A Formal Model For Declarative Workflows

Dynamic Condition Response Graphs

Raghava Rao Mukkamala

A PhD Dissertation
Presented to the Faculty of the IT University of Copenhagen
in Partial Fulfillment of the Requirements of the PhD Degree

Advisor	
Dr. Thomas T. Hildebrandt, PhD	IT University of Copenhagen, Denmark
Evaluation Committee	
Dr. Andrzej Wąsowski, PhD	IT University of Copenhagen, Denmark
Dr. Richard Hull, PhD	IBM T.J. Watson Research Center, NY, USA
Dr. Hagen Völzer, PhD	IBM Research – Zurich, Switzerland

IT University of Copenhagen February 2012

Current business process technology is pretty good in supporting well-structured business processes and aim at achieving a fixed goal by carrying out an exact set of operations. In contrast, those exact operations needed to fulfill a business process/workflow may not be always possible to foresee in highly complex and dynamic environments like healthcare and case management sectors, where the processes exhibit a lot of uncertainty and unexpected behavior and thereby require high degree of flexibility. Several research groups have suggested declarative models as a good approach to handle such ad-hoc nature by describing control flow implicitly and there by offering greater flexibility to the end uses.

The first contribution of this PhD thesis is to formalize the core primitives of a declarative workflow management system employed by our industrial partner Resultmaker and further develop it as a general formal model for specification and execution of declarative, event-based business processes, as a generalization of a concurrency model, the classic event structures. The model allows for an intuitive operational semantics and mapping of execution state by a notion of markings of the graphs and we have proved that it is sufficiently expressive to model ω-regular languages for infinite runs. The model has been extended with nested sub-graphs to express hierarchy, multi-instance sub processes to model replicated behavior and support for data.

The second contribution of the thesis is to provide a formal technique for safe distribution of collaborative, cross-organizational workflows declaratively modeled in DCR graphs based on a notion of projections. The generality of the distribution technique allows for fine tuned projections based on few selected events/labels, at the same time keeping the declarative nature of the projected graphs (which are also DCR graphs). We have also provided semantics for distributed executions based on synchronous communication among network of projected graphs and proved that global and distributed executions are equivalent.

Further, to support modeling of processes using DCR Graphs and to make the formal model available to a wider audience, we have developed prototype tools for specification and a workflow engine for the execution of DCR Graphs. We have also developed tools interfacing SPIN model checker to formally verify safety and liveness properties on the DCR Graphs. Case studies from healthcare and case management domains have been modeled in DCR Graphs to show that our formal model is suitable for modeling the workflows from those dynamic sectors.

This PhD project is funded by the Danish Strategic Research Council through the Trustworthy Pervasive Healthcare Services project (www.trustcare.eu).

A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs

Raghava Rao Mukkamala

2012

IT University of Copenhagen
Theoretical Computer Science

A Formal Model For Declarative Workflows

Dynamic Condition Response Graphs

Raghava Rao Mukkamala

ITU DS: D-2012–80
ISSN: 1602-3536
ISBN: 978-87-7949-268-4

A Formal Model For Declarative Workflows
Dynamic Condition Response Graphs

Current business process technology is pretty good in supporting well-structured business processes and aim at achieving a fixed goal by carrying out an exact set of operations. In contrast, those exact operations needed to fulfill a business process/workflow may not be always possible to foresee in highly complex and dynamic environments like healthcare and case management sectors, where the processes exhibit a lot of uncertainty and unexpected behavior and thereby require high degree of flexibility. Several research groups have suggested declarative models as a good approach to handle such ad-hoc nature by describing control flow implicitly and there by offering greater flexibility to the end uses.

The first contribution of this PhD thesis is to formalize the core primitives of a declarative workflow management system employed by our industrial partner Resultmaker and further develop it as a general formal model for specification and execution of declarative, event-based business processes, as a generalization of a concurrency model, the classic event structures. The model allows for an intuitive operational semantics and mapping of execution state by a notion of markings of the graphs and we have proved that it is sufficiently expressive to model ω-regular languages for infinite runs. The model has been extended with *nested sub-graphs* to express hierarchy, multi-instance *sub processes* to model replicated behavior and support for *data*.

