancel its

dependent object's usage via the dependency arc, while in P-graph, the operation representing the usage of the object will not be enabled, because of the faulty input material.

Note, that in case of the dependency relation connects an object and an activity, then a simple arc is enough to represent the dependency.

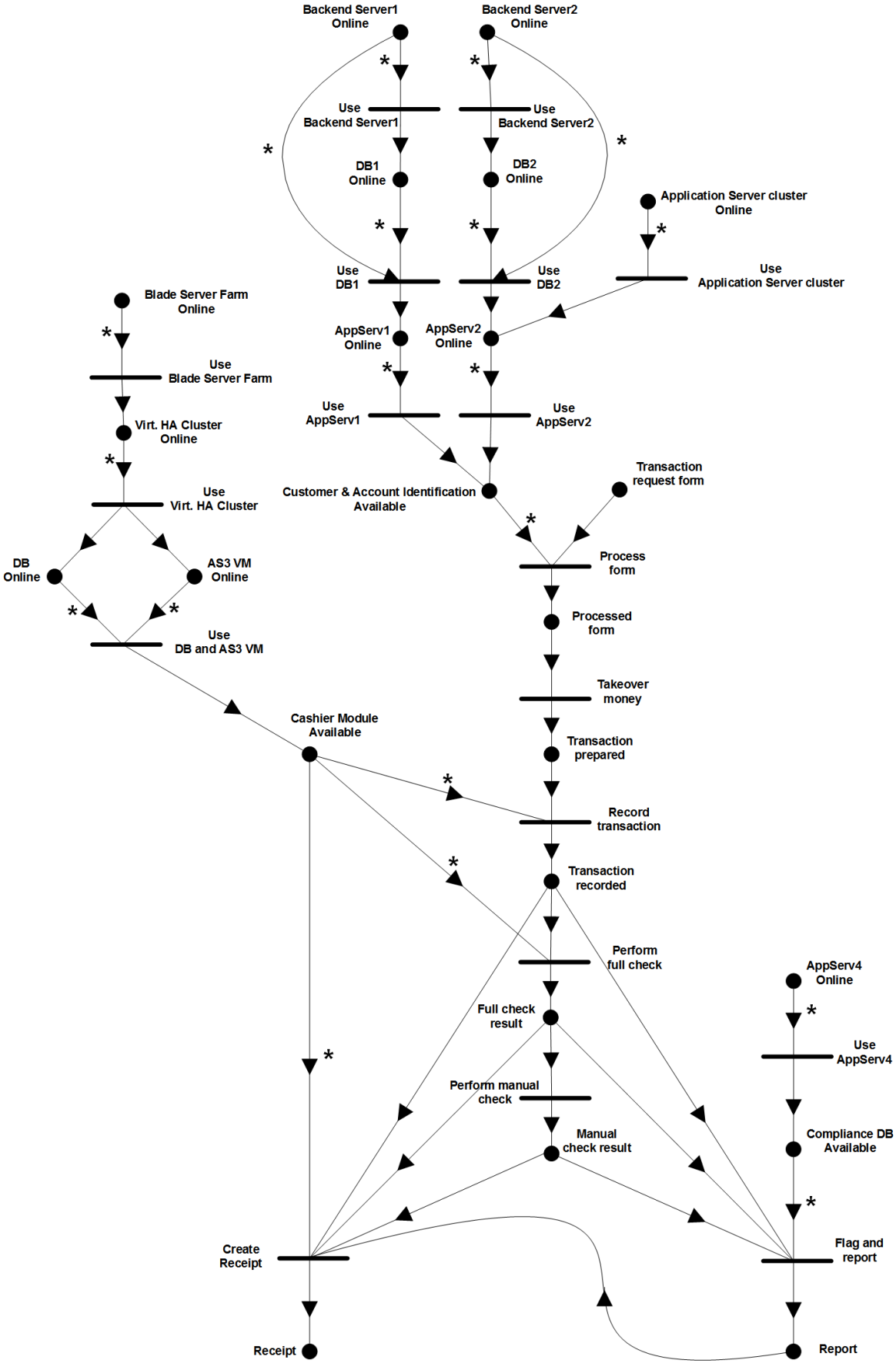


Figure 4.1. P-graph translation of the case study

4.3 Defining P-graph in Alloy

Overview Our approach to modelling a P-graph (M, O) is to model the sets M and O as abstract Alloy signatures, and create separate signatures for each concrete materials and operations. The SS and MS are represented as signatures with relations to their Material and Operation components. Furthermore, the SS must satisfy the axioms defined, while the MS is generated with the help of auxiliary structures.

4.3.1 Materials and Operations

As P-graphs are bipartite graphs, we use this aspect on our model, namely every Material and Operation is a `Node` signature. Consequently, an `arc` relation is present between two nodes only if a directed arc is present in the P-graph as well. Also, we enforce the bipartite structure with an Alloy fact, i.e. every `arc` is between a `Material` and an `Operation`. An `arc` from a `Material` m to an `Operation` o means that the o uses m as input, and on the other way around it means that m is produced by o. We define the three group of Materials as extensions of the `Material` signature. Moreover, `Raw` materials have a sanity constraint, that no `Operation` can produce them.

