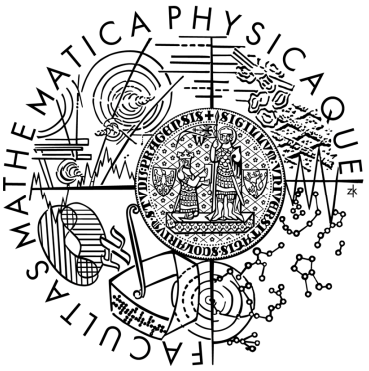
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Vladimír Rovenský

Workflow Modelling

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Roman Barták Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2011

I would like to thank the following people for their support: Mr. Roman Barták for supervising this thesis, Mr. Con Sheahan and Mr. Dang Thanh-Tung for their help with the FlowOpt project and Mr. Filip Dvořák for his insights on the workflow verification algorithms.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date.....

Název práce: Modelování workflows

Autor: Vladimír Rovenský

Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: doc. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem diplomové práce bylo navrhnout a implementovat grafický editor umožňující modelování pracovních postupů (workflows) s důrazem na efektivitu práce, jednoduchost a použitelnost pro běžného uživatele. Výsledná aplikace je integrována do systému FlowOpt, ve kterém lze vytvořené pracovní postupy použít k řízení výrobních procesů v malých a středně velkých továrnách. Vlastní editor by měl sloužit mj. jako demonstrace použití modelu Nested TNA v reálném protředí. Součástí práce je funkční implementace vlastního editoru, navržení a implementace procedury pro automatickou verifikaci pracovních postupů a podpora standardního formátu XPDL (BPMN) pro ukládání pracovních postupů (import i export).

Klíčová slova: Pracovní postupy, FlowOpt, XPDL

Title: Workflow modelling

Author: Vladimír Rovenský

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis was to design and implement a graphical editor for workflow modelling, focusing on productivity, simplicity and usability for the common user. The resulting application is integrated into the FlowOpt project, in which the workflows can be used to manage manufacturing processes in small and medium size factories. The workflow editor should serve among other things as a proof of concept of practical usability of the Nested TNA workflow model. The main parts of the thesis include a working implementation of the editor, a procedure for automatic verification of the workflows and support of the XPDL (BPMN) standard for saving workflows.

Keywords: Workflows, FlowOpt, XPDL

Table of Contents

Introduction

Workflows are a very important tool in business modeling, since they provide intuitive visual representation of business processes. Such a representation can be used for many purposes. Simple workflows may serve just for illustration of a particular process, while more complex ones can actually be used to simulate and even schedule the execution of a process.

Since utilizing such a tool in a business can increase its overall performance considerably, various workflow solutions have been developed, both standardized and proprietary, good examples being BPMN[1], YAWL[2] or MAKE[3].

The purpose of this thesis was to implement a similar system for workflow modeling, but with particular software requirements in mind. The presented system was to be simple, effective and expressive enough to represent manufacturing processes (workflows used in factories to manufacture various products). In particular, it should utilize the Nested TNA[4] workflow model presented in [1] and show its usability in practical scenarios.

In the first part of the thesis, we describe general workflows and their various uses in more detail to provide the necessary context for this thesis.

We briefly describe the background of this thesis and its connection to the FlowOpt software project and the MAKE system, into which the presented application is fully integrated, since these two systems determined most of the software requirements on the presented application.

Then we mention some of the widely used workflow modeling tools that served as an inspiration for this thesis while pointing out which of their features were of particular importance for the presented application.

The rest of the thesis will describe all of the presented application's most important features, particularly the implemented workflow model, workflow visualization, workflow verification and support for the standardized XPDL[5] format.

Full documentation of the presented application is available as one of the attachments of the thesis, together with the application itself.

---

[1] Business Process Modeling Notation

[2] Yet Another Workflow Language

[3] ManOpt MAKE – a commercial system for modeling of manufacturing processes

[4] Temporal Networks with Alternatives

[5] XML Process Definition Language

1. Workflows in General

A workflow is essentially a special kind of graph describing some work process. The nodes in such graphs usually represent units of work (commonly called activities) and links represent the order of execution of these activities.

Depending on the complexity of the particular workflow model, other types of objects may be defined, like events that happen while performing the process or resources used by the activities.

As a visual representation, workflows are much more illustrative and concise than full textual description of the process. Depending on how much information the workflow contains, they can be used for anything from simple visual demonstration to guiding and optimizing the actual execution of the process.

To illustrate the concept, we provide an example of a simple workflow describing the general process of handling an order in a shop.

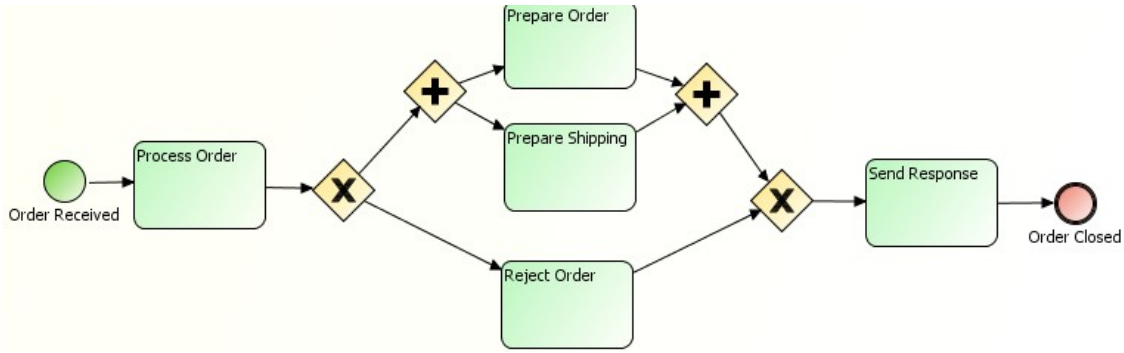


Figure 1: Example of a workflow

Figure 1 shows a simple workflow that was created using the BPMN notation, which is widely used and understood by many users and organizations.

The workflow starts when the shop receives an order. This event is explicitly represented by the green circle. After that, the order is processed to determine whether the requested goods are available.

The order processing is an example of an activity – some unit of work that needs to be done. Activities are the basic building blocks of all workflows and in this particular notation, they are represented as green rounded rectangles.

Note that the order of execution is given by the black arrows – these are called flow links in BPMN. Since the order of execution isn't always linear, it is necessary to have some constructs that can represent branching. In BPMN, these are called gateways and they are represented as diamonds.

We can see that when the order is processed, the execution comes to one such gateway – the ones marked with an 'X' symbol are exclusive gateways, meaning that the flow can go in exactly one of several possible ways.

In this case the worker may determine that it is possible to fulfill the order or that the order has to be rejected, for instance because the requested goods aren't available at the moment.

Notice that after the flow diverges in the exclusive gateway, it later converges again in a gateway of the same type. Such patterns will be referred to as *nests* in this document. Nests are a very important concept for the presented application, as we will show in later chapters.

If the order can be fulfilled, the execution comes to another type of gateway – a parallel gateway. This means that all the subsequent activities are to be performed in parallel, since they are independent on each other.

Here we would prepare the order (a worker has to bring all the requested items from the warehouse) and prepare shipping for the package. Since these activities are independent, it is more effective to have them done in parallel.

After that, the execution flow comes to another parallel gate, which represents the synchronization point for the two activities we performed in parallel. Both of the activities must be finished before the process continues past the synchronization point. The two parallel gateways and the activities between them are another example of a *nest*.

When the order is handled (either fulfilled or rejected), we send a response to the customer and mark the order as closed. The red event indicates the end of the process.

Although this example is greatly simplified, it should be enough to illustrate the general purpose and strengths of workflows (note that the textual description of the process far exceeds the workflow in size, even for this simple example).

Depending on our requirements, we could utilize this workflow merely to explain the process to new employees, or we could have complex software that automatically receives orders, schedules the activities in time in the order described by the workflow and assigns them to specific employees. The larger the shop gets, the more important it is to do these things effectively.

Even though we used a specific workflow notation (BPMN), the objects introduced in this workflow (activities, gateways, flow links and events) are defined by most of the existing workflow models. They represent the basic building blocks that are commonly used to create workflows.

2. FlowOpt and MAKE

This chapter briefly describes the background of this thesis to give the reader a general idea of the context in which the presented application was developed.

The presented workflow editor is a part of the FlowOpt software project. The general purpose of FlowOpt was to provide a feature-rich, easy to use framework for designing and scheduling of *manufacturing processes*.

Manufacturing processes are a subset of general business processes. They describe the manufacturing of various products in factories. The workflow models representing them are usually not as expressive as those representing general business processes, since manufacturing processes are not as complex.

Instead, the focus is on the ability to efficiently schedule the work described by the workflow in time - there are specialized tools that can automatically generate a complete plan of works based on a workflow. The workflow editor must support this process with a suitable workflow model.

The FlowOpt project includes five cooperating modules – a workflow editor (the presented application), a work order manager for entering orders placed by the customers, an optimizer module that schedules the workflows, a schedule visualizer that displays the schedules as Gantt diagrams and an analyzer module that can be used to optimize the process.

The intended way that FlowOpt should be used is as follows:

1) The user creates a workflow in the workflow editor. This workflow describes how a particular product is manufactured. It contains all the information necessary to create a schedule, like resource requirements and durations of various activities.
2) A work order is created in the work order manager module. A work order is simply a list of requested products with quantities and some kind of deadline.
3) The optimizer creates a schedule for this work order. A schedule assigns an exact time and performer for each activity that has to be done in order to fulfill the work order. In other words, it automatically assigns work to employees. Furthermore, the optimizer tries to assign the work so that the overall cost of fulfilling the order is as low as possible, so its performance has great impact on the productivity of the factory.
4) The schedules created by the optimizer can be visualized and modified in the schedule viewer.
5) The schedules can also be analyzed by the analyzer module. This analysis yields suggestions as to what actions should the factory perform to make the process more effective, like buying a new resource for example.

There is a close connection between the FlowOpt project and the ManOpt MAKE system – since the general goals of the two systems are quite similar, we cooperated with the ManOpt Company during the development of FlowOpt.

ManOpt developers provided us with valuable technical insights, shared their experiences in workflow modeling and provided us with some powerful commercial

tools that we could use to develop our application. In exchange, we integrated our modules (including the presented application) into the MAKE system. This meant unifying the workflow model (and the data model in general) across all the FlowOpt modules. It also brought additional requirements on functionality and usability, since the MAKE application is a commercial product.

The FlowOpt software project was successfully defended in June 2011. Once all of its modules are fully integrated into the MAKE system, it will be possible to evaluate its performance on real manufacturing scenarios.

Further description of the FlowOpt project is beyond the scope of this document, however its full documentation (user, development and programmer) and installation are available as an attachment to this thesis.

3. Existing Solutions

Since workflows are such a powerful concept, naturally there are many systems that support and utilize them. An important part of this thesis was analyzing some of the more successful workflow systems and combining their principles with some of our own ideas to create an application that best suits our particular requirements.

We discuss the properties of four existing workflow models (and systems that implement them) – MAKE, Nested TNA, BPMN and YAWL. These were the most significant inspirations for the presented application.

3.1 MAKE

MAKE is a complex commercial system developed by an Irish company called ManOpt[1]. It includes a workflow editor and a proprietary workflow model that focuses on manufacturing processes.

Besides a workflow editor, MAKE provides other modules that let the user manage work orders or even assign work to employees based on the defined workflows. It is an example of a proven, powerful application that uses workflows to increase business performance considerably.

This system was of particular importance, because we cooperated with ManOpt from the very start of the development process and the presented application was intended to provide an alternative to MAKE's built-in workflow editor to determine the viability of some proposed ideas (mainly the Nested TNA model).

The workflow model of MAKE is quite simple, as mentioned above it doesn't model general business processes, but it focuses on manufacturing only. This means that it isn't as expressive as other, more general models, but it is sufficient for its specific purpose.

Furthermore, the simplicity of the model makes it possible to schedule and assign work based on the defined workflows as mentioned above – this would be extremely complicated for more general models like BPMN.

To illustrate what MAKE workflows look like, we present the same workflow as in the introductory chapter, modeled in MAKE.

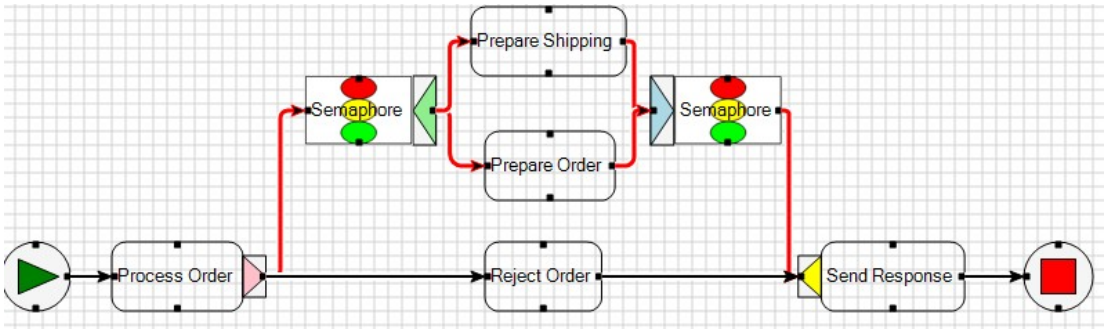


Figure 2: MAKE workflow example

[1] Recently renamed to Entellexi (http://www.entellexi.com)

We can see that the workflow in Figure 2 looks very similar to the introductory example. That is because all the general concepts are the same – there are activities representing the actual work, links representing the order of execution and gateways (here they are called semaphores) that affect the execution flow.

An important property of the activities is the list of their resource requirements (not visible in the figure) - it is possible to specify which resources the activity needs in order to be performed (employees, machines). These requirements are used later to create a detailed schedule of works.

The only objects that weren't in the previous example are the decorators – MAKE workflow model allows for the gateways (semaphores) to actually be placed directly on activities – they "decorate" them, hence the term decorators. This makes sense, since gateways generally do not represent any actual work, so it is convenient to just place them on other objects.

In MAKE, decorators look like rectangles attached to an activity's border with a triangle sign representing the decorator's type (green / blue represent parallel nests, pink / yellow represent exclusive nests).

These are actually all the objects defined by the MAKE workflow model, which makes it very simple and easy to work with, yet expressive enough to describe even complex manufacturing processes.

3.2 Nested TNA

Nested TNA (Temporal Networks with Alternatives) is an academic workflow model presented in [1]. It represents mostly manufacturing processes in a way resembling the MAKE model. Its key distinguishing feature is a tree-like hierarchy of the workflows - unlike the other presented models, Nested TNA actually requires that the workflow is organized hierarchically.

Simply put, a Nested TNA workflow is a tree structure whose inner nodes represent work that has to be broken down into smaller units, which are represented by the children of the node. In the leaves of this tree are the activities – elementary units of work that we do not decompose further. It is a concept similar to the well-known WBS[1].

The fact that the workflow has to be organized like this limits the expressivity of Nested TNA to a certain degree, but it has considerable advantages, like more effective scheduling or easier workflow verification.

Nested TNA is especially important for this thesis, since it is (with some extensions) implemented by the presented application to demonstrate its usability and various consequences of the nested structure.

A notable feature of the Nested TNA model is the simple and intuitive way in which workflows are defined. Other models usually define all the objects that can exist in a workflow separately, together with the semantics on their use. Nested TNA uses a different approach.

[1] Work Breakdown Structure

It first describes a trivial workflow that is a valid Nested TNA by definition. In the original article, this workflow consists of two nodes connected by a link. In the presented application, it is a single node. This minor difference will be explained further in this document.

The other part of the definition of Nested TNA is a precise set of actions that can be performed on a valid Nested TNA to extend it that preserve validity. These actions are called decompositions. Decomposition can convert a single node into a nest of new nodes, building the workflow and the tree hierarchy (top to bottom).

This incremental definition is convenient to implement, since it explicitly defines how the user should build the workflow.

An example of the ordering process in the Nested TNA model is given below.

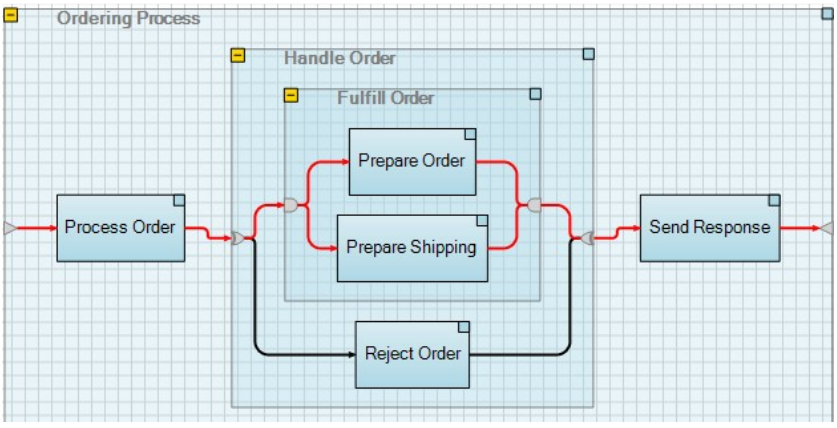


Figure 3: Nested TNA workflow example

Figure 3 shows a Nested TNA workflow that was created in the presented application. It looks very similar to the MAKE workflow, because it defines the same basic objects: activities represent work (blue rectangles) and links represent the order of execution.

The only major difference between the above workflow and the workflow in Figure 2 is the workflow hierarchy – nests that were created via decomposition are represented explicitly in this model (visually emphasized by gray boxes).

The flow control is not represented by semaphores or decorators. It is given by the type of decomposition – serial, parallel and alternative decomposition represents sequences, parallel and exclusive branching respectively.

Like in the MAKE workflow model, the focus of the Nested TNA model is on simplicity. While Nested TNA is not powerful enough to represent general business processes, it suffices for manufacturing processes, which is what the model was meant for.

Nested TNA together with our extensions and modifications will be described in more detail in the chapter dedicated to the FlowOpt Workflows.

3.3 BPMN / XPDL

BPMN (Business Process Modeling Notation) is an example of a standardized workflow model. Currently it is the most widely used notation representing business

processes. It is maintained by the OMG[1] group, which also develops the widely used UML[2] notation. It is interesting to note that UML also defines a special kind of diagram to represent workflows, which is in many ways similar to BPMN.

BPMN is a very general and powerful model, which can be used to represent a much wider range of processes than the MAKE model or Nested TNA. It defines all of the basic concepts (activities, gateways, events…) and it adds some more advanced objects like message flow to represent communication, pools and lanes to represent the performers of a particular part of the workflow or subprocesses to bring hierarchy to the workflow (this hierarchy is not required though).

Technically, BPMN is not a single system like MAKE or YAWL. It is a standard on visual representation of workflows, which is implemented by many software solutions (for example Together Workflow Editor[3] or BizAgi Process Modeller[4]).

Since version 2.0, the BPMN standard also includes the definition of several formats for data exchange. However in previous versions the standard only defined what the workflows should look like, but not how they should be stored. That is why the XPDL format was developed. XPDL is a standardized XML format for saving workflows adhering to the BPMN workflow model.

We've already mentioned that the first example used was actually created using BPMN. We present one more example to show some of the more advanced features of this notation.

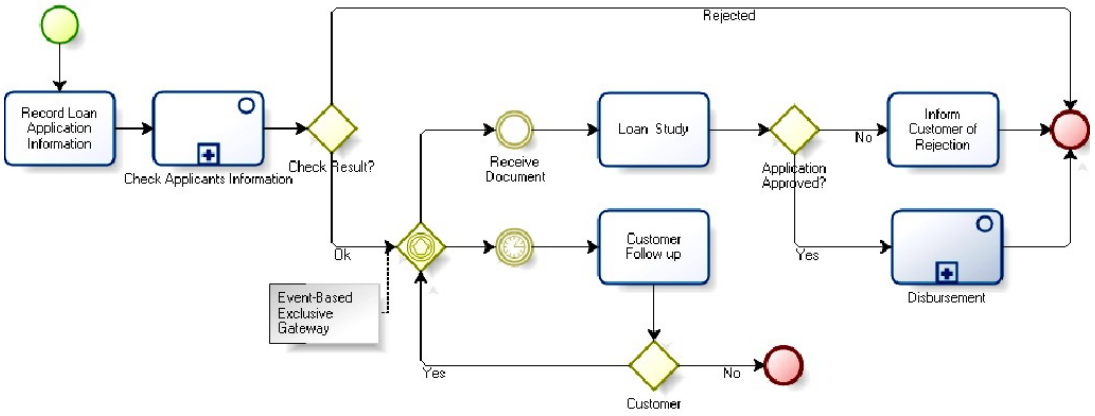


Figure 4: A more complex BPMN example

The example in Figure 4 describes the process of asking a hypothetical bank for a loan. Source: http://www.bizagi.com/eng/downloads/BPMNbyExample.pdf.

First step is to record the information on the new loan application. Then the information provided by the applicant is verified to determine whether a loan can even be considered (the applicant must be a client of the bank for instance).

---

[1] Object Management Group

[2] Unified Modeling Language

[3] http://www.together.at/prod/workflow/twe

[4] http://www.bizagi.com/index.php?option=com_content&view=article&id=95&Itemid=107

This is done in a separate subprocess. A subprocess is a self-contained workflow embedded into the parent workflow that is to be performed in place of the placeholder activity (these look like ordinary activities, but they are marked with a '+' sign in the middle center part).

Subprocesses allow modelers to create hierarchical workflows, which is very useful for more complex processes. BPMN standard accounts for both a collapsed view of subprocesses (like in the example above) and for an expanded view, in which the nested workflow is actually visible within the placeholder activity. This concept is very important for the presented application, since we implemented the hierarchy in our workflows in a similar way.

Getting back to the example, the loan application verification can have one of two results (indicating an exclusive gateway) – if the loan is not possible, the process ends. Otherwise the process continues.

At this point, we need a more detailed document with information about the applicant to determine the exact amount we can loan, the interest etc. The problem is that providing the document is up to the applicant, we have no control over it. This is a typical situation in which events have to be used.

In this case, two possible events can happen – either the user provides the necessary documents, or some specified period of time passes (a timeout occurs). If the documents are provided, we can analyze them and, based on the information, we can either reject the loan or agree on specific terms of the loan (this is again done in a separate subprocess).

If the user does not provide the necessary documents in time, a follow up is made with the client to determine whether the request for loan stands. If not, the process ends. If it does, the flow returns to the state of waiting for the necessary documents. This is an example of a cycle in a workflow – a pattern that allows for (conditional or unconditional) looping of various parts of the workflow.

It should be apparent from this example that BPMN describes more general processes than MAKE or Nested TNA. While we were able to describe the entire workflow model of MAKE on a simple example, the full extent of BPMN is beyond the scope of this thesis. For further reference, please see [2].

3.4 YAWL

YAWL (Yet Another Workflow Language [12]) is a workflow model with academic origins. It was created in part by Wil van der Aalst, one of the leading authorities on workflow design.

The research of Wil van der Aalst is particularly important since it identifies a number of standard workflow patterns that are commonly used to describe workflow processes. For more information on these, please refer to [3].

The support of these patterns can be a viable measure of a workflow model's expressivity (we will show which of these patterns can be represented by our model later in this document). YAWL was designed with the specific goal of supporting these patterns while presenting a relatively simple workflow model.

Besides the workflow model, YAWL also implements a complex system similar to MAKE including a workflow editor and other modules that utilize the created workflows for business management.

An example of the ordering process modeled in YAWL is presented below.

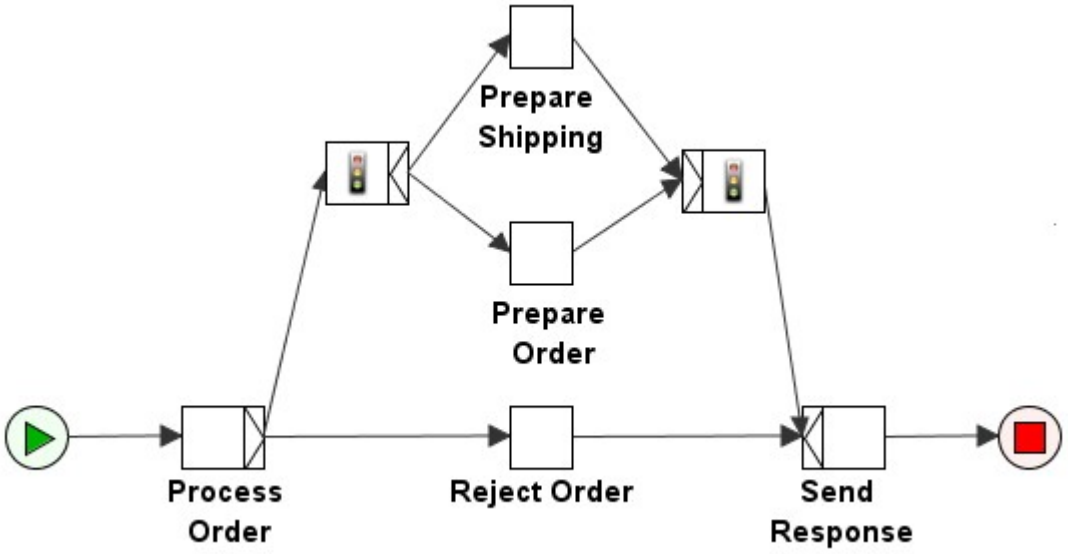


Figure 5: YAWL workflow example

As we can see from Figure 5, the model is visually very similar to the model of MAKE. All of the basic concepts are here (activities, gateways / semaphores, decorators). YAWL is more expressive though and can represent processes beyond the scope of manufacturing. It supports additional objects similar to BPMN's events and subprocesses, making it a very flexible solution.

3.5 Comparison

This chapter showed several different workflow models that served as an inspiration for the presented application. MAKE and Nested TNA are rather simple workflow models that are meant to represent a specific subset of general business processes (manufacturing processes). BPMN and YAWL are examples of more powerful and complex workflow models that can represent wider array of processes.

