Kinerja: A Workflow Execution Environment

by

Sam Procter

B.S., University of Nebraska at Lincoln, 2009

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2011

Approved by:

Major Professor
John Hatcliff

Copyright

Abstract

Like all businesses, clinical care groups and facilities are under a range of pressures to enhance the efficacy of their operations. Though there are a number of ways to go about these improvements, one exciting methodology involves the documentation and analysis of clinical workflows. Unfortunately, there is no industry standard tool which supports this, and many available workflow documentation technologies are not only proprietary, but technologically insufficient as well. Ideally, these workflows would be documented at a formal enough level to support their execution; this would allow the partial automation of documented clinical procedures. However, the difficulty involved in this automation effort is substantial: not only is there the irreducible complexity inherent to automation, but a number of the solutions presented so far layer on additional complication.

To solve this, the author introduces Kinerja, a state-of-the-art execution environment for formally specified workflows. Operating on a subset of the academically and industrially proven workflow language YAWL, Kinerja allows for both human guided governance and computer guided verification of workflows, and allows for a seamless switching between modalities. Though the base of Kinerja is essentially an integrated framework allowing for considerable extensibility, a number of modules have already been developed to support the checking and executing of clinical workflows. One such module integrates symbolic execution which greatly optimizes the time and space necessary for a complete exploration of a workflow's state space.

Table of Contents

List of Figures

List of Algorithms

Acknowledgments

This thesis, and the software developed to support it, could not have happened without the support, help, and caring of my teachers, family and friends. First and foremost I would like to thank Dr. John Hatcliff, my primary adviser, whose help and guidance was absolutely crucial not only on this project, but on a number of smaller projects which led to my graduate work at Kansas State. I also want to explicitly thank Dr. Robby, whose technical advice was essential during the development of the more intricate parts of Kinerja. I'd also like to thank Dr. Virgil Wallentine for his advice, and for serving on my committee. I'd like to thank my family for their understanding, support and advice, particularly when it came to the machinations of graduate school. Finally I'd like to thank my friends, who listened to not only my complaints when my program was functioning incorrectly, but also my celebrations when things finally worked.

Dedication

To my parents, who were my first teachers, and to my brother, who before anyone else showed me that learning was cool.

Chapter 1

Introduction

1.1 What is Workflow?

We define a workflow as a series of steps necessary to complete some task as well as the rules governing transitions between those steps. An alternative definition, from *Modern Business Process Automation* is (paraphrased): a precise process description used to guide the execution of activities[14]. The task might be something quite simple with a small number of steps (e.g., filling a cup with coffee) or it may be a very complex, long process (e.g., launching the space shuttle). Each step in a workflow can itself contain a number of sub-steps; i.e., each step in a process can itself be a fully defined workflow. This gives the creator of a workflow the ability to define a process at an arbitrary level of abstraction, and it allows for a gradual refinement (or abstraction) over time as needs dictate.

A workflow model, then, is a representation of a workflow for the purpose of communication or storage. Though a model may be a diagram on a piece of paper or a whiteboard, it might also be machine readable, in a standardized format referred to as a workflow modeling language. Some of these languages, such as Business Process Modeling Notation (BPMN)[31], are developed by third party consortia for the purpose of standardizing workflow models. In recent years there has been a growing interest in getting workflows off-the-page (or off-the-whiteboard, as the case may be) for ease of sharing and archiving. This has led to a number of new workflow modeling languages which are designed strictly for use in various work-

flow modeling tools[2,14,18,26]. These languages are similar in a number of ways to computer programming languages, in that they:

- Have a defined syntax and structure
- Enjoy rich tool support, including editors, translators, pretty printers, etc.
- Can often be generated from higher-level abstractions
- Have (possibly formally defined) semantics
- Can be simulated, verified, etc.

It should also be noted that without a well-defined semantics, simulation, verification and execution of workflows is difficult, if not impossible.

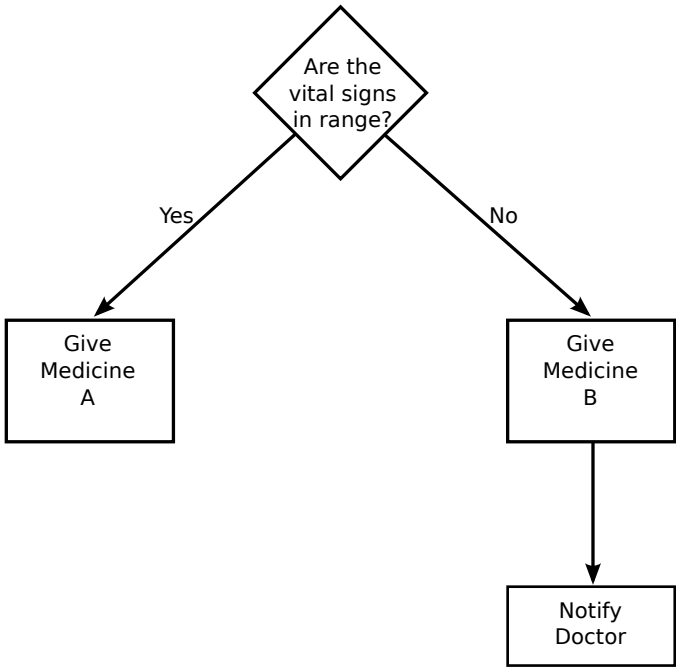


Figure 1.1: *An example workflow*

There are a number of industries to which workflow modeling is particularly well suited, and interest in many of these industries is high. Often the best process for completing a

given task is dictated by some domain expert, so in these industries the sharing of best-practices (in the form of workflow documentation) can be of substantial benefit. Workflow modeling has been successfully used in a broad range of commercial and industrial contexts, ranging from professional cinema[23] to business process management[29] to various medical contexts (e.g.: gynecological[14]).

There are a number of organizations involved in advancing workflow modeling. These include commercial entities like the Object Management Group (OMG[12]) as well as academic institutions like the University of Massachusetts (UMass[33]) and Queensland University of Technology (QUT[28]). Similarly, a range of tools have been developed, such as the Little-JIL[33] or BPMN Modeler[27] tools – both are plug-ins for the popular Eclipse integrated development environment. There are also standalone tools like the Java Application Building Center (jABC[25]) or Yet Another Workflow Language (YAWL[28]) editor and engine.

1.2 Workflow in Health Care

The documentation of work processes is important in a range of industries, not least among them medicine. There are four main ways the documentation of workflow is especially useful:

1. Communicating best practices from domain experts to medical workers – The range of procedures various medical personnel are tasked with is vast. Though many have a great deal of expertise in a number of areas, there are experts in each of the more specific domains found in health care. Training with workflows developed, critiqued, or otherwise reviewed by these experts allows for an efficient and novel dissemination of the professional's expertise.

2. Communicating best practices from device manufacturers to medical workers – Device manufacturers and those in the medical services field have a vested interest in making sure that their products are used correctly and safely. The use of workflow modeling technologies in conjunction with standard training techniques allows for longer term and more thorough reinforcement after the training session has completed.

3. Documenting a hospital's current practices for quality auditing organizations – Having on-hand documentation of existing processes can minimize the disruption caused by the various audits medical centers are often required to undergo. The documentation can also provide a "paper trail" useful for explaining what went wrong when a problem is discovered. This paper trail can allow issues of liability to be settled more quickly and accurately, as time-stamped comparisons of what should have occurred versus what actually occurred can be easily created.

4. Enhancing operations management and efficiency planning – Workflows can aid in the creation of simulations which allow management to more easily test out different resource allocation strategies, increasing the facility's efficiency and helping to keep costs low. They can benefit planners by not only detecting flaws earlier in the workflow design process, but by designing workflows with more clarity and specificity in the original case.

In many medical centers, workflows are poorly documented[24], and many aren't documented at all, but are instead simply routines well-known to experienced workers. These steps to perform a given task are passed on through word-of-mouth and on-site training. If a process requires in-depth analysis, it may be drawn up on a whiteboard or poster, but the knowledge contained in the diagram is lost as soon as the board is erased or the poster is taken down. The most advanced technology commonly used is basic graphing software (e.g., Microsoft Visio) which allows for sub-processes, but little else.

One area where currently popular workflow modeling technologies (graphing software as well as whiteboard diagrams) fall considerably short of more specialized technologies is the formality of their semantics. As mentioned previously, a formal semantics allows for a range of benefits:

1. Simulation – Since a workflow specification written in a language which has formalized semantics has a completely unambiguous meaning, a computerized interpreter can be built which allows for simulation of the workflow.

2. Verification – The simulation mentioned in (1) can be re-executed repeatedly to check for errors or verify that certain properties never hold, hold in all cases, or hold in certain special situations.

3. Code Generation – An unambiguous workflow specification can be interpreted by a computer and then re-generated in some other unambiguous language (e.g. a programming language like Java or Scala). This allows for large amounts of tedious, error-prone (and boring) code to be generated rather than implemented by a human.

Unfortunately, the solution to this problem is far more complex than simply using existing formal modeling technologies. Specifically, the problems with formally modeling clinical workflows include:

1. Frequent exceptional occurrences – With the variety of patient disorders and possible comorbidities, workflows cannot be standardized to nearly the same degree that they can in other industries.

2. Frequent concurrency – Many medical workflows have a high degree of concurrency. Further complicating the processes is the large amount of integration between the concurrent workflows as inter-process communication is extremely common.

3. Wide variety of potential workflows – Many roles in medical centers require a broad range of abilities, and repetition can be infrequent. This means individual workflows may receive less attention than those in other industries due to the number of processes to document.

1.3 Contributions

This thesis introduces and explains a new tool, developed by the author, called Kinerja. Kinerja is a workflow execution environment. That is, it is not an editor for developing workflows, but rather an interpreter for workflows with well-defined semantics. While its

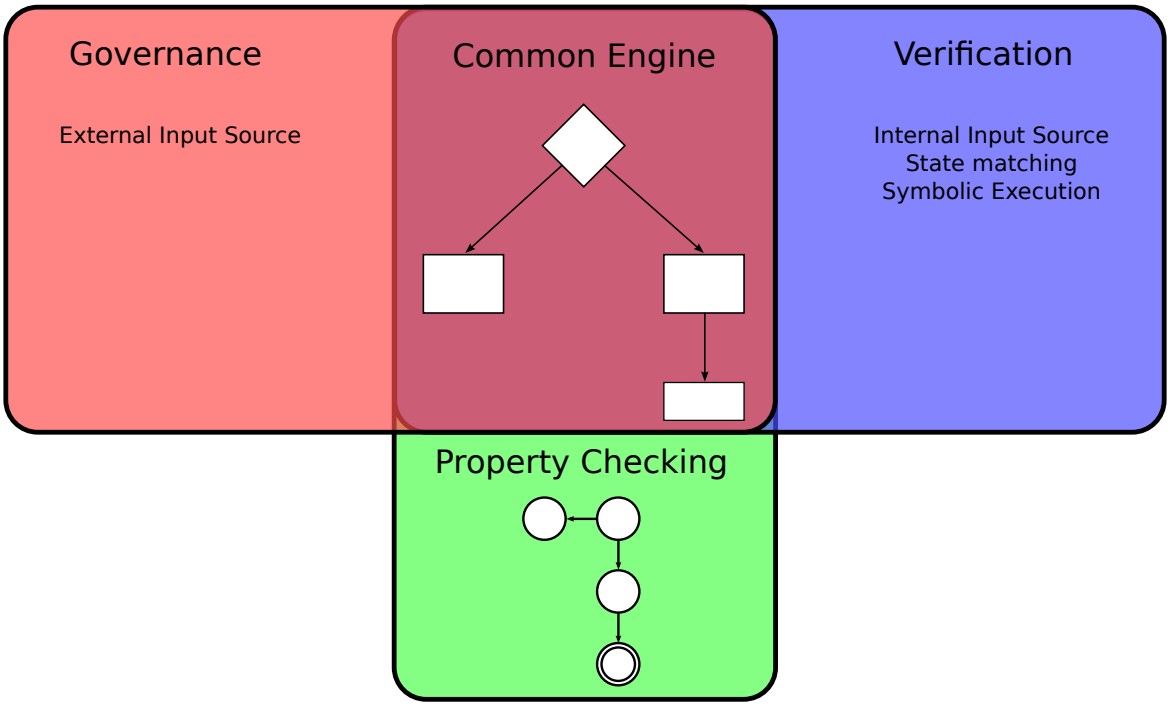


Figure 1.2: *The Kinerja vision*

use is not tailored to any particular industry, the examples and focus in this thesis are clinical in nature.

Kinerja itself has two important aspects:

1. Two engine modes –

   (a) Verification – In verification mode, Kinerja systematically explores all possible program states.

   (b) Governance – In governance mode, Kinerja directly executes the workflow.

   The key difference between engine modes is what happens when a point of nondeterminism is reached – that is, what happens when the system does not contain enough information to figure out how to proceed on its own. In verification mode, execution takes all possible routes. That is, if the workflow asks for a patient's heart rate as some number between 1 and 300, Kinerja continues executing with every possible heart rate. In governance mode, Kinerja will poll an outside agent for the information. For

example, the patient's heart rate could be entered by a clinician, or read directly from a compatible heart rate monitor.

Note also that both of these modes will use the same program logic, guaranteeing that all states reachable in governance mode are checkable by the verification mode.

2. Property Checking – Kinerja is able to verify properties[9] of the workflows it executes, and these properties are implemented as finite-state machines.

Chapter 2

Literature Review

2.1 Execution of Workflow

In addition to the benefits cited in the previous section, an additional, and significant, benefit of formalizing workflows is the ability to "execute" them by defining actions (either manual or automated) to associate with steps in processes. This execution can be quite basic, like simply working through a checklist generated from the flow of steps in a given process, or it can be far more complex, like an execution environment with automatic and manual task integration handled by a central dispatch. It is similar to the simulation of workflows discussed earlier, except rather than simulating the outcomes of tasks, the steps of the workflow are instead actually performed, and the result is loaded back into the system so the process can continue.

2.1.1 Status Quo

While the concept of performing medical procedures as a series of steps has been around as long as medicine itself, medical devices have only (relatively) recently been electro-mechanical devices. As the machines, and eventually computers, controlling medical devices have increased in complexity, so has their flexibility. No longer are devices limited to the simple display of visual information – many medical devices can report report their status along some electronic communication channel as well (e.g.: a serial or Universal Serial Bus

(USB) port). This allows these devices to be networked with each other and with central servers.

These communication links can remove much of the tedious and error-prone data entry steps in medical workflows – not only making getting patient physiological data into medical systems faster, but more accurate as well. This communication allows for a far more smoother integration of automated and manual tasks in a workflow (since manually entering the results of an automated step is counter-intuitive and error-prone).

Some workflow technologies (e.g.: YAWL) allow custom automated tasks. These technologies define a mapping from the workflow's state variables to a user-defined program (which may be an interface to a physical device) and from the output of the custom program back into the workflow state. In this mapping, there is a realization of the goal of a smooth integration of automated and manually executed tasks.

2.1.2 Future Developments

One interesting application currently on the "bleeding-edge" of development is the ability to coordinate the control of medical devices with the Integrated Clinical Environment (ICE) standard or a similar framework (see e.g.:[15]). The technological piece necessary for this advancement is the addition of control data to the computer-interface in a medical device. Though many medical devices support streaming data meant for display, relatively few support receiving commands that modify the device's behavior. This advancement is interesting for three reasons:

1. The power of such a system is impressive – Connecting and integrating medical devices allows for everything from safer surgeries (e.g.: disabling a medical laser if a patient's oxygen line is still active) to more accurate diagnostic images (e.g.: a respirator signaling an x-ray machine when a patient's lungs are inflated to a certain amount[17]). With such a system, both the likelihood and impact of human error can be reduced, and the speed of many operations increased.

2. A "closed-loop" medical system is risky – The idea of a computer error costing a human life is particularly horrifying, and relinquishing any amount of control over a medical procedure is likely to cause considerable unease.

3. Verification of such a system is unprecedented – The verification required to eliminate risk in such a system is certain to be difficult. Worse still, the current regulatory framework is not prepared to deal with workflows that create closed-loop systems[15]. Since medical devices have historically been monolithic units, they are tested as such and there is no way to verify the correctness of individual elements of a heterogeneous system.

It's logical that workflow coordination languages should aim to be "device-aware" at some future point, and to be able to handle the coordination of devices into closed-loop systems.

2.2 Evaluation Metrics

There exist a number of workflow modeling languages and tools; and they range from those designed for use within the medical profession to those designed for business to those designed for computer scientists. Ultimately we settled on three order qualifiers:

1. Expressiveness – The language must be able to capture the complexities and diverse range of workflows in the medical profession. This means the tool must support a number of different workflow "constructs" such as:

   (a) Explicit flow control – The workflow author should be able to explicitly control the flow in the process. This entails not only the ordering of tasks, but also any synchronization requirements as well. This is often achieved by embedding different "split" and "join" logics, which allows the transitions between steps in a workflow to be deterministic at run time.

   (b) Explicit sequence control – The workflow author should be able to model multiple tasks that execute in parallel or in sequence.

(c) Inter-process communication – The workflow author should be able to model channels of communication between tasks that are executing simultaneously.

(d) Acquisition of resources – The workflow author should be able to acquire resources for shared or exclusive use.

(e) Exceptional control flow – The workflow author should be able to cleanly and succinctly model behavior where errors have occurred.

2. Ease of use – The language must be easily understandable by someone in the medical field. If the language is highly expressive but is too complex to be rapidly learned and understood by workers in industry, then it is ultimately of little value. Uses one and two in the Workflow in section 1.2 (communication of best practices from either domain experts or device manufacturers to medical workers) are directly contingent upon workflow models being adopted by non-technical medical professionals, and ease of use is critical to their adopting any language.

3. Tool support – The language must have a rich tool set in place to support the creation, maintenance, and preferably execution of workflows. Ideally this tool would also be easily used by medical personnel, since this would lead to domain experts requiring less training and non-domain professionals providing better feedback, editing, and revisions. Better tool support would also translate into users being more likely to continue using the workflow modeling language, leading to increased standardization over time. This support might also include a model checker / other verification system integrated with a given execution environment.

Note that another means of evaluation include the multitude of workflow patterns explored by W.M.P. van der Aalst et al[29].

There were a number of potential workflow modeling languages we considered:

- BPEL[4] – An XML standard for describing interactions between a business process and

a web service. It has no set graphical notation, although BPMN is partially capable as serving as a graphical layout of a BPEL workflow.

- Little-JIL[34] – Little-JIL refers to itself as an agent coordination language. It focuses on coordinating the actions of various agents, which are defined in an environment external to Little-JIL.