Listing 4.1. Metamodel of P-graph

```
abstract sig Node {
    arc: set Node
}
/********
MATERIALS
********/
abstract sig Material extends Node{}

//Material subtypes
abstract sig Raw, Inter, Product extends Material{}

/****************
OPERATIONS
****************/
abstract sig Operation extends Node{}

/****************
    FACTS
****************/
// Materials can only be connected to operations and vica versa
//Sanity constraint: Raw materials cannot be produced by any Operation
fact PGraphProperties{
    all m: Material | m.arc in Operation
    all o: Operation | o.arc in Material

        all r : Raw | arc.r = none
}
```

In order to capture the structure of a concrete P-graph as Alloy model, and control precisely what instances are created in Alloy, we defined for every P-graph element a concrete signature, which is instantiated only once. As seen in the source code on Listing 4.1 the signatures discussed so far are all abstract on purpose. Conversely, all concrete elements are defined as signatures that are not abstract, and extended from either one of the three `Material` types, or from the `Operation`. Moreover, these signature have a multiplicity

of `one`, in order to be instantiated exactly once. Hence, the `arc` relations defined are created correctly as well. In conclusion, our model is ultimately a static definition, that is referenced and used by the other structures discussed later.

4.3.2 Example translation of a P-graph

In Figure 4.2 an example P-graph is shown. R stands for Raw material, Op for Operation, I for Intermediate, P for Product respectively. The corresponding source code of the concrete elements is presented on Listing 4.2. Note, that we can write every $arc = X + Y$ safely, because the multiplicity is `one` for every concrete `Node`.

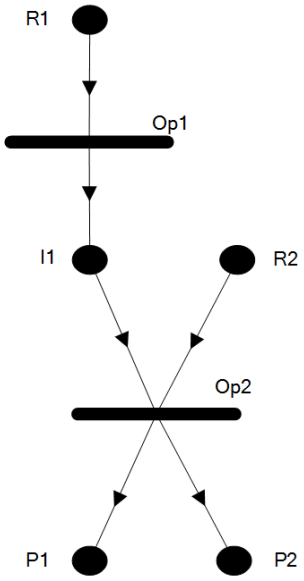


Figure 4.2. An example of a P-graph

Listing 4.2. Concrete P-graph elements

```
/*****************
Concrete elements
*****************/
one sig R1 extends Raw{}{arc = Op1}
one sig R2 extends Raw{}{arc = Op2}

one sig I1 extends Inter{}{arc = Op2}

one sig P1 extends Product{}{arc = none}
one sig P2 extends Product{}{arc = none}

one sig Op1 extends Operation{}{arc = I1}
one sig Op2 extends Operation{}{arc = P1 } P2}
```

4.3.3 Maximal Structure and Solution Structure

The MS and SS are structures that both contain a set of Materials and a set of Operations. Hence, in Alloy they both are represented as signatures with relations that specify which `Materials` and `Operations` they contain. In addition, the concrete instances of SS and MS are calculated by Alloy, and we do not specify concrete signature descendants, i.e.

subsets, as previously with the `Nodes`. Instead, we define constraints on the structure of SS and MS, and Alloy will construct correct instances from our defined model. Therefore, their signatures are not abstract.

Constraints on MS MS constraints are defined as to reflect the MSG steps presented in Chapter 5.

Constraints on SS SS constraints in Alloy are based on the Axioms of PNS discussed in Chapter 3. Additionally, these constraints are formulated as facts of a predicate outside the definition of the SS signature. This predicate only requires the set of `Materials` and `Operations` as parameters. Therefore, we are able to check without an actual SS instance whether an arbitrary set of `Materials` and `Operations` define a valid SS or not.

Listing 4.3. The PNS Axioms in Alloy

```
pred SolStructAxioms [mats : set Material, ops : set Operation] {

	//A1 - Every product is present in the SS
	Product in mats

	//A2 - If a Material has no producer from the set of ops <=> that Material is Raw
	all m : mats | (arc.m & ops = none) <=> (m in Raw)

	//A4 - From every Operation, there is a route to a Product
	//NOTE: The route must only be composed of the arcs between the ops and mats!!
	let n = (mats + ops) | ops in Product.*(~((n <: arc) :> n))

	//A5 - Every Material is connected to an ops Operation
	//A3 - Everfy operation has all of its inputs and outputs
	//NOTE: Operations can only be used as they were defined
	//i.e. every one of its input and output Materials must be included!
	mats = arc.ops + ops.arc
}
```

4.4 Defining Resource States

After creating the basic P-graph model, we continue with the notions of error and availability. Contrary to the classic PNS problem, where unusable elements are deduced from incomplete or syntactically problematic elements of the static problem definition, our model's structure can change dynamically and therefore elements may have an error, or be unavailable at one time, and be usable in other times. As already established, Resources are represented as Materials, hence we define availability and error on Materials only.

Error An error can occur when an inner or outer event happens that makes a Resource faulty. Therefore, a Material has error, if it is no longer usable in its current form.

Availability If no other external process instance is using a Resource, then it is available to us as a Material. Logically, for our purposes the MS is calculated only from the set of available Materials. However, the availability of a Material is not a sufficient condition for usability, as the Material may have error or may require other elements that are connected

to them and are unavailable or unusable at the moment of calculation. As the definition states, the Materials that are used by our process instance is available to us. In addition, the Materials that are not part of our SS, i.e. process instance, but are available to us form the set of Resources called the Reserve.

Environment In order to represent the various states of Materials at different times, i.e. different Resource configurations, we define an `Environment` signature, that describes the concrete state of the whole system at a given moment. Specifically, it describes which Material is available, and which has error. Furthermore, an MS and SS can only be interpreted over a specific `Environment`. Therefore, for every `Environment` we can generate one MS and multiple SS-s (depending on the specific `Environment`) if the constraints are satisfied. The MS of an `Environment` contains all elements that can be used for our purposes, i.e. it is the union of every combinatorially feasible structures. Thus, every SS of that `Environment` works on the elements of the MS of that `Environment`.

Simulation By not specifying the concrete errors and available elements, only the existence of the `Environment` we can simulate different error and available elements configurations. Due to less `Environment` model constraints, Alloy can construct an arbitrary instance model, or multiple models each time we make a request. Therefore, with the `Environment` signature we are able to model and simulate structural dynamic system changes and configurations in Alloy.