There is a subtle but important difference between MAKE / Nested TNA and BPMN / YAWL. Both MAKE and Nested TNA utilize constraint programming for flow control. This means that the links in both the workflow models represent a temporal relation – a link from an activity A_1 to an activity A_2 means that A_1 should be executed *before* A_2 (in the general case the link may carry the minimal and maximal temporal distance between the two activities).

BPMN and YAWL do not work like that. They are based on Petri nets [4], which use different execution semantics. In Petri nets, flow links do not represent precedence relation. Instead, they actually define the exact way that the workflow has to be executed.

Intuitively, Petri nets define an execution token that travels within the workflow, moving over the flow links and causing whatever activities it encounters to execute. Although this may not seem particularly important, this semantics can be more powerful.

For example it is impossible to model looping when the execution flow is represented (just) by precedence constraints – a cycle of precedences is always invalid. However if we use the execution semantics of Petri nets, looping is well defined, the execution token simply moves in a cycle, causing the activities to execute in a loop.

Since the purpose of the presented application was to model manufacturing processes in particular, Nested TNA was our chosen workflow model. This model is very simple while being able to represent the vast majority of manufacturing processes. In order to increase its expressivity, we extended this model by some additional concepts, which will be described later in this document.

MAKE workflow model is also important for this thesis, since it represents the same class of processes. The main difference is that MAKE workflows do not utilize the nested hierarchy as much as Nested TNA does. In particular, the authors of MAKE were interested in utilizing the nested model in scheduling and schedule visualization.

That is why the presented application is fully integrated into the MAKE system and it is possible to import and export workflows from and to the MAKE model, allowing the potential user to utilize both models.

BPMN and YAWL served as an inspiration for additional features both of the workflow model and of the workflow editor. Fully implementing one of them seemed to be much too complicated and unnecessary, considering the specific requirements of manufacturing processes.

Knowing how these models represent some of the more advanced workflow concepts can be very useful however, since the same approaches can be utilized in certain parts of our model. For instance the visual representation of workflow hierarchy in our application follows the example of BPMN and YAWL subprocesses.

Also, examining multiple systems allowed us to get a general overview of the most useful features of a typical workflow editor. We tried to implement all the features we encountered that were common or seemed especially useful, so that the potential user of the presented application find its environment familiar.

In the future, we would like to implement some of the more advanced workflow patterns that are supported by BPMN/YAWL. In particular the patterns involving multiple instances and reusable subprocesses (see the Future works chapter for details).

4. FlowOpt Workflows

This chapter informally describes the workflow model of the presented application. As mentioned previously, the model is based on the Nested TNA model, but it is extended with some additional features to be more expressive.

The presented model is extremely simple. There are only three types of objects in our workflows – activities, tasks and custom links. It is designed to be roughly as expressive as the MAKE model, meaning it should be sufficient to represent typical manufacturing processes.

All of the objects that are used to build FlowOpt workflows are informally described below. A formal description of the model is provided in a separate chapter.

4.1 Activities

Activities serve the same purpose as in all the other workflow systems presented in this document – they represent elementary units of work that need to be done in order to complete the workflow.

For scheduling purposes, activities contain a list of resources on which they can be performed. These resources can be employees, machines or their combination. Every activity also has a duration.

When we described the MAKE workflow model, we listed the same properties for activities. This is not a coincidence – the two systems actually share the activities, since they represent the same concept in both MAKE and FlowOpt. This has some nice benefits, like simplifying the process of converting a workflow from one model to the other or consistent use of activities from the user's perspective.

4.2 Tasks

Tasks are the biggest difference between the presented workflow editor and the one in MAKE (and most other workflow solutions). We have emphasized the tree structure of the Nested TNA workflows as the key feature of our workflow editor multiple times in this document. Tasks are the objects that we use to represent this structure.

A task represents some portion of the workflow – when the user creates a new workflow, it contains a single task that represents the whole process. Every task is created empty, which means that it has no defined "implementation" of the work it represents. This implementation can be either via an elementary activity, or through decomposition into child tasks that will represent smaller portions of the work. In order for a FlowOpt workflow to be complete, all the empty tasks have to be either decomposed, or populated with activities.

There are three types of decomposed tasks – serial, parallel and alternative. These hold the execution flow, in place of semaphores or decorators, which do not exist in FlowOpt workflows. Based on the type of the task, the execution semantics are as follows:

- A serial task executes all of its children in a sequence. The order of this sequence is a part of the serial task's definition.
- A parallel task executes all of its children in parallel.
- An alternative task executes exactly one of its children (it forms an exclusive nest in terms of BPMN).

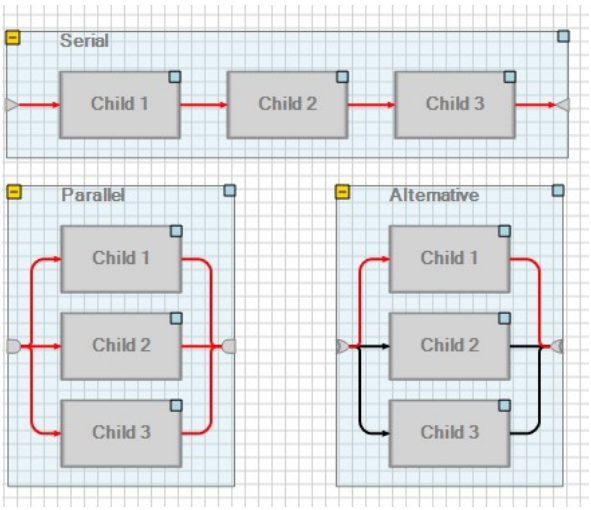


Figure 6: Task types

Figure 6 shows all three types of decomposed tasks. As we can see, decomposed tasks are represented as gray boxes containing the nest of their child tasks. Serial tasks form a sequence to indicate the order of execution of their children, parallel and alternative tasks form a fan in / fan out subgraph (in Nested TNA terminology).

As a further indication of the decomposition type, the input and output ports of the nest have specific shape – a triangle for serial task, a little AND gate for parallel tasks and a little OR gate for alternative tasks. For convenience, every task can have a name that appears in the top part of the task.

The combination of using decomposition and elementary activities to model the process naturally leads to a top-down kind of approach to workflow design, in which the user starts with a complex task and keeps decomposing it into smaller and smaller tasks, until the units of work are so finely grained that they can be represented by an activity. The concept is similar to the well-known Work Breakdown Structure [5].

A FlowOpt workflow can be thought of as a tree. The inner nodes of this tree are (decomposed) tasks, while activities are in the leaves. Technically, the leaves are also tasks that are populated with a single activity. The root of this tree is referred to as the *root task* in this document. It shouldn't be hard to see that tasks are an explicit representation of the nests introduced in the Nested TNA model.

There are two minor differences between our model and the original Nested TNA that should be pointed out:

1) The original model decomposes links, whereas we chose to decompose nodes instead. We think that decomposing nodes is more intuitive for the user, as in most workflow models, the work is stored in the nodes, not in the links.
2) The original Nested TNA model uses general temporal links (meaning that the links specify the minimum and maximum temporal distance between the

tasks they connect) while our model only uses general precedences (meaning that one task should be performed *before* the other). This change was made simply because the users didn't seem to utilize the general temporal constraints too much.

In terms of the presented workflow models, tasks could be compared to BPMN or YAWL subprocesses, except for the fact that they play a much more crucial role in FlowOpt, since their use is not optional - every FlowOpt workflow has to have the tree hierarchy of nested tasks.

While we are aware of the fact that imposing such a hierarchy on the workflow model may potentially be limiting for the user, there are many advantages to this approach. The workflow editor itself utilizes it to simplify workflow creation, navigation and layout. Other FlowOpt modules take advantage of the tasks as well. For instance the optimizer uses them to create schedules more effectively and the schedule viewer uses them for better visualization.

We believe it could be a viable alternative to other commonly used systems. Providing a working implementation as a proof of concept for this model is the main goal of this thesis.

4.3 Custom links

The third and final type of objects that can appear in FlowOpt workflows are custom links. These aren't a part of the original Nested TNA model. They are our own attempt to extend the model and make it more expressive. We made no other extensions to the Nested TNA model – if the user does not use custom links, the workflow conforms to Nested TNA.

Custom links are a special kind of links that the user can place between almost[1] any pair of tasks in the workflow to impose a specific (binary) constraint on these two tasks. There are several types of constraints that can be created using custom links:

- Precedence – general precedence relation, indicating that one task should be performed before the other.
- Logical constraints:
    - Implication – if a certain task is performed, some other task also has to be performed.
    - Equivalence – a certain task has to be performed if and only if some other task is also performed.
    - Mutual exclusion – at most one of a given pair of tasks can be performed (not both).
- Synchronization constraints – these indicate that two tasks should be synchronized in time. It is possible to synchronize tasks in the following ways:
    - Start-Start – both tasks start at the same time.

[1] The verification procedure will not allow the user to create some custom links that would create an invalid workflow. See the chapter on verification for details.

- End-End – both tasks end at the same time.
- End-Start – the second task starts precisely when the first ends.
- Start-End – the first task starts precisely when the second ends.

Custom links are meant to make the model more flexible and usable in more complex manufacturing processes. Figure 7 shows an example of a custom synchronization link of type End to End (between activities 'Sawing' and 'Welding').

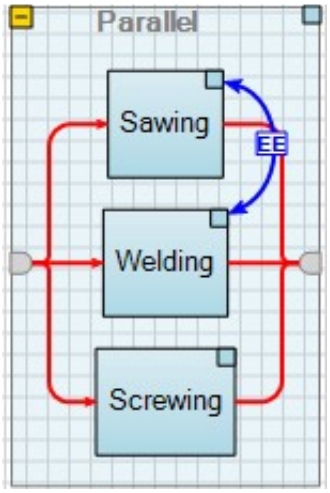


Figure 7: Custom synchronization link of type End to End

Figure 7 shows a simple task that executes three activities in parallel. Two of them (sawing and welding) will be planned so that they end at the same time. The custom links can be easily identified, since they are connected to a special port in the top right of every task (the blue square port). The type of a custom link can be determined from its color and label.

Custom links are called custom to emphasize the fact that they can be created and deleted by the user and connected to any task in the workflow. All the other links in the workflow are integral parts of the tasks, the user cannot create or modify them.

4.4 Example

As we can see, the workflow model of FlowOpt is extremely simple, containing only three types of objects. It was designed to be as lightweight and intuitive as possible, while preserving (and extending) the expressivity of Nested TNA. Now that we described the entire model, we provide an example of a complete FlowOpt workflow that utilizes all of the described objects.

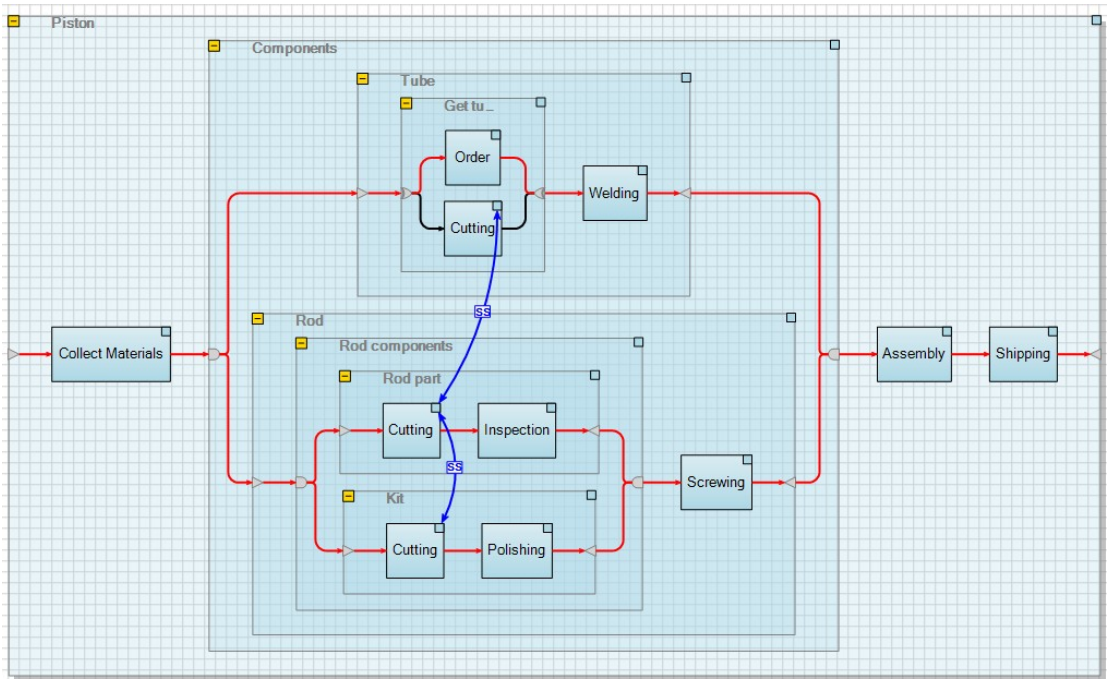


Figure 8: A complete FlowOpt workflow

The workflow in Figure 8 describes the process of manufacturing a piston. The structure of the workflow (the way it is built) follows the top to bottom approach to problem solving – we decompose larger processes into smaller ones and repeat until the units of work are small enough to be represented by activities.

First we have to collect the necessary materials, then manufacture the components of the piston, assemble them together and finally ship the piston to the customer. These tasks have to be done in a sequence, therefore the root task is serial.

Collecting materials, assembly and shipping are specific enough to be represented by elementary activities, but creating components is more complex. There are two components of a piston – a tube and a rod. These can be manufactured independently, so we used a parallel task.

To manufacture the tube, we first need to get the tube part and then weld it on the kit that attaches to the piston. The welding is an activity, but getting the part is not as easy. Let's say that there are two ways to obtain the part – we can make it ourselves (cut it from a piece of metal) or we can order it from a third party. This leads to alternative decomposition (the 'Get tube' task).

To make the rod, we have to once again get the parts somehow and then screw them together, which we model as an activity. The parts can be created in parallel, so we create a parallel task. We will need a rod part and a kit.

To create the rod part, we cut it from metal and make sure that its dimensions are just right to fit in the tube (inspection). To create the kit, we also have to cut some metal, plus we have to polish it so it can be attached to the rod.

Finally, we can also see that some custom links were used – in this case we know that cutting is expensive, so we make sure that we start all the cutting at the same time, so that the tool used to do the cutting doesn't have to run multiple times.

Once we have a workflow, we can utilize other FlowOpt modules to plan and visualize the manufacturing process. First, we can use the FlowOpt optimizer module

to create a schedule of works that will assign the workflow's activities to specific employees at specific times, effectively managing all the work that needs to be done in order to manufacture a piston.

Once we have the schedule, we can use the FlowOpt schedule visualizer module to display it as a Gantt charter[1]. One possible schedule for the piston workflow is shown in Figure 9.

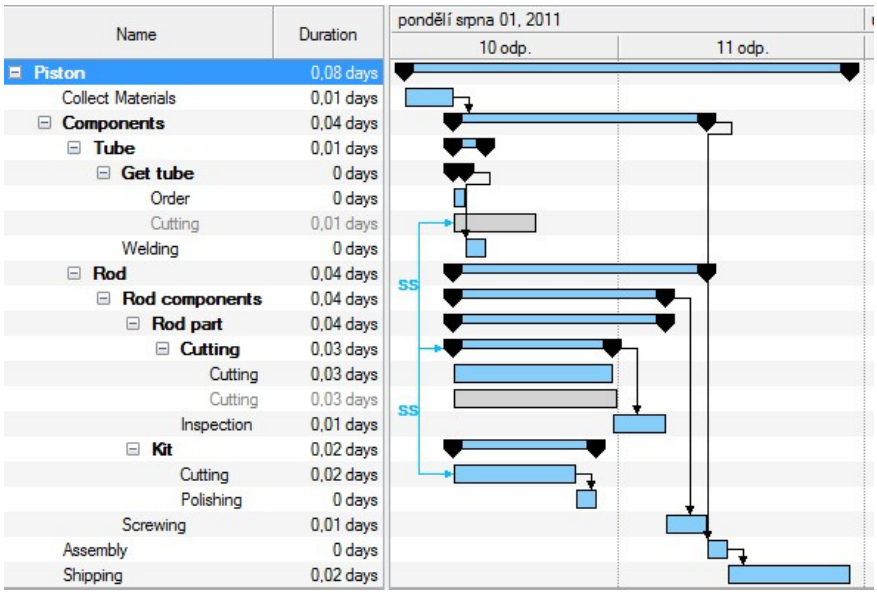


Figure 9: A schedule for a FlowOpt workflow

Figure 9 shows a schedule generated by the FlowOpt optimizer module for the workflow in Figure 8. In the left part, we can see all the tasks that were defined in the workflow. In the right part, we can see the exact times when the tasks should run, together with their expected durations (represented as a simple bar graph).

Notice that the workflow structure created in the workflow editor is still present here in the schedule – the user can still see the task hierarchy and any custom links between the tasks. In this example, we can see that the three 'Cutting' activities were synchronized to start at the same time (the blue link labeled 'SS').

Two activities are grayed out in the above schedule (both are named 'Cutting'). That is because they are children of an alternative task and they weren't chosen to be performed.

[1] A well-known and widely used schedule notation.

5. Formal Definition of the Workflow Model

Let us now describe our model more formally. This is done in three parts – first we define the three types of objects that can exist in a FlowOpt workflow (activities, tasks and custom links). Then we define all the constraints on these objects that have to hold in a FlowOpt workflow. Finally, we describe the process of building FlowOpt workflows (which similar to that in Nested TNA).

5.1 Workflow objects

We have already introduced the three types of objects that can exist in a FlowOpt workflow (tasks, activities and custom links) in the FlowOpt Workflows chapter. We will now define these objects and their properties formally.

A FlowOpt workflow is a tuple *(Activities, Tasks, Constraints)*:

- *Activities* is the set of all the activities in the workflow.
- *Tasks* is the set of all the tasks in the workflow.
- *Constraints* is the set of all the custom constraints (constraints defined by custom links) in the workflow.

The semantics of our workflow model is given by a number of constraints on these objects and their properties.

First, every task can be in one of five states, depending on the way in which it should be implemented. It can be empty, meaning it has no defined implementation yet, it can be implemented via an activity, or it can be decomposed in one of three ways (serial, parallel, alternative):

- $\text{Empty} = \{\text{T} \in \text{Tasks} | \text{T is empty}\}$
- $WithActivity = \{T \in Tasks | T\ is\ implemented\ through\ an\ activity\}$
- $Serial = \{T \in Tasks | T\ is\ decomposed\ using\ serial\ decomposition\}$
- $Parallel = \{T \in Tasks | T\ is\ decomposed\ using\ parallel\ decomposition\}$
- $Alternative = \{T \in Tasks | T\ is\ decomposed\ using\ alternative\ decomposition\}$
- $Decomposed = Serial \cup Parallel \cup Alternative$
- $Tasks = Empty \cup WithActivity \cup Decomposed$
- *Serial, Parallel, Alternative, Empty* and *WithActivity* are mutually disjunctive

Similarly, custom links are divided into several sets based on their type. There are precedence custom links, logical custom links (these can represent an implication, an equivalence or mutual exclusion) and synchronization custom links (of types start-start, end-end, start-end and end-start):

- $Precedences = \{C \in Constraints | C\ represents\ a\ precedence\}$
- $Implications = \{C \in Constraints | C\ represents\ an\ implication\}$

- $Equivalences = \{C \in Constraints | C\ represents\ an\ equivalence\}$
- $MUTEXes = \{C \in Constraints | C\ represents\ a\ MUTEX\}$
- $Logical = Implications \cup Equivalences \cup MUTEXes$
- $SS = \{C \in Constraints | C\ represents\ an\ SS\ synchronization\}$
- $EE = \{C \in Constraints | C\ represents\ an\ EE\ synchronization\}$
- $SE = \{C \in Constraints | C\ represents\ an\ SE\ synchronization\}$
- $ES = \{C \in Constraints | C\ represents\ an\ ES\ synchronization\}$
- $Synchronizations = SS \cup EE \cup SE \cup ES$
- $Constraints = Precedences \cup Logical \cup Synchronizations$
- *Precedences, Implications, Equivalences, MUTEXes, SS, EE, SE* and *ES* are mutually disjunctive

Now that we have introduced all the objects that can exist in a FlowOpt workflow, we define their properties. We will later introduce a number of constraints on these properties that will define a valid FlowOpt workflow (see Constraints on workflow objects and Workflow Verification chapters).

Every activity has a duration that indicates how long does it take to perform it. This duration is measured in some abstract units (same for every activity).

$\forall a \in Activities: Duration(a) = the\ duration\ of\ activity\ a$

For every task implemented via an activity, we introduce a predicate that will reference the activity.

$\forall t \in WithActivity: Activity(t) = a \in Activities, a\ implements\ t$

Every decomposed task is implemented by a nest of child tasks. The following predicate defines this (ordered) set of children.

- $\forall t \in Decomposed: Children(t) = (c_1, c_2, \dots c_n), c_i \in Tasks,$
  $c_i\ is\ within\ the\ nest\ of\ t$

Since the tasks form a tree hierarchy, we will also need to reference the parent task for a given child task.

- $\forall t \in Tasks$:
  - $\mathrm{Parent(t)} = p \in Tasks | t \in Children(p)$
    $or\ NIL\ if\ no\ such\ p\ exists$
  - $\exists!\ RootTask \in Tasks: Parent(RootTask) = NIL$

For a custom link, we will need to reference the pair of tasks that it connects.

- $\forall c \in Constraints$:
  - $From(c) = t \in Tasks | c\ goes\ from\ t$
  - $To(c) = t \in Tasks | c\ goes\ to\ t$
  - $Other\big(c, From(c)\big) = To(c), Other\big(c, To(c)\big) = From(c)$
  - *Parent(c)* = the task containing c - the first common parent of *From(c)* and *To(c)*.

- *FromTop(c)* = the direct child of *Parent(c)* from which *c* leads.
- *ToTop(c)* = the direct child of *Parent(c)* into which *c* leads.

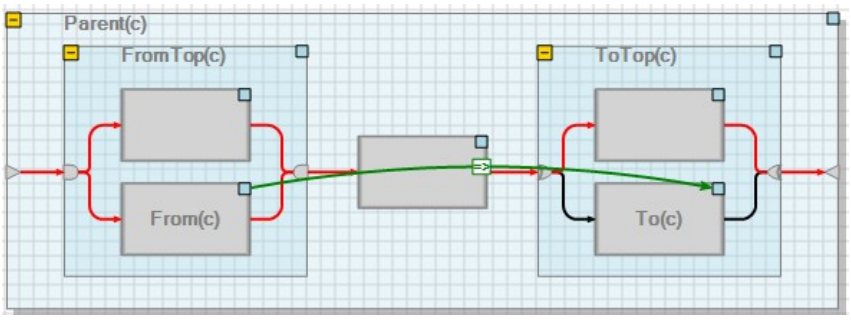


Figure 10: Custom link notation example

Figure 10 illustrates the introduced notation for custom links. The logical implication link is referred to as *c*.

Since the workflow may contain alternative tasks, we have to be able to determine whether a task should be performed or not. This information is stored in the *logical domain* of a task, i.e. a subset of *{true, false}* indicating whether the task is to be performed.

- $\forall t \in Tasks: \mathrm{LogicalDomain(t)} \in 2^{\{\mathrm{true,false}\}}$
  - $LogicalDomain(t) = \{true, false\} \Leftrightarrow t\ may\ or\ may\ not\ be\ performed$
  - $LogicalDomain(t) = \{true\} \Leftrightarrow t\ must\ be\ performed$ *(t is* ***active)***
  - $LogicalDomain(t) = \{false\} \Leftrightarrow t\ must\ not\ be\ performed$ (t *is* $\boldsymbol{inactive}$)
  - $LogicalDomain(t) = \emptyset\ indicates\ a\ constraint\ conflict (invalid\ workflow)$

Also, since the workflow has to eventually be scheduled, we have to maintain some temporal information about when a particular task should be executed. Specifically, we define two time points corresponding to the start and end time of each task (again in some abstract units, like the durations).

- $\forall \mathrm{t} \in \mathrm{Tasks}:$
  - $\mathrm{Start(t)} = \mathrm{start\ time\ of\ t}$
  - $\mathrm{End(t)} = \mathrm{end\ time\ of\ t}$
- $\mathrm{TimePoints} = \{\mathrm{Start(t)} | \mathrm{t} \in \mathrm{Tasks}\} \cup \{\mathrm{End(t)} | \mathrm{t} \in \mathrm{Tasks}\}$

We will not explicitly represent the (temporal) domains of these time points. Instead, we will place constraints on distances between them.

- $\forall \mathrm{p_1, p_2} \in \mathrm{TimePoints}: \mathrm{Dist}\ (\mathrm{p_1, p_2}) = [\mathrm{min, max}]$
  - $\mathrm{min} \leq \mathrm{p_2} - \mathrm{p_1} \leq \mathrm{max}$

5.2 Constraints on workflow objects

FlowOpt defines a number of constraints that describe a valid workflow. These constraints restrict both logical domains of tasks and temporal distances between the time points given by the tasks. They are all implied by the informal definitions of the workflow objects, but we need to express them formally to be able to verify or schedule the workflow properly. The constraint definitions are listed below.