- jABC[25] – jABC is developed at the University of Dortmund. It is an all-purpose "Java Application Building Center," allowing computer programmers to define nodes which are linked together by non-technical domain experts. These nodes are backed by java classes, and can be linked together graphically in the supplied editor.

- BPMN[12] – BPMN is a standardized notation designed to build diagrams of business processes. There are a wide range of graphical editors, though, and while some attempts at formalizing BPMN's semantics have been made, they greatly reduce the language to a less-expressive subset[8]. It would likely be extremely difficult or impossible to formally define the entirety of the BPMN specification.

- YAWL[28] – YAWL was created to be able to easily support nearly all of the 26 original workflow patterns[29]. It is an actively-developed, free, open-source workflow editor and includes a process server capable of executing workflows. There also exist basic model-checkers for its control aspect.

Three of these technologies were selected for a more close examination: BPMN, Little-JIL, and YAWL (see the section below).

2.3 Workflow Language Examples

This section contains an example of a process represented in three different workflow modeling languages. In this process, a nurse checks a patient to see if her vital signs are within

an expected range (see Figure 1). If they are, then he administers a dose of medicine A. If her vital signs are abnormal, however, a doctor is notified while medicine B is delivered.

2.3.1 BPMN

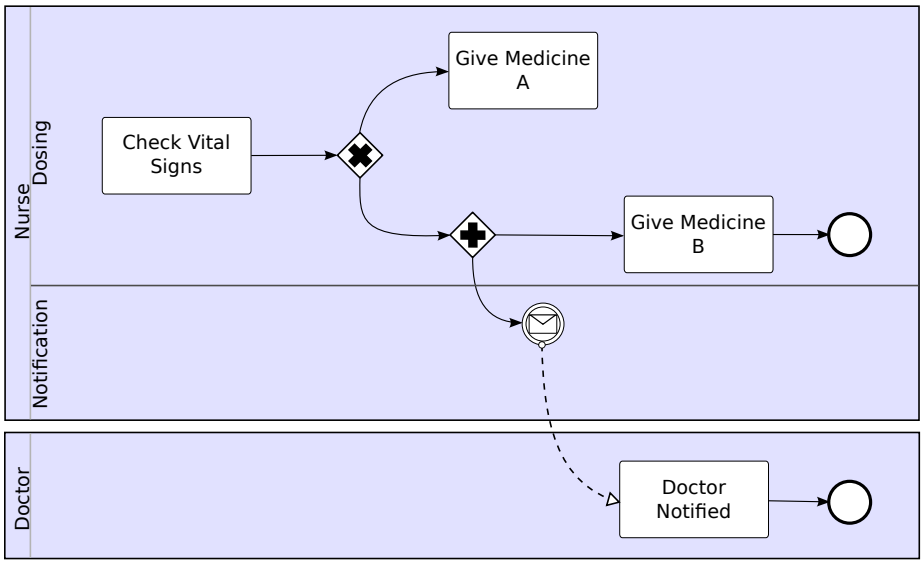


Figure 2.1: *A simple BPMN diagram.*

Business Process Modeling Notation (BPMN – see Figure 2) is a modeling language created by the Business Process Management Initiative (BPMI) for standardizing the many diagram formats used in business[31]. It is now supported by the Object Management Group (OMG), which merged with BPMI after the creation of the language. The language is made up of graphical elements that are similar to those found in flow charts. Diagrams in BPMN are not meant to be executable.

Though there are four types of constructs in BPMN, many simple diagrams can be represented with only the first two types (flow and connecting objects). The four types of constructs are:

- Flow Objects – These are objects that represent the steps in a workflow. There are three types of flow objects:

  - Activities – Rounded rectangles represent activities. An activity is a "task" or unit of work in the workflow. Tasks with decompositions (that is, tasks which themselves are made up of subtasks) are referred to as sub-processes and are denoted by a small + on the rounded rectangle.
  - Events – Circles represent events e.g.: a message being sent or received. There are three types of events:
    * Start Events – These are events that start a series of steps when they occur.
    * Intermediate Events – These are events that occur mid-flow, and signal neither the start nor the end of a workflow. They can, however, affect which branch a workflow takes.
    * End Events – These events signal that a workflow has completed, and should be terminated.
  - Gateways – Diamonds represent gateways. These can be either decision points (which direct the flow through a process) or control points (which manage the joining and splitting of flow paths). Symbols on the diamond specify the type of gateway, e.g.: a + denotes a parallel gateway.

- Connecting Objects – These are branches that connect the flow objects. There are three types of connecting objects:
  - Sequence Flow – The typical flow connector (represented by a solid line), this indicates the order in which flow objects are to be executed.
  - Message Flow – A message flow (represented by a dashed line) indicates communication between different entities or roles in the diagram.
  - Association – An association (represented by a dotted line) specifies that a given artifact belongs to the indicated flow object.

- Swimlanes – Swimlanes are used to denote the responsibilities of actors in a BPMN workflow. There are two types of of swimlanes:
  - Pool – A pool contains all tasks a process participant is responsible for. In the diagram, the two pools are named "Nurse" and "Doctor."
  - Lane – A lane is a subsection of a pool, and is used solely for organizing activities; it does not denote a different participant. In the diagram, the Nurse pool has two lanes, denoted "Dosing" and "Notification." The Doctor pool has only one (anonymous) lane.
- Artifacts – Artifacts are graphical enhancements to BPMN diagrams, and do not affect the flow through a process. The set of artifacts is intended to be tailored to the application. The set of artifacts included in the BPMN 1.0 standard are data objects, groupings, and comments / annotations[31].

As BPMN is a business-oriented standard[12], tool support for the editing of BPMN diagrams is extensive. These tools range from components of commercially developed systems[27] to freely available standalone diagram editors[5]. No tools which allow simulation / verification of a BPMN model are freely available.

BPMN is highly readable, and its standardization is one of its selling points. Its similarity to existing flow diagrams, and its compatibility with Unified Markup Language (UML) make it quite desirable, unfortunately there are no formally defined semantics. Though there have been attempts to define formal semantics for BPMN, they are defined for only a subset of the language, and there is no tool support for this strict subset[8]. This makes BPMN essentially impossible to simulate or model check, which means that it is not a strong candidate.

2.3.2 Little-JIL

Little-JIL is (see Figure 3) defined by its creators as an agent coordination language[33]. It was designed by the Laboratory for Advanced Software Engineering Research (LASER)

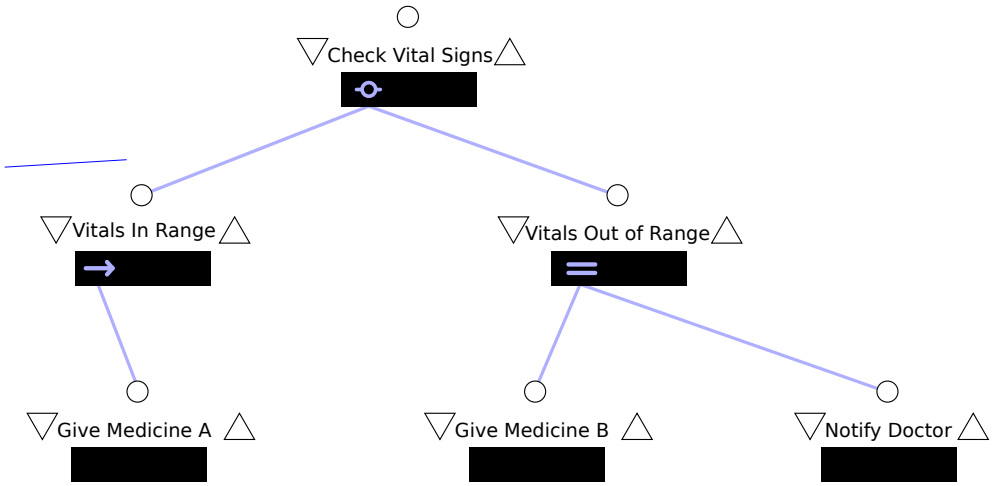


Figure 2.2: *A simple Little-JIL diagram.*

group at the University of Massachusetts at Amherst (UMass) as an easily-read graphical tool to support process programming by people who are not computer scientists. It has formally defined semantics and is supported by a set of analysis tools as well as a graphical editor.

There are two basic elements in Little-JIL: agents and steps. Steps are laid out (with the Visual-JIL plug-in for the Eclipse IDE) graphically and represent units of work or tasks. A group of steps combine as a tree into an agenda, which is assigned to an agent. Agents are responsible for (attempting to) execute steps, and can be human but are not necessarily, as tasks could potentially be completed by a computer or mechanical system.

- A step represents a single task.
  - A badge on the left side of a step denotes the execution sequence of its children. Execution sequences can be sequential ($\rightarrow$), parallel ($=$), mutually exclusive ($\twoheadrightarrow$), or try-until-success ($\circ$).
  - Steps can be external to the current agenda, and groups of external steps are referred to as modules.
  - Steps can have parameters, which are modal: a parameter can be designated for input, output, input and output, or local/temporary uses.

  - Steps possess a cardinality which denotes the number of times the step should be executed. This cardinality can be known at design time or determined dynamically.
  - Steps have pre- and post-requisites (as well as predicates). Requisites are denoted by triangles to the right and left of node names.
  - Steps can have deadlines, which are written with clock hands on the interface circle above a step's name. Deadlines are given as a length of time relative to a step's enablement.

- Resources are written as call-outs with a line connected to the circle above the name of a given step. Resource use, acquisition, and collections are all denoted by manipulations of a circle icon. Note that agents can be passed as resources.

- Channels are written in a similar fashion to resources, and allow non-hierarchical communication between nodes.

- Potentially thrown exceptions are denoted with badges on the right side of steps. Exception handlers are linked to this badge much the same way sub-steps are linked to a node's sequencing badge.

- Little-JIL also has message passing / handling. Messages that are sent are denoted with a lightning bolt attached to a call-out coming from the interface circle. A message handler has a lightning bolt badge in the center of the step (between the sequencing and exception badges) which is linked via call-out to the message handler.

Two tools for working with Little-JIL have been developed by UMass. The first, Visual-JIL, is an Eclipse plugin which enables a process programmer to build a Little-JIL program[34]. The second is a set of two tools for verifying Little-JIL programs. The first tool is a finite state verification system, and the second is a fault tree generator / analysis system[3,10]. The agent environment UMass uses is called Juliette and is not publicly released.

Little-JIL has none of the problems that BPMN has – Little-JIL's semantics are well defined and rich. Interesting work has been done with Little-JIL[34], but it has not taken off in industry. This may be due to the structure of Little-JIL diagrams: though information-rich, they can be difficult to understand quickly. Interpreting a large set of tasks can pose a challenge when they are presented in a control-flow order, but can be outright confusing when presented as a tree designed to be traversed in preorder. Further, since the only denotation of sequencing is done with a small badge on the left side of a step, parallel, exclusive-choice, and sequentially ordered agendas all look virtually identical at first glance, and following an execution process requires a steady concentration. Since we ultimately want to work with members of industry who will need to pick up our notation quickly and since we desired the ability to simulate workflow models, we decided that Little-JIL was unfit for our purposes.

2.3.3 YAWL

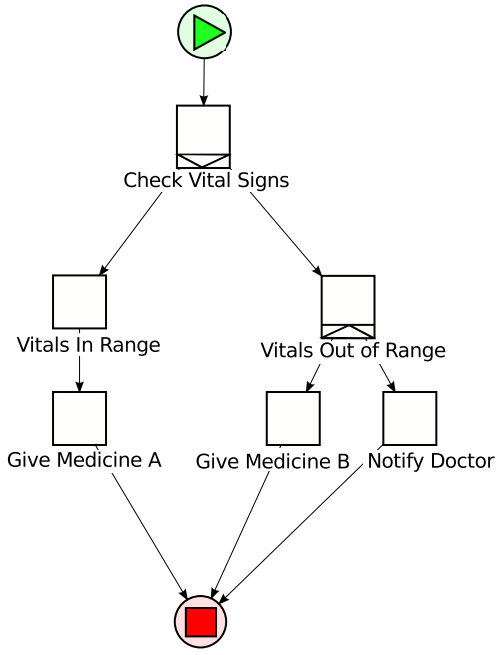


Figure 2.3: *A simple YAWL diagram.*

Yet Another Workflow Language (YAWL) was developed in 2002 as a joint project be-

tween professors at the Eindhoven University of Technology and the Queensland University of Technology[1]. It was built with two chief considerations: support for the common workflow patterns[29], and a rigorous set of semantics based on colored (or stateful) petri-nets. The makers of YAWL had to modify colored petri-nets in novel ways to support all of the workflow patterns, and as YAWL is still under heavy development, they continue to extend their semantics to support new features.

There are three perspectives in YAWL[22]:

- Control-flow Perspective – This is the perspective most often thought of when the term workflow is used, and it is also the main perspective of the YAWL editor. YAWL workflows are organized into nets, which are essentially pages. The basic constructs of the control-flow perspective are[1]:
  - Conditions – Known as places in the language of Petri nets[28], conditions are akin to resting states. Each workflow has two special conditions – an input condition which is where the control flow begins, and an output condition that, when reached, terminates the workflow. Places are indicated as circles on YAWL nets.
  - Tasks – Tasks are similar to the Petri net concept of transitions. Tasks are where the actual work of the workflow gets described, and are denoted as squares. Note that unlike in Petri nets, it is not necessary to have a condition between tasks as an implicit condition can be created if tasks are connected directly. A task will also be:
    - Composite or Atomic – An atomic task is entirely self-describing; it is not an abstraction for a sub-process. A composite task, on the other hand, is merely an abstraction. A composite task is mapped to a separate workflow (defined on a different net) which contains the steps necessary to complete it.

    * Single or Multiple Instance – A single instance task will occur once and a multiple instance task can occur multiple times. The semantics for controlling the number of simultaneous instances are specific enough to allow minimum and maximum numbers of instances, minimum numbers of instances that must complete before the task itself is considered complete, and continuation thresholds.

  - Flows – Akin to arcs in YAWL parlance, control flows are indicated as solid, directional lines with arrowheads. The splitting and joining of flows is explicit in YAWL, and there are three types of splits and joins:

    * AND – AND splits enable all outgoing flows in parallel. Similarly, an AND join will wait for all incoming flows to be activated before the task it precedes will execute.

    * XOR – XOR splits enable a single outgoing flow, and the others are never enabled. XOR joins initiate the task attached to them as soon as any of the incoming flows are enabled.

    * OR – OR splits trigger some outgoing flows. Each outgoing flow has a condition attached to it (e.g.: myVar > 7), and each flow's condition is tested. Each flow with a true condition will be enabled. OR joins block until all incoming flows are either enabled or deadlocked (that is, they will never be enabled).

  - Cancellation Regions – Denoted by a dashed line around a set of nodes, a cancellation region allows for a set of tasks to be canceled after a certain task completes execution.

- Data Perspective – Recognizing that workflows in the real world rely on data to guide decisions, the creators of YAWL built a data perspective into the YAWL editor. The base of the data perspective is the decomposition[1]. A decomposition is the work

required to complete a task, and they use parameters which are taken as input and produced as output. Parameters are akin to variables in traditional programming, and have many of the same characteristics, such as a name, a type, a scope, and a designation. A parameter's designation is identical to the concept of modality in traditional programming: a parameter may be an "in" variable, an "out" variable, an "in/out" variable or for local use only. Decompositions can be small XPath or XQuery strings, references to web services, or even java classes.

- Resource Perspective – In industry, assignments to workers are often based on their roles, capabilities, memberships in organizations, etc.. In order to best model this, the creators of YAWL implemented the resource perspective which allows a workflow designer to define a range of organizational structures, and to let the system assign tasks automatically using a person's membership in these structures as criteria. The four types of organizational structures are[1]:

  - Roles – A role is a job to be performed by a given user. Note that a user can have multiple roles, and that roles can be part of other roles (e.g.: the role of nurse's assistant belongs to the role of nurse).
  - Capabilities – A capability is simply a flag that a user has some skill (e.g.: a nurse has pediatric training).
  - Positions – A position is a set job in a set hierarchical structure (e.g.: the nurse's assistant reports to the chief of nursing who reports to the director of a hospital).
  - Organizational Groups – Organizational groups are groups of positions. Organizational groups can contain positions as well as other organizational groups (e.g.: the nursing group would contain the positions of nurse's assistant and chief of nursing, and be contained by the hospital group).

It should also be noted that what YAWL terms resources are often also called agents.

They are not resources that are used or consumed in a process, but rather things that are capable of completing tasks independently.

There are a broad range of tools designed to work with YAWL – the official distribution includes both an editor for YAWL nets and an engine to allow them to be executed. The editor is the reference implementation, and supports all currently available language features. The engine allows an organization to execute workflows by posting tasks to user's worklists, executing automated tasks, and keeping track of organizational data. There are also a number of services designed to work with the engine, ranging from interfaces with Twitter to digital signature validators[1]. Further, there are tools which are external to the main YAWL distribution but which coordinate with it to, e.g.: run simulations and analysis on data derived from YAWL execution traces[35].

YAWL marries the benefits of the two previously discussed technologies (Little-JIL and BPMN): it has formally defined semantics as well as an easily understandable graphical notation. There are numerous cases of it being used in a range of industries (e.g.: the film industry[23]), and its facilities for logging and simulation are very exciting from an operations management perspective. Though YAWL lacks some desirable features (e.g.: inter-process communication) it is the best notation available.

2.4 Symbolic Execution

Note that this section is reprinted, with modifications, from the SAnToS technical report "Efficient and Formal Generalized Symbolic Execution" by Xianghua Deng, Jooyong Lee, and Robby[7].

King initially came up with the idea of symbolic execution in 1976[16]. The essential idea of symbolic execution is demonstrated on a simple imperative method in Figure 2.4. In contrast to concrete (traditional) execution, symbolic execution reasons about all possible values when a concrete value isn't known. These unknown values are represented as symbols instead of e.g. integers.