The part of our Alloy metamodel presented in this chapter is shown in Figure 4.3.

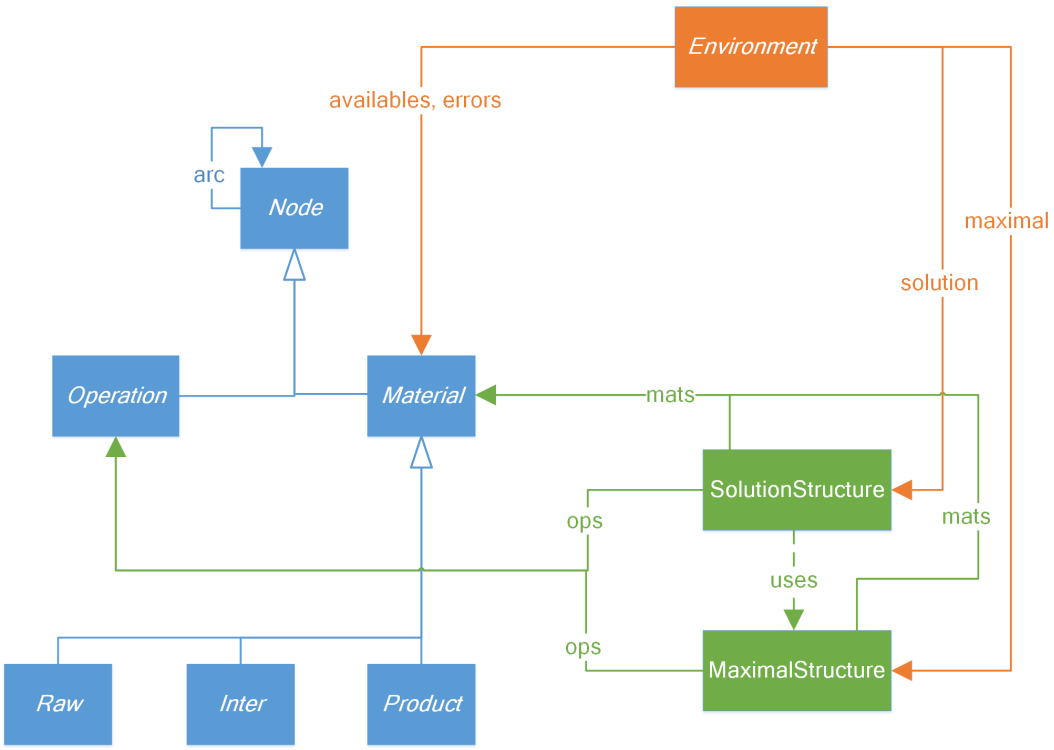


Figure 4.3. Part of the metamodel in Alloy

Chapter 5

Optimization and Reconfiguration method in Alloy

5.1 Overview

As discussed in Chapter 2, our main goal is to support the optimization and the reconfiguration of our process instance by structurally reducing the search-space, i.e. calculating the new MS. We also return a SS that can function as a potential optimization or reconfiguration alternative, which is then used by the numerical optimization phase as an initial bound. In situations where there are only limited resources at our disposal, this initial bound is close to the theoretical optimum. We achieve this in the reconfiguration case by reusing elements from the initial SS, i.e. the process instance that has error and must be reconfigured, that are still usable, i.e. are not in the error confinement region of the initial process instance. Therefore, it can greatly reduce the search-space, and ensure speed-up in the execution of the numerical optimization phase. Figure 2.1 shows our method workflow.

We define the signatures `ReconfEnv` and `OptEnv` as extensions to the `Environment` signature, where `OptEnv` is used by the Optimization, the `ReconfEnv` used by the Reconfiguration method. In the latter, we define a `initialSS` relation that contains the initial SS.

5.2 Maximal Structure Generation

As Alloy has a declarative modelling language, the MSG algorithm is implemented by defining appropriate structural elements and constraints. Based on these definitions Alloy finds and constructs a model instance, that reflects the steps and the outcomes of the generation. Similarly as in the classic MSG algorithm, we generate MS with two main steps. However, as the consequence of the sanity constraint on `Raw` materials, there is no need to filter out the `Operations` that produce `Raw` material.
Main steps are the following:

1. Iteratively identify the elements, both `Materials` and `Operations`, that are unusable with the propagation rules. We represent these propagations iteration-by-iteration as

a set of Nodes, contained in an IterationSet. These IterationSets are chained into a sequence, where each IterationSet is implied by the previous sets. This whole sequence is contained by the UnusableElements signature.

2. Examine which remaining elements can be reached from a Product. More precisely, an Operation has a valid path, if it is not unusable and can reach a Product through not unusable elements, on the other hand a not unusable Material has a valid path to a Product if it is a Product or connected to an Operation that has a valid path to a Product. These Nodes make up the MS. Note, that if not all Products are amongst the set of these elements, no MS exists.

5.2.1 Propagation rules

The propagation rules of the original MSG is not sufficient for our model with errors and availability, thus we define the following extended propagation rules for finding the unusable elements.
A Material becomes unusable if any of the following holds:

- it is not available to us
- it has an error
- it is not Raw, and all of its producer Operations are unusable

An Operation becomes unusable if any of the following holds:

- at least one of its input Material is unusable
- at least one of its output Material is unusable

5.2.2 Auxiliary structures

The first part of the MS generation requires the definition of two auxiliary structures, which supports the selection of unusable elements.