1) Any *serial or parallel* task T is active if and only if all of its children are also active:
   $\forall t \in Serial \cup Parallel$:
   $LogicalDomain(t) = \{1\} \Leftrightarrow \forall c\ in\ Children(t)$: $LogicalDomain(c) = \{1\}$

2) Every active *alternative* task must have exactly one active child (this child is called the *active child*):
   $\forall t \in Alternative$:
   $LogicalDomain(t) = \{1\} \Leftrightarrow !\exists c_i \in Children(t)$:
   $LogicalDomain(c_i) = \{1\}\ \&\ \forall c_j, j \neq i$: $LogicalDomain(c_j) = \{0\}$

3) If any task becomes active, all the tasks on the path from this task to the root task must also be active:
   $\forall t \in Tasks$: $LogicalDomain(t) = \{1\} \Longrightarrow$
   $\forall p \in \{ Parent(t), Parent(Parent(t)), \dots, RootTask\}$:
   $LogicalDomain(p) = \{1\}$

4) The children in any active *serial* task must be performed in the order given by the serial task:
   $\forall t \in Serial, LogicalDomain(t) = \{1\}, Children(t) = (c_1, c_2, \dots, c_n)$:

$$\text{Dist}(End(c_1), Start(c_2)) = [0, \infty],$$
$$\text{Dist}(End(c_2), Start(c_3)) = [0, \infty],$$
$$\dots$$
$$\text{Dist}(End(c_{n-1}), Start(c_n)) = [0, \infty]$$

5) Every active task T containing (directly) an activity A must have its duration equal to the duration of the activity:
   $\forall t \in WithActivity, LogicalDomain(t) = \{1\}$:
   $Dist(Start(t), End(t)) = [Duration(Activity(t)), Duration(Activity(t))]$

6) Any active decomposed task *t* must start exactly when the first of its children starts and end exactly when the last of its children ends. For different task types, this yields different constraints:

   a. *t* is serial – synchronize the start of the parent with the start of the first child in the sequence and the end of the parent with the end of the last child in the sequence:
   $\forall t \in Serial, LogicalDomain(t) = \{1\}, Children(t) = (c_1, c_2, \dots, c_n)$:
   $Dist\big(Start(t), Start(c_1)\big) = [0,0]$
   $Dist\big(End(t), End(c_n)\big) = [0,0]$

   b. *t* is alternative – synchronize the start of the parent with the start of the active child and the end of the parent with the end of the active child:
   $\forall t \in Alternative, LogicalDomain(t) = \{1\}$:
   $Dist\left(Start(t), Start\big(ActiveChild(t)\big)\right) = [0,0]$
   $Dist\left(End(t), End\big(ActiveChild(t)\big)\right) = [0,0]$

   c. *t* is parallel – all the children must be performed after the parent starts and before the parent ends. Furthermore, synchronize the start of the parent with the start of the first child and the end of the parent with the end of the last child:
   $\forall t \in Parallel, LogicalDomain(t) = \{1\}$:
   $\forall c \in Children(t): Dist\big(Start(t), Start(c)\big) = [0, \infty]$
   $\forall c \in Children(t): Dist\big(End(c), End(t)\big) = [0, \infty]$
   $Dist\left(Start(t), Min_{c \in Children(t)}\big(Start(c)\big)\right) = [0,0]$
   $Dist\big(Max_{c \in Children(t)}\big(End(c)\big), End(t)\big) = [0,0]$

7) Any precedence link between active tasks t_1 and t_2 means that t_1 must be performed before t_2:
   $\forall c \in Precedences$:
   $LogicalDomain\big(From(c)\big) = \{1\}$ & $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow$
   $\text{Dist}\left(End\big(From(c)\big), Start\big(To(c)\big)\right) = [0, \infty]$

8) Any logical implication link between tasks t_1 and t_2 means that if t_1 becomes active, t_2 must also become active:
   $\forall c \in Implications$:
   $LogicalDomain\big(From(c)\big) = \{1\} \Longrightarrow LogicalDomain\big(To(c)\big) = \{1\}$

9) Any logical equivalence link between tasks t_1 and t_2 means that t_1 is active if and only if t_2 is active:
   $\forall c \in Equivalences$:
   $LogicalDomain\big(From(c)\big) = \{1\} \Longleftrightarrow LogicalDomain\big(To(c)\big) = \{1\}$

10) Any logical mutex link between tasks t_1 and t_2 means that at most one of the tasks can be active at a time:
   $\forall c \in MUTEXes$:
   $LogicalDomain\big(From(c)\big) = \{1\} \Longrightarrow LogicalDomain\big(To(c)\big) = \{0\}$
   $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow LogicalDomain\big(From(c)\big) = \{0\}$

11) Any SS synchronization between active tasks t_1 and t_2 means that the tasks start at the same time:

$\forall c \in SS$:

$LogicalDomain\big(From(c)\big) = \{1\}$ & $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow$

$Dist\Big(Start\big(From(c)\big), Start\big(To(c)\big)\Big) = [0,0]$

12) Any EE synchronization between active tasks T_1 and T_2 means that the tasks end at the same time:

$\forall c \in EE$:

$LogicalDomain\big(From(c)\big) = \{1\}$ & $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow$

$Dist\Big(End\big(From(c)\big), End\big(To(c)\big)\Big) = [0,0]$

13) Any SE synchronization between active tasks T_1 and T_2 means that T_1 starts precisely when T_2 ends:

$\forall c \in SE$:

$LogicalDomain\big(From(c)\big) = \{1\}$ & $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow$

$Dist\Big(Start\big(From(c)\big), End\big(To(c)\big)\Big) = [0,0]$

14) Any ES synchronization between active tasks T_1 and T_2 means that T_1 ends precisely when T_2 starts:

$\forall c \in ES$:

$LogicalDomain\big(From(c)\big) = \{1\}$ & $LogicalDomain\big(To(c)\big) = \{1\} \Longrightarrow$

$Dist\Big(End\big(From(c)\big), Start\big(To(c)\big)\Big) = [0,0]$

We will refer to the constraints above as *workflow constraints***.** We can divide workflow constraints into two categories – *general constraints*, which are defined by the task hierarchy and task decomposition – that is constraints 1) through 6) and *custom constraints*, which are defined by custom links – that is constraints 7) through 14).

5.3 Building FlowOpt workflows

We have defined the three basic objects that can exist in a FlowOpt workflow, as well as the constraints on their properties. We will now briefly describe the process of building the workflow.

This chapter's purpose is to emphasize the connection between Nested TNA and FlowOpt workflow model - as we will see, the way workflows are built is almost identical in both these models. For a description of the process from the user's point of view, please refer to the Building workflows chapter.

A newly created FlowOpt workflow contains a single empty task. At any time, the user may perform one of three basic actions:

- Decompose an *empty* task to convert it into a *decomposed* task with given type (serial, parallel or alternative) and a given number of children.
- Assign an activity into an *empty* task.
- Connect two tasks with a custom link of given type (precedence, implication, equivalence, MUTEX, SS, EE, SE or EE).

The workflow editor actually provides a number of additional actions that can change the workflow structure, but they are all just for user convenience, they will not create any workflow that cannot be created using the above actions (this is apparent after a brief analysis of the other actions, we didn't feel it was necessary to show this formally).

It shouldn't be hard to see that this way of building a workflow leads to a tree hierarchy of tasks, where the inner nodes are decomposed, while the leaves have assigned activities (or are empty).

If we assume no custom links in the workflow, we can see that the underlying network corresponds to a Nested TNA, since the way of building workflows in our model is defined analogically to the way used to define Nested TNA.

6. Features of the Workflow Editor

This chapter describes the most important features of the workflow editor. It is only meant as an overview of the workflow editor's capabilities, complete user documentation is beyond the scope of this document, but it is one of its attachments.

Two of the features are particularly interesting (workflow verification and import / export to and from other workflow models), because they were implemented outside the scope of the FlowOpt software project, specifically for this thesis. That is why they will be described in separate chapters.

6.1 Visualization

Perhaps the most basic, yet crucial feature of any workflow editor is its ability to visualize workflows in a user-friendly way. The bigger the workflow gets, the more important it is to layout its parts so that the workflow can still be interpreted easily.

The FlowOpt workflow editor utilizes the task hierarchy in visualization. Since the structure of workflows is so clearly defined, it is possible to implement an automatic layout procedure that takes care of positioning the tasks on the screen. The layout in the previous piston example was done by the application, not by the user.

This is one of the more significant differences between the FlowOpt workflow editor and other similar systems – the workflow editor fully controls the visualization, the user can only influence it marginally. For instance, it is possible to move the top-level tasks on the canvas, change the order of the child tasks within their parent or collapse and expand tasks, but the exact positions and sizes of the workflow objects are calculated automatically by the application.

On one hand, this may be a bit limiting in some scenarios, but it increases the productivity considerably, since the user doesn't have to worry about the layout at all, it is updated automatically as the workflow changes.

Many other workflow systems also implement some kind of automatic layout procedure, but it is usually more general and static (it doesn't update automatically to reflect the structure of the workflow). This means that the results typically aren't as good and the user has to make manual adjustments to the layout. Since our procedure is so closely connected to the workflow's nested structure, the layout is much more precise.

To make the visualization as convenient as possible, the workflow editor allows the user to choose orientation (horizontal or vertical) and the way child tasks are aligned with respect to their parent.

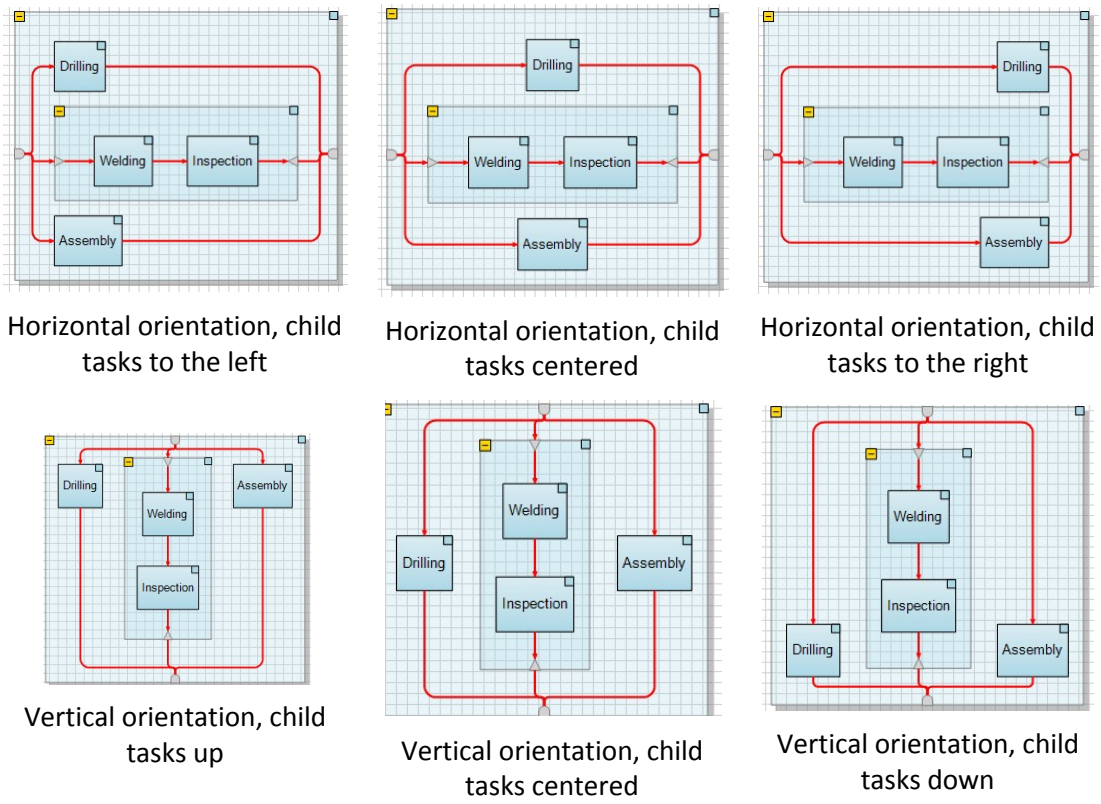


Figure 11: Orientation and child task align

Figure 11 shows different layouts of the same task based on changing orientation and child task alignment.

The editor also supports standard features like zooming (zoom in, zoom out, zoom to fit…), box selection, panning (moving the view by dragging the mouse) and collapsing / expanding tasks arbitrarily.

6.2 Navigation

Since workflows can get very large, it is important to provide features which make it easier to navigate them. The nested hierarchy makes this task somewhat easier, since navigating a tree is simpler than navigating a general graph.

Probably the most basic feature in terms of navigation is task collapsing. If the workflow gets large, it is useful to be able to collapse some portions of it to hide the details and only focus on tasks on a higher level.

Another feature that should make navigating larger workflows easier is the mini overview. It is a smaller view of the workflow at reduced scale, so that the user can see a larger area. It also lets the user quickly move the view by dragging the selection within the overview.

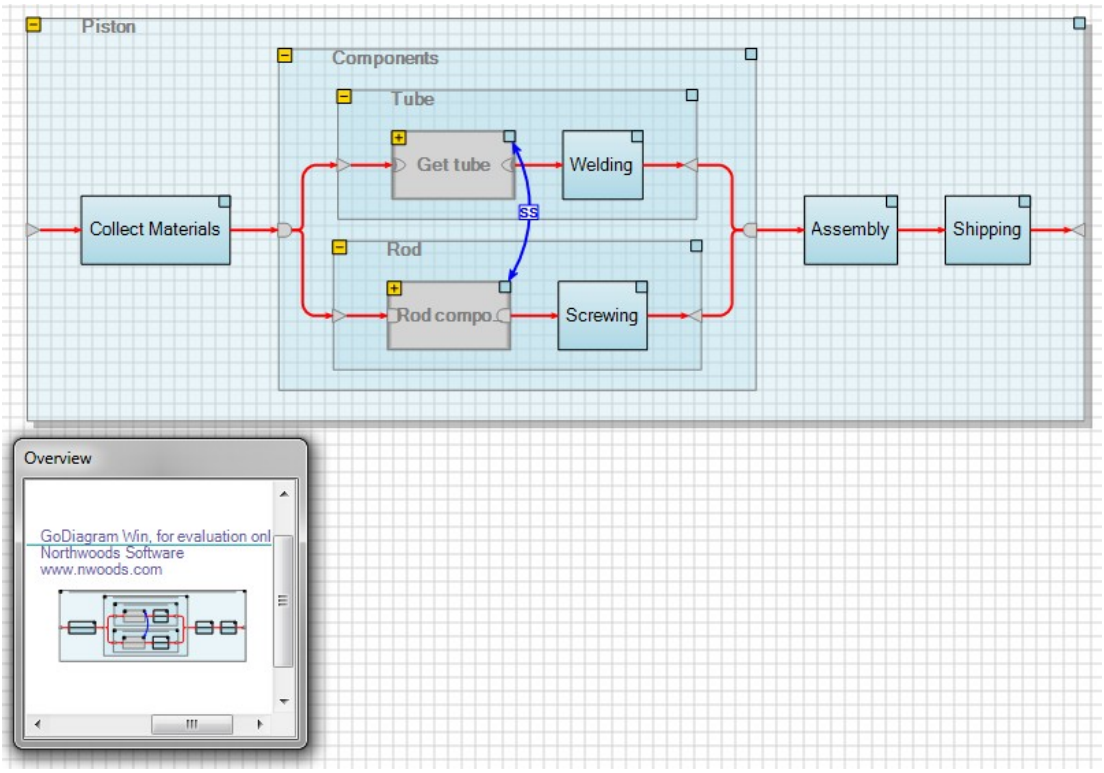


Figure 12: Task collapsing and overview

Figure 12 shows the piston workflow with two tasks collapsed and the overview visible.

We mentioned that the task hierarchy is also utilized to make navigating large workflows easier - the user can display a simple outline that displays the workflow's structure in a practical tree view.

Selecting a node in this outline also selects it in the workflow and the view moves over the selected task. Right clicking on a node in this view shows the same context menu as right clicking on the corresponding task in the workflow. This makes the outline a useful tool when the workflow gets more complex.

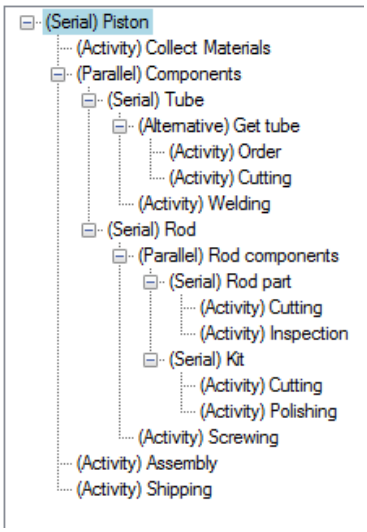


Figure 13: Workflow outline

Figure 13 shows what the outline looks like for the piston workflow example.

One more tool that should make navigating through larger workflows easier is the ability to focus on a particular task. When the user focuses on a task, all the tasks outside the selected task's subtree will be hidden. This makes it easier to concentrate on one particular task without accidentally changing other tasks.

Once the modifications to that task are done, the focus may be cleared and the whole workflow becomes visible again. It is even possible to focus multiple times, diving deeper and deeper into a particular subtree.

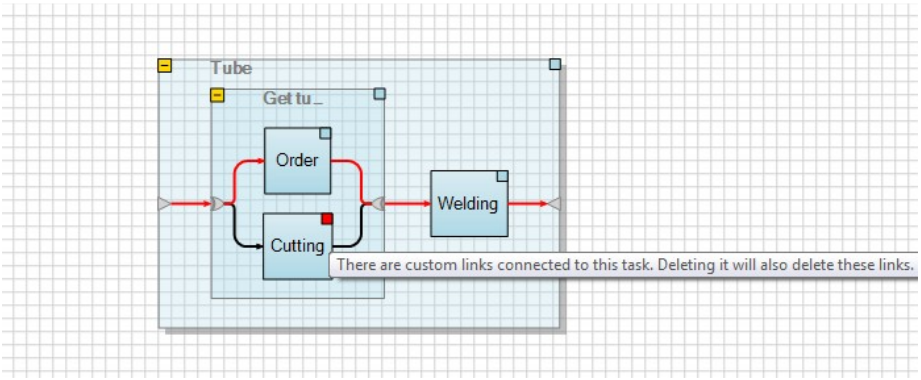


Figure 14: Task collapsing

In Figure 14, we focused on the 'Tube' task in the previous piston example. This caused the rest of the workflow to become hidden. Notice how the port on the 'Cutting' activity is highlighted to indicate that there are custom links connected to the activity and deleting the activity will also delete these links.

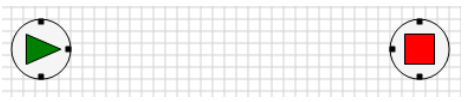
6.3 Building workflows

This chapter describes the actual process of building workflows in the FlowOpt workflow editor. Like visualization, this process is quite different to the approaches used in most other workflow solutions and it is also based on the nested structure of our workflow model.

There is one thing that almost all the workflow models presented earlier in this document have in common - the way of building workflows (technically, BPMN defines no such thing, but all the solutions that use BPMN that we examined followed this approach).

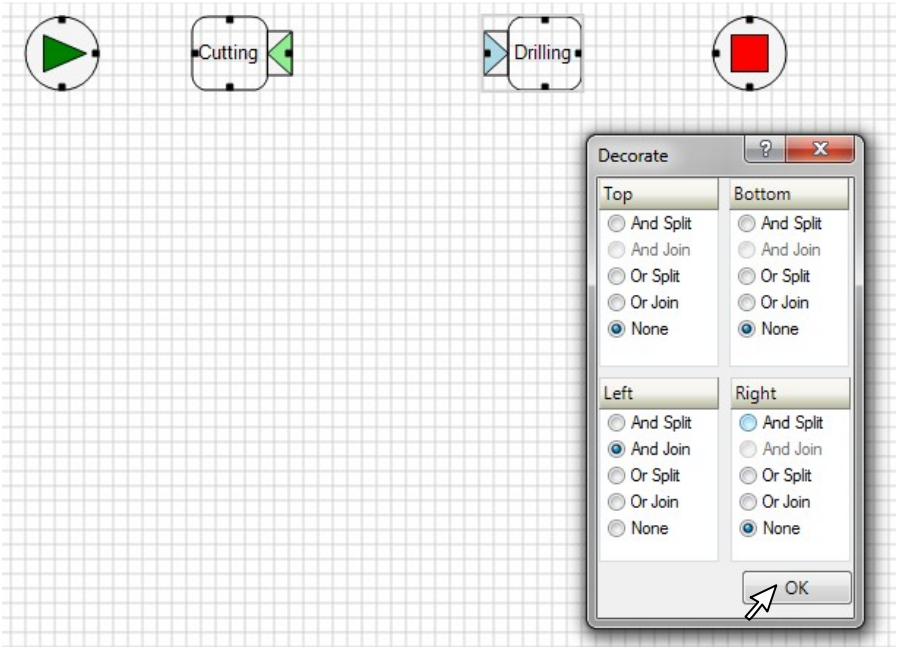
The usual approach lets the user of the workflow editor arbitrarily create and link workflow objects. The user first places activities, semaphores, events or other workflow objects on the canvas and then connects these objects with links to mark the order of execution. A simple example of this process is presented below.

The user starts with some trivial workflow, usually just the start and end activities, with nothing in between.

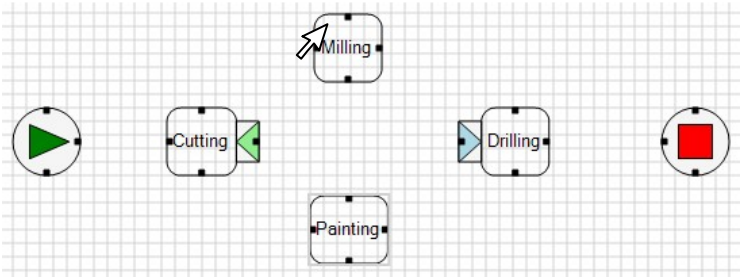


The user adds some activities to the canvas.

The activities are either decorated, or semaphores are added to create branching.



More activities are added as the workflow is being built.



The user connects the activities with links to mark the execution flow.

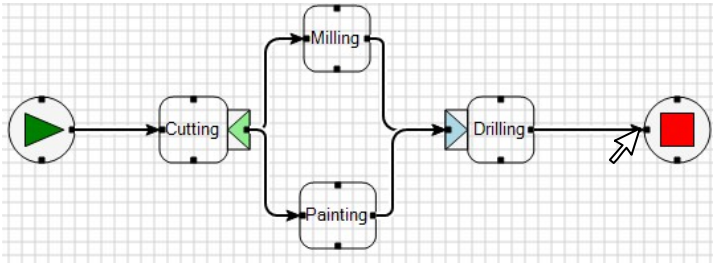


Figure 15: Traditional way to build workflows

The examples in Figure 15 are from the MAKE application, but the process is very similar in most of the other systems we tried out (the exception being Nested TNA).

While this approach is very intuitive and unrestrictive for the user, it doesn't seem very productive, considering the fact that the user has to create all the objects within a workflow manually.

In a general graph, the number of links is $O(n^2)$, where n is the number of nodes. Workflows are generally not dense in terms of the number of links, but the total number of objects is still rather large. Let us consider three of the most basic and common patterns present in virtually every workflow model: the sequence, the parallel nest and the alternative (exclusive) nest.

- In a sequence of n activities, there are n tasks plus $n-1$ links, that is $2n-1$ objects in total.
- In a parallel or an alternative nest, we have n tasks, 2n links (two links for every child task), and two decorators / semaphores to mark the routing type. That is $3n+2$ objects.

That is a lot of objects to create manually. FlowOpt workflow editor attempts to utilize the nested hierarchy inherent to FlowOpt workflows to alleviate this problem. The user doesn't build workflows in the traditional way described above. Instead, the workflow editor defines a set of actions that can be used to build workflows, following the definition of the Nested TNA model.

There are a number of those actions, but only a few of them are needed to build workflows, the others are only for users' convenience. There are three basic actions: *task decomposition*, *activity assigning* and *custom link creation*. Using just these three actions, the user can create any workflow in the FlowOpt model. It should be apparent from the names of these actions that they create the three basic objects of the FlowOpt workflow model – tasks, activities and custom links.

Task decomposition

We already introduced decomposition when we described the Nested TNA model. This action implements the Nested TNA decomposition, except for the fact that it decomposes nodes (tasks) instead of links.

Decomposition can be performed on any empty task. The user chooses the type of decomposition (serial, parallel or alternative to determine the routing within the task) and the number of children that should be in the resulting nest. The workflow editor automatically creates the resulting nest.

There are two ways to do this. The user can invoke a special dialog in which the number of children and the decomposition type can be chosen, but there is also a more streamlined way: the workflow editor overrides the resize function on empty tasks, so that the user can simply drag the border of an empty task to decompose it. Dragging horizontally creates a serial task, dragging vertically creates a parallel task (which can be quickly changed to alternative by double clicking on one of its ports).

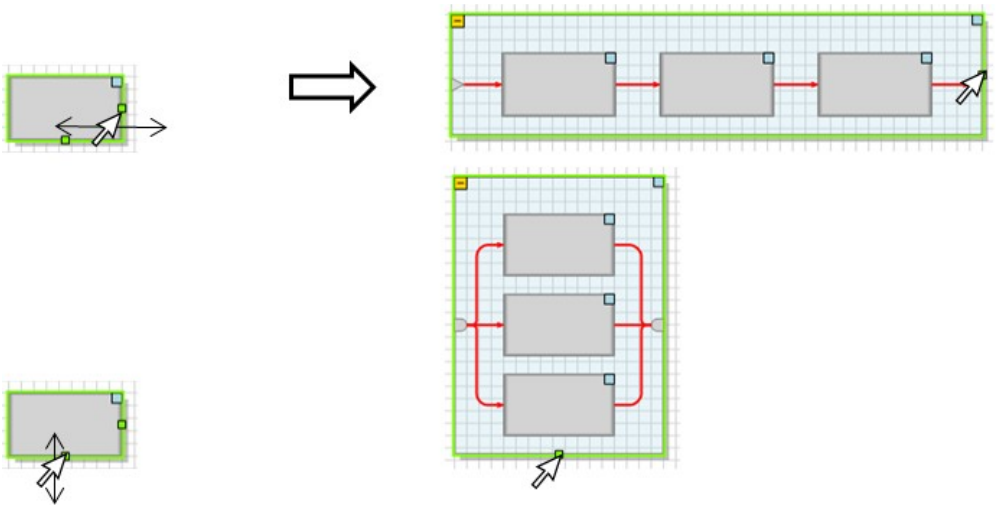


Figure 16: Decomposing tasks

Figure 16 illustrates the decomposition process.