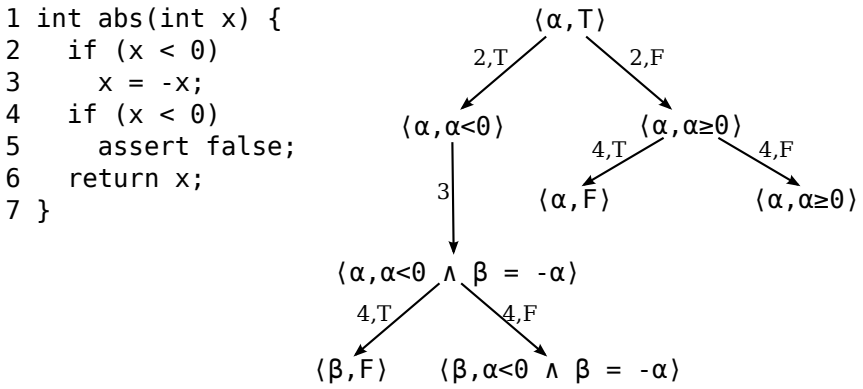


Figure 2.4: *An imperative program and its symbolic execution tree*

In the example, each tree node is a symbolic state $\langle x, \phi \rangle$. The first term is a variable identifier (x, in this case) and the second is a predicate ϕ which constrains the value of x. As there is no initial information about x, the predicate is TRUE, and there are no constraints on ϕ. When execution arrives at line two, there isn't enough information to know which execution path to take, thus both paths are explored and appropriate constraints are added to the symbolic state.

As execution progresses, the constraint is augmented by predicates which correspond to the logical condition that caused the particular branch to be explored – thus ϕ is often referred to as the path condition because it fully describes the conditions that must be met for an execution trace to reach the current state. Note that if a constraint ever becomes false, that means that that branch is no longer logically consistent (that is, it's unreachable) and as such can be discarded. In our example, line five is unreachable.

Decision Procedure: Decision procedures are used to determine which branches are infeasible, and which should be followed. Similarly, they are also used to determine when a constraint is technically valid (i.e., is a valid program state) but is inconsistent with stated properties (i.e., an error has been detected).

Termination: One of the major challenges with symbolic execution is (the potential lack of) termination. State matching, a common method for detecting duplicate states, is not

immediately applicable since additional constraints (which have no effect) may be added by an additional iteration of a loop or an additional recursive call. Instead of testing for equality, however, testing (by the decision procedure) can be done for *state subsumption*. That is, if a state includes entirely a smaller state, then the former is said to *subsume* the latter. As an example, consider the situation where we have two states: $\gamma = \langle x, x > 3 \rangle$ and $\delta = \langle x, x > 5 \rangle$. In this situation, γ subsumes δ – that is, every possible state δ encompasses is also encompassed by γ, and as such, δ can be disregarded.

Chapter 3

Kinerja

Kinerja (sometimes written / abbreviated as $\varkappa$) is a software model-checker, designed and developed by the author, for workflows written in YAWL-formatted XML. Workflows can thus be designed in the YAWL-editor, and then executed (in one of multiple modes) in Kinerja. This approach is novel in a number of regards, primarily in the ability to verify a workflow and then immediately execute it in a human-guided governance mode in the same engine. This allows for increased confidence in the results of the model checker, as the execution trace which is used to prove or disprove properties in verification mode can be mapped directly to a set of inputs which, when given as input to Kinerja's governance mode, will recreate the execution trace exactly.

3.1 Design Overview

Kinerja's execution begins by parsing its input, and transforming it into a valid colored petri-net. Parsing is currently only implemented for YAWL, and is responsible for enforcing a number of the language's definitions and requirements (see section 3.2.2).

Depending on the execution mode, the control-aspect is run via different implementations (see figure 3.1). Note that there are two key differences between the execution modes: the source of input and the use of subsumption / state matching.

- Source of input: In verification mode, the engine exhaustively checks all possible

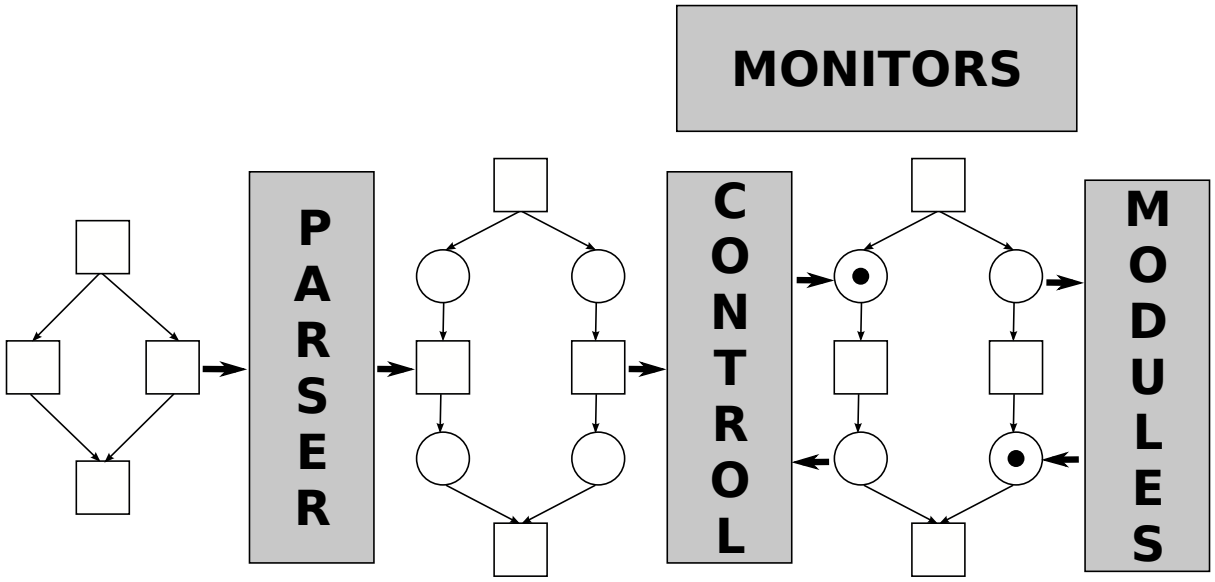


Figure 3.1: *The high-level architecture of Kinerja.*

states. Thus, input is automatically generated, and is symbolic rather than literal. In governance mode, the engine polls external sources for its input – sources can include humans or other machines.

- Use of subsumption: As soon as a loop has been detected, Kinerja will, in verification mode, back out and resume execution after the loop's completion. In governance mode, however, there is no checking for looping, and execution will continue as long as input guides it in a repetitive path.

Execution in Kinerja consists of two distinct phases which explicitly treat two aspects of workflow-modeling: the control aspect (which governs moving execution according to a program's various control-flow paths) and the data aspect (which governs the state of a workflow's variables). The data aspect is implemented as an additional "module" – while other aspects (such as one that might track resources or agents) are possible, they have not been implemented. On top of these two phases sit Kinerja's monitors, which watch all stateful aspects of execution.

3.2 Description of YAWL-Subset

3.2.1 Informal Description

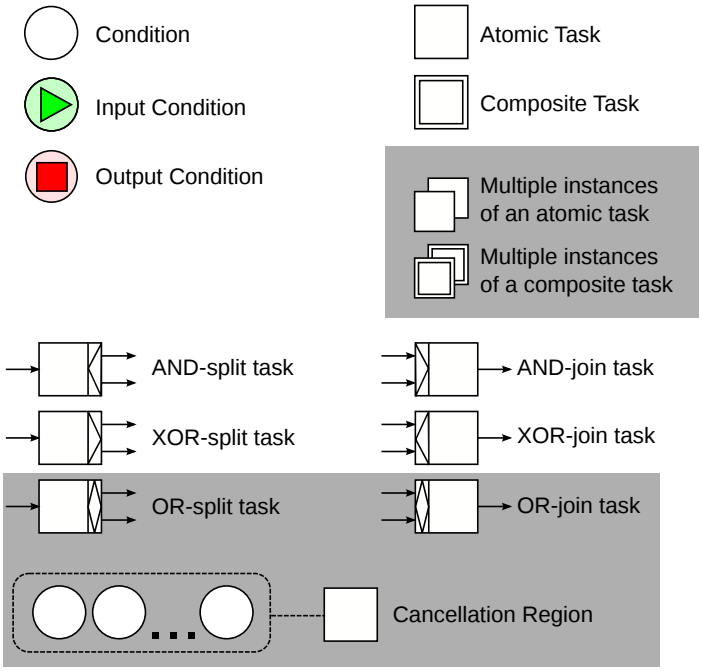


Figure 3.2: *The subset of YAWL*[28] *that Kinerja supports.*

Kinerja supports a majority of YAWL's features, as shown by the unshaded areas of the above figure. Or-joins and cancellation regions were not supported because their implementation requires non-local semantics. Multiple-instance tasks may be supported at some future point, but they were not required by our working examples, and became too low of a priority to be included in the initial scope.

3.2.2 Formal Description

Figure 3.3 shows the YAWL example from Chapter 2, unfolded with implicit conditions and collector tasks. These implicit conditions are added by the Kinerja parser, and this will be the format of examples in this section.

Note that these definitions are reprinted, with modifications, from "YAWL: Yet Another Workflow Language" by W.M.P. van der Aalst and A.H.M. ter Hofstede[28].

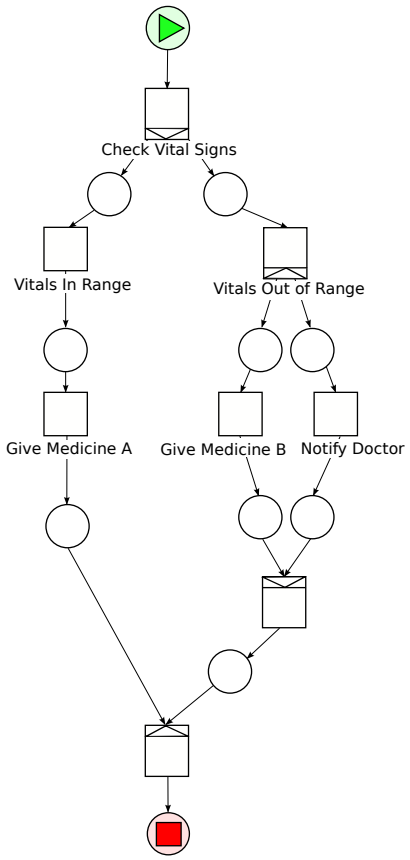


Figure 3.3: *The YAWL example from chapter two, as parsed*

Definition 1: An extended workflow net N is a tuple $(C, i, o, T, F, split, join)$ such that:

- C is a set of conditions,
- $i \in C$ is the input condition,
- $o \in C$ is the output condition,
- T is a set of tasks,
- $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation,
- every node in the graph $(C \cup T, F)$ is on a directed path from i to o,
- $split : T \rightarrow \{AND,\ XOR\}$ specifies the split behavior of each task, and
- $join : T \rightarrow \{AND,\ XOR\}$ specifies the join behavior of each task.

Figure 3.4 highlights the various parts of Definition 1. As the first definition, it simply lays out the most basic parts of a workflow – the nodes, arcs, and rules governing their joining. Note the special conditions i and o, which are the net's input and output conditions, respectively.

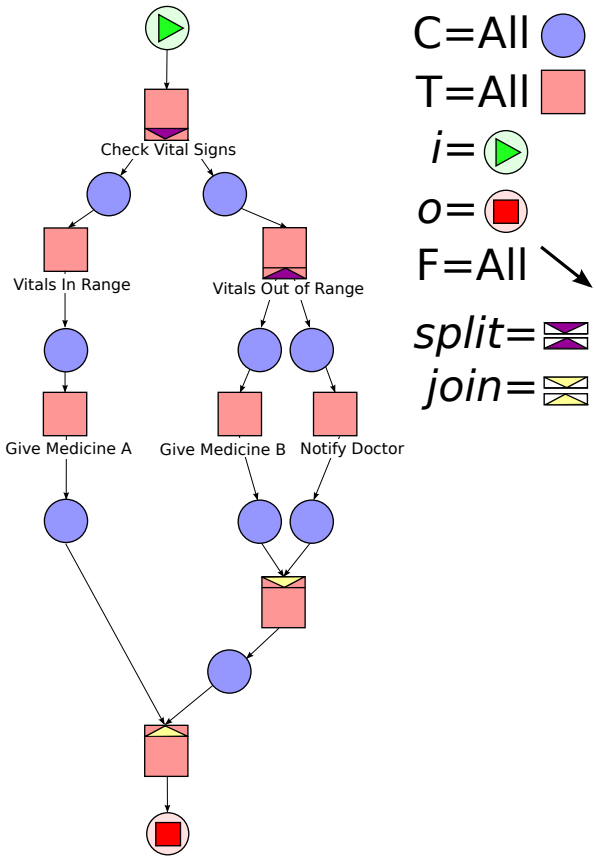


Figure 3.4: *The YAWL example, highlighting definition one*

If you're familiar with YAWL, you may notice some things missing (see Figure 3.2) – these features were removed for a number of reasons, and will not function correctly if used with Kinerja.

Definition 2: A workflow specification S is a tuple $(Q, top, T^{\diamond}, map)$ such that:

- Q is a set of EWF-nets,
- top $\in Q$ is the top level workflow,
- $T^{\diamond} = \cup_{N \in Q} T_N$ is the set of all tasks,
- $\forall_{N_1, N_2 \in Q} N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$ i.e., no name clashes,
- map: $T^{\diamond} \not\rightarrow Q \setminus \{top\}$ is a surjective injective function which maps each composite task onto an EWF net, and
- the relation $\{(N_1, N_2) \in Q \times Q | \exists_{t \in dom(map_{N_1})} map_{N_1}(t) = N_2\}$ is a tree.

This defines how multiple workflow nets can be combined into a single workflow. I make the simplifying assumption that there is no nesting for workflows in Kinerja, so to enforce this the parser it employs "flattens" input workflows. This allows workflows to still use nesting when being designed, but effectively eliminates the constraints of this definition from the Kinerja engine.

Definition 3: Let $N = (C, i, o, T, F, split, join)$ be an EWF-net. $C^{ext} = C \cup \{c_{(t_1,t_2)} | (t_1, t_2) \in F \cap (T \times T)\}$ and $F^{ext} = (F \setminus (T \times T)) \cup \{(t_1, c_{(t_1,t_2)}) | (t_1, t_2) \in F \cap (T \times T)\} \cup \{c_{(t_1,t_2)} | (t_1, t_2) \in F \cap (T \times T)\}$. Moreover, auxiliary functions $\bullet\_, \_\bullet : (C^{ext} \cup T) \to \mathbb{P}(C^{ext} \cup T)$ are defined that assign to each node its preset and postset, respectively.

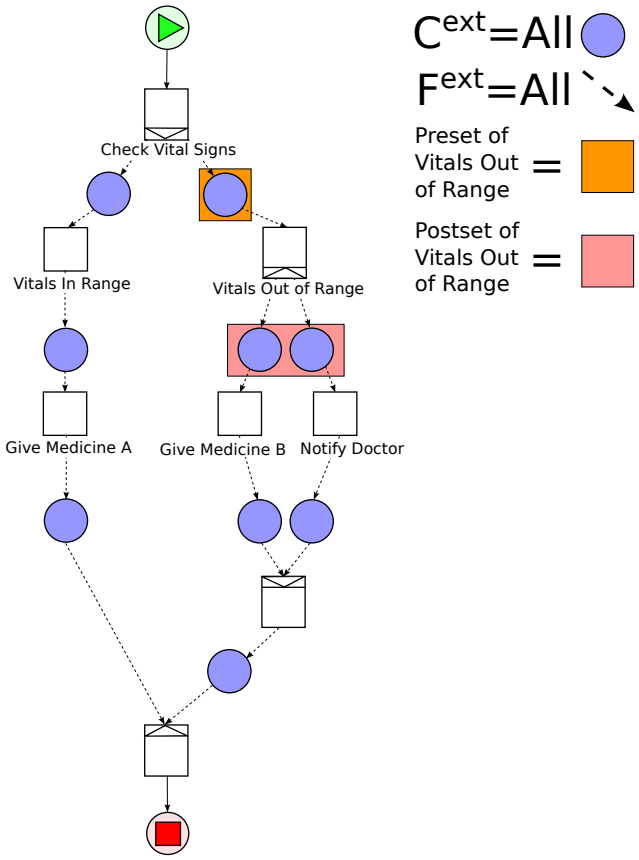


Figure 3.5: *The YAWL example, highlighting definition three*

Definition 3 accomplishes two things. First, it explains how to generate a valid colored

petri net from a YAWL workflow by creating implicit conditions (states) between tasks (transitions), and updating F, the flow relation, accordingly. It also defines two functions: $\bullet\_$, and $\_\bullet$, which map a node (either a condition or task) to its preset and postset, respectively.

Note that in our example (Figure 3.5), nearly every old element in F has been replaced – the only elements which persisted are the arcs which were in contact with either i or o. Also note that, like Definition 2, the constraints of this definition are handled by Kinerja's parser, so the engine itself only deals with valid petri nets.

Definition 4: Whenever we introduce a workflow specification $S = (Q, top, T^\diamond, map)$, we assume $T^A, T^C, C^\diamond$ to be defined as follows:

- $T^A = \{t \in T^\diamond | t \notin dom(map)\}$ is the set of atomic tasks,
- $T^C = \{t \in T^\diamond | t \in dom(map)\}$ is the set of composite tasks, and
- $C^\diamond = \cup_{N \in Q} C_N^{ext}$ is the extended set of all conditions.

This definition is straightforward – for the purpose of brevity in future discussions, it defines:

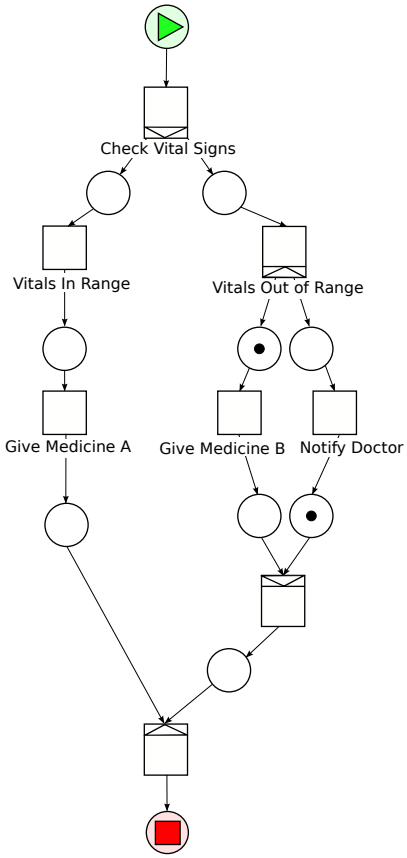
- T^A: The set of all atomic tasks (across all workflow nets),
- T^C: The set of all composite tasks (across all workflow nets), and
- $C^\diamond$: The set of all conditions (across all workflow nets).