IterationSet An IterationSet signature has a relation to either a set of Materials or a set of Operations. This set of Nodes represents the outcome of a single iteration of unusable elements selection based on the propagation rules. IterationSets are sorted into a sequence, where every IterationSet is implied by the previous sets in the sequence. Furthermore, no Nodes are repeated, every iteration contains exactly the new Nodes that became unusable as a consequence of previous sets. Trivially, every consecutive IterationSet is of a different type, i.e. a set of Materials imply a set of Operations, and vice versa. Note, that the maximal number of IterationSets is the number of Nodes, i.e. it has a finite scope and can be calculated in Alloy.

UnusableElements The `UnusableElements` signature contains (has relations to) `IterationSets`, and has a ternary relation called `implyEdge`, that links together these `IterationSets` into a sequence discussed previously. Every `Environment` contains one `UnusableElements`, which supports the generation of the `Environment`'s MS. Furthermore, in order to ensure correctness, the structure of the sequence is set rigorously, i.e. we define constraints on adjacent `IterationSets`, on the first and last `IterationSet` and on the sets between them. We determine the first set of `Nodes` by checking which elements are initially unusable in the P-graph, by definition, these elements are `Materials`. Also, every interim `IterationSet` must have exactly one preceding and one following neighbor in the sequence and no more.

5.2.3 An iteration example

Figure 5.1 shows an example P-graph, where the name of each node through A to F represents exactly one iteration, i.e. `IterationSet` of `UnusableElements`.

- Materials (A1 - A2): these are the `rootElements` of the `UnusableElements`, they are either unavailable or faulty.
- Operations (B1 - B4): B1 becomes unusable because of A1, while B2, B3, B4 operations are unusable because of A2. As we can see if one input or output material of an operation becomes unusable, then that operation becomes unusable as well.
- Materials (C1 - C2): note, that E1 and P do are not unusable (yet), because they still have producers left, and A2 is not included in this iteration, because it is already a part of a previous iteration. However, in the case of C1 and C2, all of their producers are unusable. Materials R and I are unaffected, because only operations that use them became unusable.
- Operation (D1): The unusability of C2 implies the unusability of D1. Notice that B3 is not repeated, because it is already a part of a previous iteration.
- Material (E1): in this iteration, E1 becomes unusable, because both D1 and B4 are unusable now.
- operation (F1): F1 is unusable, because E1 is unusable. This is the last iteration, because P still has producers left, and no new Node becomes unusable, therefore the propagation stops.

Moreover, the second part of the MS generation excludes Op1 and I, because they have no path to a product, thus the MS consists of R, Op2 and P.

5.2.4 Second part of MS generation

The `IterationSets` of the `UnusableElements` contain all elements that are unusable. The `Materials` and `Operations` of `MaximalStructure` are every element that is not unusable, and has a path to a `Product`. Therefore, on the set of `Product` we use transitive closure

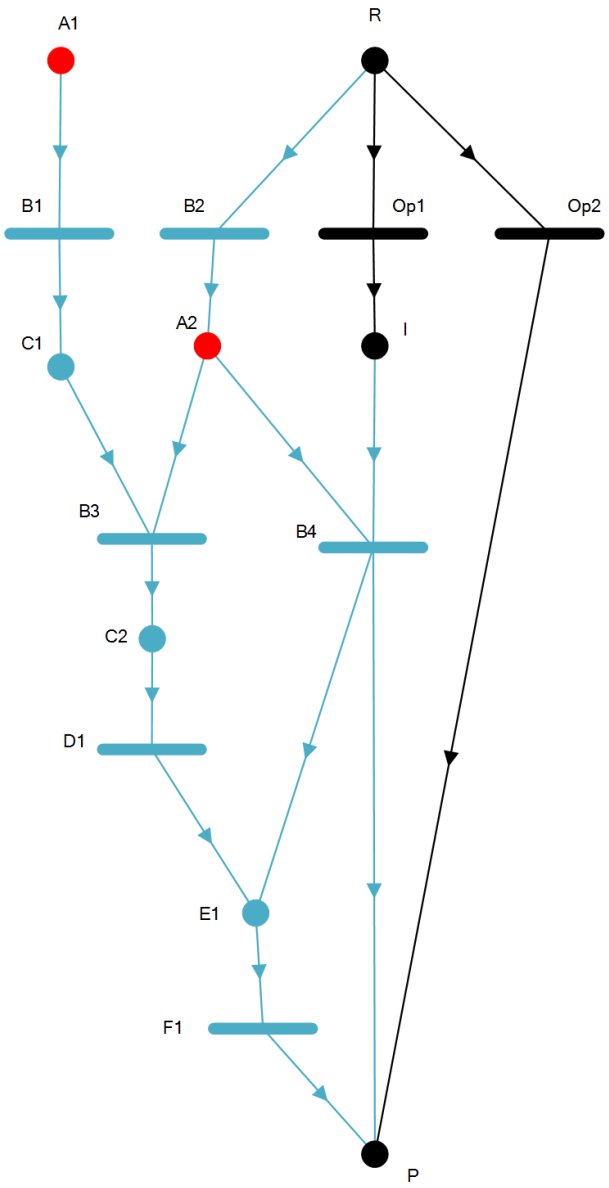


Figure 5.1. Iteration of unusable elements

with the `arcs` that are defined between two still usable nodes, i.e. two nodes from the complement of `UnusableElements`. By using transitive closure on the usable `arcs`, we get all elements that are usable, and have a path to a `Product`, i.e. the elements of the MS. Note, that we have to transpose the `arcs` in order to be able to define a valid transitive closure on `Product`.

5.2.5 Metamodel in Alloy

We can see the complete metamodel in Figure 5.2.