This way the user can create the entire nest with a single drag of the mouse. It is still necessary to assign activities in the child tasks (or decompose them), but the process is generally much faster than creating the workflow manually (not to mention the automatic layout). Using this approach, the user only has to perform $O(n)$ actions to create a workflow containing n objects.

This way of building workflows is a bit more restrictive for the user than the traditional approach, but its advantages should be worth the compromise. You may notice that this is a similar idea as the one we used in Visualization – we try to make the work faster and more productive at the cost of restricting the user a bit more than the other editors do.

Note that once again, this way of doing things takes advantage of the strict hierarchy of the nested workflows. Features like these are what distinguishes our workflow editor from other workflow solutions.

Activity assigning

Once the tasks are created, we have to say how they should be "implemented". We can either decompose them (see above) or assign an elementary activity into them. The latter means that the task can be performed by performing the activity – assigning an activity into an empty task essentially creates a leaf in the tree hierarchy.

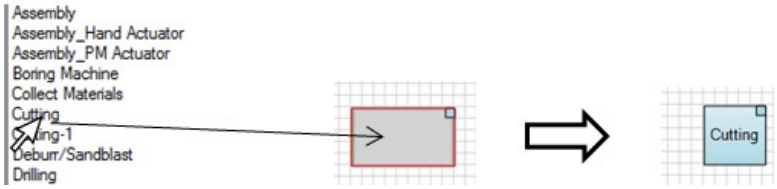


Figure 17: Assigning an activity

The way to do this is simple - the user just drags the activity from the list of defined activities into an empty task, as shown in Figure 17.

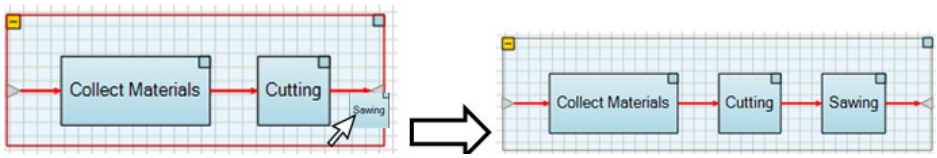


Figure 18: Streamlined decomposition

To streamline the process further, the user can also drop activities directly into decomposed tasks. This is illustrated in Figure 18.

Custom link creation

Custom links are created in the traditional way – connecting nodes manually. Since the user can link any two tasks with a custom link, there is probably no simpler way.

To create a custom link, the user simply drags the mouse between two task ports (blue squares in the top right of all tasks) and chooses the type of the link from the context menu that appears.

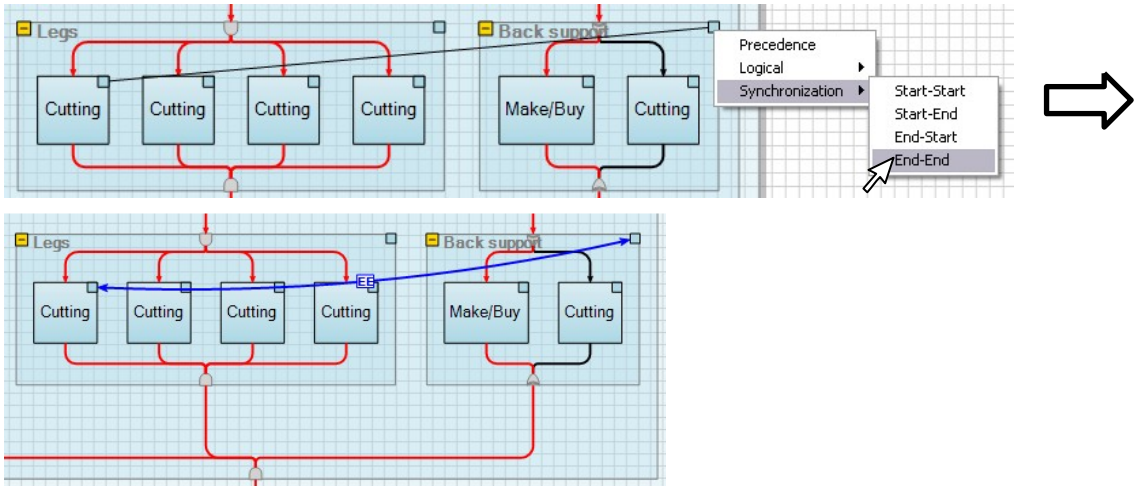


Figure 19: Custom links creation

Figure 19 shows the process of creating a custom end-end synchronization link in a workflow.

Other actions

Though the three actions described above are sufficient to build any FlowOpt workflow, the editor defines many more actions for user convenience. This chapter briefly goes through the more interesting ones.

To provide an alternative to the top to bottom approach implied by decomposition, we provide a way to create workflows the other way – bottom to top. The user can first create the child tasks and then *group* them into the parent task. This is more intuitive for some users, since it may be easier to start by adding activities to the workflow and build the task hierarchy from them.

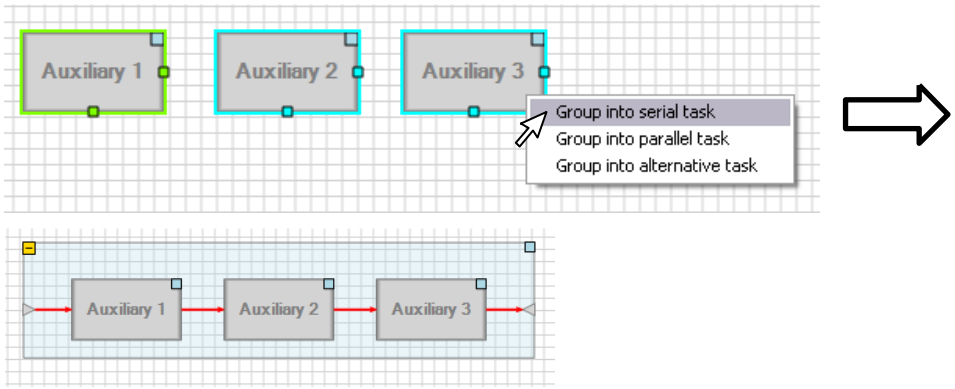


Figure 20: Task grouping

Figure 20 shows how several tasks can be grouped into a new decomposed (serial) task. The workflow editor preserves the order of the children based on their positioning before they were grouped.

Other actions implemented in the workflow editor include:

- Deleting tasks
- Changing order of the child tasks by dragging them within their nest.
- Inserting new tasks into the nest at any position.
- Changing the decomposition type or removing the decomposition entirely and making the task empty again.
- Copy, cut and paste operations on all the task types in order to make it easier to duplicate or move an entire subtree of the workflow hierarchy.
- Saving tasks as separate workflows so they can be reused later.
- Embedding (copying) other workflows into empty tasks.
- Enclosing existing tasks within the workflow in a new parent that is inserted in place of the original task.

It is likely that this list will be extended in the future based on the users' priorities.

6.4 Miscellaneous features

The workflow editor implements many other features that weren't listed above. While these features are mostly technical and not particularly interesting from the workflow design point of view, we would still like to briefly point them out, since they may be interesting from the implementation point of view.

Some of the more notable features are:

- Saving and loading of workflows to and from a database.
- Full undo and redo support for all the actions that the user can perform.
- Organizing workflows and activities into folders for user convenience.
- Maintaining preferred route - the user can mark one of the children of any alternative task as preferred, which may be somehow interpreted by the optimizer.
- Creating a picture from the workflow. Supported formats are PNG[1], SVG[2] and JPEG[3]. The last may seem a strange choice for representing diagrams, but for very large workflows it can produce considerably smaller files due to its flexible compression.
- Printing the workflow (can be used to create a PDF[4] file from the workflow with the help of some virtual printer like PDFCreator[5]).

[1] Portable Network Graphics, a raster format with lossless compression

[2] Scalable Vector Graphics, a vector format

[3] Joint Photographic Experts Group, a lossy raster format

[4] Portable Document Format, a vector format for document exchange

[5] http://sourceforge.net/projects/pdfcreator/

7. Workflow Verification

Workflow verification is the process of determining whether there are any errors in the design of a given workflow. Although the workflow model of FlowOpt is relatively simple and transparent, it is possible to create such errors – specifically by adding conflicting custom constraints.

As a consequence of imperfect design, some of the workflow's tasks may never be performed or, in the worst case, the entire workflow may be impossible to schedule.

That is why it is necessary to provide the user with a way to determine whether a particular workflow is valid or whether there are any problems that should be fixed prior to running the optimizer.

This chapter works with terms defined in the Formal Definition of the FlowOpt model. Please refer to that chapter for more information on the model or the notation used.

7.1 Problem definition

Problem: Decide whether a FlowOpt workflow is valid.

Definition: A *process* is a workflow in which the root task is active and all the workflow constraints hold. Specifically, every alternative task has exactly one active child in any process.

Intuitively, a process corresponds to a specific way how the workflow may be executed (specific assignment of values to *LogicalDomains* and *Dist* so that all the workflow constraints hold).

Definition: A FlowOpt workflow is valid if and only if the following conditions hold:

a) The workflow contains no empty tasks

b) $\forall t \in Tasks\ \exists\ process\ p: LogicalDomain(t) = 1\ in\ p$

The first part of this definition states that a valid FlowOpt workflow cannot contain empty tasks. This should be obvious - we cannot schedule an incomplete workflow.

The second part states that in a valid workflow, all the tasks can actually be performed in some process – there is no constraint conflict that would mean that some task can never become active. Specifically, this condition requires that every child of an alternative task can become active in some process. In the rest of this document, we will call the children of an alternative task *alternatives*.

If an alternative is valid in the sense of b), we will call it an *accessible* alternative, otherwise we will call it *inaccessible*.

The presence of empty tasks (part a) can be determined trivially in linear time by going through all the tasks in the workflow. For the rest of this chapter, let us assume that there are **no empty tasks in the workflow**.

The second part is the more difficult one and determining whether all the alternatives can be a part of some process is the core of the verification.

7.2 Verifying workflows without custom links

Let us first focus on the case when the user does not create any custom links. As it turns out, verifying workflows with no custom links is trivial, because they are always guaranteed to be valid (provided that they do not contain empty tasks).

Proposition: A workflow containing no empty tasks and no custom links is always valid.

Proof: We can prove the above proposition through simple induction over complexity of the workflow. We will follow the way that FlowOpt workflows are built (see Building FlowOpt workflows) and show that at every point, we have a valid workflow.

We assume that the workflow contains no empty tasks, otherwise the workflow is by definition invalid. In other words, all empty tasks have either been decomposed or assigned an activity. It should be apparent that the induction is still correct, since the structure of the workflow is built through decomposition only, assigning an activity cannot change it.

A trivial workflow consists of a single task with an assigned activity. We can create a process for this workflow by performing just this one task. All the workflow constraints will trivially hold.

Let us assume we decomposed an empty task *T* in a valid workflow (and assigned activities in the new tasks). From induction we know that for every task in the old workflow there is a process that performs it. We now extend each of these processes to include the tasks newly created by decomposition.

If *T* is performed in the original process then:

a) If *T* is serial, perform all the tasks in the newly created nest in the order given by *T*.

b) If *T* is parallel, perform all the tasks in the newly created nest in parallel.

c) If *T* is alternative, perform any single task in the newly created nest (create a new separate process for each of the alternatives).

We need to show that this way of performing the new tasks will not create a constraint conflict (in other words, we will still have a process after the above extension). If that holds, we know that we have a process for every task in the new workflow and it is therefore valid.

To show that no constraint conflicts can occur, we can first note that if we assume no custom links, only the general constraints are in effect. We can also notice that due to the tree structure of the workflow, the only task that has any effect on the new tasks what so ever (in terms of logical domains or temporal distances) is their parent. The other tasks cannot restrict the new tasks in any way due to the definition of the general constraints. The reasoning for all six types of general constraints is given below:

- Constraints of types 1 and 2 cannot create a conflict, because the only task that has any impact on the new task's logical domains is their parent. Since the new process is created exactly so that the parent's constraint holds, there cannot be a conflict.

- Constraints of type 3 will cause no conflict due to the fact that we only perform the new children if their parent is performed in the old process.
- Constraints of types 4, 5 and 6 will also not create a conflict, since there is no upper bound on the tasks' durations and the workflow forms a DAG[1], so no cycles can appear either. Both of these facts can be easily observed from induction (they hold for a single activity, decomposing an empty task will not change them). This means we can set the durations and order of the new tasks however we want.

Since there are no constraint conflicts, the new process is valid and therefore (from induction) the new workflow is also valid.

This result is not too surprising, due to the fact that our model is based on Nested TNA. However, notice that unlike our model, Nested TNA does not guarantee temporal consistency, which is due to the fact that it uses general temporal constraints on its links (they specify minimal and maximal distance between the nodes they connect). Our model is less expressive in this regard, since we only use general precedences. Consequently, there is no upper limit on the durations of the tasks, which results in guaranteed temporal consistency.

To summarize, unless the user creates a custom link in the workflow, checking validity is trivial – we simply go through all the tasks in the workflow and look for empty tasks and custom links. If there are none, the workflow is valid.

Unfortunately, we have reason to believe that if the user does use a custom link, the problem becomes NP-complete (proof will be presented in [13]). It seems that any kind of custom link (precedence, logical or synchronization) is enough to move the problem into the NP-complete class.

In the following sections we describe the general algorithm used in FlowOpt to verify workflows that do contain custom links, then we show how the workflows can be simplified without the loss of generality prior to running the algorithm to speed it up considerably.

7.3 General workflow verification algorithm

The full verification process consists of several parts:

1) Preprocessing
2) Generating a common FlowOpt representation of the workflow (CDM[2]).
3) Core verification algorithm

Preprocessing is essentially a number of simple checks that make sure that there is no custom constraint that is obviously invalid or redundant. These ad-hoc checks are useful in the sense that the user can immediately see the problem and the core verification algorithm need not even start. They are simple and fast and can even be

[1] Directed Acyclic Graph

[2] Common Data Model of FlowOpt, for details please see FlowOpt development / programmer documentation.

performed while the user works with the workflow. Currently, the following problems are detected by the preprocessing phase:

- Any duplicate link (a link of the same type is already in place).
- Any link *L* where *Parent(L)* is an alternative task. Such a constraint is always redundant, because at most one of the tasks it affects will ever be active.
- Any logical link *L* where both *From(L)* and *To(L)* are always active, meaning there is no alternative task on the path from *From(L)* to the root task or from *To(L)* to the root task. Since both the tasks are active, the constraint is either redundant (implication, equivalence) or invalid (mutual exclusion).
- Any precedence link *L* where *Parent(L)* is a serial task. Such a constraint is always either redundant (if the link follows the sequence direction) or invalid (if it goes in the wrong direction)[1].
- Any synchronization link *L* where *Parent(L)* is a serial task and type of L is not *ES or SE*. Also the link has to go in the right direction (*ES* links must follow the sequence, *SE* link must go in the opposite direction). The reasoning here is similar as in the point above.
- Any precedence link where *From(L)* is a (direct or indirect) parent of *To(L)* or vice versa. Such a constraint is always invalid, since the children must always occur within the temporal bounds of their parent.

If the preprocessing step finds no errors, the second part of the algorithm starts. In this phase, the workflow is translated into the data model used across FlowOpt modules (CDM – please see FlowOpt project documentation for details).

While there is no semantic difference between the data model used by the workflow editor and CDM, this step is still worth mentioning from the technical point of view, since it has some notable advantages:

- CDM represents the actual data model of FlowOpt as a whole, so if we verify a workflow in CDM, we know it will be valid in all the other FlowOpt modules besides the workflow editor.
- The workflow editor's data model carries a lot of additional information that isn't semantically important for the verification purposes (for example visual representation of the workflow). On the other hand the CDM is designed to be as simple and lightweight as possible, which in turn makes the verification process simpler and more transparent.
- It is likely that the workflow editor's model will be extended in the future with some additional workflow patterns. As long as CDM remains the same, no changes in the verification algorithm will be needed, only the procedure that generates CDM from the workflow editor model will have to be extended.

---

[1] One could argue that if all the tasks within the sequence have zero duration, then the precedence might not be outright invalid, as all the tasks could be scheduled into a single point in time. However we still consider this a conceptual error, because an uninterruptible cycle in the workflow usually implies an error in design.

The third and final part of the verification process is the core verification algorithm, which is described in the next chapter.

7.4 Core verification algorithm

The algorithm used to verify FlowOpt workflows is based on the techniques commonly used to solve the disjunctive temporal problem (DTP) – constraint propagation with forward checking [6].

The algorithm basically goes through all the possible valuations of the tasks‘ logical domains while trying to create a process by propagating all of the workflow constraints. As a consequence of the definition of the general constraints (particularly constraint types 1 and 2), this boils down to trying every possible choice of active child for every alternative task in the workflow while making sure that all the workflow constraints hold.

If a constraint conflict is detected at some point, the algorithm goes to the next possible combination of active children. If all the constraints are propagated successfully, the algorithm marks a new process and remembers the alternatives that were performed in that process – we know that these are accessible.

This explicit evaluation of (potentially) all alternative combinations means that the algorithm is exponential, which is to be expected, since the problem it solves is NP-complete.

Eventually we either find a process for every task in the workflow or we examine all of the possible alternative combinations. The rest of this chapter describes this algorithm formally.

Propagating constraints

When we propagate the workflow constraints, we have to make sure that they do not conflict with each other in any way - detecting a constraint conflict is a crucial part of the verification algorithm. In general, we will be propagating two kinds of constraints – logical constraints, which restrict logical domains of tasks, and temporal constraints, which restrict distances between the time points.

It is relatively easy to detect a conflict when changing the logical domains of tasks – all we have to do is make sure that we aren’t trying to activate a task that was previously deactivated or deactivate a task that was previously activated. In other words, we cannot assign logical domain of $\{1\}$ to a task that has logical domain of $\{0\}$ or vice versa. This check can of course be done in constant time.

Detecting a temporal conflict is a bit more difficult. It essentially means solving the Simple Temporal Problem [7]. Let us define this problem formally.

An instance of the simple temporal problem (STP) consists of a set of time points $P = \{p_1, p_2, \dots, p_n\}$ and a set $C = \{c_{i,j} | p_i, p_j \in P\}$ of binary temporal constraints on time differences between these time points.

Each constraint $c_{i,j}$ has a defined $weight \in \mathbb{Z}$, which is an upper bound on the time difference between the time points p_i and p_j:

$$weight(c_{i,j}) = n \Leftrightarrow p_j - p_i \leq n$$

A solution to an STP instance is an assignment of exact values to all the time points so that all the constraints hold. An instance of STP is called consistent if it has at least one solution.

Detecting a conflict while propagating temporal constraints is equivalent to deciding whether a specific instance of STP is consistent. This instance is defined by the time points and temporal constraints imposed by the workflow:

$$P = TimePoints$$
$$C \sim Temporal\ workflow\ constraints$$

In other words, we will maintain an instance of STP describing the temporal properties of the workflow that we are verifying. Every time we need to propagate a temporal constraint, we do so by adding this constraint into the STP instance. The propagation succeeds if and only if the resulting STP instance is still consistent.

There are many ways how we can solve an STP, for instance the well-known Bellman-Ford algorithm. The problem is that most of these traditional algorithms work on a static temporal network, meaning we would have to re-run the whole temporal verification every time we add a new constraint to the network. This would be very inefficient, since the workflow verification procedure gradually builds the temporal network and the temporal verification is invoked very often.

For our purposes the incremental algorithms presented in [7] are much more interesting, since they take advantage of the fact that we are trying to add a constraint to an already consistent temporal network and generally perform better as a result.

That is why the verification procedure currently implements one of these algorithms (namely IFPC[1]).

The Verify method

Let us now describe the core verification algorithm together with pseudocode of its key methods. First we have a method called `Verify` that is to be called on the workflow's root task and that encapsulates the whole verification process.

```
FUNCTION Verify(Task rootTask):

Inaccessible = ⋃_{t∈Alternative} Children(t)
TODO = ∅
∀t ∈ Tasks: LogicalDomain(t) = {true, false}
∀p1, p2 ∈ TimePoints: Dist(p1, p2) = [−∞, +∞]

IF (NOT ActivateTask(RootTask))
      RETURN false

IF (IterateAlternative())
      IF(|Inaccessible| > 0) RETURN ProcessExists
      ELSE   RETURN FullyVerified
ELSE   RETURN NoProcessExists
```

Figure 21: The Verify method pseudocode

The method in Figure 21 accepts the root task of the workflow and returns either `NoProcessExists` if there is no process what so ever for the workflow due to some serious constraint conflict, `ProcessExists` if some process does exist, but

[1] Incremental Full Path Checking

some alternatives are inaccessible or `FullyVerified` if all the alternatives are accessible.

First, we initialize some values that will be used throughout the verification procedure. To keep track of which alternatives are inaccessible, we will define a set of tasks called `Inaccessible` which will hold them. When the verification starts, let all the alternatives be inaccessible. As the algorithm progresses, we will remove those alternatives that we know are accessible (whenever we find a process, we remove all alternatives that are active in that process).

We will also need to maintain a set of alternative tasks that have yet to be examined. This set is called `TODO`. Whenever we encounter an alternative task, we will add it to this set so that we can later determine which of its children are accessible.

When the verification starts, all the logical domains are set to *{true, false}* and all the temporal weights are equal to infinity (meaning there is no restriction on any of the values).

We start by activating the root task and propagating all the workflow constraints that this implies. This is done by the `ActivateTask` method, which accepts a task that should be activated and returns true if the propagation succeeds or false if it fails (that is if some kind of conflict is found).

If the root task was activated successfully, we start examining all the possible alternative combinations in the workflow to determine which ones are accessible – this is done in the `IterateAlternative` method. `IterateAlternative` succeeds if at least one process exists for the workflow.

If that is the case, we can check whether there are any inaccessible alternatives. If not, the workflow is valid. If there are inaccessible alternatives, the workflow is not valid in the sense of the definition we used, but we can still let the user know that at least one process exists.

Notice that the `Verify` method directly follows the definition of a process that we established: we first activate the root task and propagate all the constraints to make sure they hold. We then go through all the possible alternative combinations, activating all the alternatives one at a time to find out which ones can be in a process. If we manage to examine all of the alternatives and propagate all the constraints correctly, the `Verify` method must also be correct.

The IterateAlternative method

The `IterateAlternative` method goes through all the unresolved alternative tasks in the workflow and determines which of their children are accessible. The method returns true if there is at least one accessible alternative, i.e. if any process exists for the workflow.

```
FUNCTION IterateAlternative():

IF(|Inaccessible| = 0 OR |TODO| = 0)
       Inaccessible = Inaccessible \ ⋂_{t∈Alternative}{ActiveChild(t)}
       RETURN true

Task next = Pop(TODO)
bool validAlternativeFound = false

IF(next already has an active child)
       RETURN IterateAlternative()

State = current values of LogicalDomains and Dist
FOREACH(Task A ∈ Children(next))
       Set LogicalDomains and Dist to values stored in State

       IF(NOT ActivateTask(A))
              Skip to the next alternative

       IF(IterateAlternative())
              validAlternativeFound = true

       IF(all alternatives are accessible)
              RETURN true

RETURN validAlternativeFound
```

Figure 22: The IterateAlternative method pseudocode

The method in Figure 22 first makes sure there are still some (potentially) inaccessible alternatives. If there are none or if there are no more alternative tasks to go through (*TODO* is empty), we have a complete process. That makes all the currently active alternatives accessible. We note which alternatives they are and end.

If there are more alternative tasks to be examined, we will take any of them and determine which of its children can be activated. Note that the decision on which alternative task to choose can affect the algorithm performance considerably. Our current implementation simply chooses the first one.

Once we have chosen an alternative task to examine, we will first check whether one of its children is already activated (this could happen during previous propagation of some custom constraint). If that is the case, we have no choice but to try a different alternative task. If no children are activated yet, we try to activate all of them, one at a time, in order to determine which ones can be in a process.

In order to do this correctly, we have to remember the values of all the logical domains and temporal distances before we start activating the children, because activating a child will cause changes to these values that we need to revert when we want to activate a different child (only one child of an alternative task can be active at a time).

If activating a child fails, we cannot mark it as an accessible alternative, so we just go to the next child. If the activation succeeds, we have to iterate through the rest of the alternative combinations in the workflow, trying to complete the process.

In the end, we return true if we managed to find any process, using any of the children.