The second contribution of the thesis is to provide a formal technique for safe distribution of collaborative, cross-organizational workflows declaratively modeled in DCR graphs based on a notion of projections. The generality of the distribution technique allows for fine tuned projections based on few selected events/labels, at the same time keeping the declarative nature of the projected graphs (which are also DCR graphs). We have also provided semantics for distributed executions based on synchronous communication among network of projected graphs and proved that global and distributed executions are equivalent.

Further, to support modeling of processes using DCR Graphs and to make the formal model available to a wider audience, we have developed prototype tools for specification and a workflow engine for the execution of DCR Graphs. We have also developed tools interfacing SPIN model checker to formally verify safety and liveness properties on the DCR Graphs. Case studies from healthcare and case management domains have been modeled in DCR Graphs to show that our formal model is suitable for modeling the workflows from those dynamic sectors.

This PhD project is funded by the Danish Strategic Research Council through the Trustworthy Pervasive Healthcare Services project (www.trustcare.eu).

Acknowledgments

It would not have been possible to finish this thesis without the help of various people.

First of all, I would like to thank my supervisor, Thomas Hildebrandt, for his constant support, guidance and encouragement throughout my PhD. Incidentally, Thomas was also my master's thesis supervisor in 2005, wherein he introduced process algebra and Bigraphs to me . Then, I realized the amazing power of formal models for the first time in my long IT career. I came from an IT background to do a PhD in application of formal methods, therefore, in initial days, I was always not sure whether I would feel comfortable in the formal methods. But he has been always supporting, encouraging me to learn new things, had time and enormous patience for discussions with me. I would be very grateful to him for all these years of working with him and also I am quite happy that I could work with him for some more time.

Further, I would like to thank my co-author and fellow PhD student Tijs Slaats for his support and his friendship. It has been always been a pleasure working with him and looking forward for a productive forthcoming year working with him. Many thanks to all the members of the Programming, Logics, and Semantics group at ITU for fruitful and very good friendly working environment. Especially, I would like to thank Hugo A. Lopez and Espen Højsgaard for sharing their thesis template, which actually saved a lot of my time and made the thesis looking nice.

As part of my PhD, I visited the IBM T.J. Watson Research Center, New York, USA in 2011 for couple months. I would like to thank Dr. Rick Hull for proving me an opportunity to visit him and his group, which gave us a chance to study and relate their work with our formal model. I also visited Microsoft Research India in 2010 for three months. I would like to thank Dr. Sriram Rajamani for being the perfect host, went far beyond his duties to offer me to share his cabin, and gave me a chance to visit their Programming Languages and Tools group and experience their scintillating work culture there.

Finally, my hearty thanks to my father and mother, who always wished me to go for higher studies. I also wish to thank my sons Siddu and Rishi for their support and their patience for not complaining even a bit when I was missing from the home for many weekends before thesis deadline. Lastly, but not the least, I would like to thank my wife Alivelu, without whose support it would't have been possible to finish my PhD. Many deep-felt thanks to her for her support and bearing with all my crazy plans of going for PhD.

Raghava Rao Mukkamala
Copenhagen, February 29, 2012

Contents

List of Tables

Listings

List of Figures

CHAPTER 1

Introduction

Organizations have always been working on improving their processes to optimize their productivity, on one hand to face the global competition and on the other hand to bring new ideas and concepts to add more value to their products and services. In order to reduce expenses and to enhance their revenues, organizations constantly look for better ways to improve their processes by automating some or whole of repeatable activities so that they can be performed at a faster rate with little or no variation. Process automation aims at streaming and standardizing the processes by reducing the human error and enhancing the operational efficiency, thereby derive a better value for products and services of a business.

A business process can be classified as a combination of a set of activities within an organization, having a clear structure identifying their logical order and dependencies to achieve a desired goal [Sara & Aguilar-Saven 2004]. The main aim of a process model is to get a clear-cut and comprehensive understanding of a business scenario or a goal. On the whole, the activities of a business process are performed in coordination in an organizational context with the help of technical environment to realize a business goal [Weske 2007].