Definition 5: Let $S = (Q, top, T^\diamond, map)$ be a workflow specification. We define the function unfold: $\mathbb{P}(T^\diamond \cup C^\diamond) \to \mathbb{P}(T^\diamond \cup C^\diamond)$ as follows. For $X \subseteq T^\diamond \cup C^\diamond$:

$$unfold(X) = \begin{cases} \emptyset & \text{if } X = \emptyset \\ unfold(X \setminus \{x\}) & \text{if } x \in X \cap (C^\diamond \cup T^A) \end{cases}$$

Definition 5 explains how to do the unfolding of composite tasks – which is handled by Kinerja's parser. It's straightforward – composite tasks are mapped to (and replaced by) the nets they stand in for.

Definition 6: A workflow state s of a specification $S = (Q, top, T^{\diamond}, map)$ is a set over $Q^{\diamond} = C^{\diamond}$, i.e., $s \in Q^{\diamond}$.



$s = \bullet$

Figure 3.6: *The YAWL example, highlighting definition six*

This defines the concept of a workflow's state: the set of all *tokens* in a workflow. A *token* can be thought of as a marker on a given condition – that is, in Figure 3.6 execution began in i, proceeded through *Check Vital Signs* (where it splits) and a doctor has been notified but Medicine B has not yet been given.

To advance the state of a workflow, a token moves along an arc, through a transition (if such a move is valid – see the next definition), splits if necessary, and then moves along the transition's outgoing arc(s), coming to rest in the transition's postset.

Definition 7: Let $S = (Q, top, T^{\diamond}, map)$ be a specification and $t \in T^{\diamond}, c, p, s \in Q^{\diamond}$. The Boolean function $binding(t, c, p, s)$ yields true if and only if some $n \in \mathbb{N}$ exists such that all of the following conditions hold:

1. Tokens to be consumed are present in the state: $c \subseteq s$
2. Tokens are consumed from the input conditions of the task involved and at most one token can be consumed from each condition in the preset: $c \subseteq \bullet t$
3. Tokens are produced only for output conditions of the task involved and at most one token can be produced for each condition in the postset: $p \subseteq t\bullet$
4. For AND-join behavior, all input conditions need to have tokens: $join(t) = AND \Rightarrow c = \bullet t$
5. For XOR-join behavior, only one input condition should have a token and that input condition should not have more than one token: $join(t) = XOR \Rightarrow c$ is a singleton
6. For AND-split behavior, tokens are produced for all output conditions of the task involved: $split(t) = AND \Rightarrow p = t\bullet$
7. For XOR-split behavior, a token is produced for exactly one of the output conditions of the task involved: $split(t) = XOR \Rightarrow p$ is a singleton

Definition seven governs how a token can advance from one condition to the next. The explanations are fairly straightforward, and Figure 3.7 shows what each rule enforces:

Definition 8: Let $S = (Q, top, T^{\diamond}, map)$ be a specification and s_1 and s_2 be two worfklow states of S. $s_1 \rightarrowtail s_2$ if and only if there are $t \in T^{\diamond}, c, p \in Q^{\diamond}$ such that $binding(t, c, p, s_1)$ and $s_2 = (s_1 - c) \uplus p$.

Definition 8 simply gives a convenient notation ($\rightarrowtail$) for denoting the transition between two states. That is, if s_1 can transition into s_2 (i.e., it meets all the requirements) then one can write $s_1 \rightarrowtail s_2$.

3.3 Evaluation Metrics Revisited

Recall that in section 2.2 a set of desiderata (spanning the range from order qualifiers to order winners) were described. These are now revisited:

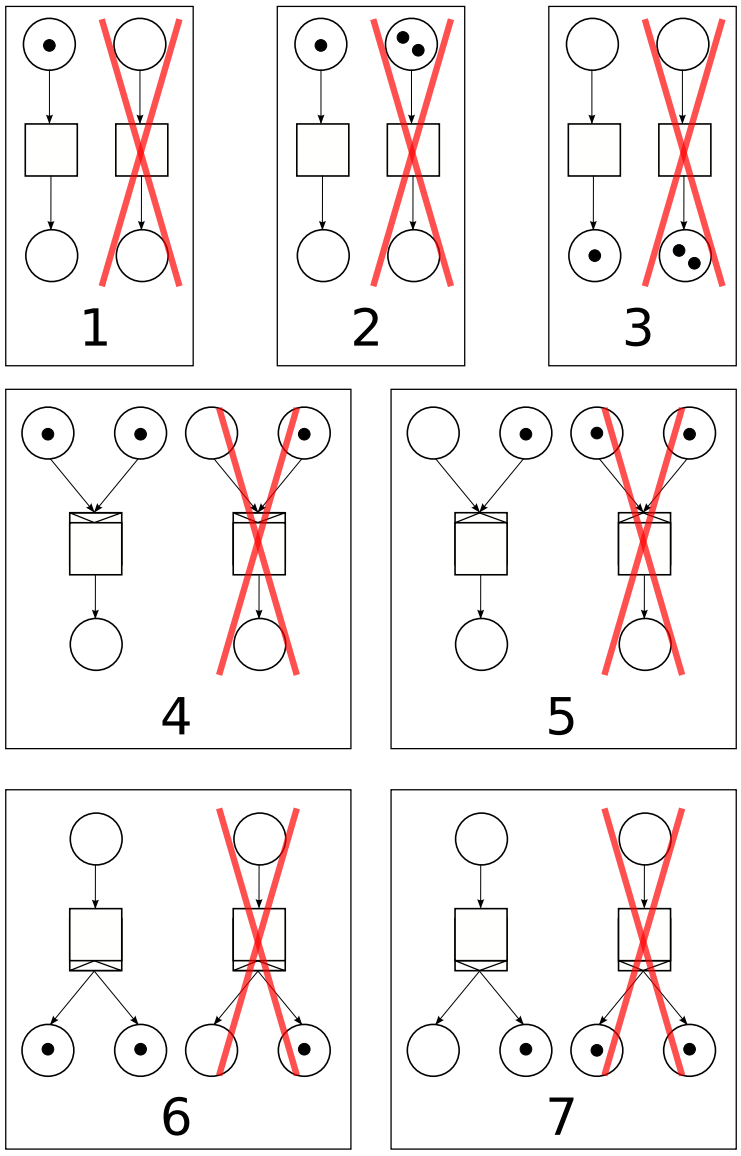


Figure 3.7: *An example of what each of definition seven's rules enforce*

- Expressiveness – Leveraging a subset of YAWL, Kinerja's supported language meets most, but not all, of the expressiveness goals:
  - Explicit flow control – Discussed in depth in the previous section, the flow control options in the supported subset of YAWL include AND and XOR splits and joins. This is satisfactory, but as will be discussed later (see Section 4.2.3) the lack of an OR join and split is noticeable.

  - Explicit sequence control – The subset of YAWL that Kinerja treats supports this.
  - Inter-process communication – This is not explicitly supported. Communication is supported, but only through the transfer of a token (which also transfers a notion of execution) making the modeling of inter-process communication less than intuitive.
  - Acquisition of resources – This is not supported, but due to Kinerja's extensible architecture, may be added later.
  - Exceptional control flow – This is not explicitly supported. Some level of exception handling can be modeled, but exceptions are not a native concept in YAWL. See section 4.1.1 for a more thorough treatment of this topic.

- Ease of use – YAWL's structure makes it straightforward to read, and similar to the basic hand-drawn format many people use on whiteboards. Additionally, YAWL has been previously used in a clinical context[14].

- Tool support – While the editor for YAWL is sufficient, the execution environments were focused in a different direction than we were interested in. Kinerja, however, fulfills the tool support goals by fully integrating a model checker with an execution environment.

3.4 Control Aspect

The control aspect of a workflow can be thought of as an imperative description of a workflow[19]. It affords no consideration for the value of the variables of the workflow, and thus any guards which would steer execution are ignored. Similarly, there is no consideration given to resourcing requirements regardless of any limits they may place on the execution of a workflow. This means that a number of the states produced by the control aspect are illogical; i.e., they cannot exist.

3.4.1 Control flow

There are three particularly important methods in the implementation of the control aspect in Kinerja. The first, DFS, governs the execution of the model-checker as it exhaustively explores (via a depth-first search, as the name implies) the entire state-space of the program under review. The second, ENABLED, determines what steps can succeed the current state. The third, SEENBEFORE, determines whether or not the current state has been "seen" by the model checker previously.

Note that algorithms are presented in two forms: in pseudocode, and in the form of an *execution tree*, that is, the possible execution steps the algorithm can take. These trees are annotated with letters corresponding to their potential states – these states are summarized to the right of the tree, and more completely explained in the text below the diagram.

DFS

This is the central, driving algorithm which propels the entire model-checker. At a high level, it takes as an argument a description of the initial state of the workflow, explores it completely and continues until all states have been explored. The depth-first search method is presented in pseudocode in algorithm 1:

Algorithm 1 The DFS algorithm

Require: $seen := \{state_0\}$
Require: $stack := [state_0]$
1: **while** $stack \neq \{\}$ **do**
2: $state := stack.\text{POP}()$
3: $workSet := \text{ENABLED}(state)$
4: **for all** $step \in workSet$ **do**
5: $state' := \text{EXECUTE}(step)$
6: **if** $state' \notin seen$ **then**
7: $stack.\text{PUSH}(state')$
8: **end if**
9: **end for**
10: **end while**

The *seen* set stores sets that the model checker has already seen, and that are either

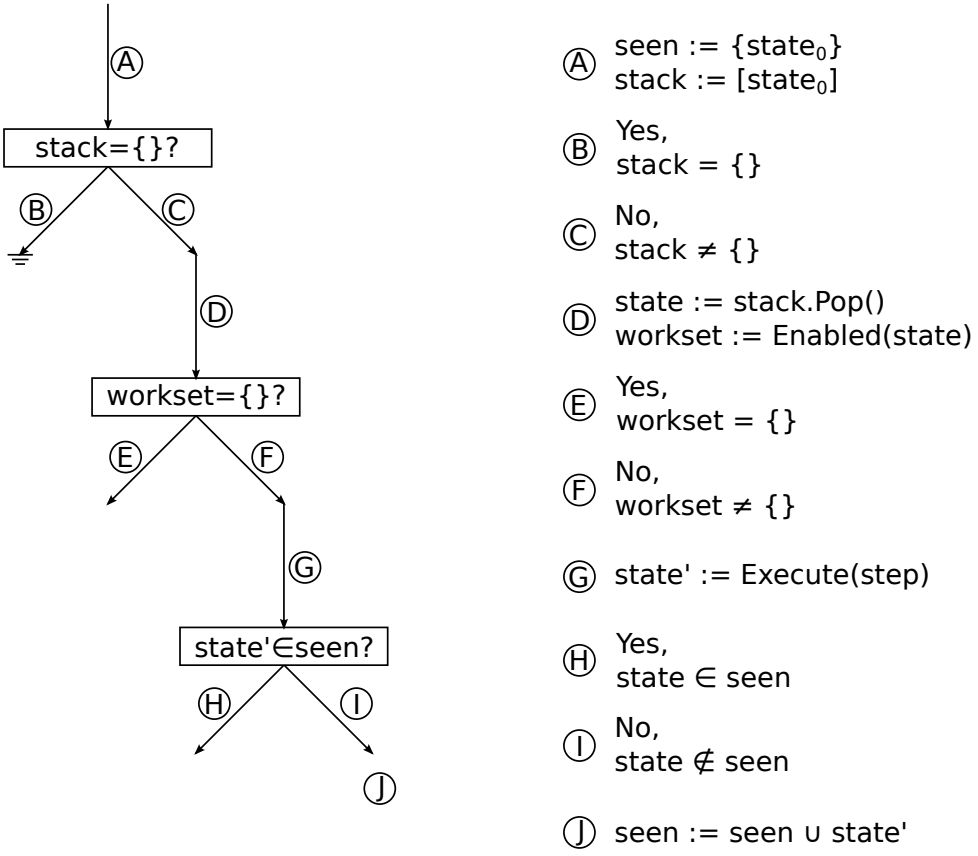


Figure 3.8: *The execution tree for the* DFS *algorithm*

fully explored or marked for future exploration. The *stack* is a stack containing states to explore. We use a stack (as opposed to a queue or other data structure) to facilitate the depth-first search strategy. The actions performed by the algorithm are (see figure 3.8):

(A) The *seen* and *stack* variables are initialized to their preliminary value: the workflow's starting state.

(B) If the stack is empty (this will never happen the first time) the search is complete and the method terminates.

(C) If the stack is nonempty, there remain unexplored states, and execution continues.

(D) The current state is set to the top of the stack, and the possible steps resulting from the current step are enumerated (see algorithm 2).

(E) If there are no possible steps to take from the current state, it is completely explored and the algorithm is ready to examine another state.

(F) If there are possible steps to take from the current state, they are iterated over, and execution begins on one of them.

(G) The current step is executed (see algorithm 4) which results in an updated version of the current state.

(H) If the current state has already been seen (see algorithm 3), then it has (necessarily) already been explored or added to the stack. Thus, we can continue execution on the next step.

(I) If the current state has not been seen, we continue execution.

(J) The current state is added to the stack, and execution continues.

Figure 3.9 shows an execution of the DFS algorithm on a simple workflow. The initial state of the workflow is shown on the left (in the bold-framed box) and the possible resulting states are shown as smaller workflow states. Since the behavior of the algorithm depends on the type of split and join used by the executed task, there are a number of possible outcomes.

Enabled

This algorithm determines which steps are possible from the given workflow state. It requires checking both the control aspect (i.e., which tasks link to which other tasks) and data aspect (i.e., whether a workflow-net's variables are in the correct state to allow entry into a given child-task) of the current state. Note also that any aspects developed in the future (e.g.: the resource-aspect) will need to be enforced through a hook in this method.

The *enabledSet* set stores which steps are possible from the given state. Accordingly, it is initialized to the empty set when the algorithm begins. The actions performed by the algorithm are (see figure 3.10):

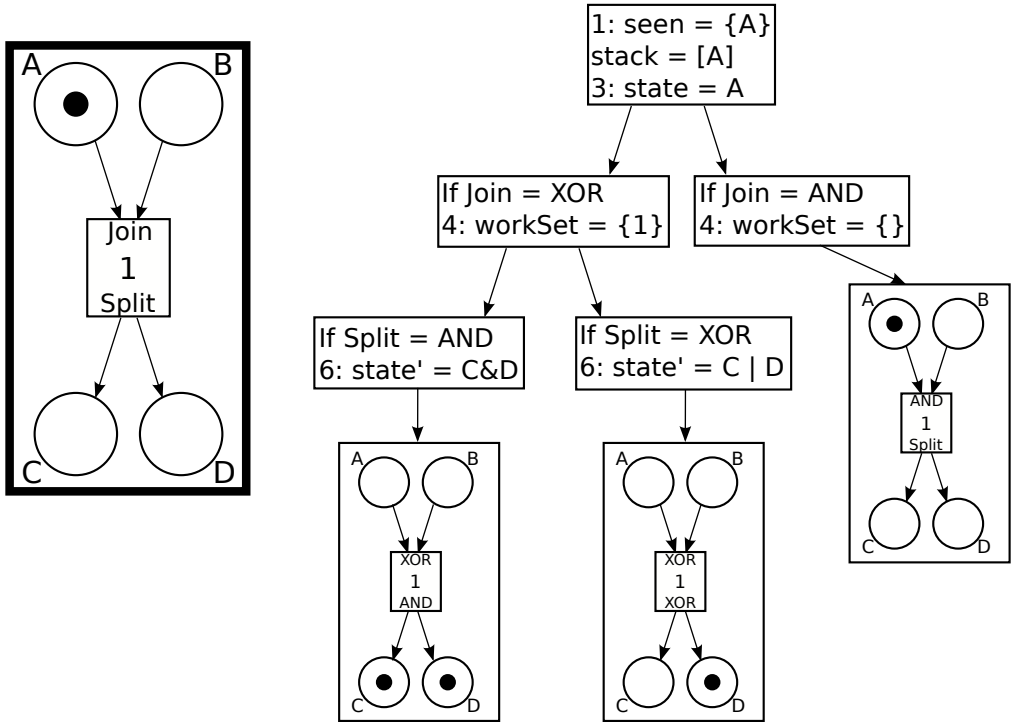


Figure 3.9: *Execution of the* DFS *algorithm on a simplified workflow*

(A) The set of enabled steps, *enabledSet*, is initialized to the empty set, and the tokens from the current state are stored in the set *tokens*.

(B) If the current state has no tokens, then we're done, and return the current set of enabled steps (possibly empty, indicating that the current state is stuck).

(C) If the current state has tokens, execution continues.

(D) The tokens are iterated over, and the current token is named τ. The current token's location is retrieved and stored in *location*. *location*'s children (which are guaranteed to be transitions) are retrieved and stored in the set *children*.

(E) If the current token has no (remaining) children, we're done with this token and can begin exploring the next.

(F) If the current token has at least one (unexplored) child, execution continues.

Algorithm 2 The ENABLED algorithm

Require: $enabledSet := \emptyset$

```
1: for all τ ∈ tokens do
2:    location := getLocation(τ)
3:    for all γ ∈ location.children do
4:       entrySet := join(s, τ, γ)
5:       exitSet := split(s, γ, entrySet)
6:       for all σ ∈ exitSet do
7:          dataSet := dataAspect(σ, s)
8:          enabledSet := enabledSet ∪ dataSet
9:       end for
10:   end for
11: end for
12: return enabledSet
```

(G) The children of the token are now iterated over, and the current child is named γ. We first check if the given transition can be activated (that is, its entry requirements are met) via the method JOIN. All resulting tokens, stored in *entrySet*, are then passed (along with the current state and γ) to the SPLIT method which computes which tokens, if any, will exit from the current child.

(H) If the current child cannot advance, we're done exploring it and can begin exploring its next sibling.

(I) If the current child can advance (that is, a valid step can be formed with it as the parent), execution continues.

(J) The current child's valid steps are now iterated over, and the current step is named σ. All non control-flow aspects are now enforced. Currently this is only the data aspect, so it is passed σ and the current state, and returns valid steps (which are updated to include the results of running the data aspect). These steps are then added to the set of enabled steps and the next advancement of the current child is considered.