The ActivateTask method

Next is the `ActivateTask` method, which activates a single task and propagates all the workflow constraints. We will emphasize the parts where some constraint is

being propagated by marking the respective line with the number of the constraint type.

```
FUNCTION ActivateTask(Task T):

IF(LogicalDomain(T) = {1}) RETURN TRUE

FOREACH(t ∈ {Parent(T), Parent(Parent(T)),… RootTask})                    (3)
       ActivateTask(t)

IF(LogicalDomain(T) = {1}) RETURN TRUE
ELSE LogicalDomain(T) = {1}

                                                                          (2)
IF(Parent(T) ∈ Alternative)
       FOREACH(Task t ∈ Children(Parent(T)), t != T)
              DeactivateTask(t)
                                                                          (6.b)
       Dist(Start(Parent(T)), Start(T)) = [0,0]                           (6.b)
       Dist(End(T), End(Parent(T)) = [0,0]

IF(T ∈ WithActivity)                                                      (5)
       Dist(Start(T), End(T)) =
            [Duration(Activity(T)), Duration(Activity(T))]
                                                                          (6.a)
IF(T ∈ Serial, Children(T) = (c1,c2,… cn))
       Dist(Start(T), Start(c1)) = [0,0]
       Dist(End(cn), End(T))) = [0,0]                                     (4)

       FOR(i = 1 … n - 1) Dist(ci, ci+1) = [0,∞]

                                                                          (6.c)
IF(T ∈ Parallel)
       FOREACH(Task C ∈ Children(T))
              Dist(Start(T), Start(C)) = [0,∞]
              Dist(End(C), End(T)) = [0,∞]

IF(T ∈ Alternative) TODO = TODO ∪ {T}                                     (1)

IF(T ∈ Serial ∪ Parallel)
       FOREACH(Task C ∈ Children(T))
              ActivateTask(C)

                                                                          (7-14)
FOREACH(newly active custom constraint C)
       PropagateConstraint(C)

RETURN true if there are no constraint conflicts, false otherwise
```

Figure 23: The ActivateTask method pseudocode

The method in Figure 23 activates a given task and propagates all the workflow constraints (with one exception - constraints of type 6.c are not fully propagated here for reasons described later). If there is any conflict during the constraint propagation, the method returns false, otherwise it returns true indicating a successful activation.

In other words, whenever any of the methods that are called in the process of activating a task fails, so will the whole activation – we require that all of the constraints hold. For the sake of simplicity, we didn't explicitly test the return values of those methods in the pseudocode above, simply having the method end whenever a conflict occurs should be understandable.

The method first checks whether the task is already activated – if that is the case, we do not have to do anything. Otherwise we activate both the task and all of its predecessors on the way to the root task to make sure that workflow constraint type 3) holds. A task could potentially have inactive parent if it was activated by a custom link.

If we want to activate a child of an alternative task, we have to deactivate all of its siblings (all the other children of the alternative task) to make sure that constraint 2) holds. Also, we must synchronize the start and end of the child with the start and end of its parent to make sure that all the constraints of type 6.b hold.

If the task contains an activity, we just set an exact distance between the task's start and end points to the activity's duration (constraint type 5).

If the task is serial, we can synchronize the task's start point with the start point of its first child and the end point of its last child with the end point of the task. We also have to propagate all the precedences given by the serial task's ordering (constraints 4 and 6.a).

If the task is parallel, then all we know at this point is that all of its children must start after the parent starts and end before the parent ends (part of constraint type 6.c), so we can only add general precedences. Note that this isn't precise enough – we need the task to start at the exact moment its first child starts and end at the exact moment its last child ends in order to enforce constraint type 6.c, but we do not have that information about the task's children yet.

We need to propagate constraints of type 6.c in later stages of the algorithm, when we know for certain which child starts first and which child ends last. Specifically, we can do it in the `IterateAlternative` method – before we mark a valid process, we try to propagate all the constraints of type 6.c. One way to do the actual propagation is described in Propagating constraint type 6.c.

After that, we propagate the activation to any children the task may have. For serial and parallel tasks we have to activate all the children. If the task is alternative we just add the alternative task to the context's TODO list of alternative tasks, so that we can later examine all of its children separately.

Finally, we must also propagate the custom constraints that became active as a result of activating some tasks. By definition, logical constraints are active if and only if

$$\text{LogicalDomain}(\text{From}(\text{C})) = \{1\} \vee \text{LogicalDomain}(\text{To}(\text{C})) = \{1\}$$

Precedence and synchronization constraints are active if and only if

$$\text{LogicalDomain}(\text{From}(\text{C})) = \{1\} \,\&\, LogicalDomain(To(C)) = \{1\}$$

This should be intuitive, since as soon as one end of a logical constraint is active, it may affect the logical domains of other tasks. On the other hand, precedence and synchronization constraints need both their end tasks to be active, since if a task is not performed, no temporal constraints should be enforced on it.

The custom link propagation is done in the `PropagateConstraint` method, which simply changes the verification context based on the constraint type. Suppose we want to propagate a custom constraint C, as a result of activating a task T (meaning that *T* is active and either *From(C) = T* or *To(C) = T*).

Based on the type of the constraint, the `PropagateConstraint` method would do one of the following actions:

Constraint	Action	
Precedence	(7)	`Dist(End(From(C)), Start(To(C))) = [0,∞]`
Logical Implication	(8)	`IF(From(C) = T)  ActivateTask(To(C))`
Logical Equivalence	(9)	`ActivateTask(Other(C,T))`
Logical Mutex	(10)	`DeactivateTask(Other(C,T))`
Synchronization SS	(11)	`Dist(Start(From(C)), Start(To(C))) = [0,0]`
Synchronization EE	(12)	`Dist(End(From(C)), End(To(C))) = [0,0]`
Synchronization SE	(13)	`Dist(Start(From(C)), End(To(C))) = [0,0]`
Synchronization ES	(14)	`Dist(End(From(C)), Start(To(C))) = [0,0]`

Figure 24: The PropagateConstraint method behavior

Figure 24 describes the behavior of the PropagateConstraint method for all the possible constraint types. We felt it wasn't necessary to provide pseudocode for this method, since it would only be a switch operation on the constraint type.

Since activating tasks is a very significant part of the algorithm, let us briefly show why this method works correctly and determine an upper bound on its worst-case complexity.

Correctness

All the workflow constraints (except 6.c) are propagated correctly, since the propagation follows the definitions of the workflow constraints. Also, tasks can only be activated via this method, so we know that whenever a task is activated at any point, all the constraints will be propagated.

Complexity

It is apparent that we can only activate each task once (we check whether the task is already activated in the very start of the method). Activating one task involves changing both logical domains and temporal distances. The former can be done in constant time, while the latter (IFPC algorithm) has worst case complexity of $O(n^2)$, where n is the number of tasks in the workflow.

There can be up to $O(n)$ children within a decomposed task, making the time complexity of a single call to ActivateTask $O(n^3)$ due to having to propagate some kind of temporal constraint to every child. However, in the worst case we have to account for the fact that the method can also call itself recursively.

To determine the complexity of all the calls, we can observe that we can call both `ActivateTask` and `DeactivateTask` at most once per task. Calling the same method twice does nothing, since both check whether the task is already active / inactive when they start. Calling both the methods (in any order) on the same task results in a conflict (a task cannot be both active and inactive).

All in all, we have $O(n)$ calls in the worst case, each one having time complexity $O(n^3)$, which makes the worst-case complexity of the method $O(n^4)$.

The DeactivateTask method

Deactivating a task is much simpler than activating one. All we have to do is set a task's *LogicalDomain* to {false} and make sure that this didn't create a logical

conflict.. We can also do some forward checking to improve performance, but this is not necessary, since *ActivateTask* will eventually detect any conflicts due to the way it traverses the workflow.

```
FUNCTION DeactivateTask(Task T)

IF(LogicalDomain(T)={1}) RETURN FALSE
LogicalDomain(T)={0}

RETURN TRUE
```

Figure 25: The DeactivateTask method pseudocode

Figure 25 shows the pseudocode for the DeactivateTask method, described above.

Propagating constraint type 6.c

The last code fragment we present here briefly describes one way of propagating constraint type 6.c. This should be done when all the other constraints have been propagated, for example right before we mark a new process.

```
FUNCTION FixParallel(Task T)

IF(LogicalDomain(T)={0})
       RETURN true

IF(T ∈ WithActivity)
       RETURN true

//task is decomposed
FOREACH(Task C ∈ Children(T))
       IF(NOT FixParallel(C)) RETURN false

IF(task ∈ Parallel)
       Task earliest = the child C of T with the minimal Start(C)
       Task latest = the child C of T with maximal End(C)
       Dist(Start(T), Start(earliest)) = [0,0]
       Dist(End(latest), End(T)) = [0,0]

RETURN true if no constraint conflict was detected, false otherwise
```

Figure 26: The FixParallel method pseudocode

The only responsibility of the method in Figure 26 is to propagate constraint type 6.c. It adds synchronization constraints to a parallel task so that it starts exactly when the first of its children starts and ends exactly when the last of its children ends.

We couldn't propagate this constraint in the `ActivateTask` method like all the other constraints, because we didn't know which child will start first or end last. This method should be called when all the other constraints have been propagated, so nothing else can change in this sense.

The process is simple, if the task is not active or contains an activity, we do not have to do anything. If the task is decomposed, we first fix all the children.

Once the children are fixed, all that is left to do is to fix the parent in case it is a parallel task. If so, we synchronize the start of the parent with the start of its first child and the end of the parent with the end of its last child.

To see that this approach is correct, note that whenever we try to add the synchronization constraints of type 6.c, all the other constraints affecting the children of the task have already been propagated. In fact, all the constraints except those of type 6.c have been propagated before this method even started.

Constraints of type 6.c are propagated by this method and we call it on the children before we call it on the parent task. Therefore by the time we try to propagate the synchronizations in the parent, we already have all the temporal information about the children – we know which one starts first and which one ends last.

We can also see that we propagate the 6.c constraints for every parallel task in the workflow, due to the way we traverse all the tasks – we call the method on the root task an move through the workflow in a DFS manner.

Core algorithm complexity

Since the problem is NP-complete, it is not surprising that the algorithm is exponential with respect to the number of tasks in the workflow in the worst case. This is due to the fact that the algorithm explicitly examines all the possible alternative combinations – the number of those grows exponentially with the number of alternative tasks.

Even if there are no alternative tasks in the workflow, the algorithm needs at least linear time to traverse the workflow, detect whether there are any custom links and if so, propagate the activation of the root task.

That is why the resulting complexity of the core verification algorithm is $O(2^a + n)$, where a is the number of alternative tasks in the workflow and n is the total number of tasks in the workflow.

Core algorithm correctness

The correctness of the algorithm comes from the correctness of the general approach to DTP solving that our algorithm is based on and from the fact that we managed to:

a) We examine every possible alternative combination and

b) We propagate all the constraints correctly.

Point a) holds due to the definition of `ActivateTask` and `IterateAlternative` methods. We essentially traverse the workflow in a DFS-like manner (in the `ActivateTask` method) and explicitly examine all the children of any alternative tasks that we find (in the `IterateAlternative` method).

To show that part b) holds, we can note that all the workflow constraints except for type 6.c are propagated in the `ActivateTask` method and constraints of type 6.c are propagated separately once we know we have enough information to do so. We have showed that all of these constraints are propagated correctly in the definition of those methods.

All together, we know we examine every possible alternative in the workflow and we propagate all the constraints correctly, therefore the algorithm must be correct.

7.5 Simplifying the workflow

Now that we have the basic algorithm defined, we would like to improve on its performance. We already showed that if a task contains no custom links (or empty tasks), it must be valid. Following that idea, we would like to determine which parts of the workflow do not have to be examined thoroughly due to the absence of any custom links.

Notation:

- Expanded $\subseteq Tasks$ – A set of tasks that need to be explicitly examined during the verification process. Alternatively, if a task is not in *Expanded*, we do not have to examine alternatives within its subtree.

In other words, we are going to determine which tasks in the workflow have to be examined thoroughly, that is in which tasks we actually have to examine all of the alternative combinations in order to get the correct results. We will call these *expanded* tasks.

On the other hand, we will be able to determine that some tasks cannot contain any inaccessible alternatives and therefore they do not need to be examined thoroughly. These will be referred to as *collapsed* tasks. The whole idea of simplifying the workflow now comes down to determining which tasks have to be expanded and which tasks can be collapsed.

Intuitively, we can collapse tasks that aren't affected by any custom constraint. In terms of the verification algorithm, custom constraints can affect logical domains and temporal distances only. Let us define the following for a task *T*:

- *Descendants(T)* = All the direct or indirect children of T[1].
- TimePoints(T) $\subseteq TimePoints, TimePoints(T) = \{Start(t) | \ t \in T.Descendants\} \cup \{End(t) | \ t \in T.Descendants\} \cup \{Start(T), End(T)\}$ – The subset of all time points that is defined by the subtree of T.

We claim that a task can be collapsed if:

a) $\forall t \in Descendants(T): t$ is not incident with any custom logical link.

b) $\forall p_1, p_2 \in TimePoints(T): Dist(p_1, p_2)$ is not restricted by any custom constraint.

The above conditions make sure that any custom links in the workflow will not affect either logical domains or temporal distances of time points within collapsed tasks, which leads to the fact that collapsed tasks cannot contain any inaccessible alternatives and therefore need not be explicitly evaluated by the verification procedure.

---

[1] Formally:

$$D_1(T) = Children(T)$$

$$D_i(T) = \bigcup_{t \in D_{i-1} \cap Decomposed} Children(t)$$

$$Descendants(T) = \bigcup_{i=1..\infty} D_i$$

Condition a) implies that any custom logical links may only affect the root of any collapsed subtree, not its descendants. This means that the only way that the rest of the workflow can affect the logical domains of *Descendants(T)* is through the logical domain of T (and propagation of constraint types 1 or 2).

In other words, custom logical constraints can cause the whole collapsed subtree to activate or deactivate, but this cannot cause any of T's descendants to become an inaccessible alternative, due to the way general constraints are defined.

Condition b) works analogically for temporal constraints. It requires that any custom temporal constraints (precedence, synchronization) cannot restrict the temporal distances of time points in *TimePoints(T)* in any way. This means that the custom temporal constraints do not affect the collapsed tasks at all. Specifically, they cannot cause an inaccessible alternative within a collapsed task.

To summarize, if the above necessary conditions hold, it is easy to see that there can be **no inaccessible alternatives in any collapsed task.** We know that if a task is collapsed, it contains no custom links or empty tasks, so on its own it must be a valid workflow with all alternatives accessible (following the reasoning from the Verifying workflows without custom links chapter).

Furthermore, since the logical domains and temporal distances of all the collapsed tasks are not restricted by the rest of the workflow, all its alternatives must still be accessible even in the context of the whole workflow, provided that there is no problem in the expanded parts (which we still verify explicitly).

Notice that condition b) is somewhat stronger and more difficult to ensure than a). That is because in our particular workflow model, logical constraints propagation is rather easy while temporal constraints propagation is more complicated and we have to be more careful while collapsing tasks to make sure we don't lose any information that we need to produce correct results.

Modified algorithm

Once we have determined which tasks have to be expanded and which tasks can be collapsed, we propose several changes to the verification algorithm:

In the `ActivateTask` method we check whether the task we are trying to activate must be expanded. If not, we just set the distance between its start and end points to some constant value, which corresponds to treating the collapsed tasks as „black boxes" with an arbitrary positive duration:

```
FUNCTION ActivateTask(Task T):

IF(LogicalDomain(T) = {1}) RETURN TRUE

FOREACH(t ∈ {Parent(T), Parent(Parent(T)),… RootTask})                    (3)
       ActivateTask(t)

IF(LogicalDomain(T) = {1}) RETURN TRUE
ELSE LogicalDomain(T) = {1}

                                                                          (2)
IF(Parent(T) ∈ Alternative)
       FOREACH(Task t ∈ Children(Parent(T)), t != T)
              DeactivateTask(t)
                                                                          (6.b)
       Dist(Start(Parent(T)), Start(T)) = [0,0]                           (6.b)
       Dist(End(T), End(Parent(T)) = [0,0]

IF(T ∈ Expanded)                                                          (5)
       IF(T ∈ WithActivity)
              Dist(Start(T), End(T)) =
                   [Duration(Activity(T)), Duration(Activity(T))]         (6.a)

       IF(T ∈ Serial, Children(T) = (c1,c2,… cn))
                     Dist(Start(T), Start(c1)) = [0,0]                    (4)
              Dist(End(cn), End(T))) = [0,0]

              FOR(i = 1 … n - 1) Dist(ci, ci+1) = [0,∞]                   (6.c)

       IF(T ∈ Parallel)
              FOREACH(Task C ∈ Children(T))
                     Dist(Start(T), Start(C)) = [0,∞]
                     Dist(End(C), End(T)) = [0,∞]

                                                                          (1)
       IF(T ∈ Alternative) TODO = TODO ∪ {T}

       IF(T ∈ Serial ∪ Parallel)

              FOREACH(Task C ∈ Children(T))                               (7-14)
                     ActivateTask(C)

ELSE Dist(Start(T), End(T)) = [1,1]

FOREACH(newly active custom constraint C)
       PropagateConstraint(C)

RETURN true if there are no constraint conflicts, false otherwise
```

Figure 27: The modified ActivateTask method pseudocode

Figure 27 describes the modified `ActivateTask` method that doesn't fully evaluate the collapsed tasks (modified parts are highlighted). Note that even if the task is collapsed, we still need to represent it in the workflow somehow – if we just deleted the collapsed tasks, the workflow could easily become invalid (for instance, we could delete all the accessible alternatives of an alternative task).

As for the duration of the collapsed tasks, since we know that there are no temporal constraints restricting them, we can set it to any value we want, it will not create a constraint conflict.

Also, when filling the *Inaccessible* collection, we only consider the alternatives in the expanded tasks as we will no longer explicitly enumerate alternatives within collapsed tasks.

This modification should increase performance considerably, as we will only explicitly evaluate the expanded tasks. This can easily be a mere fraction of the original workflow, depending on how many custom links were used. The worst-case complexity remains unchanged of course.

To show the correctness of this approach, we need show that both verification algorithms (the original and the modified) detect the exact same inaccessible alternatives.

First, we can note that both the algorithms work identically on expanded tasks. Collapsing some tasks cannot change this, since we only collapse tasks that aren't affected by any custom link, meaning both the algorithms propagate the exact same custom constraints in the same tasks.

Second, we know that the collapsed tasks cannot ever contain an inaccessible alternative. In other words, any inaccessible alternatives have to be in expanded tasks, where both the algorithms behave identically and thus yield identical results.

Simple way of collapsing tasks

As a simple way of improving the performance of the verification algorithm, we compute *Expanded* as follows:

$$Expanded = \emptyset$$

FOREACH $(C \in Constraints)$

$$Expanded = Expanded \cup \{Parent(C), Parent(Parent(C)), \dots, RootTask\} \cup Descendants(Parent(C))$$

Figure 28: Simple way of collapsing tasks

Figure 28 shows a basic way of expanding tasks that simply makes sure that every task that contains a custom constraint (directly or indirectly) is expanded. We go through every custom constraint defined and expand both the path from the root task to the link's parent and the parent itself together with all of its children (recursively).

Below is an example of how a workflow with a single precedence link L would look like after applying this simple procedure.

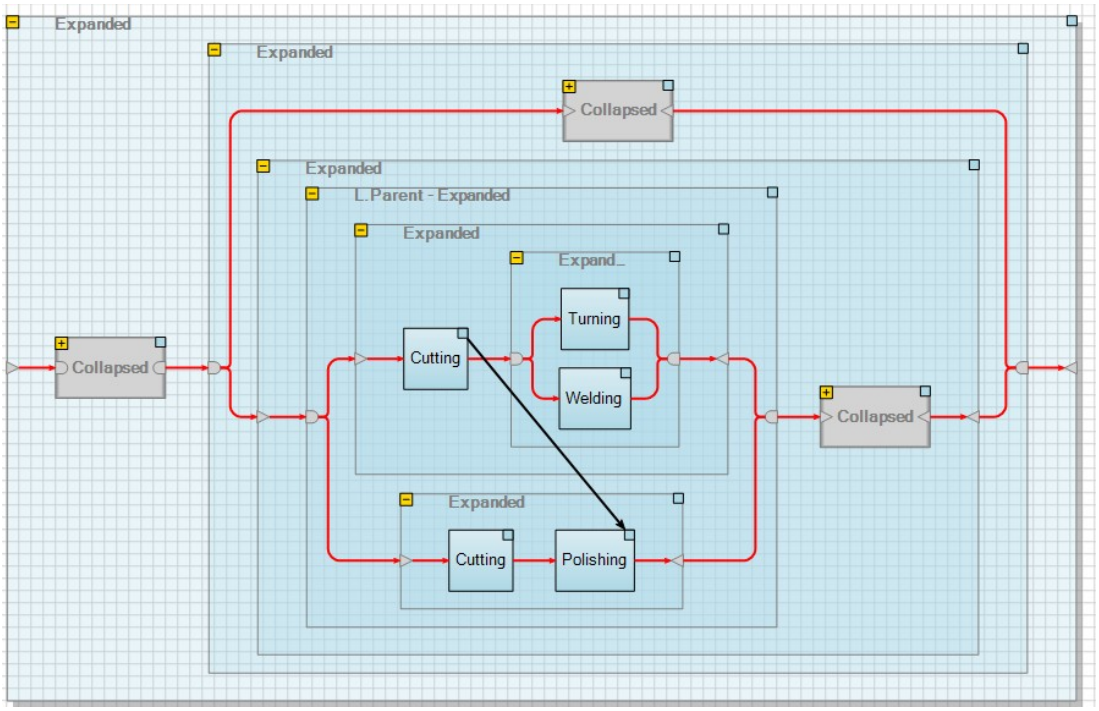


Figure 29: Example of task collapsing

Figure 29 shows an example of the simple collapsing procedure described in Figure 28. The tasks that are not in *Expanded* after the procedure finishes are shown collapsed in Figure 29 for illustration.

To show the correctness of this procedure, we have to show that we only collapse tasks that cannot contain an inaccessible alternative, that is tasks that aren't affected by any custom link.

Suppose we have a precedence or synchronization custom constraint *C*. We know that this constraint only affects temporal distances between tasks. Since both the tasks it connects are contained within the constraint link's parent, we know that the link cannot directly influence any other tasks beyond those within the subtree of *Parent(C)*, due to the tree structure of the workflow and the definition of the workflow constraints.

The only way that the constraint affects the rest of the workflow beyond the subtree of *Parent*(C) is by changing *Dist(Start(Parent(C)), End(Parent(C)))*. This doesn't restrict temporal distances between any other time points that aren't in the subtree of *Parent*(C) though.

For a logical constraint, the situation is similar, except instead of temporal distances we have logical domains. Using the same reasoning as before, a custom logical constraint can only affect the rest of the workflow beyond *Parent(C)* by changing *the* logical domain of *Parent(C)*. The only way that this can affect other task's logical domains is through propagation of constraint type 4. This doesn't affect the tasks' descendants though, so both the necessary conditions we defined for collapsing tasks hold and the new algorithm provides equal results as the old one.

Further optimizations

The above way of collapsing tasks is by no means optimal. We can certainly collapse more tasks based on some further observations.

For instance, it isn't hard to see that we do not need to fully expand tasks containing custom logical links. Based on condition a) for collapsing tasks, we can expand much fewer tasks, as shown in the following method:

$$Expanded = \emptyset$$

FOREACH $(C \in Constraints \setminus Logical)$

$$Expanded = Expanded \cup \{Parent(C), Parent(Parent(C)), \dots, RootTask\} \cup Descendants(Parent(C))$$

FOREACH $(C \in Logical)$

$$Expanded = Expanded \cup \{Parent(From(C)), Parent(Parent(From(C))), \dots, RootTask\} \cup \{Parent(To(C)), Parent(Parent(To(C))), \dots, RootTask\}$$

Figure 30: Expanding tasks with custom logical links

Figure 30 shows a procedure that determines which tasks have to be expanded based on all the custom logical links in the workflow. The other custom links (precedence and synchronization) still use the procedure in Figure 28.

This procedure will only expand tasks on the paths *(From(C), RootTask]* and *(To(C), RootTask]*. Notice that this is the bare minimum, if we expand anything less, we lose the information about which tasks the custom link connects.

This still satisfies both the necessary conditions for collapsing a task that we established, since the procedure for temporal constraints remains unchanged and condition a) obviously holds.

The situation with custom temporal constraints (precedences and synchronizations) is more complicated, since propagating temporal constraints is not as simple as propagating logical constraints. We certainly cannot use the same procedure as above, as illustrated by the simple contra-example below.

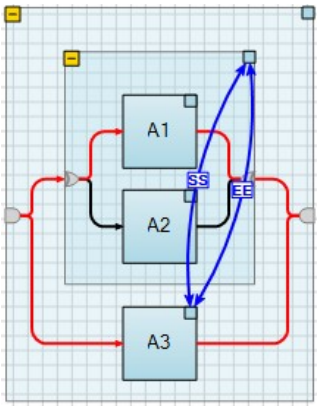


Figure 31: Problem with synchronizations

Figure 31 shows a simple workflow where the user requires that two tasks start and end at the same time (in other words, that they have identical durations). In order to verify this workflow, we need to fully expand both tasks incident with one of the links, otherwise we do not know their durations.

In general, temporal constraint propagation is much less transparent and straightforward than logical links propagation. For this reason, the verification procedure currently uses no further optimization for temporal constraints.

7.6 Verification from the user's perspective

Despite the underlying complex processes, we tried to make verification as simple as possible for the user. The user only has to invoke the verification dialog and run the process. When the verification ends, the result is displayed next to the workflow.

There are three possible results (see The Verify method for details). The workflow can either be valid (all alternatives accessible), or invalid with either some or all alternatives inaccessible (the former means that at least one process exists for the workflow, while the latter means that there is no process at all). The results are simply visualized as green, orange and red "tick" respectively.

If any problems were detected, the editor displays them in an interactive list, which can be used to navigate to the particular problem (the view automatically moves over the relevant object) and to get a simple description of the problem.

Specifically, all the inaccessible alternatives will be in this list. In some cases, this list will even contain the custom links that caused some inaccessible alternative, but that is not always possible (due to the way constraints are propagated).

The results of the verification process are saved together with the workflow, so they can be reviewed later.