With the help of modeling, often a very complicated business scenario can be translated into a simplified model. We can reason about a simplified model much more easily than what we can reason about the very complex scenario itself. In other words, models help us to manage complexity and also to make decisions based on the well-understood and explicitly formulated essentials of the modeled situation [Kilov 2002]. On the whole, good models helps us handle complicated problems in a clear and explicit manner. In general business process models [Weske 2007] are the primary artifacts for implementing the business processes and they contain a set activity models and execution constraints prescribing the logical order between them.

According to Gartner [Hill *et al.* 2006], in a management perspective, Business Process Management (BPM) is a management discipline that treats business processes as assets to be valued, designed, and exploited in their own right. It is a structured methodology to employ both management practices and software tools continuously to model, manage and optimize the activities and processes that interact with people and systems both within and across organizations. But a more concrete definition of BPM from the scientific community point of view [van der Aalst *et al.* 2003, Weske 2007] is that, the BPM is a methodology to support business processes using methods, techniques and software to design, enact, control and analyze operational processes involving humans, organizations, applications, documents and other source of information. In the thesis, we will adhere to the later definition of the

BPM.

Before looking further into the business process technology, let us delve down into it's historical perspective to get a better understanding of how process-centric thinking has evolved during the course of time to the state of art of current BPM methodology.

1.1 Brief Historical Perspective of Business Processes

Even though the importance of business processes were first mentioned by a management theorist, Levitt [Levitt 1960] as early as in 1960, but it was not until the1980s that the process orientation acquired real importance in the design of organizations [Sara & Aguilar-Saven 2004]. However during the 1970s, there was a lot of interest in Office automation initiative with a motivation to enhance productivity of office workers by automating the office procedures. It also received the attention of Computer Science [Zisman 1977, BURNS 1977, Ellis 1979, Ellis & Nutt 1980] with a key research focus on design methodology, software tools, and system integration techniques. Even though there was great optimism about the success of office automation, only quite few systems were successful. The systems developed in 1970s were quite rigid, embedded with complex specifications of the organizations office procedures which interfered with the work routines rather than expedite them and further more, the networking facilities and application technology were not sufficiently mature enough for the success of office automation [Ellis & Nutt 1996, van der Aalst *et al.* 2003].

In 1980s, Michel Porter introduced the concept of value chain, which is the first groundwork for the emphasis on the comprehensive understanding of a business processes that spread across the functional or departmental boundaries [Harmon 2007]. In the end of 1980s, Rummler and Brache [Rummler & Brache 1990, Rummler & Brache 1995, Harmon 2007], provided a detailed methodology on how to analyze an organization with a process-centric view and how to redesign and improve processes. They focused on organizations as systems and worked from top down to develop a comprehensive picture of how organizations were defined by processes. In the same period, Six Sigma Movement is one of the important contributions from the quality control management perspectives. Even though Six Sigma Movement has evolved as best practices from the quality control initiatives, but it failed to make a significant influence on process-centric initiatives due to its origins in quality control and a heavy emphasis on the statistical techniques [Harmon 2007].

Apart from those mention above, the most important and notable initiative is Business Process Reengineering (BPR) movement which began in 1990s. The main motivators for BPR initiative are Champy [Hammer & Champy 1993], Davenport [Davenport 1993] and Hammer [Hammer 1990], who strongly argued that organizations must think in terms of comprehensive processes, in the similar lines of Porter's value chains and Rummler's organization level. The methodology proposed under BPR is that, the processes should be conceptualized as complete entities and then, Information Technology (IT) should be used to integrate these comprehensive processes. Further BPR theorists had observed that IT applications could cut across departmental boundaries to eliminate inefficiencies and yield huge gains in coordination [Harmon 2007].

In the 1990s, along with BPR movement, there was again a huge interest in the IT field to build systems to support business processes, which gave birth to new type of software applications called business process management systems (BPMS) and we will explore them in the next section.

1.2 Business Process Management and IT

In mid 1990s, most of the developments in business processes were driven by Information Technology. We can observe two broad categories in the software applications that emerged in the initiative of business process management and redesign. The first category of systems is Enterprise resource planning (ERP) systems. These systems are based on modules such as inventory, accounting and human resources and they are suitable for the standardized processes that are most common between the organizations and they can be considered as integrated business process management system [van der Aalst *et al.* 2003].