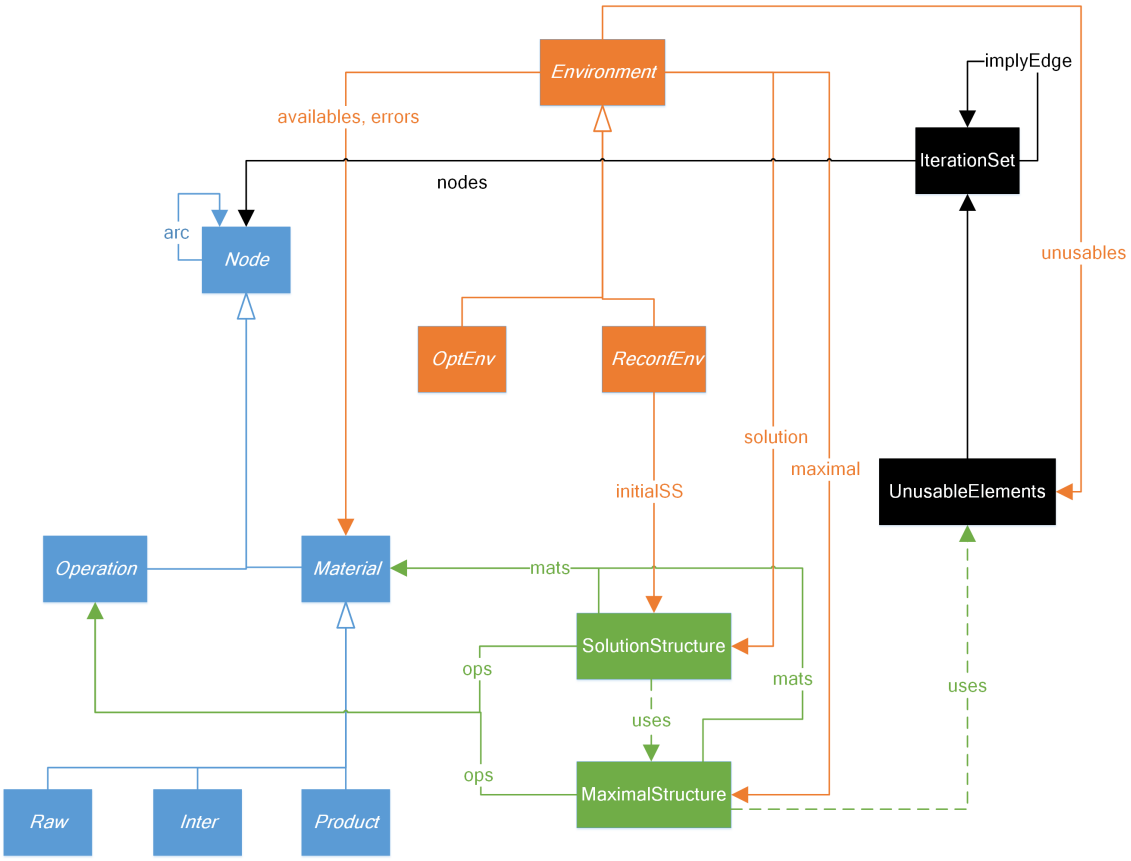


Figure 5.2. The metamodel in Alloy

5.3 Optimization method in Alloy

For a given `Environment` where our process instance is not specified initially, we support the initial optimization, i.e. process instance calculation, by generating the MS from the currently available elements. Therefore we combinatorially reduce the search-space for the following numerical optimization phase, which calculates the final optimal process structure.

In Alloy, we define an `OptEnv` with the specified or simulated errors and available elements. The `UnusableElements`, the `MaximalStructure`, and the `SolutionStructure` of `OptEnv` is calculated by Alloy. Furthermore, the `UnusableElements` is calculated based on the errors and available elements of the `OptEnv`, the `MaximalStructure` is calculated based on the nodes of the `UnusableElements`, and the `SolutionStructure` is calculated based on the `MaximalStructure`.

5.3.1 Resource states during the execution of our method

In Figure 5.3 we present through multiple diagrams the changes in Resource states during the execution of our method. Note, that while the sets discussed in the following are about Resources only, our Alloy model handles both `Materials` and `Operations`.

Initial Resource configuration In Figure 5.3a we can see the initial Resource configuration. The set *Others* represent the Resources that are used by other process instances, hence are not available to us. We do not handle the errors nor the Resource changes in *Others*, it is separate from our process instance. The complement of *Others* is the set *Available*, which contains all Resources that we can work with. Finally, the set *Errors*, which can be empty, represents all initial Resource errors in *Available*.

After first part of MS generation We use `UnusableElements` for calculating every unusable Resources. Figure 5.3b shows the outcome of the first part of the MS generation. Additionally, we can see that the set *Unusable* includes the set *Errors* as well. The initially unusable Resources are the Resources with errors, and the the unavailable Resources, i.e. the sets *Errors* and *Others*. Consequently, all other unusable elements are derived from these initial elements using the propagation rules.

After second part of the MS generation After finding all unusable elements, all remaining *Available* elements must be examined, whether they have a valid path to a Product or not. In Figure 5.3c the set *Available* is reconstructed into three disjunct sets. The set *Unusable* is the same as in the previous Figure, set *MS* is composed of every Resource that is part of the MS, and the *Not connected* contains all Resources that have no valid path to a Product, i.e. they are excluded by the second part of the MS generation. Note, that it is possible, that no MS exists, i.e. *MS* is empty.