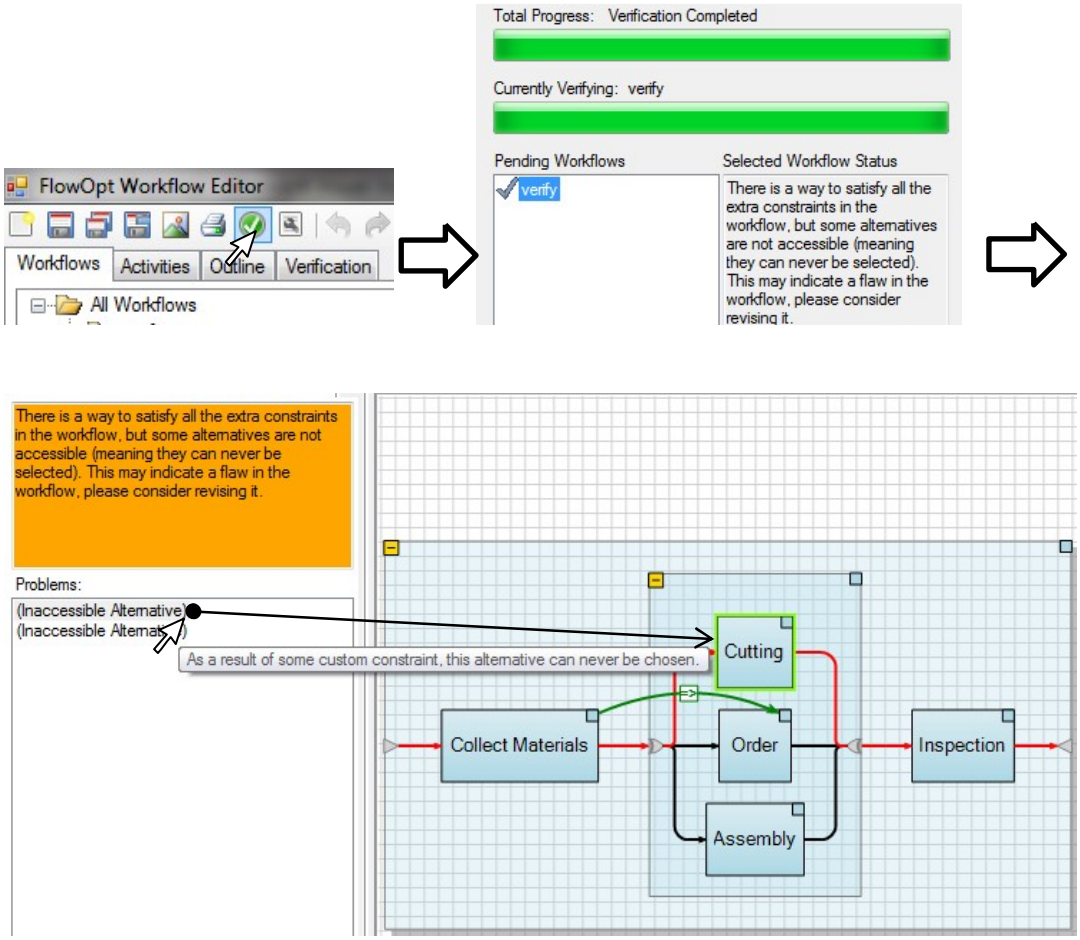


Figure 32: Verification example

In Figure 32, the user created an implication custom link from a task that will always be selected to a child of an alternative task, making the other children inaccessible. The verification procedure detects this and lets the user know where the problem is.

The workflow editor also allows the user to verify a specific way of choosing alternatives in the workflow. Specifically, it is possible to have the verification algorithm always choose the preferred alternative (the task on the preferred route) as the active child in any alternative task.

Due to the fact that the verification procedure only evaluates a single combination of alternatives, the process becomes polynomial. Also, it can only yield one of two results – workflow is fully verified, or workflow is invalid (as soon as there is an inaccessible alternative, the workflow becomes invalid, since we cannot choose another alternative instead).

This partial verification may be useful if the user is only interested in a particular way that the process can be executed.

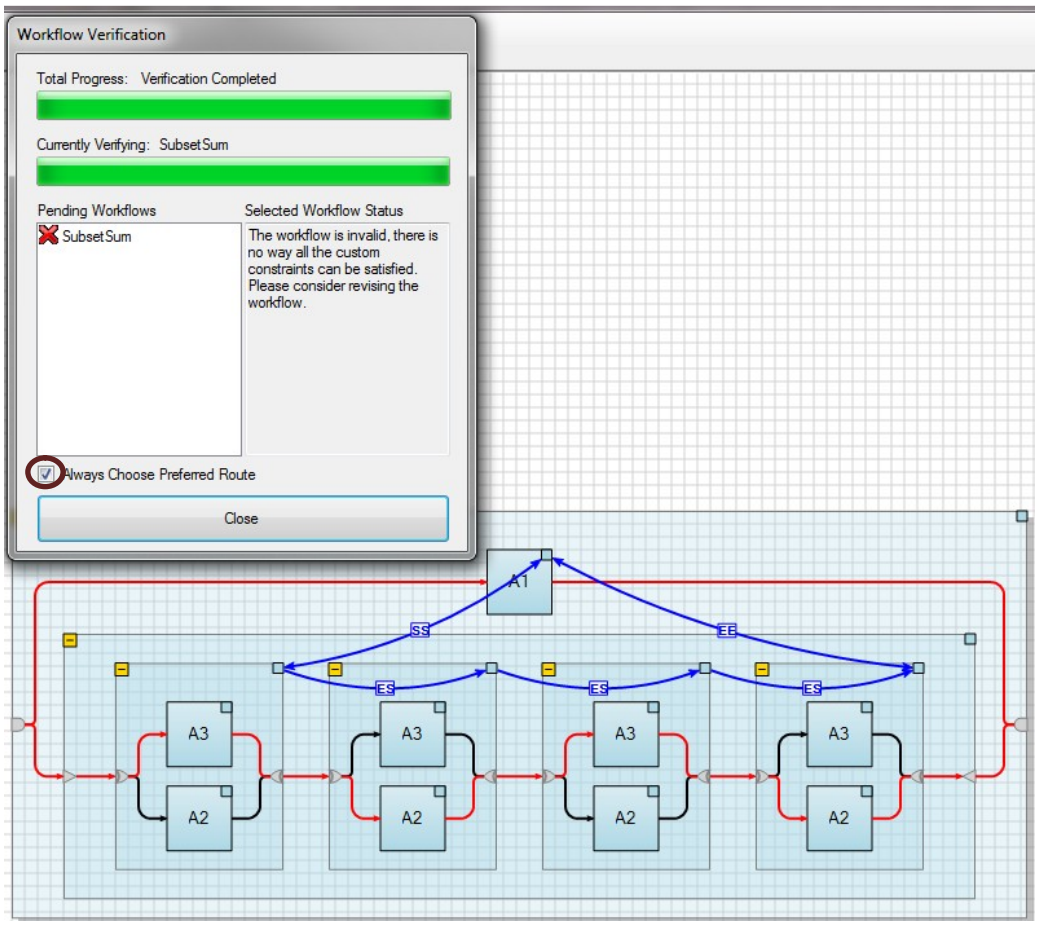


Figure 33: Partial verification

Figure 33 shows a workflow that was verified using only the preferred alternatives (preferred route is painted red). Activities A1, A2 and A3 have durations equal to 4, 0 and 1 respectively.

The partial verification procedure marks the above workflow as invalid for this particular choice of alternatives. Although there is a way to satisfy all the constraints (namely by performing A3 in all the alternative tasks), partial verification only considers the alternatives on the preferred route, and those do not satisfy the constraints (A1 has duration of 4, the alternative tasks have total duration of 2 and the synchronization constraints require those to be equal).

8. Import / Export of the Workflows

FlowOpt workflow editor implements both import and export of workflows to (currently) two formats:

1) MAKE workflows
2) XPDL 2.1

The need to support MAKE workflows comes from the fact that one of the goals of the FlowOpt workflow editor is full integration into the MAKE application. Since MAKE uses a different workflow model than FlowOpt, it is very convenient to have a way of converting one into the other, otherwise FlowOpt workflows couldn't take advantage of some of MAKE's advanced features and vice versa.

XPDL is supported because it is a well-known, standardized format used and understood by many users and applications, so being able to import / export to and from XPDL greatly adds to FlowOpt workflow editor's usability.

8.1 MAKE Import / Export

The workflow model used in MAKE is similar to that of FlowOpt, but it lacks the tree hierarchy that is characteristic for FlowOpt workflows. Essentially, MAKE workflow model is a general temporal network with alternatives, as defined in [8].

There are several kinds of objects in MAKE workflows:

- Activities – elementary units of work just like in FlowOpt. Correspond to nodes in a Simple TNA.
- (Temporal) links – define the order of execution through the precedence relation (again, same as in FlowOpt). Correspond to temporal links in Simple TNA.
- Start and end event marking the start and end of the workflow respectively. There is exactly one start and exactly one end event in a MAKE workflow.
- Activity decorators – any activity can have a parallel or alternative split decorator and a parallel or alternative join decorator. These mark the parallel / alternative subgraphs in Simple TNA terminology.
- Semaphore – a special activity that doesn't correspond to any real work, but is only meant to carry some decorator. Route activities in XPDL or gateways in BPMN are a similar concept.

All things considered, MAKE workflows are relatively close to FlowOpt workflows, which is not surprising considering the fact that FlowOpt workflow editor was greatly influenced by its counterpart in MAKE. Some workflow objects are even shared between the two models (activities for instance), which also helps to simplify the implementation part of the import process.

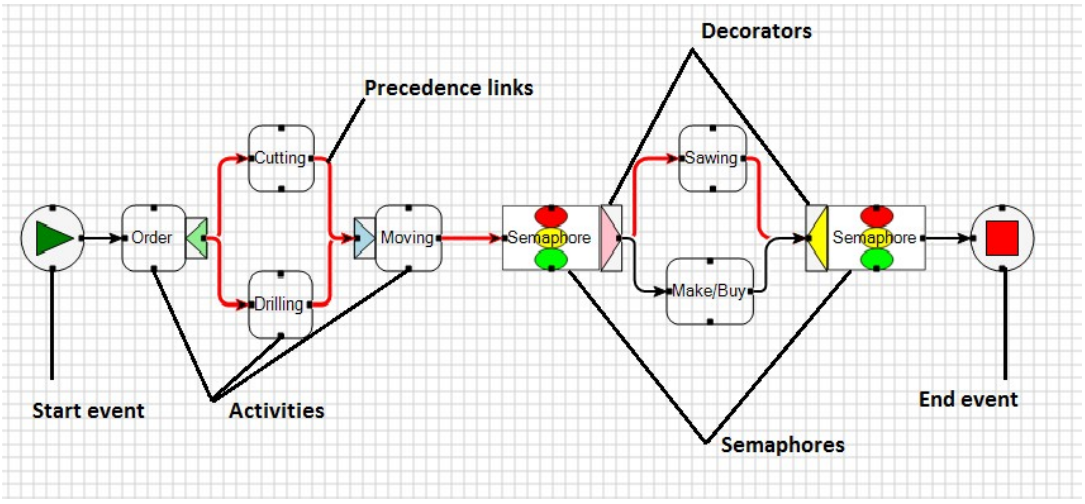


Figure 34: MAKE workflow example

Figure 34 shows a MAKE workflow together with names of the objects in it.

There are several restrictions on the degrees of nodes (activities) in a MAKE workflow that the import and export procedures take advantage of:

- An activity with no decorators must have input degree = output degree = 1. In other words, every activity with no decorators must have exactly one incoming and exactly one outgoing link.
- A start event must have exactly one outgoing link and no incoming links.
- An end event can have no outgoing links.
- An activity decorated with a split decorator must have at least two outgoing links.
- An activity decorated with a join decorator must have at least two incoming links.

Export into MAKE

Export into MAKE is rather straightforward – we can take advantage of the tree structure of FlowOpt workflows and define the procedure of exporting a task recursively. Exporting a workflow of course means exporting its root task. It also involves setting some properties that both workflow models have in common to match, but this isn't semantically important.

We will describe the export procedure in detail below. We will use the following notation:

- $For\ t \in Tasks$: *Exported(t)* = the fragment of the exported MAKE workflow that corresponds to *t* (subgraph that was created by exporting *t*).
- $For\ t \in Tasks$: *First(t), Last(t)* = first and last node of the exported subgraph corresponding to *t*. The export procedure uses these to connect the exported links correctly.

Pseudocode for the procedure that exports a task into MAKE is given below.

```
Procedure ExportTask(Task T)

IF(T ∈ WithActivity)
       Export Activity(T)
       First(T) = Last(T) = Exported(T)

IF (T ∈ Serial, Children(T) = (c1,c2,…,cn))
       ExportTask(ci) for all i = 1..n
       FOR (i = 1..n-1)
              Add a link between Last(ci) and First(ci+1)
       First(T) = First(c1)
       Last(T) = Last(cn)

IF(T∈ Parallel ∪ Alternative)
       Create a semaphore S with a parallel / alternative split
       Create a semaphore J with a parallel / alternative join
       FOREACH(Task c ∈ Children(T))
              ExportTask(c)
              Create a link between S and First(c)
              Create a link between Last(c) and J
       First(T) = S
       Last(T) = J
```

Figure 35: Procedure for exporting FlowOpt workflows into MAKE

Figure 35 shows the pseudocode of the procedure that can export a task into the MAKE workflow model.

In other words, if we want to export a task *T* into MAKE, then depending on the type of the task we do the following:

- If the task contains an activity, we don't have to do anything, just export the activity. Note that MAKE and FlowOpt actually use the exact same activity definitions, so this is trivial.
- If the task is serial, export all of its children and connect them with precedence links in order given by the serial task.
- If the task is parallel or alternative, we need to do several things:
    a. Export all of its children
    b. Create a semaphore holding a parallel / alternative split depending on the type of decomposition.
    c. Create a semaphore holding a parallel / alternative join depending on the type of the decomposition.
    d. Create a link from the split semaphore into every exported child and from every exported child into the join semaphore.

In the end, we also have to add the start and end events and connect it to the exported root task to create a valid MAKE workflow.

We do not define an export procedure for incomplete workflows, that is workflows that contain an empty task. We could easily modify the above procedure to change this, for instance by exporting empty tasks as some special activities, but that did not seem necessary.

Unfortunately the export procedure ignores custom links, since the MAKE workflow model does not explicitly support them.

First, MAKE only defines temporal links, not logical links. Second, the restrictions on node input and output degrees listed above complicate things further. For

instance, every non-decorated activity must have exactly one incoming and exactly one outgoing link in MAKE. It should be apparent from the export procedure above that it will never create an activity with zero incoming or outgoing links. This means that even if we managed to export the custom link somehow, we couldn't connect it to the exported activity without decorating it.

When we decorate an activity, we can connect it to multiple links, however decorating an activity carries some additional logical constraints – a parallel decorator implies that all of the nodes connected to it should be executed in parallel, whereas an alternative decorator implies that exactly one node of those connected to it should be executed.

This makes it very difficult to find an intuitive way of exporting the custom links that wouldn't violate MAKE workflow model and still be intuitive for the user. That is why we opted not to export the custom links at all. If they prove to be a useful construct, it should be rather easy to add them to MAKE workflow model, which would in turn make it easy to export them in a much cleaner way.

All the other information besides the custom links is exported – aspects like layout, preferred route or activity to resource mapping are preserved. The way this is implemented is mostly technical and not very interesting conceptually.

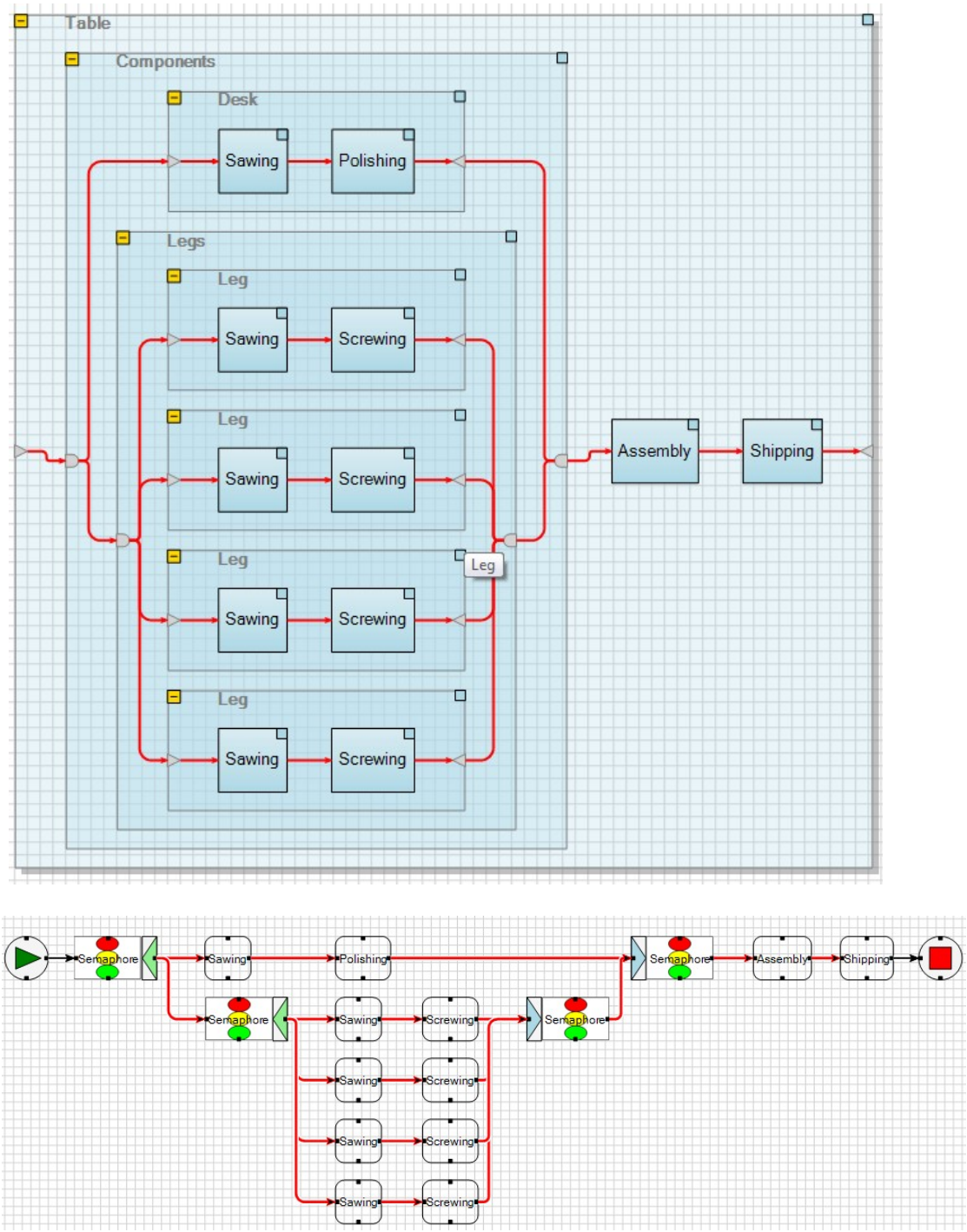


Figure 36: A workflow exported into MAKE

Figure 36 shows a FlowOpt workflow and its equivalent exported into MAKE.

Import from MAKE

Import is the more interesting problem, because MAKE workflow model does not define hierarchy and we therefore have to build it somehow. Since MAKE workflows are essentially Simple TNAs, we can use the algorithm described in [1] that manages to convert a Simple TNA into a Nested TNA (~FlowOpt workflow model) in polynomial time, provided that the Simple TNA has a nested structure.

Let us now briefly describe the procedure currently used in the presented application to import workflows from other formats (specifically MAKE and XPDL) into FlowOpt.

It is based on the algorithm presented in [1], but we made some slight modifications to reflect the details in which our model differs from Nested TNA – mainly the fact that we decompose nodes instead of links. Also, the algorithm currently used in the presented application automatically filters out semaphores / gateways, transferring the information they carry to the type of the created task.

Below is the pseudo code of the original algorithm presented in [1]. We use the same notation as in [1], that is:

- The workflow being imported is represented as a graph G whose nodes are the activities and arcs are the temporal links.
- $\forall node\ x: pred(x) = \{y: (y,x)\ is\ a\ link\}$ - predecessors of x in the graph
- $\forall node\ x: succ(x) = \{y: (x,y)\ is\ a\ link\}$ - successors of x in the graph

```
algorithm DetectNested(input: graph G, output: {success, failure})
1. select all nodes x in G such that |pred(x)| = |succ(x)| = 1
2. sort the selected nodes lexicographically according to index
       (pred(x), succ(x)) to form a queue Q
3. while non-empty Q do
4.     select and delete a sub-sequence L of size k in Q such that
              all nodes in L have an identical index ({x}, {y}) and
              either |succ(x)| = k or |pred(y)| = k
5.     if no such L exists then stop with failure
6.     if k > 1 & outLab(x) ≠ inLab(y) then stop with failure
7.     remove nodes z∈L from the graph
8.     remove nodes x, y from Q (if they are there)
9.     add arc (x,y) to the graph (an update succ(x) and pred(y))
10.    if |pred(x)| = |succ(x)| = 1 then insert x to Q
11.    if |pred(y)| = |succ(y)| = 1 then insert y to Q
12.end while
13.if the graph consists of two nodes connected by an arc then
14.    stop with success
15.else stop with failure
```

Figure 37: The original algorithm for recognizing Nested TNA as presented in [1]

Figure 37 shows pseudocode for the `DetectNested` procedure that can detect whether a given Simple TNA is a Nested TNA and if so, it can be used create the nested structure. The importing procedure for MAKE (and for XPDL as well) is based on this procedure.

The modified algorithm that is used in the presented application is shown below.

```
algorithm ImportTNA(input: graph G, output: {success, failure})
1. select all nodes x in G such that |pred(x)| = |succ(x)| = 1
2. sort the selected nodes lexicographically according to index
       (pred(x), succ(x)) to form a queue Q
3. while non-empty Q do
4.     select and delete a sub-sequence L of size k >= 2 in Q such that
              all nodes in L have an identical index ({x}, {y}) and
              either |succ(x)| = k or |pred(y)| = k
5.     if no such sequence exists then
              select and delete a sub-sequence L = (x = c1, c2, …, y = ck)
              of size k >= 2 in Q such that ∀i = 1..k − 1: (ci, ci+1) is an arc in G
              if no such sequence exists either then stop with failure
6.     else if outLab(x) does not match inLab(y) then stop with failure
7.     remove nodes z∈L from the graph
8.     remove nodes x, y from Q (if they are there)
9.     add a new node d and arcs (x,d) and (d,y) to the graph
       Create a new task from L and associate it with d
10.    if |pred(x)| = |succ(x)| = 1 then insert x to Q
11.    if |pred(y)| = |succ(y)| = 1 then insert y to Q
       if |pred(n)| = |succ(n)| = 1 then insert n to Q
12.end while
13.if the graph consists of a single node then
14.    stop with success
15.else stop with failure
```

Figure 38: Modified importing procedure

Figure 38 describes the procedure that is used by the presented application to import MAKE (and XPDL) workflows. It is almost identical to the original algorithm in Figure 37, but there are some modifications.

The algorithm starts in the same way – we initialize the queue of nodes that have a single successor and a single predecessor and sort it according to the same index. Then we repeatedly try to detect a nest and perform a contraction (an opposite of decomposition – see [1]).

First modification is in the way we detect nests. The original algorithm does not distinguish serial nests, it treats them as a series of parallel nests of size 1. While this is semantically correct, it is better to try to detect the serial nests separately if we want to visualize the workflow.

We first try to detect a parallel / alternative nest like in the original algorithm (requiring that the nest's size is at least two, since if it is one it indicates a serial nest). If we cannot find one, we look for a serial nest instead – we search the queue and select a sequence of nodes that form a serial nest. If such a sequence does not exist either, it means there are no more nests and the workflow is not Nested TNA, so we end with failure.

Another modification is on line 6 – we no longer require that the labels are equal. We relaxed this requirement a bit and we only need the labels to 'match'. This change is introduced for the sake of importing XPDL workflows and will be thoroughly described later in this document. When importing MAKE workflows, matching is effectively the same as equality. Notice that we only check the labels if we found a parallel or alternative nest, serial nests do not have to have matching labels (same as in the original algorithm).

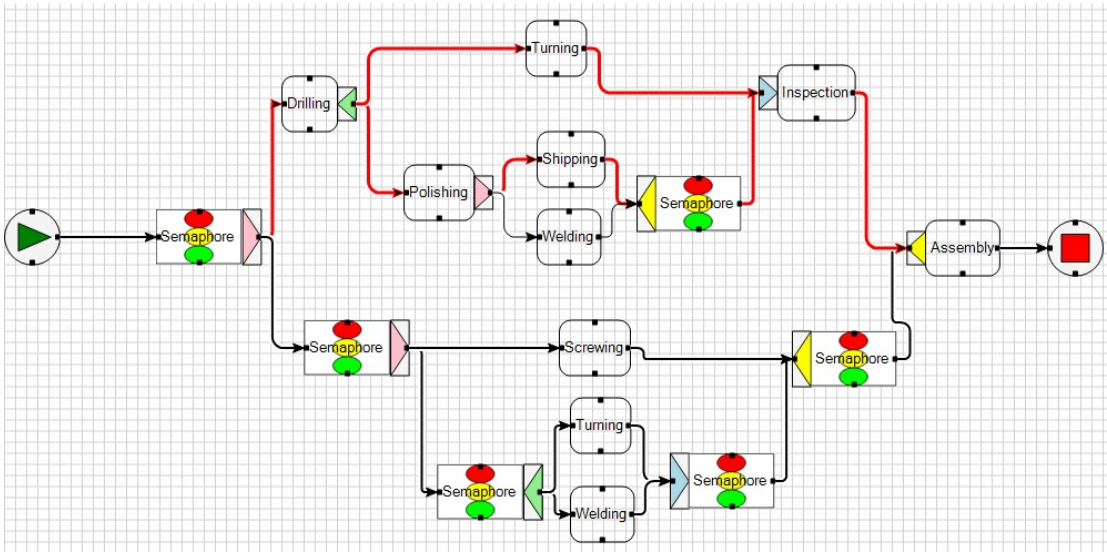
Next change is on line 9 – instead of contracting the nest into an arc like in the original algorithm, we create a new node that represents the nest (the decomposed task). That is a consequence of the fact that we decompose nodes rather than links.