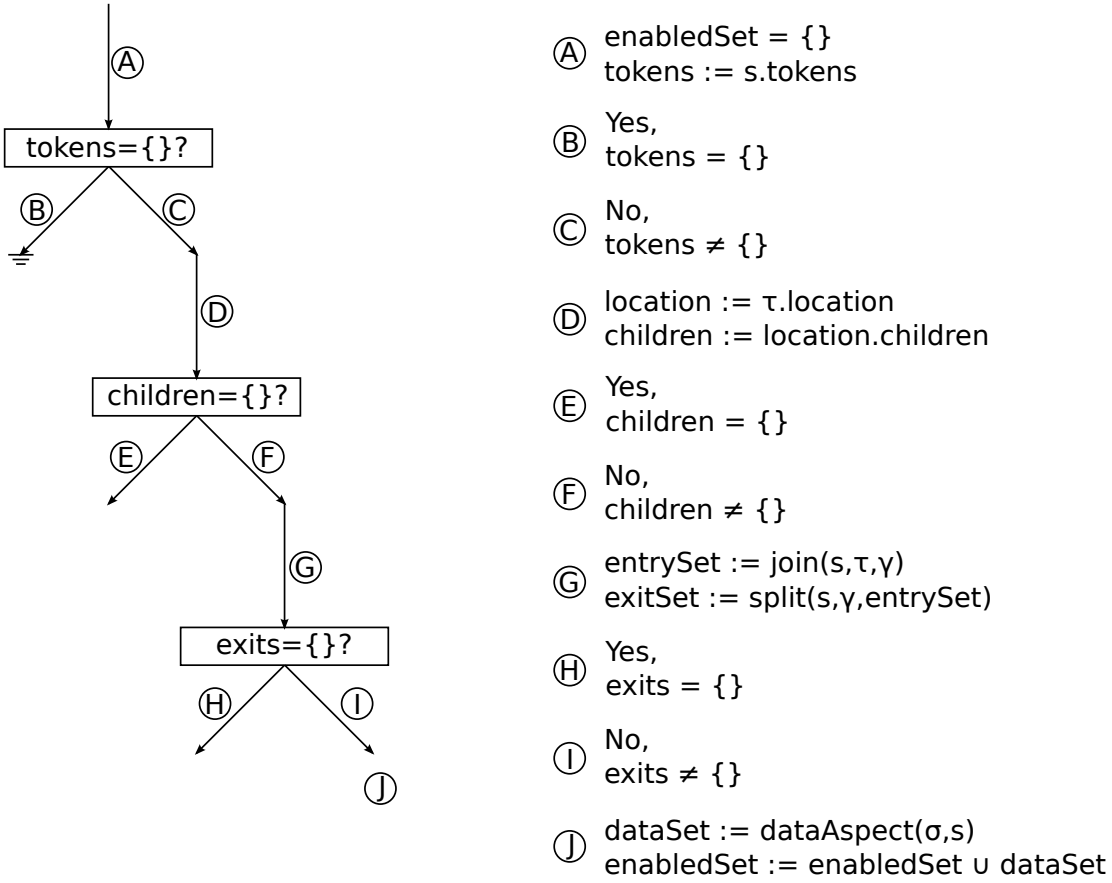


Figure 3.10: *The execution tree for the* ENABLED *algorithm*

SEENBEFORE

The SEENBEFORE algorithm calculates whether or not a state has been seen previously – that is, has the state already been explored / marked for exploration. If it has, we can safely disregard it – in fact, we have to disregard it in order to avoid getting stuck in an infinite evaluation loop.

Checking to see if a state has previously been seen can be idealized as checking if the current state is a member of the *seen* set, which is maintained by the DFS algorithm (see algorithm 1). Unfortunately, since Kinerja uses symbolic execution via Kiasan[6] (and thus, maintains its state symbolically) state matching alone is not strong enough.

Note that this algorithm only runs if the control-aspect of the current state has been

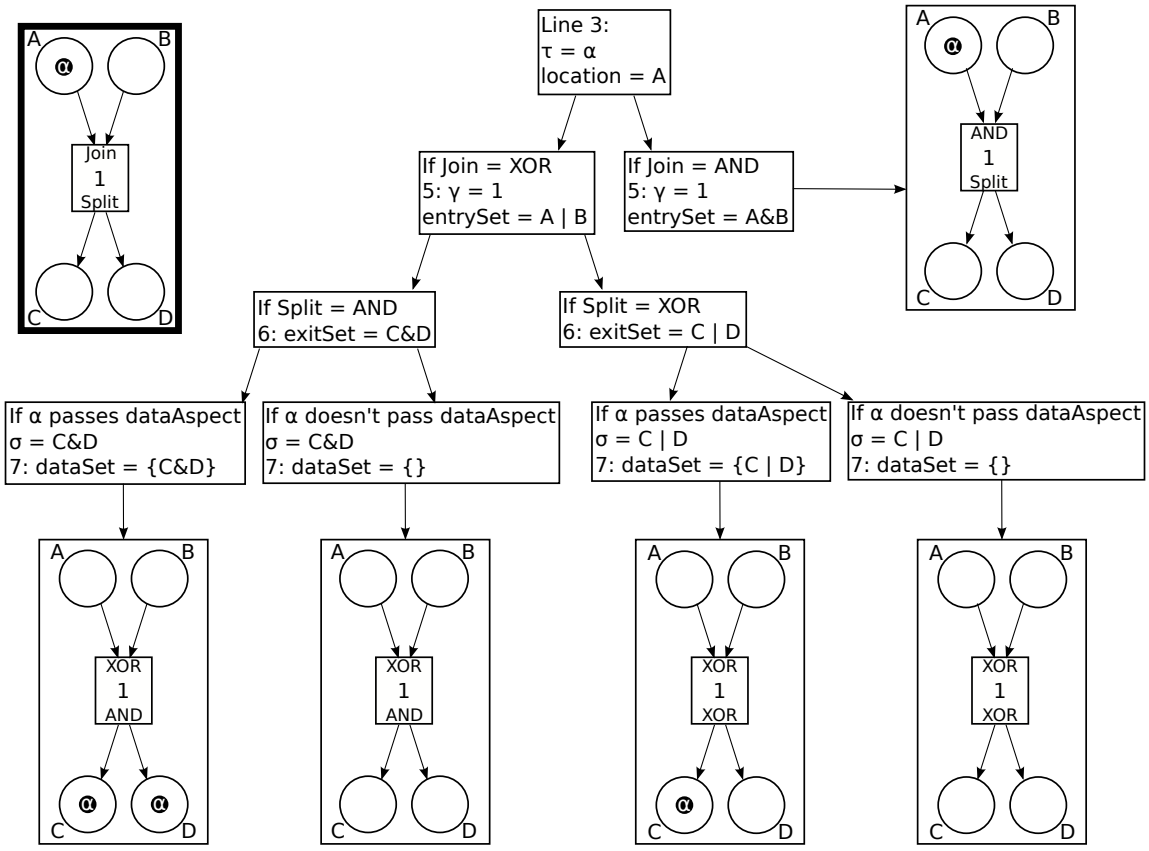


Figure 3.11: *Execution of the* ENABLED *algorithm on a simplified workflow*

seen before. Since the control-aspect of a state is simply a collection of tokens, a test for the control-aspect state's membership in the *seen* set is sufficient. Thus, this algorithm assumes identical control-aspect states, and is testing for data-aspect subsumption.

Consider the case where we have previously explored a state with a single constraint which has a single variable: $x > 3$. If we then generate a new state where $x > 5$, our states won't match exactly, but we do not need to explore the new state since there is no number that is larger than five that is not larger than three. We say that the first state subsumes our new state, and thus we do not need to explore it.

The SEENBEFORE algorithm has two phases: first, it converts the current (symbolic) state into its logical representation; second, it checks to see if the current state does not imply any of the previous states.

This implication may seem confusing at first, so a small example should be considered. Consider again our two states: $x > 3$ and $x > 5$. Ideally we would simply ask our decision procedure "Are there any states where x is greater than five, but not greater than three?" Unfortunately, there are two problems with this query: first, it's not stated as a logical statement. Thus, we convert our query to $(x > 5) \rightarrow (x > 3)$? Second, the decision procedure cannot work on such open ended queries, but rather must consider cases where a single counterexample is enough to prove inconsistency. We then rephrase our question to $\neg((x > 5) \rightarrow (x > 3))$. Our decision procedure will of course tell us that this is true, and that means our new state is subsumed by our old state and should not added to the list of states to explore.

Algorithm 3 The SeenBefore algorithm

Require: $logicalState := \emptyset$
1: $varMap\{symVar \rightarrow netVar\} := buildVarMap(state.vars)$
2: **for all** $\alpha \in state.pathConditions$ **do**
3: **for all** $\beta \in \alpha.constraints$ **do**
4: **for all** $\gamma \in \beta.substitutions$ **do**
5: $\gamma := varMap\{\gamma.symVar\}$
6: **end for**
7: **if** $symVars \cap \beta \neq \emptyset$ **then**
8: $logicalState := logicalState \wedge \beta$
9: **end if**
10: **end for**
11: **end for**
12: **if** $\neg((logicalState \wedge monitorState) \rightarrow (seen_1 \vee seen_2 \vee \ldots \vee seen_n))$ **then**
13: **return** true
14: **else**
15: $seen := seen \cup logicalState$
16: **return** false
17: **end if**

The *enabledSet* set stores the logical representation of the constraints that make up the state of the workflow. Accordingly, it is initialized to the empty set when the algorithm begins. The actions performed by the algorithm are (see figure 3.12):

(A) A mapping from symbolic variable name to net variable name is built with the variables

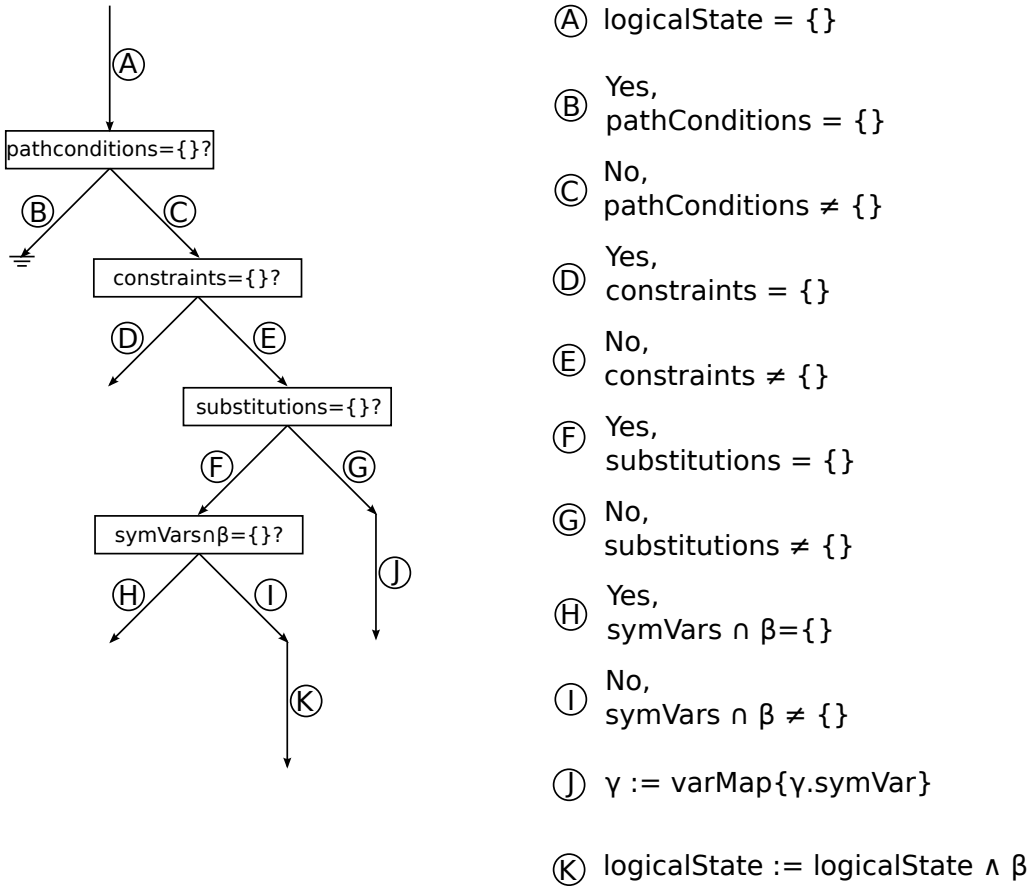


Figure 3.12: *The execution tree for the logical state conversion component of the* SeenBefore *algorithm*

in the current workflow state.

(B) If there are no (remaining) path conditions, this phase of the algorithm is complete, and execution moves to the subsumption checking phase (see figure 3.13).

(C) The path conditions are iterated over, with the current condition being named α. The number of constraints within this path condition is then checked.

(D) If there are no (remaining) constraints, then the exploration of this path condition is complete.

(E) If there are constraints that still need to be checked, they are iterated over, with the current constraint being named β. The number of substitutions within this constraint

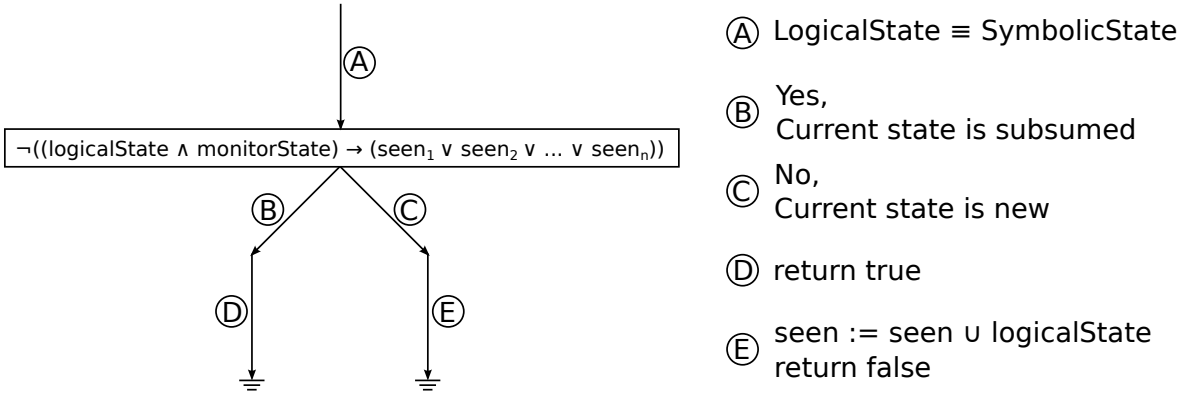


Figure 3.13: *The execution tree for the subsumption checking component of the* SeenBefore *algorithm*

is then checked.

(F) If there are no (remaining) substitutions, the intersection of the symbolic variables and the current constraint is computed.

(G) If there are substitutions that need to be checked, they are iterated over, with the current substitution being named γ.

(H) If the intersection of β and the symbolic variables is the empty set, this iteration is complete and the next constraint can be evaluated.

(I) If the intersection of β and the symbolic variables is not the empty set, execution can continue.

(J) All substitutions are updated to use the names from the current variable mapping. This replaces internal, meaningless names with external, meaningful variable names.

(K) The logical state is updated to include β, which now has all of its constraint names updated to match the workflow's variable names.

After the logical state conversion is complete (see figure 3.12), the algorithm is ready to see if the current state is subsumed by some previously visited state (see figure figure 3.13):

(A) The logical state has been updated to match the symbolic state of the workflow.

(B) The current state is subsumed, and execution continues.

(C) The current state is not subsumed, and execution continues.

(D) True is returned, and the algorithm is finished executing.

(E) The current state is added to the set of seen states (as it is now marked for exploration) and false is returned. The algorithm is finished executing.

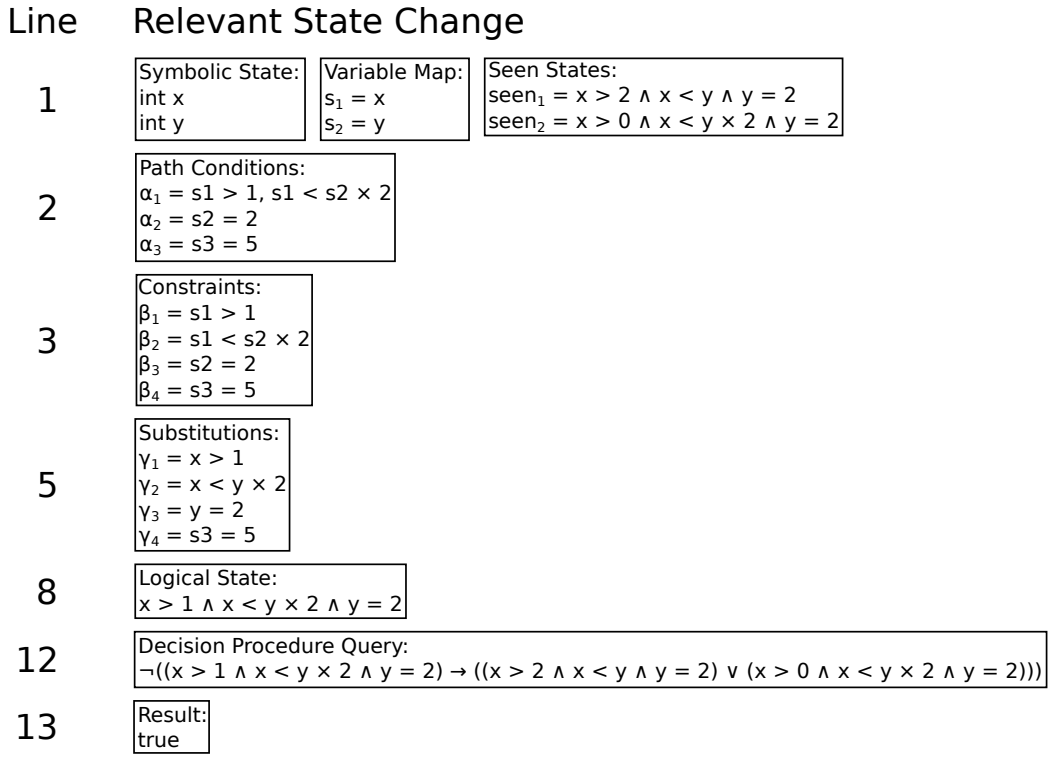


Figure 3.14: *Execution of the* SeenBefore *algorithm on a simple data-aspect state*

3.4.2 Data Structures

There are three data structures which are of particular importance to the implementation of the control-aspect of Kinerja. The first, *ModelCheckerState*, is a representation of the state of the model-checker. The second, *SingleStep*, is the information necessary to advance

through a workflow's state space by one step. The third, *IVarState* is an interface that workflow designers must supply an implementation of; this allows the engine to work with their particular workflow's state and executors.

ModelCheckerState

The *ModelCheckerState* class stores the state of the model-checker. That is, it holds all the information necessary to recreate the current state of the model-checker. The class consists of three fields and various utility functions which manipulate the fields. The fields are:

1. *netState*: This is an instance of the *State* class, and it stores the state of the net. It is itself made up of three fields and their associated utility functions. The fields are:

   (a) *tokens*: This is a collection of the tokens currently in the net. A token is simply a marker on a condition (or place) in the net.

   (b) *netVariables*: This is a mapping from a variable name (stored as a string) to the value of the associated variable. The value is stored as an instance of the *StateObject* class, which is simply a wrapper for the various types of variables a net-designer may want to use.