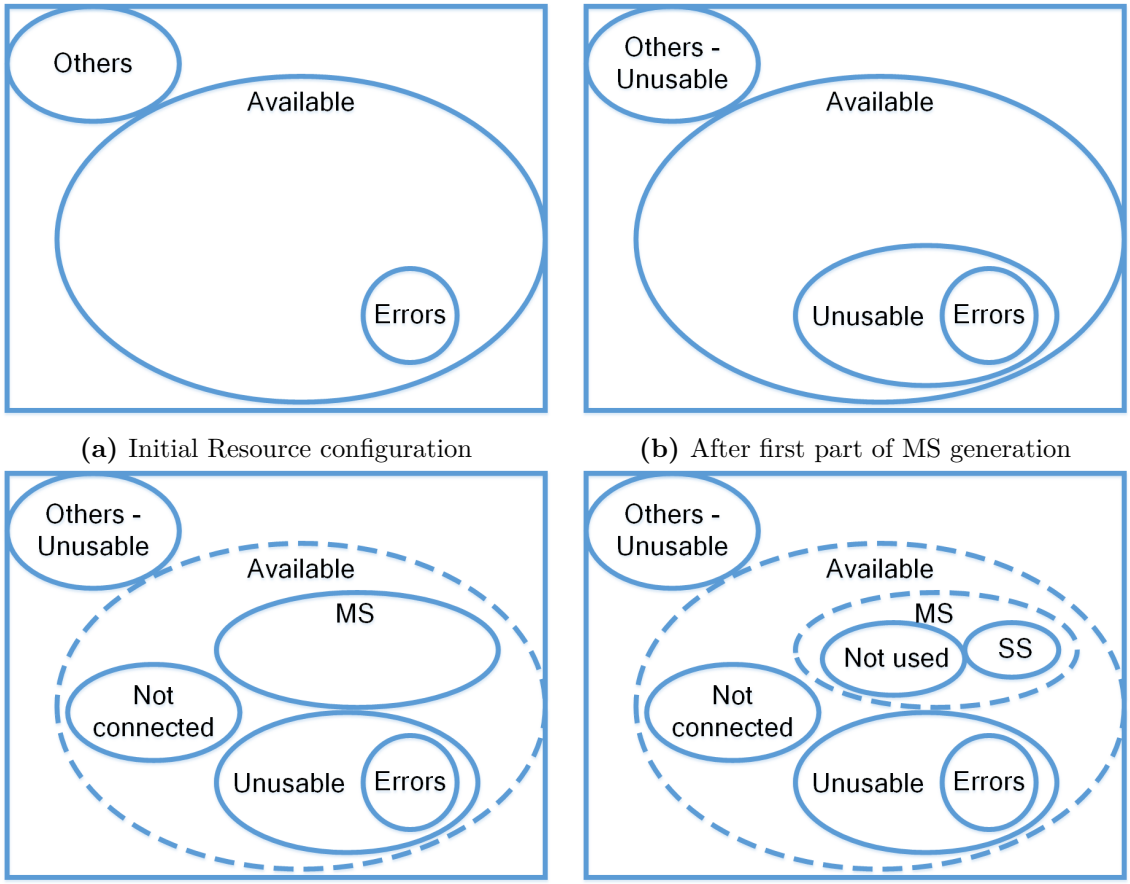
SS bound generation To support the bounding of the numerical optimization phase, we generate an initial SS from the set *MS*. Figure 5.3d shows how the set *MS* is made up of sets *SS* and *Not used*.

After numerical optimization The outcome of the numerical optimization is a final SS, which will be used by our process instance. The final configuration is similar to Figure 5.3d, but the sets *SS* and *Not used* might be different depending on the final SS.

5.4 Reconfiguration method in Alloy

After an initial SS becomes faulty and the Resource configuration changes, we want to support the reconfiguration by calculating the new MS and generate a SS that uses elements from the original SS, i.e. gives a sharper initial bound for the numerical optimization phase. We define a `ReconfEnv` where we set the initial SS as the faulty SS and the `errors` and `availables` are set as the new Resource configuration. Moreover, the initial SS is a SS that can also be defined through the optimization of an earlier `OptEnv`, or specified directly by defining an `OptEnv` with this SS, or we can completely simulate the `OptEnv`. The errors and availability of the `ReconfEnv` can also be specified in advance, or simulated.

As discussed in the case of `OptEnv`, the `UnusableElements`, `MaximalStructure`, `SolutionStructure` of the `ReconfEnv` are calculated by Alloy accordingly.



(a) Initial Resource configuration

(b) After first part of MS generation

(c) After second part of the MS generation

(d) Generate a SS

Figure 5.3. Resource configurations during Optimization

5.4.1 Resource states during the execution of our method

Again, in Figure 5.4 we show only Resources, but in Alloy both `Materials` and `Operations` are handled.

System during operation Figure 5.4a shows a snapshot of the system with a Resource configuration, and our process instance represented as set *Process*. Moreover, the Resources that are in *Available*, but not used by our process instance are in the set *Reserve*, which contains Resources that can be used for our reconfiguration purposes. During the operation of the system, the sets *Others* and *Reserve*, i.e. the structure of our system can change dynamically depending on the other process instances, which we do not handle, or as a consequence of events that cause errors.

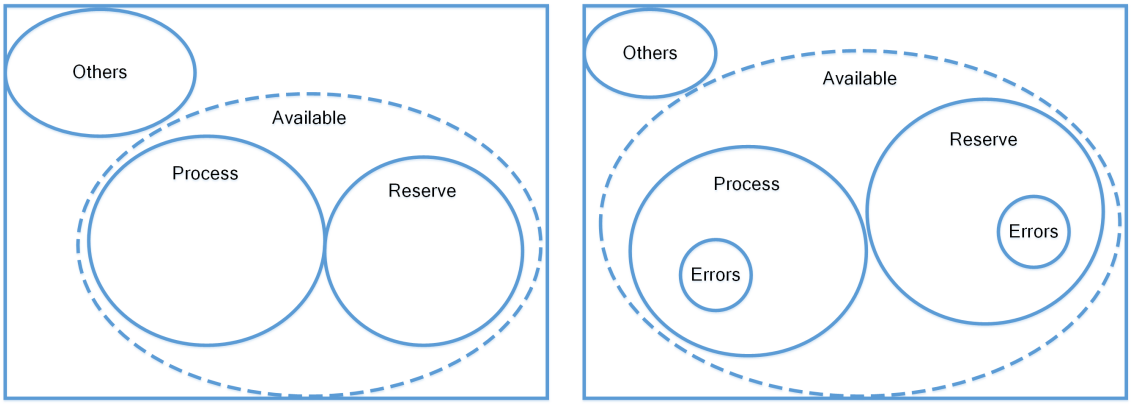
Errors in our process The system changes, if our process instance cannot function with the generated errors, we must reconfigure. As shown in Figure 5.4b, first we determine the available elements we can work with, i.e. the *Reserve* and its *Errors*. After that, we begin the calculation of MS.