We also create a new FlowOpt task from the sequence we found. The new task is either serial, if we detected a serial nest, or parallel / alternative based on the labels if we detected a parallel / alternative nest. It is created by grouping the tasks associated with the nodes within the detected nest, discarding any semaphores. We associate this new task with the new node somehow so that we build the imported workflow as the procedure progresses. Initially, every node is associated with a task carrying the activity represented by the node.

Finally, the change in line 13 is also a consequence of decomposing nodes rather than arcs – we end when there is a single node left in the graph, rather than a single arc. The imported FlowOpt workflow's root task is the task associated with this single remaining node.

The correctness of this procedure comes from correctness of the original algorithm, the proposed modifications are mostly technical, they do not change the basic principles of the algorithm.

The only information omitted by the import procedure are the exact temporal constraints on the links – MAKE allows specifying both minimal and maximal distance on any temporal link, whereas FlowOpt only has general precedences and synchronizations. For the sake of simplicity, we opted to omit the intervals and use precedences everywhere.



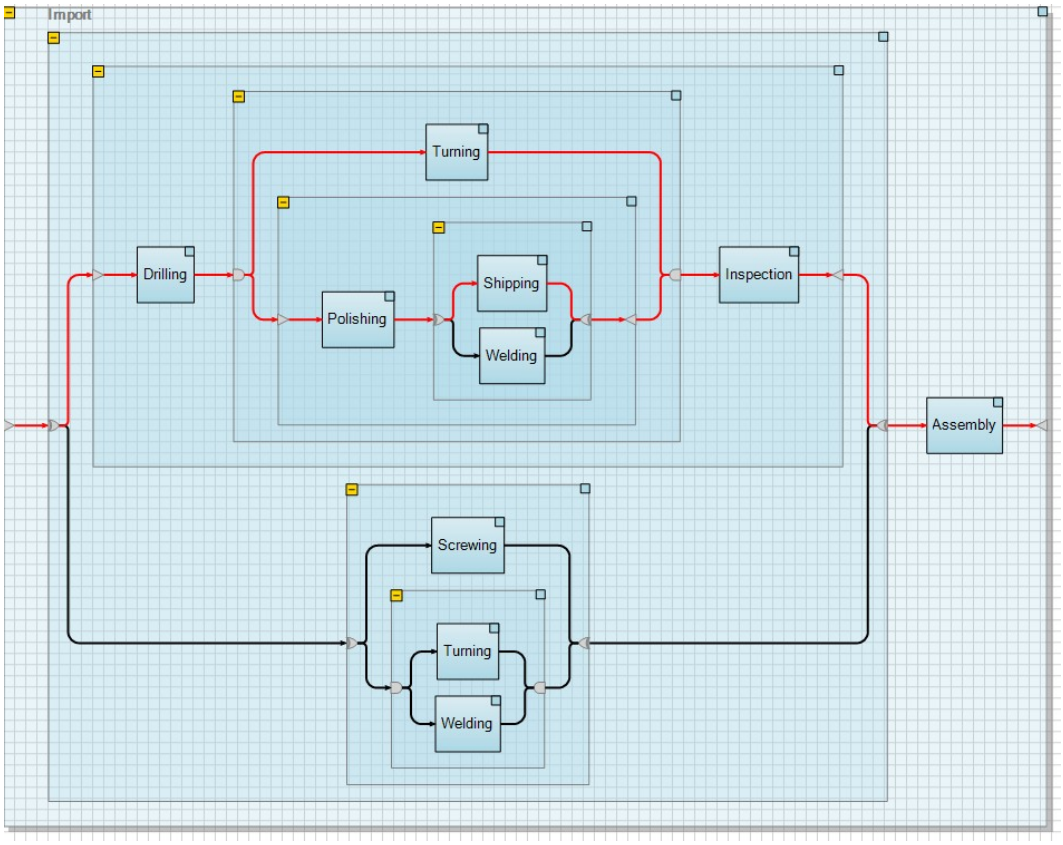


Figure 39: A workflow imported from MAKE

Figure 39 shows a MAKE workflow and its equivalent imported into FlowOpt.

8.2 XPDL Import / Export

XPDL is a widely used standard for storing BPMN diagrams that is based on XML[1]. Its full specification can be found in [9]. FlowOpt currently supports import and export to / from XPDL 2.1, which represents BPMN 1.1 workflows.

It should be noted that BPMN specification in version 2.0 defines its own XML schema for data exchange. However when we were trying to make a decision on which format to support, this standard was very young and therefore not as widely supported as XPDL. XPDL was already a proven industry standard at the time, so we elected to support it over other formats.

Export into XPDL

The key question in exporting workflows into XPDL was again the mapping of FlowOpt objects to XPDL objects. First of all, FlowOpt activities are mapped to XPDL activities (specifically to activities with no implementation). XPDL activities support the notion of resources (called performers / participants in XPDL terminology), so we can export those as well.

[1] eXtensible Markup Language, a standard format for data exchange

MAKE / FlowOpt defines three types of resources – a single resource, a resource group and a resource mode. Fortunately, all of them have their equivalents in XPDL, so we can export them with little trouble.

A single resource in MAKE is mapped to a single resource in XPDL. A resource group in MAKE is essentially a set of resources that have some common capability (operators, workers …). If an activity has a dependency on a resource group, it means we can allocate an activity on any of the resources within the resource group. This corresponds to the concept of a role in XPDL.

The third resource type in MAKE are modes, which is again a set of resources, but to satisfy a dependency on a mode, all of the resources in it have to be allocated. This corresponds to a resource set in XPDL terminology.

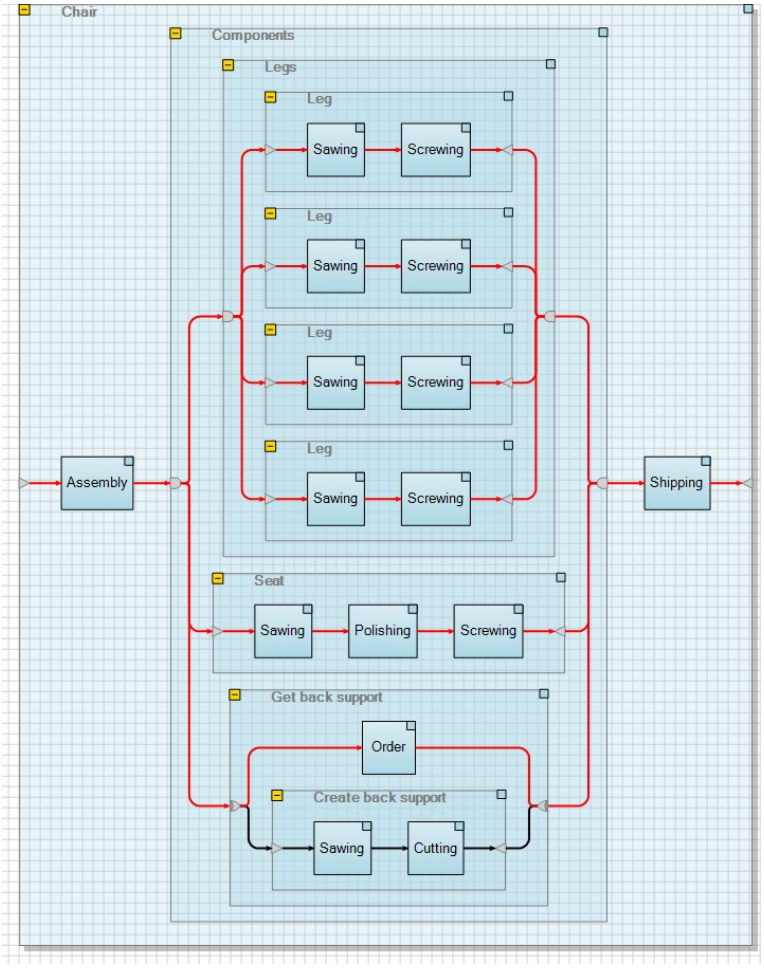
Moving on, FlowOpt precedence links are mapped on flow links in XPDL. It should be pointed out that the semantics aren't exactly the same. FlowOpt precedence links represent just a simple precedence relation, whereas XPDL flow links are defined in a way based on links in Petri nets (see the Comparison chapter). In simple nested workflows like those created by the FlowOpt workflow editor, the difference is not too significant though. Besides, XPDL only defines flow links, so there is little choice in the matter.

FlowOpt tasks are mapped on XPDL block activities / activity sets, which is a very similar concept. A block activity in XPDL is an activity that contains some independent workflow, which is called an activity set. In order to execute the block activity, the activity set has to be executed. In BPMN, this corresponds to a sub-process. FlowOpt tasks are almost identical to BPMN processes in terms of both semantics and the way they are displayed, so mapping them on each other is very intuitive for the user.

Finally, the custom links again aren't exported. There are two problems in exporting the custom links. First, XPDL doesn't define any direct equivalents of FlowOpt custom links. As stated above, flow links are the only link type supported, so any mapping would have to be done through some complex conditions on the links or through extra events simulating the original custom links, which would be rather unintuitive for the user.

The other problem is that XPDL doesn't allow any links to cross the boundaries of an activity set (a sub-process). Since custom links in FlowOpt can (and usually do) cross the boundaries of tasks, exporting custom links would mean not mapping tasks onto activity sets. Seeing as task hierarchy is the core defining feature of FlowOpt workflows, we definitely wanted it preserved in the exported document wherever possible.

In terms of conformance, the structure of FlowOpt workflows corresponds to the *FULL_BLOCKED* graph conformance class of XPDL. The BPMN model portability conformance is set to standard. FlowOpt currently produces XPDL files that are valid with respect to XPDL schema of version 2.1. Exported workflows were tested in two third party applications (Together Workflow Editor and BizAgi Process Modeller) and both displayed them correctly.


Chair
Components
Legs
Leg
Sawing
Screwing
Leg
Sawing
Screwing
Leg
Sawing
Screwing
Leg
Sawing
Screwing
Assembly
Shipping
Seat
Sawing
Polishing
Screwing
Get back support
Order
Create back support
Sawing
Cutting

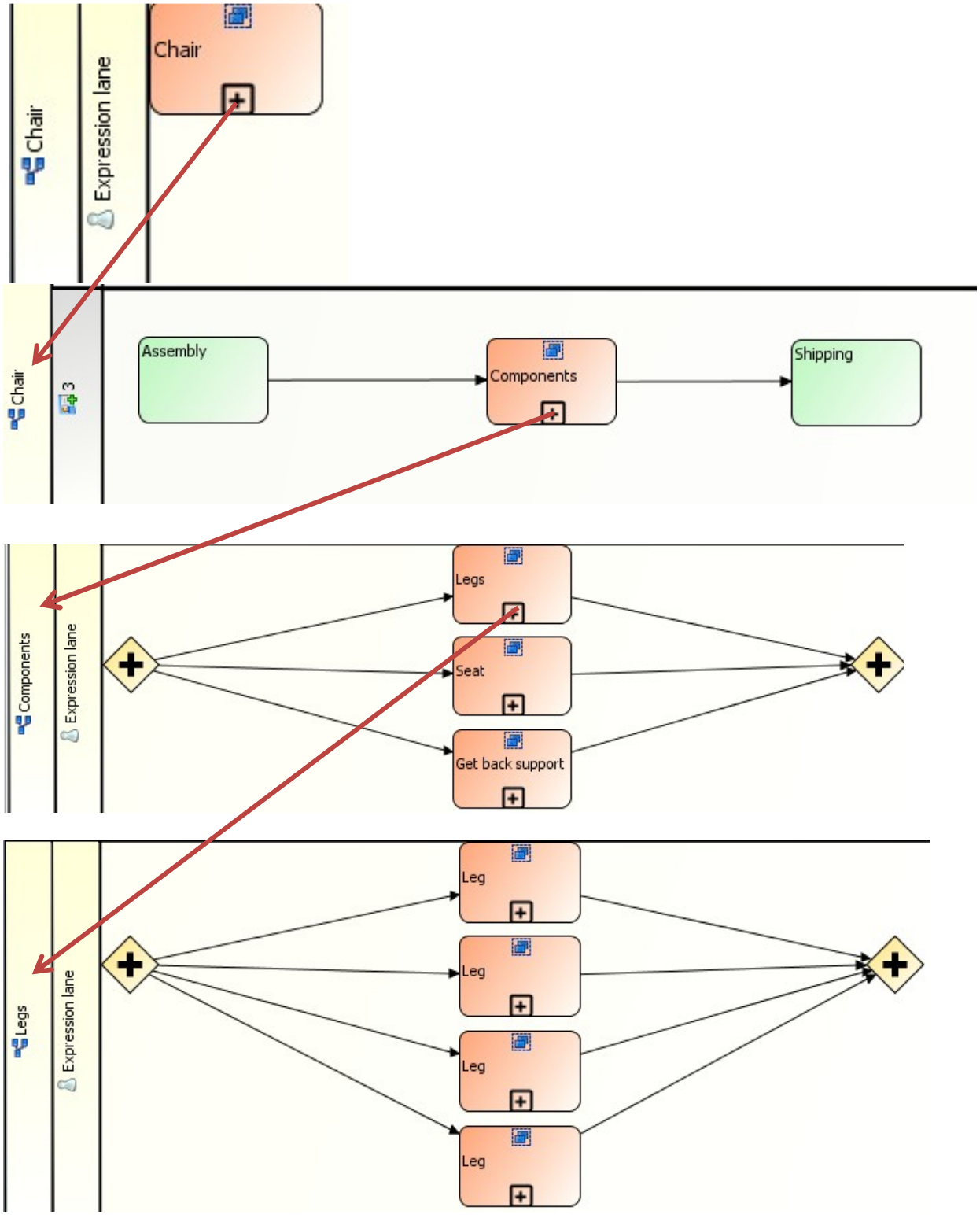


Figure 40: A workflow exported into XPDL (parts)

Figure 40 shows a FlowOpt workflow exported into XPDL. The result is displayed in the Together Workflow Editor and due to the size of the resulting diagram we only show parts of it (TWE displays subprocesses on separate screens).

Import from XPDL

Importing workflows from XPDL (BPMN) is significantly more complicated than importing them from MAKE, since the XPDL workflow model is much richer. Seeing as XPDL defines many objects that have no equivalents in the FlowOpt workflow model and the semantics of even the most basic workflow objects is somewhat different, we have to accept the fact that some information will be lost during the import procedure and some workflows simply cannot be imported.

That is why our goal wasn't to fully map the XPDL model to the FlowOpt model, but rather to create a procedure that preserves as much information as possible while still producing workflows that are visually similar to the original. That way the user will be able to import a workflow reasonably close to the original in most cases.

It should be pointed out that the exact results of the import procedure depend on how close the input XPDL file is to the XPDL 2.1 specification. Unfortunately, full conformance doesn't seem to be a standard, but the files are usually close enough.

FlowOpt workflow editor can validate input files against the XPDL 2.1 schema and if the validation fails, the user may decide whether the import procedure should be attempted anyway. The reason for this is that some tools produce files that do not fully conform to the schema, but they are close enough for the import procedure to work.

The XPDL files listed in this section were created in the Yaoqiang XPDL editor [10], which seems to produce reasonably accurate files.

Import procedure

We used the same basic algorithm for importing nested workflows as in the MAKE import procedure, but we made a small extension in order to support some XPDL patterns that aren't present in the Simple TNA model.

The problem is that the original algorithm (see Figure 37) only recognizes two types of routing – parallel and alternative. However in XPDL / BPMN, there is another type – inclusive. Inclusive routing allows any (non-empty) subset of all the relevant activities to execute.

This pattern falls somewhere "between" the parallel and alternative nest patterns recognized by FlowOpt - the former forces all the child nodes to be performed and the latter forces exactly one child node to be performed. An inclusive nest allows any possible non-empty subset of the child nodes to execute.

If we want to import inclusive nests into FlowOpt, we have three choices:

1) Extend FlowOpt workflow model. This choice was generally out of the question, since for the moment, the FlowOpt data model is final (other FlowOpt modules' functionality relies on this fact).
2) Map inclusive nests into either parallel or alternative nests. This would mean creating a workflow that is not equivalent to the original, but it would be similar.
3) "Simulate" the inclusive nests by creating an alternative nest with one alternative for every possible subset of the child tasks of the original nest. This would create a workflow that is semantically closer to the original at the cost of creating an exponential number of tasks.

We chose the second option, because an exponential increase of size between the original and the imported workflow is unacceptable for the potential user. It is also very ineffective, since the workflow eventually has to be scheduled and/or saved to be of any use.

Note that even if we did convert the inclusive nests by explicitly creating an alternative for every way they can be performed, the result still wouldn't be equivalent to the original, because we have no way of importing the conditions on the links that XPDL uses to determine which subset eventually gets executed.

Now that we established that we need to convert XPDL inclusive nests into either parallel or alternative FlowOpt nests, we have to decide which one is the better option.

In BPMN/XPDL, all splits are inclusive by default and all joins are exclusive by default (this default behavior is commonly referred to as uncontrolled flow). This means that if we statically map inclusive nests to either parallel or alternative, some standard pattern won't be imported:

- If we map to parallel nests, we won't be able to import the following pattern (called multiple merge):

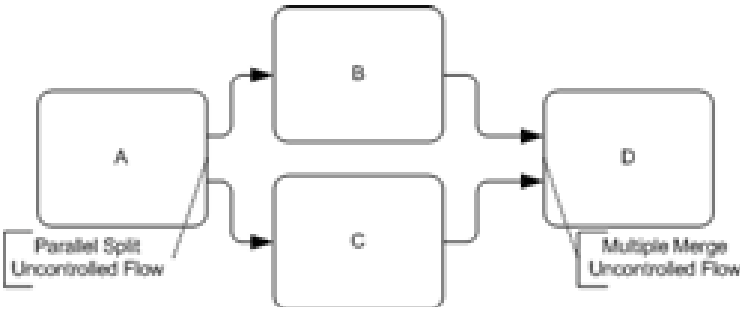


Figure 41: XPDL inclusive routing 1

In Figure 41, we map the (inclusive) split on activity 'A' to parallel, while the join on activity 'D' is by default exclusive (alternative in FlowOpt terminology). The original algorithm would report an error when trying to import this workflow, since the two routing types are different.

- On the other hand, if we map inclusive to alternative nests, the situation is even worse - we wouldn't be able to import the standard parallel split pattern:

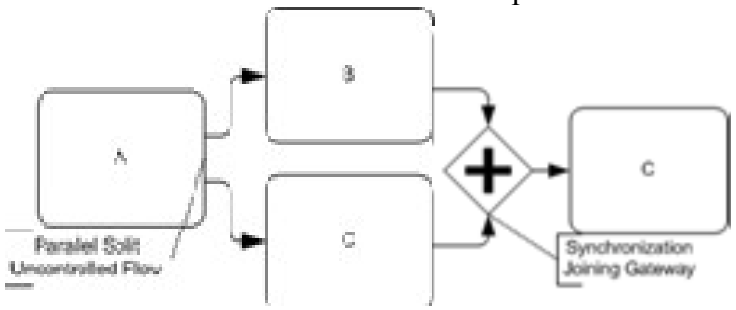


Figure 42: XPDL inclusive routing 2

In Figure 42 we map the (inclusive) split on activity 'A' to alternative and the join on the synchronization gateway is parallel by definition. This would again produce an error in the original algorithm.

To summarize, if we want to be able to import both the above patterns, we cannot map the inclusive nests statically. The import procedure we used tackles this problem by extending the original import algorithm with an extra label type called 'Any'. If a node has (input or output) label equal to 'Any', it means that it can create either a parallel or an alternative nest, depending on the label of the other principal node (we are using terminology introduced in [1]).

To implement this behavior, we relax the original algorithm a bit – we no longer require the labels of both the principal nodes to be equal when trying to create a nest. In case either of them is set to 'Any', we create a nest determined by the type of the other principal node.

If both labels happen to be set to 'Any', we can choose whether to create a parallel or an alternative nest. We chose to create a parallel nest in such a case, since the uncontrolled flow is so often used to model a parallel split and the user would probably expect such behavior.

If neither label is set to 'Any', we require them to be equal, like in the original algorithm.

The above functionality is referred to as 'matching' of labels in the algorithm presented in Figure 38 and its description. It didn't matter when importing MAKE workflows, since the MAKE model is simpler and does not define inclusive routing. In other words, no node has the 'Any' label when importing MAKE workflows, so 'matching' becomes the same as equality in that particular case.

XPDL to FlowOpt mapping

We already described the mapping of inclusive nests. This section describes the mapping of the other XPDL objects into FlowOpt workflow model.

XPDL packages and processes

An XPDL package is a top level container for workflow processes, which correspond to individual workflows. Obviously workflow processes are mapped to FlowOpt workflows.

Packages are mapped to FlowOpt workflow editor folders, but note that these are only defined by the workflow editor and they aren't a part of FlowOpt data model, so this is only for the user's convenience.

XPDL activities

In XPDL, activities are a much broader concept than in FlowOpt. They are the only type of nodes present in XPDL diagrams, but based on their properties, their semantics can vary drastically. There are several different kinds of activities in XPDL:

- Simple activities
- Block activities / Subflows (Embedded / Reusable subprocesses)
- Gateways
- Events

Simple activities in XPDL correspond to activities in FlowOpt, representing some elementary unit of work to be done. As such, they are mapped onto FlowOpt activities.

Note that the terminology can be a bit misleading here, since both FlowOpt and XPDL define the concepts of a task and an activity, but they aren't the same.

In XPDL, a task is a specific way of implementing an elementary activity (manually, via a service, via a script…). In other words it is just an attribute of an activity. Tasks aren't imported from XPDL, since FlowOpt activities do not have any alternative means of implementation (it is always manual).

Block activities and subflows are a concept similar to FlowOpt tasks – they represent an independent nested workflow that is to be executed in place of the activity. The difference is that block activities represent other parts of the same workflow (called activity sets in XPDL terminology), while a subflow refers to an entirely different workflow process.

In BPMN, the corresponding terms are embedded and reusable subprocesses. The former roughly corresponds to block activities, the latter to subflows.

Currently, both embedded and reusable subprocesses are mapped onto tasks, since the concepts are very similar. However, one of the planned extensions of the FlowOpt workflow model is the ability to link a task to entire independent workflow, which would be a more appropriate mapping for the subflows / reusable subprocesses.

Gateways are a concept similar to MAKE's semaphores – activities representing no work, but affecting the execution flow somehow. In FlowOpt, these flow constraints are stored in tasks, so semaphores do not directly translate into anything. They just define the types of tasks that are created.

Finally, events in XPDL represent something that happens during the workflow execution. There is no such concept in FlowOpt, so any mapping of these is problematic. There are three kinds of events – start events, end events and intermediate events.

The first two serve to mark the beginning and end of a workflow and as such, they aren't mapped to anything, FlowOpt workflows implicitly start when their root task starts and end when their root task ends.

Intermediate events represent things that can occur while the workflow is being executed. As stated above, there is no such thing in FlowOpt, since FlowOpt workflows do not have any connection to runtime. We chose to map intermediate events to empty tasks, so that at least some information is preserved, but the resulting workflow can be significantly different from the original. Since the workflow editor allows the user to delete all empty tasks in a workflow at once, this shouldn't be too limiting in terms of usability.

XPDL links

XPDL defines the following types of links:

- Sequence flow
- Message flow
- Associations

Sequence flow links are the most important, as they define the order of execution. As such, they are mapped on FlowOpt precedence links. As we already mentioned in the description of the XPDL export procedure, these are not semantically equivalent.

FlowOpt precedence links simply represent a (temporal) precedence relation, whereas XPDL sequence flow links explicitly define the order of execution in a way similar to that of Petri nets – an execution token travels over the sequence flow links, causing whatever activities it encounters on the way to execute (please see XPDL/BPMN specification for a more formal definition).

This difference doesn't really matter when workflows follow a simple workflow model like that of FlowOpt, so it wasn't much of an issue when we exported into XPDL. However when we want to import from XPDL, we have to realize that some of the more complex workflow patterns that can be modeled in XPDL will not translate into FlowOpt due to this difference. We will provide a list of standard workflow patterns and how they can translate from XPDL later in this chapter.

Message flow links represent message passing between two participants and they can only connect activities in different pools. FlowOpt defines no way to visualize pools or message passing, so we chose to omit the message flow links.

Associations serve to connect an activity with an artifact, which is some kind of additional information that is not semantically a part of the workflow, such as a comment. Once again, FlowOpt defines no such concepts, so we do not import those.

Routing information

XPDL defines three types of routing in case an activity has multiple incoming or outgoing (flow) links:

- Parallel
- Exclusive
- Inclusive

Parallel and exclusive correspond to the semantics of FlowOpt parallel and alternative tasks respectively, so naturally we map them to the corresponding task type.

Inclusive routing means that (potentially) any non-empty subset of all the connected activities is to be executed. We already described the mapping of those in the general description of the import procedure.

Participants

XPDL participants are a concept similar to FlowOpt resources – some people or machines capable of performing an activity. As such, they should be mapped onto FlowOpt resources, but this feature is currently not implemented due to some technical complications. It is a work in progress though.

Pools, lanes, applications…

In XPDL, pools and lanes visualize which participants perform which parts of the workflow. In FlowOpt, we have no way of doing that, therefore pools and lanes aren't imported.

XPDL applications aren't imported for the same reason - there is no equivalent, since FlowOpt resources can only be people or machines, which reflects the fact that FlowOpt workflows primarily describe manufacturing processes.

Any other XPDL objects not explicitly listed above are also not imported, either because they have no equivalent or because they aren't very significant.

Standard workflow patterns

To illustrate how the import procedure works, this section briefly describes how the standard workflow patterns [11] are imported when the procedure from Figure 38 is used.