   (c) *pathConditions*: This is a field that is only used when the model-checker is in verification mode. It stores the path conditions which must be supplied to the theorem-prover for the symbolic variables (stored in *netVariable*s) to maintain their correct value(s).

2. *monitorState*: This is an instance of the *MonitorState* class, and it stores the state of the various monitors that are being tested. It is made up of one field and utility functions which manipulate it. The field is:

   (a) *varMap*: This maps variable names (stored as strings) to their values (wrapped as *StateObject*s). It is intuitively very similar to the net's collection of variables,

except that the variables it stores are used by the various monitors that are being tested. For example, if a net's designer wants to test if a certain node is reached five different times in all execution-paths, the node's hit-count is stored in a variable in this map.

SingleStep

The *SingleStep* class represents one step (through a transition / task) in the execution of a workflow. Instances are created when the control-aspect is first determining what steps are possible (in the ENABLED method) and are passed to the data-aspect for modification. Once a step has been completely determined, it is interpreted by the MOVE method to advance the state of the workflow accordingly. The class itself is made up of five fields and the utility methods which manipulate those fields:

1. *tokens*: This is an ordered set of the tokens that will be advancing. While this will normally be only a single token, a task with an AND-Join will consume as many tokens as it has parents, and we store references to those tokens here.

2. *children*: This is an unordered set of the conditions that will have a token after this step has taken place. This will normally be only a single condition, but a task with an AND-Split will produce as many tokens as it has children, and references to those children are stored here.

3. *transitions*: This is an unordered set of the transitions that will have a token move through them as this step executes. Note that this set will always be a singleton in the current implementation of Kinerja, as it only supports moving a single token at a time.

4. *futureVarMap*: This is the same as the previously described *varMap* (which maps variable names as strings to their values as *StateObjects*) except that it stores the values of variables after the step's execution has completed.

5. *pathConditions*: This is an unordered set that is only used when the model-checker is in verification mode. It stores the path conditions which must be supplied to the theorem prover for symbolic variables to maintain their value(s).

3.5 Data Aspect

The data aspect of a workflow is the consideration of the data values of a workflow, as opposed to the flow of control. While certainly there is some overlap between it and the control aspect, there is support in the literature for giving the data generating / enforcing parts of a workflow equal treatment[20]. The data aspect can be conceptualized as two unique, equally important parts: the generation and modification of data values (termed "executors" in Kinerja) and the modification of control-flow based on these values (termed "flow guards"). The data aspect runs after the control aspect in Kinerja, and thus directly modifies candidate actions generated by the control aspect. This modification can be in the form of changing (or outright eliminating) candidate steps, or in certain modes of execution (e.g.: verification) it can also mean the generation of additional steps.

3.5.1 Formalization of Symbolic Execution

This section holds the formal definitions of the symbolic and logical state representations used by Kinerja. Translating between the two is straightforward, though it should be noted that translation from a logical state to a symbolic state is never required. This translation is sketched, at a high level, in the pseudocode of algorithm 3 and fully detailed in the code of Kinerja.

Symbolic State

$\tau \in Types = \mathbb{Z}$

$sobj \in StateObjects = \{sobj|\tau\}$

$sop \in SymbolicOperators = \{ADD, SUB, MUL, DIV, REM, BIT\_AND, BIT\_OR, BIT\_XOR\}$

$scon \in SymbolicConnectives = \{LE, LT, GT, GE, EQ, NEQ\}$

$ssymb \in$ The set of valid identifiers

$see \in SymbolicExternalExp = \{see|\mathbb{Z}\}$

$sbexp \in SymbolicBinaryExp = \{(sop \vee scon) \times sexp \times sexp\}$

$sexp \in Expression = \{see \vee sbexp\}$

$netVar \in netVars = \{ssymb \rightharpoonup sobj\}$

$pc \in PathConditions = \{netVar \rightarrow sexp\}$

$pcs =$ The set of path conditions

Logical State

$lconc = \mathbb{Z}$

$lsymb =$ The set of valid identifiers

$lcon \in LogicalConnectives = \{\leq, <, >, \geq, =, \neq\}$

$lop \in LogicalOperators = \{+, -, \times, \div, \%, \wedge, \vee, \oplus\}$

$lee \in LogicalExternalExp = \{lee|lconc \vee lsymb\}$

$lbexp \in LogicalBinaryExp = \{lbexp|lexp \times lop \times lexp\}$

$lexp \in LogicalExpression = \{lee \vee lbexp\}$

$stmt \in Statement = \{stmt|lee \times lcon \times lexp\}$

$stmts =$ The ordered set of statements

3.5.2 Control Flow

There is one particularly important method in the implementation of the data aspect: Execute. This method first runs the executor associated with a given task and then runs its associated flow guard. It also handles various utility functions, like loading a task's variables from the net-state and renaming them to what the task expects.

Execute

Algorithm 4 The EXECUTE algorithm

Require: $nameToExec := \{String \rightarrow Executor\}$
Require: $nameToFlow := \{String \rightarrow FlowGuard\}$
1: $preStateVars := loadTaskVars(taskName, controlState)$
2: $newStep := controlStep \cup preStateVars$
3: **if** $executionMode = VERIFY$ **then**
4: $executor := VerifyingExecutor$
5: $flowGuard := VerifyingFlowGuard$
6: **else if** $executionMode = GOVERN$ **then**
7: $executor := getAPIImplementation(nameToExec\{taskName\})$
8: $flowGuard := getAPIImplementation(nameToFlow\{taskName\})$
9: **end if**
10: $execSteps := executor.exec(controlState, newStep, nameToExec\{taskName\})$
11: $postFlowSteps := flowGuard.checkFlow(execSteps, nameToFlow\{taskName\})$
12: **return** postFlowSteps

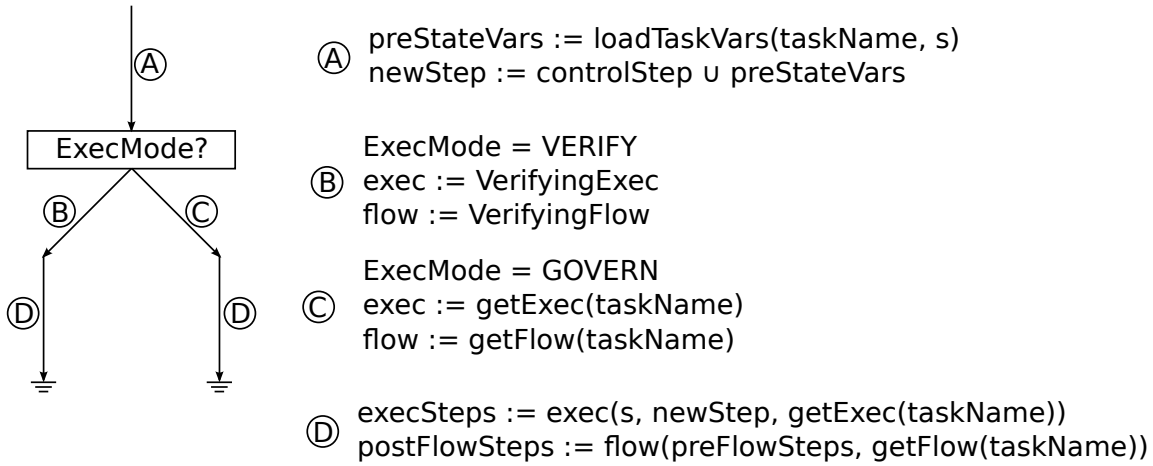


Figure 3.15: *The execution tree for the* EXECUTE *algorithm*

The EXECUTE algorithm calculates and enforces the requirements of the data aspect. It does this is two main steps – running a task's executor, and then running that task's flow guard. The steps which are generated by the executor and then modified (and deemed possible) by the flow guard are returned to the main algorithm and execution continues.

There are two external data structures used by EXECUTE. The first, nameToExec, maps a task's name to an instance of its executor. An executor is a custom java class which implements the IExecutor interface, and fully describes the behavior of the task.

Similarly, the second external data structure is nameToFlow, a mapping from a task's name to an instance of its flow guard. This guard implements the IFlowDetail interface, and is a custom java class which fully describes the requirements a token must meet before it advances out of the given task.

The steps the algorithm takes are:

(A) The task's variable map is loaded. This consists of mapping the values of net variables to task variables, and modifying the names as appropriate. A new step is then created which is identical to the step created by the control aspect except it has the task's variable names and values loaded.

(B) If the system is in verification mode, the executor and flowGuard are initialized to instances of the VerifyingExecutor and VerifyingFlowGuard class (respectively).

(C) Alternatively, if the system is in governance mode, the executor and flowGuard are bound to the specific instance of the GoverningExecutor / GoverningFlowGuard class associated with the current task. This allows the governance-mode executors to be stateful.

(D) The executor is then run, taking as arguments:

  (a) The state, as generated by the control aspect.

  (b) The next step, as generated by the control aspect and modified to have the task's variables.

  (c) The actual executor implementation.

  The flow guard is then run, with the steps generated by execute as well as the actual flow guard implementation as arguments. It returns a subset of potentially modified steps.

  Finally the generated, checked steps are then returned.

3.5.3 Data Structures

The only important data structure in the implementation of the data aspect is the interface *IVarState*, which must be implemented by the designer of a workflow's back end.

IVarState

Implementations of the *IVarState* interface hold the state representations of any user-defined executor. It contains a number of utility methods which are used for accessing and modifying the variables that make up the state of a workflow. It is far simpler than the other implementations of a net's state, and is thus appropriate for use by Kiasan, Kinerja's symbolic-execution back-end[6].

3.6 Monitors

Kinerja's monitors can be thought of as small state machines whose transitions are activated via hooks into the workflow engine. These state machines can then be checked after execution of a workflow (or even during) to see if certain desirable properties hold. A workflow designer might, for example, have a very simple requirement that no execution through her workflow enters a certain undesirable state. Alternatively, her requirements might get quite complex, and specify things like a response condition that must be reached if a particular action condition gets executed. The monitors Kinerja supports are a subset of the patterns identified by Dwyer, Avrunin and Corbett[9].

Figure 3.16 shows how a very basic monitor works on a very basic workflow (the state of the monitor and workflow is shown before execution on the left, and after execution on the right):

The workflow (the directed-flow diagram on the left) starts at the initial condition (the green circle), moves through the task (the white square), and concludes in the sink condition (the red circle). The current state is indicated by the token, denoted with a yellow star ($\star$). The workflow communicates with the monitor via an engine event announcement, which

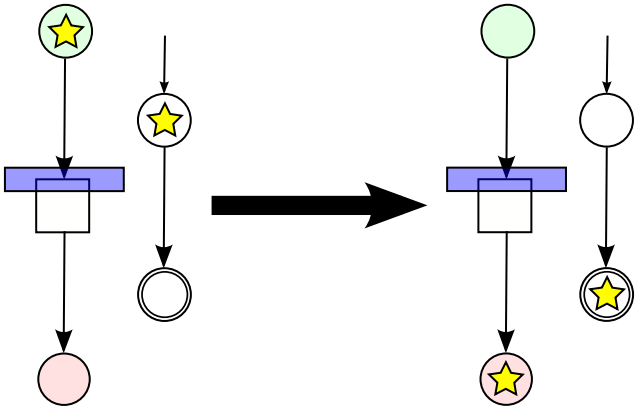


Figure 3.16: *A simple monitor*

takes place when control flow enters the task (denoted by the blue rectangle).

The monitor (the directed-flow diagram on the right) starts at the top state (indicated by the sourceless incoming arrow) and terminates at the second state (the concentric circles). When the workflow executes the token moves from the source condition through the task (triggering the announcement, which moves the monitor into its accepting state) and into the sink / final state.

More complex monitors work in a similar fashion – there can be (many) more states in each monitor (which allows for interesting behavior like scoping), multiple active monitors, much larger workflows, and (potentially) more types of engine announcements.

Figure 3.17 shows two example monitors, M1 and M2. The monitors are the finite state machines on the sides of the YAWL diagram (which is our persistent example, though nodes have been renamed for clarity). The blue zones at the top of tasks are engine events; they serve as guards for the transitions between monitor states. Note that engine events can guard any number (ranging from zero up) of monitor transitions.

M1 shows an example of a scoped existence pattern[9]. It checks for the existence of node D in the scope of B through H (this is referred to as a "between" scope). The finite state machine has three states – out of scope (which is also the initial state), in scope, and existence (which is also the accepting state). When the workflow enters task B, the first transition in M1 is activated, and the monitor (in state 2) is said to be "in scope." When

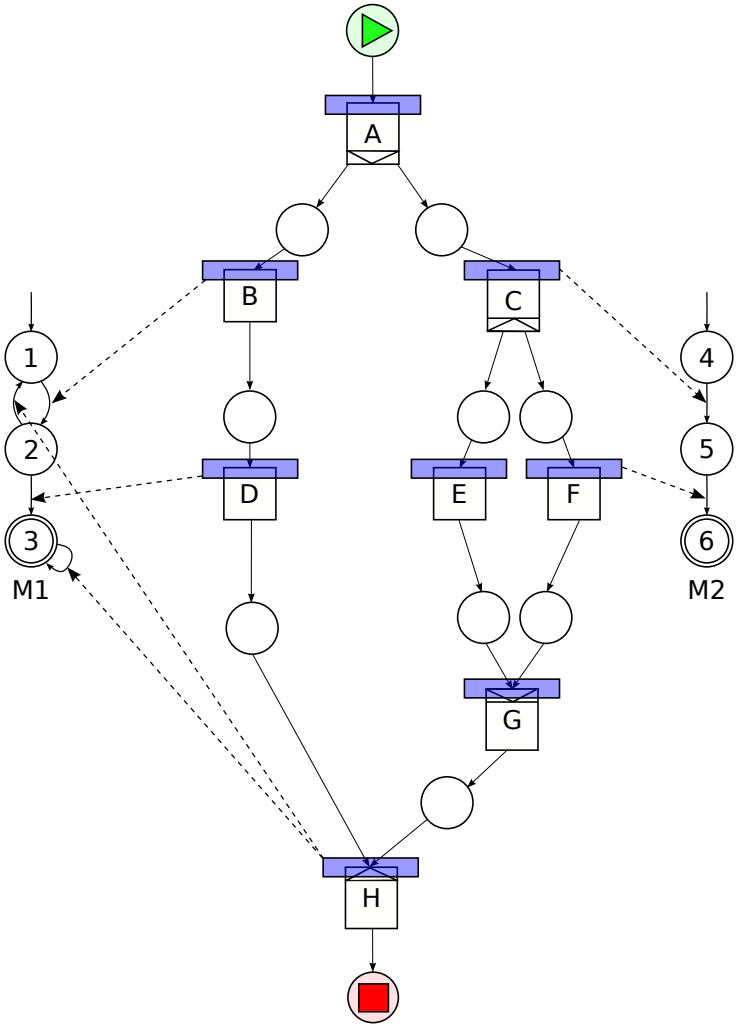


Figure 3.17: *Two example monitors*

the workflow enters task D, the existence is verified, and M1 moves to its accepting state, 3. When the workflow enters H, nothing happens if D has already been visited or if B has never been visited. Had B been visited but D not, the monitor would rest in an unaccepting state, and the property it was checking would have been found to be violatable.

M2 shows a "response" pattern[9] – in this pattern, C is referred to as the "action" and F the "response". It checks that every time task C is executed, task F is also, subsequently executed. When task C is executed, M2 moves from its initial state, 4, to state 5. If F is executed, M2 will move to its accepting state, 6. If F is never executed, this monitor would

have detected a violatable property.

3.7 User Interface

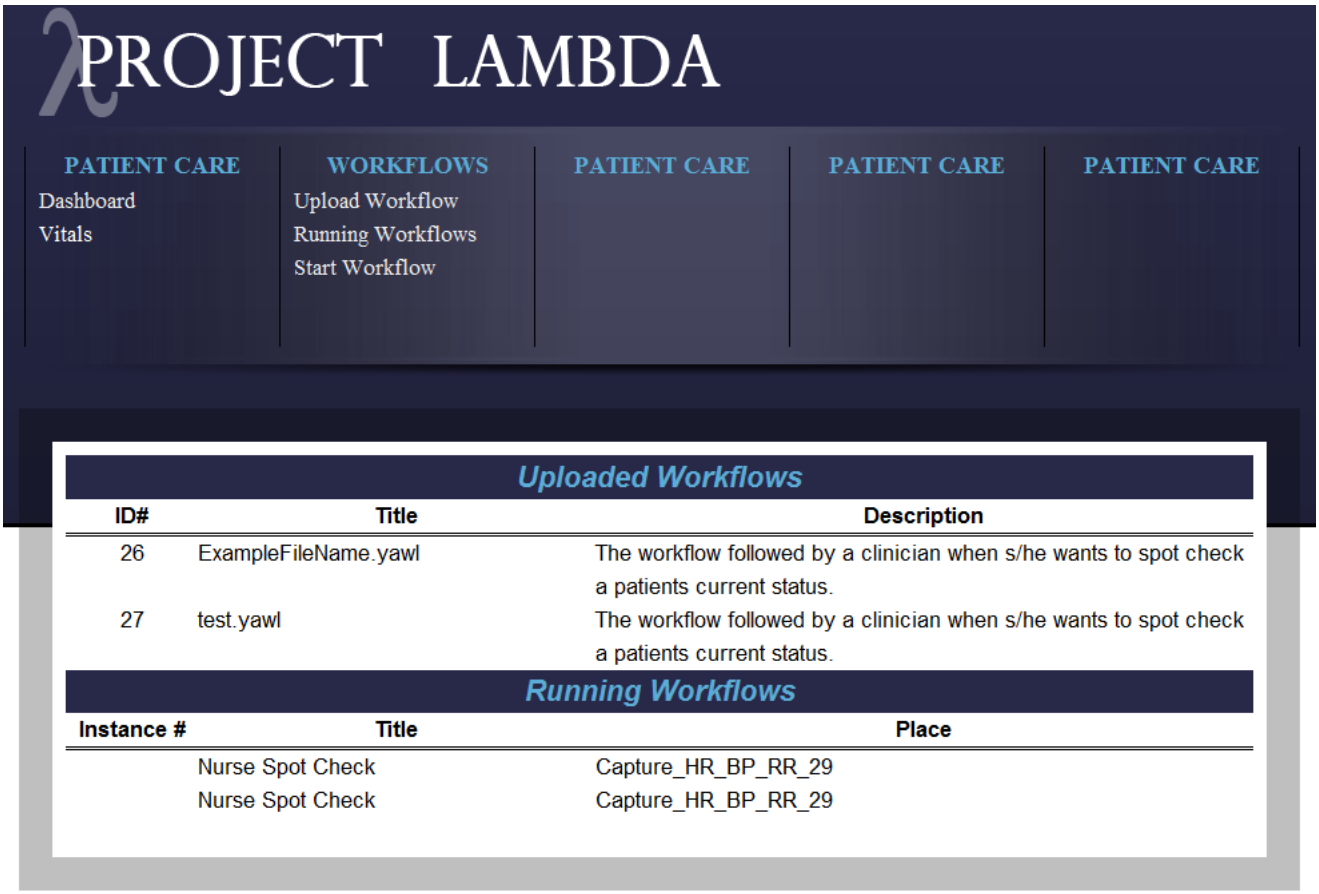


Figure 3.18: *The Kinerja (named Lambda when this screenshot was taken) web interface.*

Kinerja is controlled entirely through the command line. There exists, however, a sophisticated framework for extending and implementing different user interfaces via XML communication over encrypted sockets.