After first part of MS generation In Figure 5.4c we can see the outcome of the calculation of the unusable elements. The `UnusableElements` comprises of the set *Others*, set *Unusable* in *Process* and set *Unusable* in *Reserve*. Moreover, the initial unusable elements are the set *Others*, set *Errors* in *Process* and set *Errors* in *Reserve*.

After second part of MS generation In the second part we search for the elements that have a valid path to a Product. Figure 5.4d shows how one part of the set *MS* is from *Process* and the other one is from the set *Reserve*. In addition, every remaining Resources are in sets *Not connected* present in both set *Process* and set *Reserve*. Set *MS* contains all Resources that can be used for constructing a reconfigured valid SS. Note, that it is possible, that *MS* is solely from *Reserve*, i.e. every Resource we use to reconfigure is from the *Reserve*, or that no MS exists, i.e. *MS* is empty.

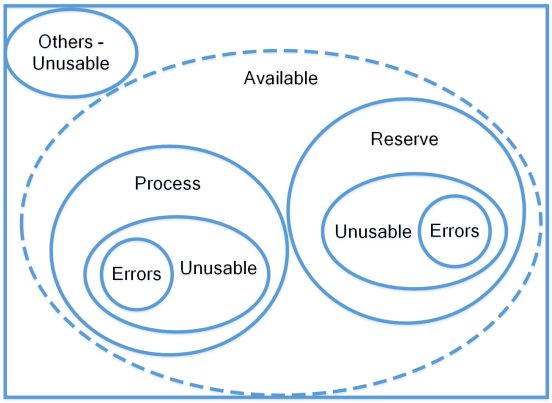
An SS bound generation From set *MS* we generate a SS that is used as a bound in the following numerical optimization phase. Furthermore, by re-using as much already used Resources as possible, i.e. constructing a SS that contains the still usable part (the *MS* part) of set *Process*, we save on the reconfiguration cost, hence the constructed SS will be a good cost-effective initial bound. Additionally, the numerical optimization phase might decide to backtrack even further, and use less Resource from set *Process*. Figure 5.4e shows a set *SS*, that contains Resources from both sets *Process* and *Reserve*. Moreover, the Resources from our process instance that we cannot use for reconfiguration is the error confinement region of that specific initial process instance, which consists of sets *Unusable* and *Not connected* in set *Process*.

After numerical optimization Numerical optimization yields the final SS that will be used as the reconfigured process instance.

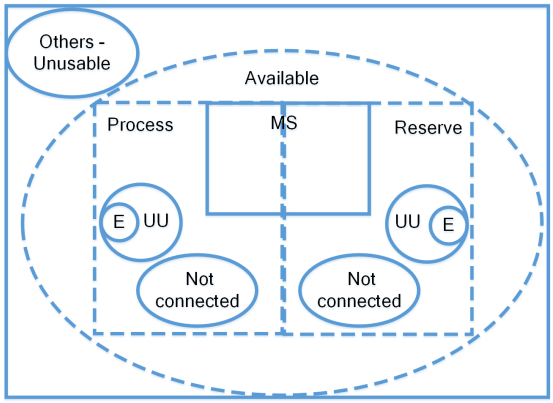


(a) System during operation

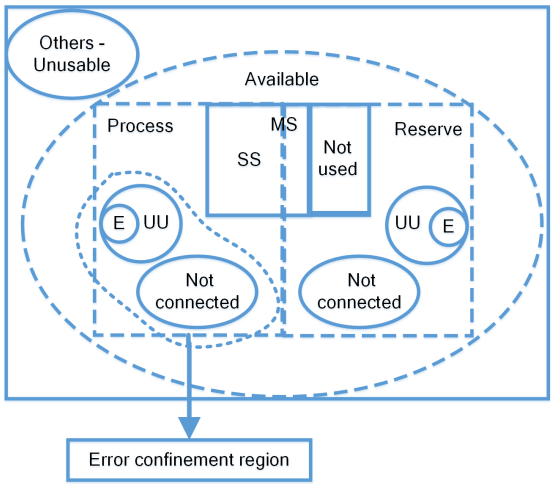
(b) Errors generated



(c) After first part of the MS generation



(d) After second part of the MS generation



(e) An SS bound generation

Figure 5.4. Resource states during the Reconfiguration

Chapter 6

Summary

6.1 Contributions

We created a method for the optimization and reconfiguration of system processes in the domain of IT infrastructures with dynamic structures, where system changes can influence both the execution and structure of the complex system. Our main work focused on the *construction phase* of this problem, i.e. supporting the numerical optimization of the *improvement phase*. This construction phase was carried out via an abstraction method we created by extending the definition of process graphs, hence our method exploits the benefits of structural relaxation possibilities of a PNS problem. This extension is primarily dedicated to the fault tolerance concepts of IT infrastructures, by the definition of error and availability. By using a modified maximal solution generation algorithm, we structurally reduced the search-space of the improvement phase, reducing its computational needs. We defined a translation from the P-graph definition to the language of Alloy, hence we were able to exploit the benefits of Alloy tool to construct the structural models and model the effects of a change defined in our method. Our method is also able to simulate the effects of failed components of a system.