Starting from the simplest patterns, *sequence* maps into a serial task, *parallel split* and *synchronization* create a parallel task and an *exclusive choice* together with a *simple merge* creates an alternative task.

This should be fairly obvious, as these are nearly equivalent concepts. As long as the workflow only contains these patterns, the resulting FlowOpt workflow should be almost equivalent to the original.

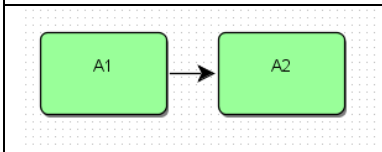
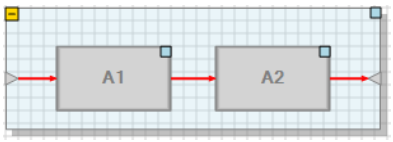
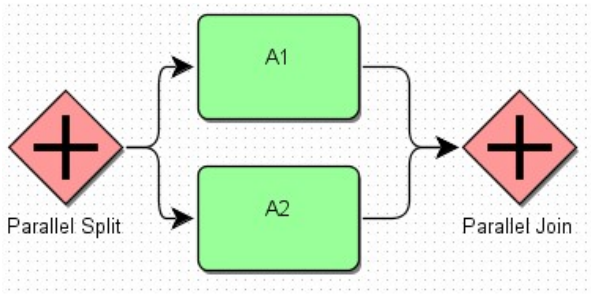
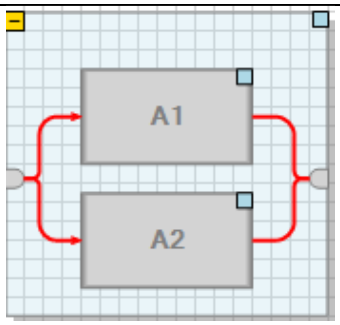
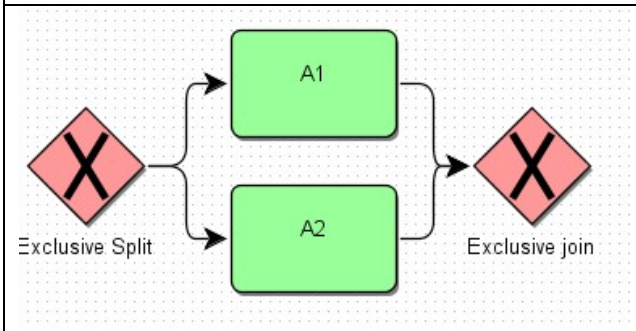
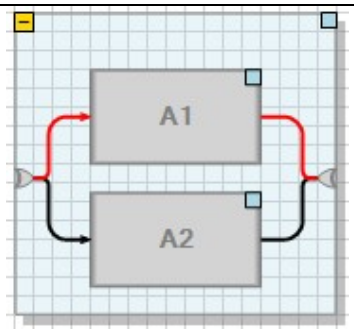
Original Pattern	Imported Workflow
	
	
	

Figure 43: Mapping of basic patterns

Figure 43 shows how the import procedure maps the most basic patterns (sequence, parallel and alternative nests).

Multiple choice and *multiple merge* are an example of inclusive routing, which means that the type of the resulting task depends on the type of the other principal node's label. If it is parallel or alternative, the resulting task is also parallel or alternative (respectively). If it is also inclusive, the resulting task is parallel (see the description of the import algorithm for details).

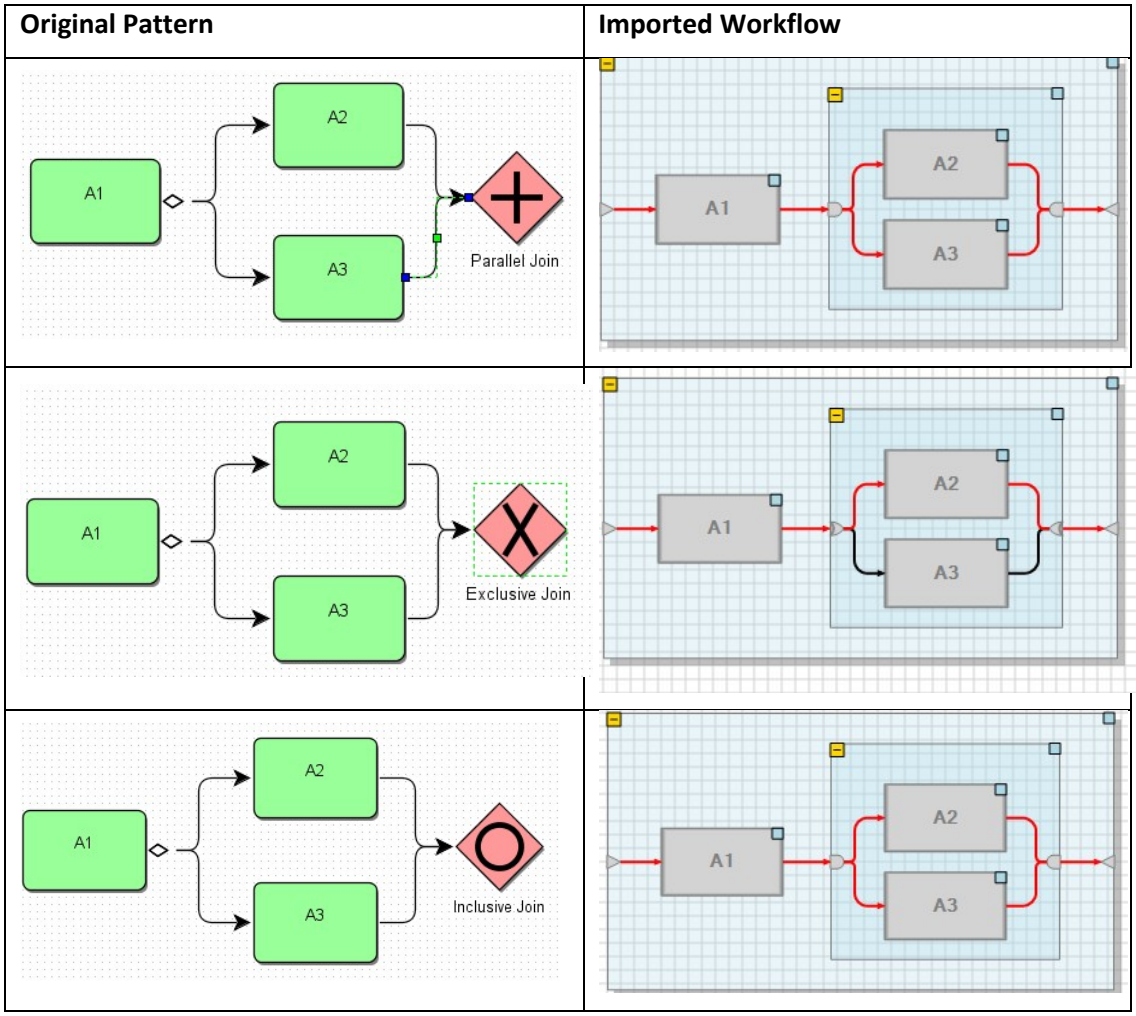


Figure 44: Mapping of multiple choice and multiple merge

Figure 44 shows how the multiple choice and multiple merge patterns are imported.

Discriminator is mapped to an alternative task, which seems to be a reasonable interpretation in terms of the FlowOpt workflow model. One could argue that since discriminator allows more than one incoming activity to execute, it should be mapped to a parallel task, but since the pattern uses an exclusive join, it seems more intuitive to use an alternative task.

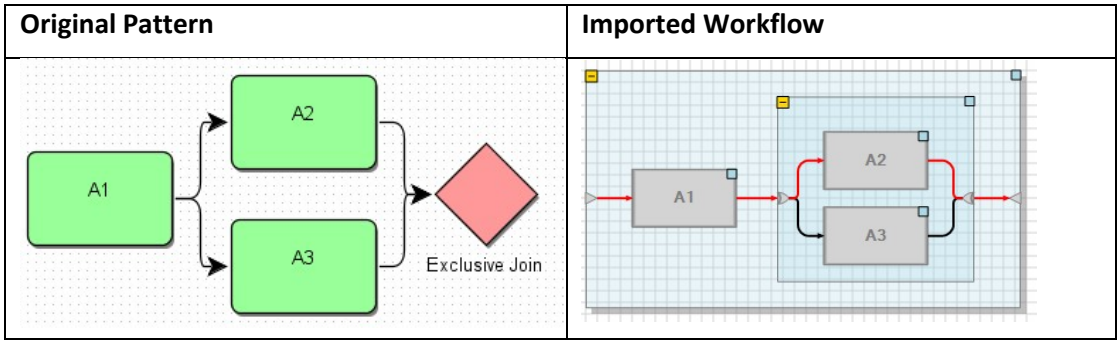


Figure 45: Mapping of discriminator

Figure 45 shows how the discriminator pattern is imported.

Due to the way we chose to map inclusive splits/joins, both the *N out of M join* pattern and the *synchronizing merge* pattern translate to parallel tasks (complex gateways are treated in the same way as if they were inclusive).

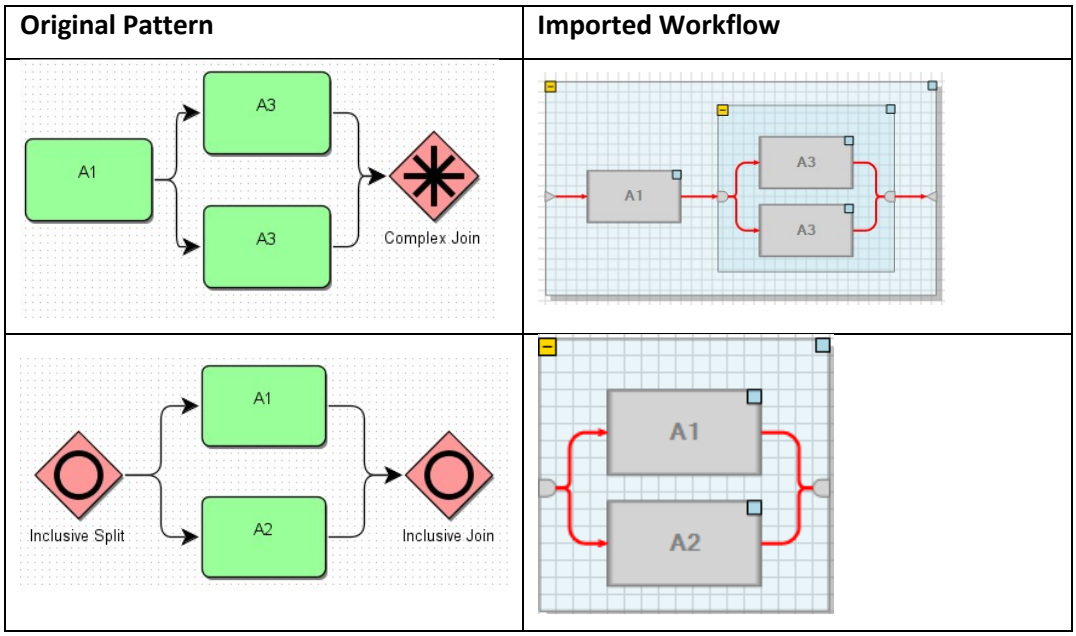


Figure 46: Mapping of N out of M join and synchronizing merge

Figure 46 shows how the N out of M join and synchronizing merge patterns are imported.

The *arbitrary cycles* pattern isn't imported – FlowOpt doesn't support cycles in workflows at all, so the import algorithm will end with an error if one is found.

One way to get around this would be to delete some of the links based on some kind of heuristics, but since this entire pattern revolves around those extra links, this approach seems to beat the whole purpose of trying to import it.

The *implicit termination* pattern also doesn't have an equivalent in FlowOpt. However the import procedure does support multiple start events and multiple end events (or no start / no end events for that matter).

It does so by automatically merging all the start / end events into a single start / end event with the 'Any' label, so in the end only one task can be created that represents the entire workflow. After that, the single start event is connected to all the nodes with no incoming links and the single end event is connected to all the nodes with no outgoing links.

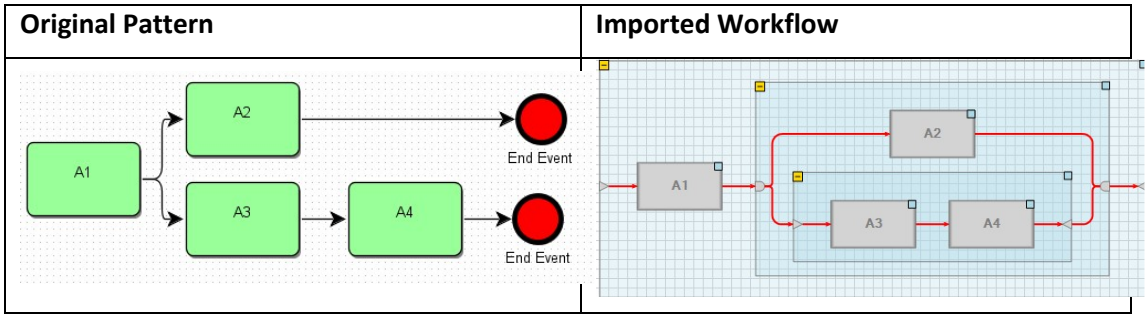


Figure 47: Mapping of implicit termination

Figure 47 shows how the implicit termination pattern is imported.

There are several patterns involving *multiple instances*, but FlowOpt currently doesn't support this concept either. Workflows containing these patters can be imported in general, but the information on how many times a particular activity is to be instantiated is lost in the process.

Supporting multiple instances is a planned feature of the FlowOpt workflow editor, so it is likely that these patterns may be imported fully at some point.

Deferred choice also has no equivalent in FlowOpt, since it relies on events, which generally do not translate. Since events map to empty tasks, we can still import the workflow, but the pattern will be lost in the process.

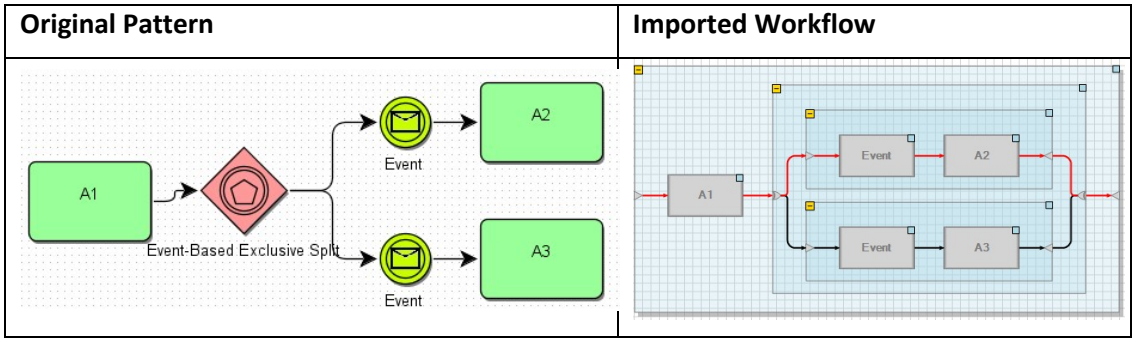


Figure 48: Mapping of deferred choice

Figure 48 shows how the deferred choice pattern is imported.

The remaining patterns do not translate well into FlowOpt, since they rely on some concept that doesn't exist in FlowOpt (usually an event of some kind).

Interleaved parallel routing doesn't translate, since in FlowOpt the order of any sequence has to be given. It is imported as any other subprocess, that is as a parallel task by default.

Milestone cannot be imported at all, since it relies on link events, which (like all events) are do not translate into FlowOpt. *Cancellation patterns* do not translate for the same reason.

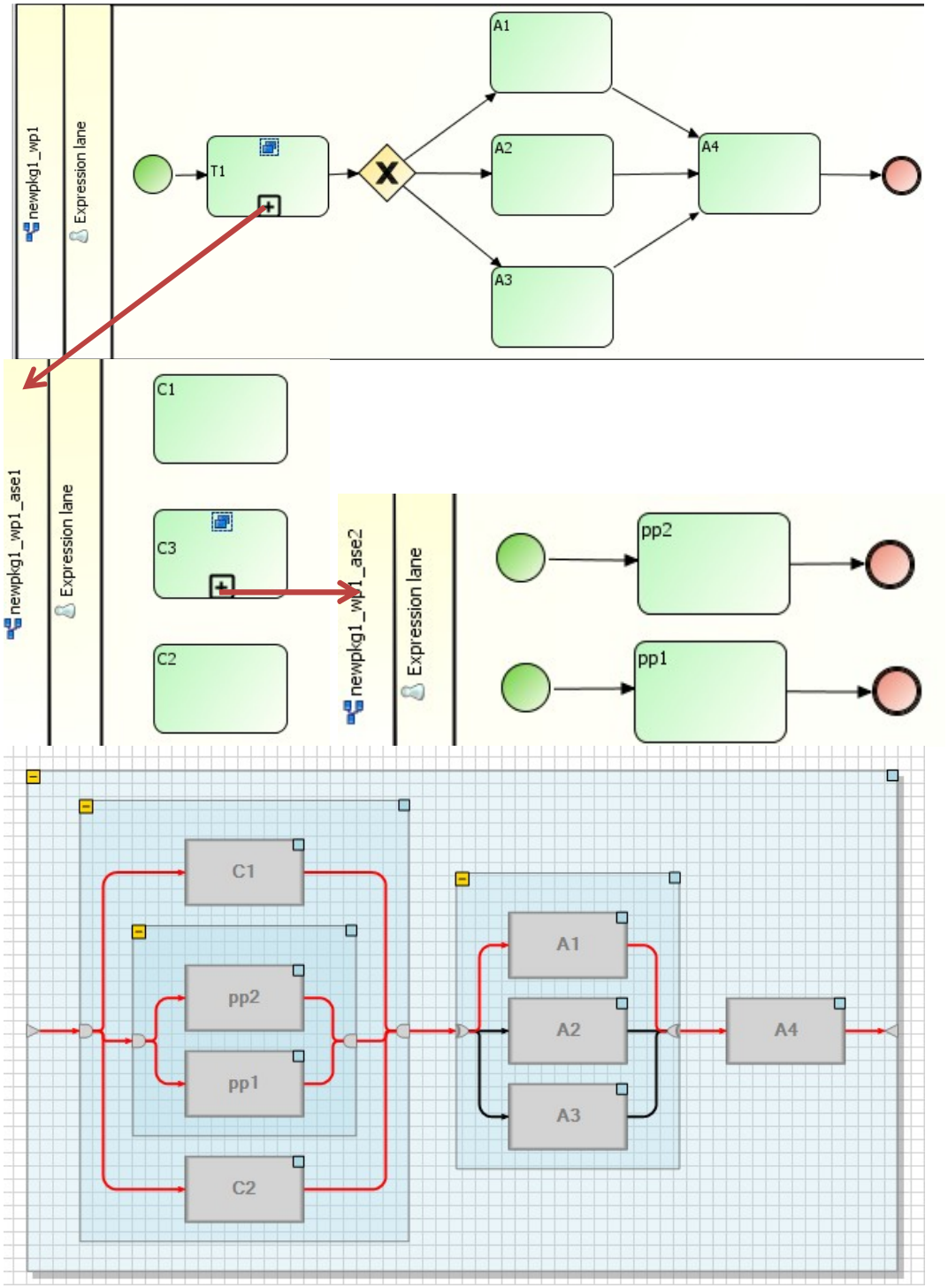


Figure 49: Imported XPDL workflow

Figure 49 shows an example of a XPDL workflow (visualized again in Together Workflow Editor) imported into FlowOpt that should illustrate how the nested structure is created by the import procedure. Notice that the order of the child tasks is preserved.

One of the attachments to this thesis is a file called '*Piston.xpdl'*. This file contains the piston workflow shown in several examples in this thesis exported into XPDL format by the presented application. Importing it yields an identical workflow to the original, except for custom links.

9. Conclusions

We believe we managed to deliver a working application that meets all the software requirements formulated in the beginning of the development process, both functional and non-functional.

It represents manufacturing processes in a simple, lightweight and efficient way while providing a different and hopefully innovative and usable approach to many aspects of workflow modelling.

This thesis adds several important features that weren't implemented during the works on the FlowOpt software project – namely workflow verification and import and export to and from other workflow formats.

From the reactions of some potential users, it appears that the application could provide a viable alternative to other workflow editors. It was presented on the ICAPS[1] conference in Freiburg [14] and the reactions were quite positive.

Some users also provided negative feedback, but it was mostly due to the fact that they were used to the traditional way of building workflows and preferred it to our proposed model. This is to be expected, our application was always meant as an alternative, not a replacement for existing solutions.

It is worth mentioning that there was no negative feedback regarding functionality, the workflow editor was tested thoroughly and the result should be of reasonable quality. The application is well documented, both for the potential user and a potential developer maintaining it.

Currently the works on integration of the FlowOpt project are being finished and once they are, all of its modules will hopefully be evaluated by a larger number of users, which will provide valuable feedback on how much potential our approach has and how it could be extended or improved.

[1] International Conference on Automated Planning and Scheduling

10. Future works

There are several planned features that should be implemented in the near future. Some of them are just for user convenience, others extend the workflow model considerably. These features include:

- Defining activities directly in the workflow editor, rather than in the MAKE application.
- Using the automatic layout and the nested structure to represent the BOM (Bill Of Materials) of MAKE. BOM is an object that describes the structure of a particular product. It is essentially a tree of various parts that the product consists of. It would be useful to provide an intuitive visualization of this concept and the nested model could definitely be utilized to do this, since it also represents a tree structure.
- Implementing tasks as workflows – the user should be able to link a separate workflow into an empty task to have it performed in place of that task. Currently the editor lets the user insert a workflow into an empty task, but this inserted workflow is an independent copy of the original. It would be useful to just create a link between the two workflows, so that updates in the linked workflow would manifest in the referencing workflow as well.
- Implementing the multiple instances patterns. This could be a very powerful tool used to model loops in our editor. The user would be able to specify that a task should be performed multiple times (either in parallel or in a sequence).
- Import work orders from MAKE. It would be convenient to be able to import whole work orders instead of just workflows. For example the user could use the FlowOpt optimizer and analyzer to schedule and optimize the work order.

Bibliography

[1] BARTÁK, Roman; ČEPEK, Ondřej. Nested Temporal Networks with Alternatives. *Hans W. Guesgen, Gerard Ligozat, Jochen Renz, Rita V. Rodriguez (Eds.): Papers from the 2007 AAAI Workshop on Spatial and Temporal Reasoning, Technical Report WS-07-12, AAAI Press, 2007, pp. 1-8 (ISBN: 978-1-57735-339-3)*. Available from <http://ktiml.ms.mff.cuni.cz/~bartak/downloads/AAAI2007ws.pdf>.

[2] *BPMN 1.1.*: Object Management Group, January 2008
Available from <http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf>.

[3] VAN DER AALST, Wil. *Workflow Patterns* [online]. 2010 [cit. 2011-07-25]. Available from <www.workflowpatterns.com>.

[4] MURATA, Tadao . Petri Nets : Properties, Analysis and Applications. *Proceedings of the IEEE*. April 1989, vol.77 no.4

[5] *Wikipedia* [online]. 2011 [cit. 2011-07-25]. Work Breakdown Structure. Available from <http://en.wikipedia.org/wiki/Work_breakdown_structure>.

[6] TSAMARDINOS, Ioannis; POLLACK, Martha. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence,* Volume 151 Issue 1-2. February 2003.

[7] PLANKEN, L.R. *New Algorithms for the Simple Temporal Problem*. Delft, the Netherlands, 2008. 75 p. Master's thesis. Delft University of Technology.

[8] BARTÁK, Roman; ČEPEK, Ondřej. Temporal Networks with Alternatives: Complexity and Model. *Proceedings of the Twentieth International Florida AI Research Society Conference (FLAIRS 2007). AAAI Press, 2007, pp. 641-646 (ISBN 978-1-57735-319-5)* .
Available from <http://ktiml.ms.mff.cuni.cz/~bartak/downloads/FLAIRS2007.pdf>.

[9] *XPDL 2.1*. Hingham, MA 02043 USA : Workflow Management Coalition, October 2008. 217 p. Available from <http://www.wfmc.org/index.php?option=com_docman&task=doc_download&Itemid=72&gid=132>.

[10] *Yaoqiang XPDL Editor* [online].

Available from <http://sourceforge.net/projects/yxe/>.

[11] WHITE, Stephen. Process Modeling Notations and Workflow Patterns. . Available from <http://www.bpmn.org/Documents/Notations_and_Workflow_Patterns.pdf>.

[12] *YAWL Foundation* [online].
Available from <http://www.yawlfoundation.org/>.

[13] BARTÁK, Roman: On Complexity of Verifying Nested Workflows with Extra Constraints. To appear in Proceedings of 14th Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty

[14] BARTÁK, Roman, et al. FlowOpt: A Set of Tools for Modeling, Optimizing, Analyzing, and Visualizing Production Workflows. *Proceedings of ICAPS 2011 System Demonstrations*. 2011, pp. 6-9.

List of Figures

List of Abbreviations

(Nested) TNA	(Nested) Temporal Networks with Alternatives
BOM	Bill Of Materials
BPMN	Business Process Modeling Notation Business Process Model and Notation
CDM	Common Data Model
DAG	Directed Acyclic Graph
DFS	Depth First Search
DTP	Disjunctive Temporal Problem
ICAPS	International Conference on Automated Planning and Scheduling
IFPC	Incremental Full Path Checking
JPEG	Joint Photographic Experts Group
OMG	Object Management Group
PDF	Portable Document Format
PNG	Portable Network Graphics
STN	Simple Temporal Network
STP	Simple Temporal Problem
SVG	Scalable Vector Graphics
TWE	Together Workflow Editor
UML	Unified Modeling Language
WBS	Work Breakdown Structure
XML	eXtensible Markup Language
XPDL	XML Process Definition Language
YAWL	Yet Another Workflow Language