As a test of this interface, a full web interface was built for a previous release of the software with the traditional three-tier web architecture. The front end, built in PHP, was displayed in a user's browser via the standard combination of HTML and AJAX. Kinerja

took the place of the data processing layer, and instead of using files as input and output, a PostgreSQL database was used to store workflows and monitors.

Development of the interface framework included introducing threading to Kinerja, to allow for multiple, concurrent jobs. Security and privacy were also large concerns, so various types of encryption, logging, and user tracking were implemented. These features can be used on any future user interface implementation, regardless of the interface type (web, desktop, mobile, etc.).

The author would like to thank Scarlett Sidwell for her part in creating the web-based user interface.

Chapter 4

Benchmarks

While the previous chapters respectively discussed the need for and functionality of Kinerja, we now turn our attention to how the software functions on actual input. This chapter discusses the performance of Kinerja on two workflows, one which was created solely to demonstrate the set of features Kinerja supports and the other which is sourced from a developing clinical scenario.

4.1 Synthetic Benchmark

This workflow was created to showcase the capabilities of Kinerja, and is not of clinical interest. The workflow, shown in figure 4.1, should be "read" in a roughly top-to-bottom manner, and has three different "paths," which increase in complexity as they progress from left-to-right. That is, the flow of execution is roughly top-down, can take any of the three branches, and the features a given branch demonstrates are more complex the farther to the right the branch is.

4.1.1 Description

Control Flow

Execution begins at the green circle with a green triangle located in the top center of the workflow. Execution continues into P1, and then into A. Node A contains an XOR-split (denoted by the outward facing triangle pointing towards the source of the three outgoing

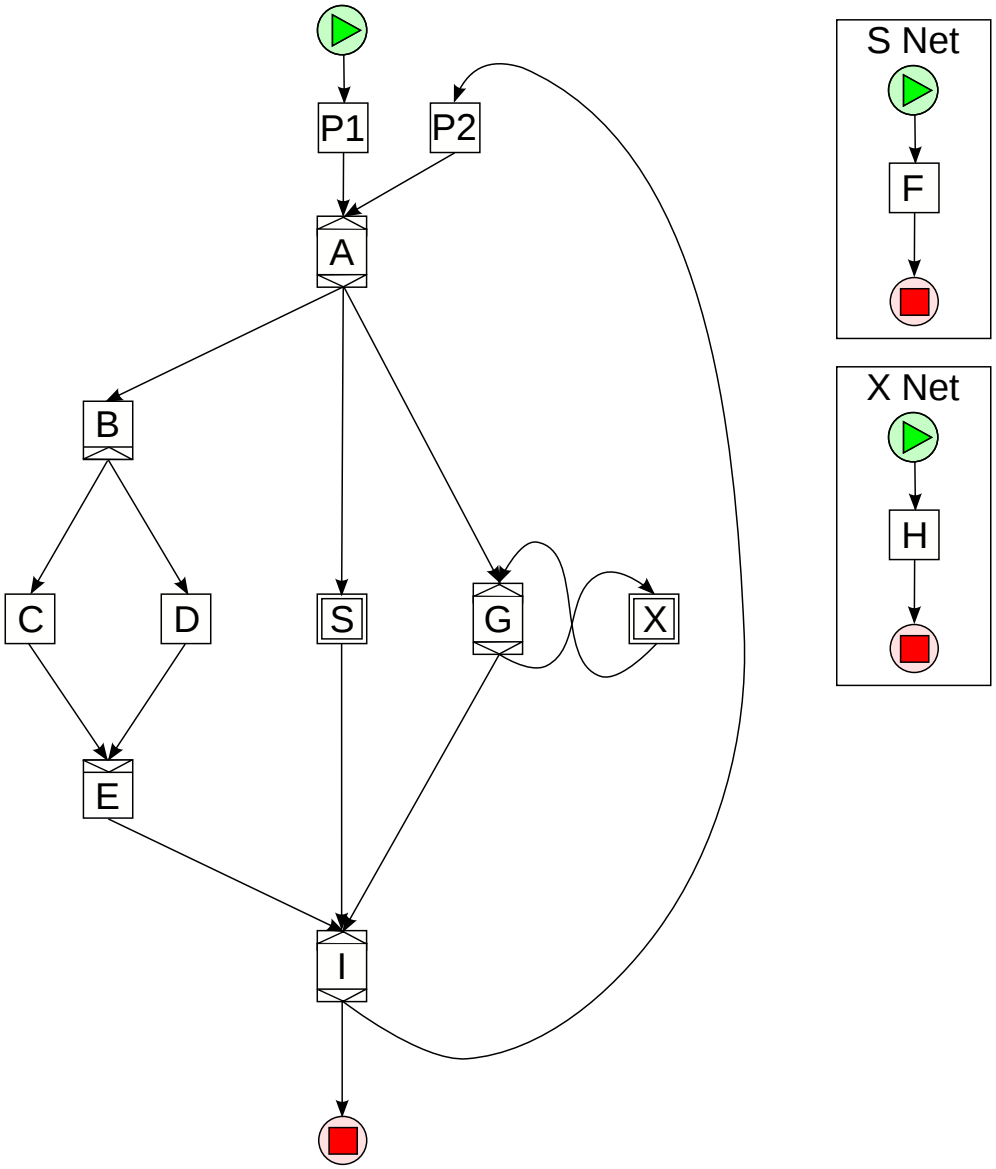


Figure 4.1: *The synthetic benchmark for Kinerja, as laid out in YAWL.*

arcs). Execution will proceed down one of the three subsequent paths (B, C, D, E; S; or G, X), each of which demonstrates a different feature of Kinerja. When the flow of execution arrives at node I, it will either terminate or loop back to node P2, which will allow the workflow to be re-executed (and potentially but not necessarily take a different path).

Nodes S and X are composite tasks, meaning that they are essentially placeholders for sub-nets (a concept quite similar, at a superficial level, to function calls in procedural

programming). Node S maps to the sub-net S Net, similarly, X maps to X sub-net.

Data Flow

It may now be clear why a description of the control-flow aspect of a particular workflow, absent any consideration of the involved data, is incomplete – any branching necessarily depends on the values of the workflow's variables. Unfortunately, the data aspect of workflow is much more difficult to represent in a way that is as easily read as the graphical layout of control flow, and as such takes longer to either scan or peruse.

The synthetic workflow's data aspect can be described simply, though interesting nuances appear upon closer examination. Node P1 initializes a variable, x, such that $1 \leq x \leq 15$. x is then tested by A to determine which path to take.

As the goal of the B, C, D, E branch is to demonstrate parallel / asynchronous flow, no modification of the variable x is necessary. Node and net S (and the embedded task F), however, do modify x (by doubling it); this demonstrates the flattening of sub-nets. The G, X branch has the potential to, but does not necessarily, modify x and demonstrates exceptional control flow.

If x is odd, task I directs execution to node P2, which re-initializes x to a subset of its original range ($2 \leq x \leq 14$). This allows for a succinct demonstration of the subsumption detection capabilities of Kinerja. If x is even, however, execution proceeds to the output condition and terminates.

Splits, Joins, and Parallel Execution

The first branch, consisting of nodes B, C, D and E, shows the basic control flow mechanisms for YAWL, and Kinerja's interpretation of them. Node B has an AND-split (denoted by the inward-facing triangle at the source of outgoing arcs) which means that C and D will be available for execution simultaneously.

In practice this does not require their simultaneous execution, rather it signifies that there is no order of execution dependency between them. Nodes C and D thus can be done

in any order – sequentially (with either C or D preceding the other) or simultaneously.

Node E can be thought of as "collecting" the parallel execution, since it has an AND-join (denoted by the inward-facing triangle at the sink for incoming arcs) which tells the Kinerja engine to wait for all immediately preceding nodes to complete before execution continues.

Had node B been labeled with an XOR-split, rather than an AND-split, only one of either C or D would have been executed. Node E would have to be similarly equipped (that is, with an XOR-join) because with an AND-join it would deadlock execution, since it is impossible for both node C and node D to complete.

Sub-nets and Embedded Workflow

The middle execution path, consisting of the composite node labeled S, was made to demonstrate how embedded workflows function. Node S is a stand in for an entire net, which could itself have sub-nets. In this case, however, node S is linked to the eponymous S net which contains only one (atomic) task, F.

Kinerja "flattens" as it parses, producing a final workflow without nesting. While this eliminates a great deal of complexity (in that it's no longer necessary to define semantics for moving between levels) it creates some issues as well, namely the possibility of variable-name collisions and the necessity of defining intermediate transitions which simulate moving from a composite node to its associated workflow. They are dealt with as follows:

- Variable name collisions – There currently is no facility for automated variable name collision elimination; this must be handled manually. It should be noted, however, that YAWL's automatic naming policy still applies to nodes, and thus conditions and tasks are guaranteed to have globally unique names.

- Intermediate transitions – Kinerja's parser will automatically generate transitions between a composite node's parent transition and the input condition of its associated workflow, as well as the embedded workflow's output condition and the immediate successor of the composite node. Without these transitions, conditions would be linked

directly to conditions, which would violate the structure of the parsed workflow. The transitions' names are automatically generated, and they take the form of either "CompositeNodeNameIN" or "CompositeNodeNameOUT" depending on whether they are transitioning into or out of the nested workflow.

The sub-net thus completely replaces the composite node in the flattened workflow, and the original node has no singular representation.

Exception Handling

The third route through the workflow, consisting of the atomic task G and composite task X, shows how exceptions can be represented in Kinerja. It is important to note that there is no "true" exception handling in Kinerja, in that there is no special construct which allows for the interruption of normal execution. Thus, the only exceptions checkable in Kinerja are those which can be detected through the evaluation of statements (e.g.: if some variable is outside of a defined acceptable range).

The route shows a typical pattern for detecting an exception (in this case if our variable state somehow indicated $x \leq 0$), executing a compensatory sub-net (which might make x positive), and resuming typical execution.

4.1.2 Execution

Figure 4.2 shows the Kinerja parser's reduction of the workflow into a flattened petri-net. Note that the arcs which join the various nodes are anonymous.

Note also that transitions are represented as rectangles, and are labeled with their user-assigned name (e.g.: A) and, following an underscore, the YAWL-assigned number which guarantees name uniqueness (e.g.: 3); these combine to form names like "A_3".

All conditions in this workflow are implicit (that is, they were not created by the workflow designer, and will not show up in YAWL's workflow editing program) and as such have auto-generated names. A condition's name is generated by simply concatenating the condition's predecessor and successor transitions' names.

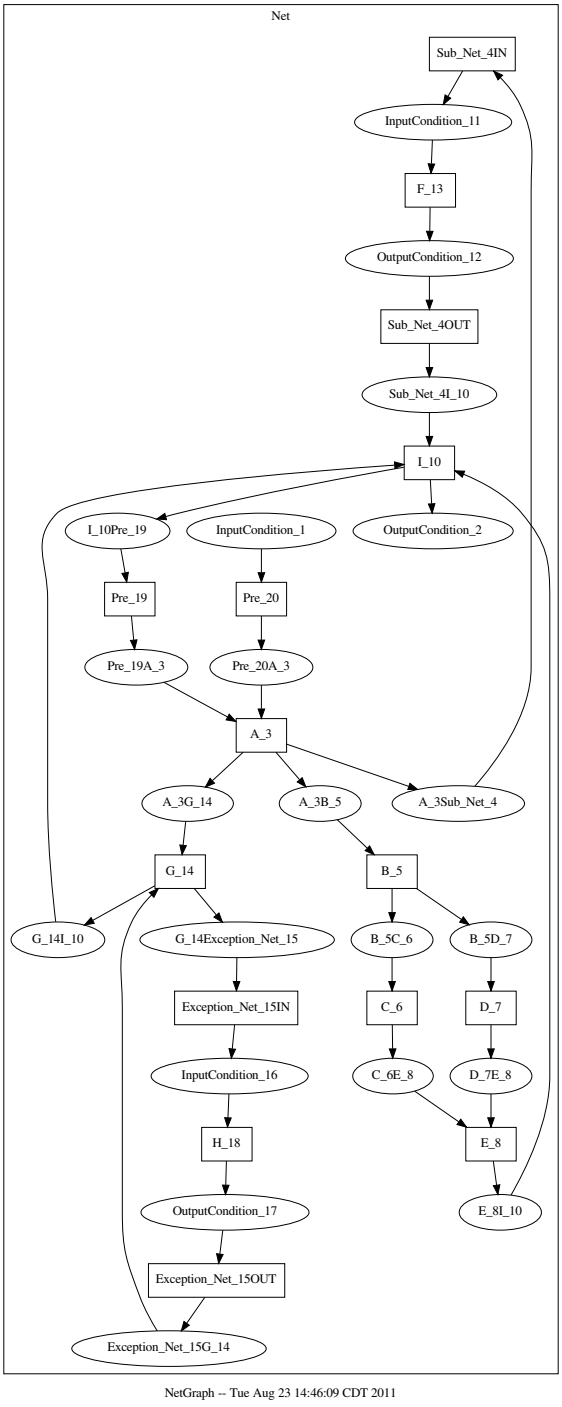


Figure 4.2: *Kinerja's parser's final interpretation of the synthetic workflow*

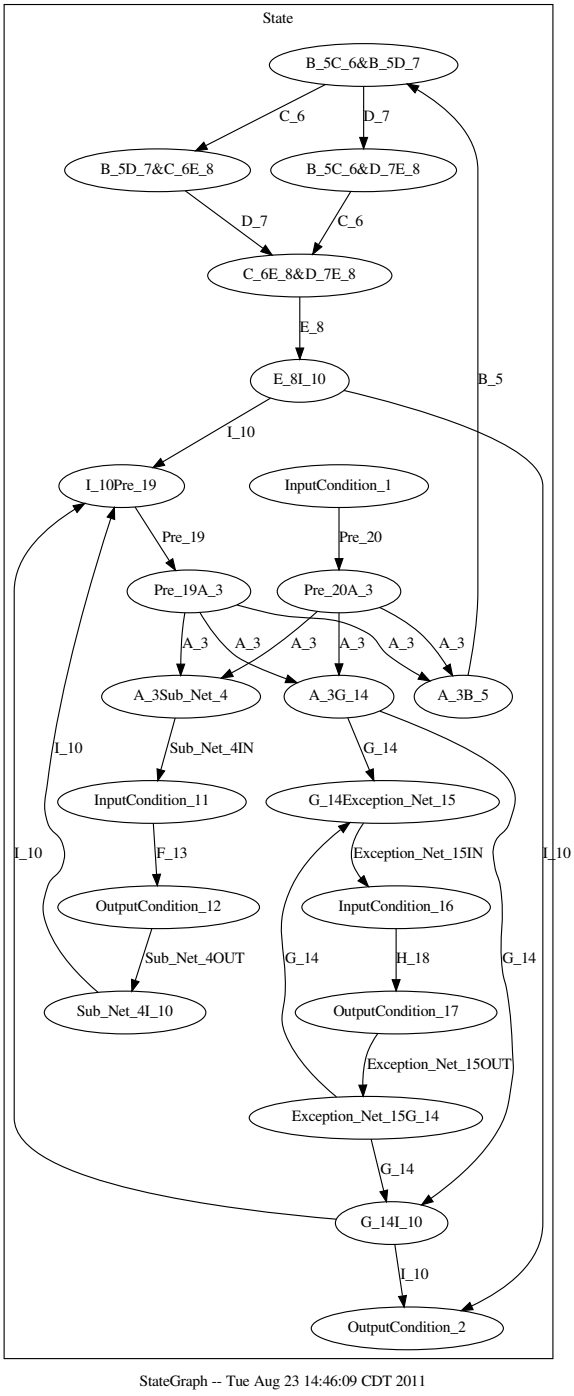


Figure 4.3: *The states of the Synthetic workflow*

Figure 4.3 shows one version of the states reached by Kinerja when executing the synthetic workflow. Note, however, that this graph shows a deceptively small number of states as it matches based on the state of the control-flow aspect alone – that is, while the data aspect is used in the execution trace, it is not considered when rendering the graph.

As execution cannot "stop" when a token is in a transition or, alternatively, the atomicity of execution is not reducible below a single "step," (which is defined as movement through a transition and into a set of resultant conditions), transitions are no longer represented as nodes on the graph. Instead, they are labels for arcs – to see what state is a successor to another state, one can simply see which are linked together. To determine which transition should be taken to move from one state to another, it suffices to read the label on the arc between nodes.

Note that nodes no longer represent conditions explicitly, but rather those conditions which are "marked" by tokens in a given workflow state. As the majority of the execution steps consist of moving single tokens, many states in this state graph have the name of only a single node. There are four, however, which consist of two tokens (corresponding to those nodes which can be executed in parallel) whose names are joined with an ampersand, e.g.: the name B_5D_7&C_6E_8 means that there are two tokens, one at B_5D_7 and a second at C_6E_8.

Execution Statistics

The synthetic workflow, with both data and control aspects considered, consists of 25 states. Kinerja creates and explores these states in slightly under six seconds, with roughly .15 second going toward the parsing and flattening of the workflow. Less than .1 second is spent in the decision procedure, which is queried 15 times – the vast majority of time, over 5.5 seconds, is spent in Kiasan, updating and examining the data-aspect state.