6.2 Future work

We will investigate options for improving our implementation in Alloy, as it has limitations when working with large problems. Other potential uses for our work includes validating the structures of process instances, determining single point of failures, simulating different error and recovery scenarios, calculating the largest set of errors that a given process instance can recover from. Furthermore, we will extend our method to be able to handle more complex systems with multiple process instances, where the system can reallocate resources from process instances with lower priority to process instances with higher priority.

We will also explore options for the *improvement* phase of the optimization, as our work focused mainly on the *construction phase*. Possible approaches include the extension of *Accelerated Branch and Bound* used in the PNS problem domain [7] [9] [10], *Branch and*

Bound for Boolean Optimization [17], *Weighted Boolean Optimization* [18], *Incremental SAT solving* [23].

6.3 Acknowledgements

We are grateful for the contribution and valuable guidance of our supervisors, Dr. András Pataricza and László Gönczy. We would also like to express our gratitude to Szilvia Varró-Gyapay for her valuable advice and assistance.

Bibliography

[1] Shahriar Asta, Daniel Karapetyan, Ahmed Kheiri, Ender Özcan, and Andrew J Parkes. Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. In *6th Multidisciplinary International Scheduling Conference: Theory & Applications (MISTA2013)*, 2013.

[2] Algirdas Avizienis, Jean claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[3] Armin Biere. *Understanding Modern SAT Solvers*, 2012. `http://fmv.jku.at/biere/talks/Biere-VTSA12-talk.pdf`.

[4] X. Blasco, J.M. Herrero, J. Sanchis, and M. Martínez. A new graphical visualization of n-dimensional Pareto front for decision-making in multiobjective optimization. *Information Sciences*, 178(20):3908–3924, October 2008.

[5] György Csertán, András Pataricza, Harang Péter, Orsolya Dobán, Biros Gábor, András Dancsecz, and Ferenc Friedler. BPM Based Robust E-business Application Development. 2002.

[6] Jacob Feldman. JSR-331 - Java Constraint Programming API. pages 1–58, 2012.

[7] F Friedler, K Tarjan, YW Huang, and LT Fan. Combinatorial algorithms for process synthesis. *Computers & chemical ...*, 1992.

[8] F Friedler, K Tarjan, YW Huang, and LT Fan. Graph-theoretic approach to process synthesis: axioms and theorems. *Chemical Engineering Science*, 1992.

[9] F Friedler, K Tarjan, YW Huang, and LT Fan. Graph-theoretic approach to process synthesis: polynomial algorithm for maximal structure generation. *Computers & Chemical ...*, 1993.

[10] F Friedler, JB Varga, E Feher, and LT Fan. Combinatorially accelerated branch-and-bound method for solving the MIP model of process network synthesis. *State of the Art in Global ...*, 1996.

[11] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability Solvers. 2008.

[12] Group Object Management. *Business Process Model and Notation (BPMN) version 2.0.* `http://www.omg.org/spec/BPMN/2.0/`.

[13] S. Gyapay and A. Pataricza. A combination of Petri nets and process network synthesis. *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483)*, 2:1167–1174, 2003.

[14] Daniel Jackson. *Alloy: a language & tool for relational models.* http://alloy.mit.edu/alloy/.

[15] Daniel Jackson. *Software Abstractions: logic, language, and analysis.* The MIT Press, 2012.

[16] Harold Henry Kollmeier. Reconfiguration for Fault Tolerance and Performance Analysis. (November), 1987.

[17] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodr. Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates. *SAT 2009 - Theory and Applications of Satisfiability Testing*, 2009.

[18] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for Weighted Boolean Optimization. pages 1–14, 2009.

[19] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc : Towards A Standard CP Modelling Language. *CP 2007 - Principles and Practice of Constraint Programming*, 2007.

[20] András Pataricza. *Model Based Dependability Analysis.* Dsc thesis, Hungarian Academy of Sciences, 2006.

[21] Domenico Salvagnin. Constraint Programming Techniques for Mixed Integer Linear Programs. 2009.

[22] Varró-Gyapay Szilvia. *Trajectory set approximation for optimization and verification of IT systems.* Phd thesis, Budapest University of Technology and Economics, 2014. Chapter 3: Process Network Synthesis, pp. 23-32.

[23] Soh Takehide. *Studies on Applying Incremental SAT Solving to Optimization and Enumeration Problems.* Phd thesis, Department of Informatics, School of Multidisciplinary Sciences, 2011.

[24] Tünde Tarczali, Zoltán Süle, and Károly Kalauz. Structural alternatives of business process models applying P-graph methodology. pages 1–11.

[25] The Open Group. *Archimate.* `http://www.opengroup.org/subjectareas/enterprise/archimate/`.

[26] Vincent T'kindt, H Scott, and Jean-Charles Billaut. *Multicriteria scheduling: theory, models and algorithms*. Springer, 2006.

[27] Emina Torlak, Felix Sheng-ho Chang, and Daniel Jackson. Finding Minimal Unsatisfiable Cores of Declarative Specifications. pages 1–16.

[28] Emina Torlak and Greg Dennis. *Kodkod for Alloy Users*, 2006. `http://homes.cs.washington.edu/~emina/pubs/kodkod.alloy06.pdf`.

[29] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pages 632–647. Wiley, 2007.

[30] Gábor Urbanics, László Gönczy, Balázs Urbán, János Hartwig, and Imre Kocsis. Combined Error Propagation Analysis and Runtime Event Detection in Process-driven Systems. 2014.