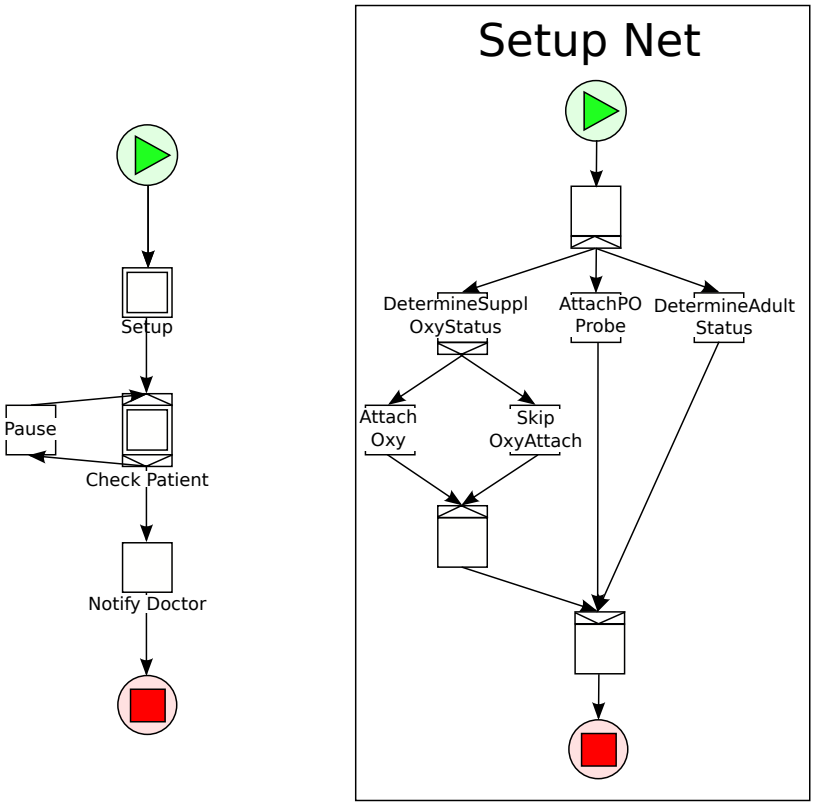


Figure 4.4: *The clinical benchmark's overview net and Setup subnet.*

4.2 Clinical Benchmark

The second example workflow considered here is less interesting from a software demonstration point of view, but more interesting from the perspective of a domain expert – e.g. a clinician, doctor, or anyone else involved in the medical industry.

4.2.1 Description

This workflow details the steps a clinician might go through to check on a patient whose heart rate, respiratory rate, and blood-oxygen saturation are being monitored. There are a number of phases, or groups of actions, described: the setup of the equipment which monitors the vital signs of the patient, the actual monitoring of the patient, and a pause which simulates the passage of time between "spot checks."

The majority of actions which require work external to Kinerja are defined in the SAnToS

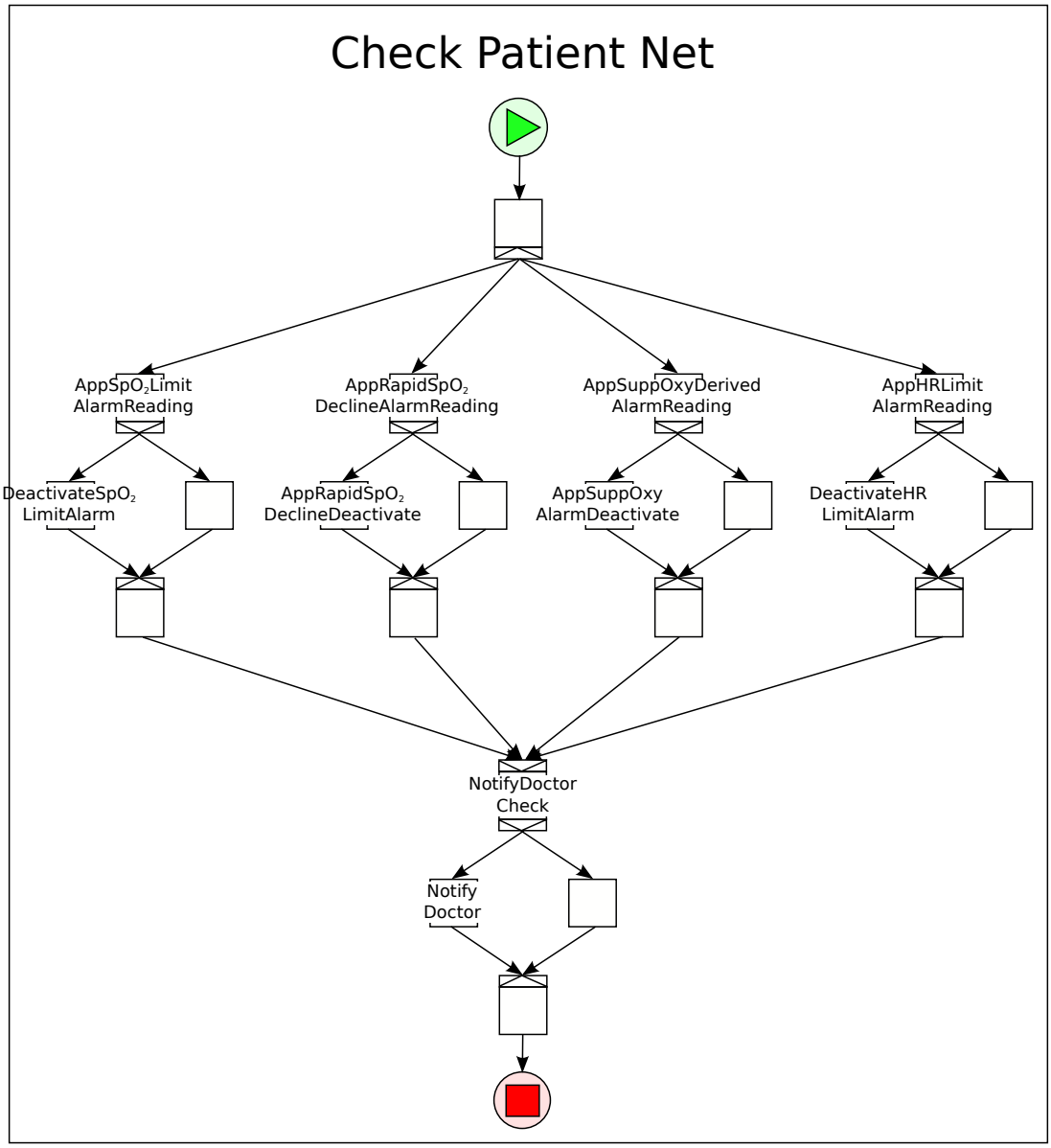


Figure 4.5: *The clinical benchmark's Check Patient subnet.*

technical report "Pulse Oximeter Monitoring Smart Alarms" by John Hatcliff[13]. Modified descriptions of these tasks are reprinted here.

Setup

The steps required to begin monitoring a patient's heart rate, respiratory rate, and blood oxygen saturation are described in the task Setup. It is a composite task which unfolds to

a net of the same name.

This subnet has three tasks (or, in the case of supplemental oxygen work, a task group):

1. DetermineSupplOxyStatus – This determines if a patient is to be placed on supplementary oxygen. After this determination, one of two actions is taken:

   (a) AttachOxy – If necessary, supplemental oxygen is attached.

   (b) SkipOxyAttach – Similarly, some patients don't need supplementary oxygen.

   In addition to physically attaching an oxygen line, this check is necessary for setting alarms correctly – an oxygenated patient's blood oxygen saturation will read as abnormal to a person or device expecting a non-oxygenated patient, and vice-versa.

2. DetermineAdultStatus – This determines if a patient is to be classified as an adult for the purpose of configuring the app smart alarm button. As this is simply determining a value which will be used elsewhere, there are no follow-up tasks.

3. AttachPOProbe – This directs the clinician to attach a pulse oximeter probe to the patient and confirm that there are valid readings registering.

Check Patient

After setting up the patient, and periodically thereafter, the clinician will actually check on the patient. This is a straightforward check, consisting of monitoring for alarms and reacting appropriately if any are active.

This subnet has four task groups, however they all have a parallel structure. Each group consists of an alarm check and (potentially) the deactivation of that particular alarm, which of course is dependent on the alarm's state. The tasks groups are:

1. AppSpO_2LimitAlarmReading – This alarm is a "dumb" alarm which activates when a patient's blood oxygen saturation goes outside of some given range.

2. AppRapidSpO_2DeclineAlarmReading – This alarm is a "smart" alarm (which might run on some ICE[32]-compliant architecture, like the MDCF[15]). "... an alarm event will be generated if the SpO_2 moving average decreases by an amount greater than the decrease bound within the configured time interval[13]."

3. AppSuppOxyDerivedAlarmReading – This is also a smart alarm, which will be triggered if the moving average of a patient's SpO_2 value, less some adjustment, falls below a lower limit parameter. Note that this parameter will be modified depending on whether or not the patient is receiving supplementary oxygen.

4. AppHRLimitAlarmReading – This is a dumb alarm which activates when a patient's heart rate is outside of a prescribed range.

After checking the state of the various alarms, a doctor is notified if necessary, and the spot check cycle is concluded. Naturally, the workflow can be restarted at some future point, though this requires re-executing the setup tasks and re-starting any necessary alarm systems.

If notifying a doctor is deemed unnecessary, then there is a pause for some amount of time, after which the patient and any attached alarms are re-checked.

4.2.2 Execution

Note that as this example is considerably larger than the synthetic workflow, the flattened net and state diagrams cannot be formatted to fit in this report, or render legibly on standard paper.

With the data aspect disabled, Kinerja generates and explores 1,333 states in three minutes and two seconds on a standard laptop. Of this, roughly .1 second was spent parsing and flattening the workflow, and the rest was spent exploring possible states. With the data aspect enabled, the execution space and time increase dramatically – it takes Kinerja nearly 54 minutes to create and explore all 10,328 states. In running this example, Kinerja makes

a total of 44,941 queries to the decision procedure, and spends nearly 18 minutes waiting on their results. Another 28.5 minutes were spent maintaining the symbolic state via Kiasan, and the remaining time was spent in Kinerja itself.

4.2.3 Evaluation

When modeling and executing the clinical workflow, a number of strengths and weaknesses of Kinerja become apparent.

Strengths

Kinerja performs well in a number of areas where other model checkers and execution environments are less successful.

- Completeness of verification – Kinerja fully explores a version of the supplied workflow that completely integrates data and control aspects.
- Integrated verification and governance – Rather than complete verification in a tool external to the execution environment (or skip either governance or verification altogether) Kinerja allows a seamless switch between environments.
- State subsumption – By using Kiasan[6], Kinerja is able to execute tasks symbolically, which allows for the testing of far more possible states than explicit enumeration. Kinerja also tests for state subsumption rather than state matching, which allows workflows that would be otherwise untestable to be verified.

Weaknesses

While this example demonstrates a number of Kinerja's strengths, it also reveals a number of missing features / weaknesses:

- Event handling – Demonstrated particularly clearly by this workflow, the somewhat awkward "polling" for events does not match the reality of event-based notifications. This mismatch at best creates workflows that are challenging (beyond any inherent

complexity) to read, and at worst can create a disconnect between the workflow and the real world, which can allow errors to occur.

- Lack of an OR-join – The subset of YAWL which Kinerja treats does not include an OR-Join (due to its non-local semantics). This lack becomes very apparent, though, because it requires all splits (both AND and XOR) to be matched with a sibling join. Thus, in the case where a sub task is only necessary part of the time (e.g., deactivating an alarm) a "dummy" task, which completes no actions, must be created.

- Speed – In the model-checking world, slightly over 10,000 states is not a terribly substantial achievement, and shouldn't take nearly an hour. Ideally verification would be quick enough that it could be run as a simple test for errors, allowing iterative testing and development.

Chapter 5

Conclusion

This thesis offers a brief survey of technologies which compete in the workflow-automation space, and offered a new entrant, Kinerja. Existing technologies were compared and discussed, and found to be lacking chiefly in the area of formal verification. What formal options exist are poorly integrated with larger execution environments.

Kinerja integrates the concepts of verification and governance into a unified infrastructure. A number of components have also been built to interact with Kinerja, including one that enables symbolic execution of the data aspect of workflows.

Finally, two exemplary workflows were produced and analyzed. Together they showcase the capabilities of Kinerja, and enable a discussion of both the strengths and weaknesses of Kinerja.

5.1 Future Work

There are three main directions that the immediate next steps could focus on:

1. Increased language support – As demonstrated by the clinical workflow example (and discussed in section 4.2.3), the small size of Kinerja's supported language makes modeling some workflows tedious. This could be remedied by adding support for more powerful language constructs. These might be features of YAWL (that were excluded from Kinerja's treated subset, like the OR-join, or the cancellation region) or they

might be features not yet in YAWL (like message passing).

2. Optimizations – Kinerja performs relatively slowly given the size and complexity of its inputs. This is a result of little attention being paid to optimizations during the development of the system, and it's likely that an optimization pass over the codebase could result in a considerable speedup. There are also algorithmic improvements that could be made, such as partial-order reduction[11].

3. Consideration of additional aspects – Kinerja currently only supports control and data aspects, but there are many more views of fully developed workflows. A workflow designer might also want to consider:

   (a) Resource management – A workflow step (or set of steps) might use a given machine or device, and it should be unavailable to other users while it completes its task.

   (b) Agent management – A workflow step might need to be delegated to an individual user or device which is capable of completing it. Task allocation strategies have been studied and implemented by the authors of YAWL[21].

   or a number of other workflow aspects.

5.2 Clinical Impact

Though considerable work remains, the benefits of a fully realized, integrated clinical environment which is managed by Kinerja are impressive. The idea of an operating room where devices communicate and a great deal of work is handled by closed-loop systems is futuristic indeed.

In the short term, any decrease in the rate of medical errors – which kill as many as 98,000 Americans each year[30] – would be beyond welcome. Adoption of Kinerja, or a similar system, would help bring about this decrease via a combination of two factors: first, by

reducing human-device interaction (e.g.: by facilitating device to device communication for the creation of closed-loop medical systems), and second, by assisting clinical care workers with complex tasks. This assistance could aid not only with tasks that are intrinsically complex, but also tasks which involve a new and unfamiliar device or modified procedure.

Bibliography

[1] Michael Adams and Arthur ter Hofstede. Yawl - user manual. User manual, The YAWL Foundation, September 2009. Version 2.0f.

[2] TRANSFLOW Nederland BV. Workflow patterns cosa workflow software. Technical report, TRANSFLOW Nederland BV, 2003.

[3] Bin Chen, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Automatic fault tree derivation from little-jil process definitions. In *Proceedings of the Software Process Workshop*, volume 3966, pages 150–158. Springer Berlin / Heidelberg, July 2006.

[4] F. Curbera, Y. Goland, J. Klein, F. Leymann, Thatte, and S. Weerawarana. Business process execution language for web services, version 1.1. Technical report, IBM, 2003.

[5] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx an open modeling platform for the bpm community. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 382–385. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85758-7_29.

[6] Xianghua Deng, Jooyong Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, Tokyo, Japan, 2006. IEEE Computer Society.

[7] Xianghua Deng, Jooyong Lee, and Robby. Efficient and formal generalized symbolic execution. Technical report, Kansas State University, April 2011.

[8] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Formal semantics and analysis of bpmn process models using petri nets. *Information and Software Technology*, 50(12):1281–1294, November 2008.

[9] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[10] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, October 2004.

[11] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, 1995.

[12] Object Management Group. Business process model and notation. Technical report, Object Management Group, January 2011.

[13] John Hatcliff. Pulse oximeter monitoring smart alarms. August 2011.

[14] A.H.M. Hofstede, W.M.P. Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and Its Support Environment*. Springer, 2009.

[15] Andrew King, Sam Procter, Dan Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, pages 141–151, Vancouver, Canada, May 2009. IEEE.

[16] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[17] Paul B. Langevin, Vashti Hellein, Susan M. Harms, William K. Tharp, C. Cheung-Seekit, and S. Lampotang. Synchronization of radiograph film exposure with the inspiratory pause effect on the appearance of bedside chest radiographs in mechanically ventilated patients. *American Journal of Respiratory and Critical Care Medicine*, 160(6):2067–2071, 1999.

[18] Inc. PECTRA Technology. Technical report, PECTRA Technology, Inc.

[19] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns a revised view. Technical report, BPM Center, 2006.

[20] Nick Russell, Arthur H.M. ter Hofstede, David Edmond, and Wil M.P. van der Aalst. Workflow data patterns. Technical report, Queensland University of Technology, Brisbane, Australia, 2004.

[21] Nick Russell, Arthur H.M. ter Hofstede, David Edmond, and Wil M.P. van der Aalst. Workflow resource patterns. Technical report, Queensland University of Technology, Brisbane, Australia, 2004.

[22] Nick Russell, Arthur H.M. ter Hofstede, and Wil M.P. van der Aalst. newyawl: Specifying a workflow reference language using coloured petri nets. In *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. CPN Group at the Department of Computer Science, University of Aarhus, Denmark, October 2007.

[23] Stefan Seidel, Michael Rosemann, Arthur ter Hofstede, and Lindsay Bradford. Developing a business process reference model for the screen business – a design science research case study. In *Proceedings of the 17th Australasian Conference on Information Systems*, number 17, Adelaide, Australia, December 2006. Australasian Conference on Information Systems.

[24] William W. Stead and Herbert S. Lin, editors. *Computational Technology for Effective*

Health Care: Immediate Steps and Strategic Directions. The National Academies Press, 2009.

[25] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jabc. In *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, HVC'06, pages 92–108, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] Kontinuum Specification Team. Implementation of standard workflow control patterns using web and flos kontinuum version 2006. Technical report, Web and Flo Pty Ltd, August 2006.

[27] Antoine Toulme. How to get the most of the bpmn modeler. Technical report, Intalio, Inc., 2008.

[28] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language (revised version). Technical report, Queensland University of Technology, Brisbane, 2006.

[29] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[30] N Saul Weingart, Ross McL Wilson, Robert W Gibberd, and Bernadette Harrison. Epidemiology of medical error. *BMJ*, 320(7237):774–777, 3 2000.

[31] Stephen A. White. Introduction to bpmn. White paper, IBM Corporation, 2004.

[32] Susan F. Whitehead and Julian M. Goldman. Medical device plug-and-play. *Patient Safety & Quality Healthcare*, February 2008.

[33] Alexander Wise. Little-jil 1.5 language report. Language report, University of Massachusetts, Amherst, MA, USA, October 2006.

[34] Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Jr. Stanley M. Sutton. Using little-jil to coordinate agents in software engineering. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 155–163, Washington DC, USA, 2000. IEEE Computer Society.

[35] M.T. Wynn, M. Dumas, C. J. Fidge, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Business process simulation for operational decision support. In *Proceedings of the Third International Workshop on Business Process Intelligence*, pages 66–77, Brisbane, Australia, September 2007. Springer-Verlag. In conjunction with Business Process Management Conference.