The second category of applications are Workflow Management Systems that provide support for automating and execution of business processes. Workflow is a concept closely related to Office automation from 1970s and the business process reengineering that began in 1990s [Georgakopoulos *et al.* 1995, van der Aalst *et al.* 2003, Russel & Ter Hofstede 2009] and according to Workflow Management Coalition [WfMC 1999], workflow is defined as "The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules".

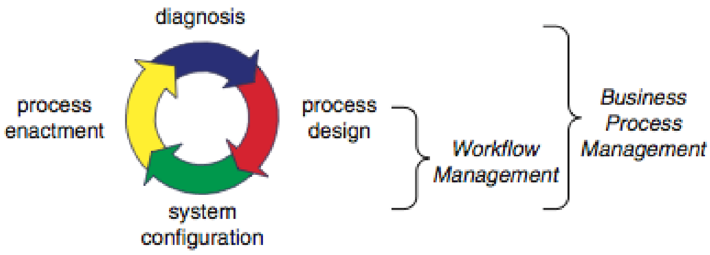


Figure 1.1: The BPM lifecyle to compare Workflow Management and BPM [van der Aalst *et al.* 2003]

The relationship between the workflow management systems and business process management systems can be explained in a better manner by using the figure 1.1, which shows four key phases of BPM life cycle [van der Aalst *et al.* 2003]. The first phase is process design, where business processes are identified and are modeled using various existing business process modeling techniques. In the configuration phase, the modeled processes will be implemented using software applications or using off the shelf BPM products. The third phase is the enactment phase where the business processes are realized and the process instances are initiated to fulfill the business goals. The last phase involves evaluation of process logs and other information produced by the process instance during enactment phase to analyze and improve the performance of a process. The focus of workflow management is mostly on implementing the lower half of BPM life cycle, from process design to process enactment, which does not generally include the diagnosis phase. On the other hand, business process management also focusses on the analysis, flexibility and other process improvement techniques.

One of the major paradigm shift during the evolution of applications in the IT is moving from data orientation to process orientation [van der Aalst *et al.* 2003, Aalst 2004]. During 1970s and 1980s the application development was dominated by data driven approaches. In those times the focus of the applications was to store and retrieve information and there by started adopting data modeling as a base for building applications in IT. These applications often neglected the process centric approach in modeling the business processes. However Business process reengineering movement evolved during 1990s strongly advocated for process centric approach and thereby more emphasis on process centric approach which can be observed in the later IT applications that were build for supporting business processes.

Another interesting paradigm for modeling business processes is the artifact-centric approach [Kumaran *et al.* 2003, Nigam & Caswell 2003, Gerede *et al.* 2007, Gerede & Su 2007, Bhattacharya *et al.* 2007b, Cohn & Hull 2009], which strongly argues that data design should be elevated to the same level as control flows for data rich workflows and business processes. Business artifacts combine the data aspects and process aspects in a holistic manner and an artifact type contains both an information model and lifecycle model, where information model manages the data for business objects and lifecycle model describes the possible ways the tasks on business objects can be invoked.

1.2.1 BPM Standardization Approaches

One of the key factors for failure of office automation in 1970s was the lack of unified standards for design methodology and modeling systems. However considerable efforts have been made in the last two decades for standardization in workflows and business process management. The Workflow Management Coalition [Workflow Management Coalition 1993] was formed in 1993 by major product vendors from workflow and BPM, with a goal of achieving interoperability and other process related standards among the product vendors. Now it has more than 300 member organizations, workflow users, interested groups from academia and one of its notable contribution is XML Process Definition Language (XPDL) [Workflow Management Coalition 2008], for exchange business process definitions between different workflow vendors.

A more later standardization effort in BPM community were focussed at developing standards for business process modeling and execution. The Web Services Business Process Execution Language (WS-BPEL) [OASIS WSBPEL Technical Committee 2007] has evolved as a standard process oriented language for service composition in the context of Service oriented architecture (SOA) and web services. Even though it has been widely adopted by different workflow product vendors, but lack of formal semantics for WS-BPEL has led to different implementations by different vendors and there by exchange of BPEL processes form one tool to other became difficult. Furthermore, WS-BPEL does not have a graphical language which makes it difficult to use it for modeling of business processes.

Further, Business Process Modeling Notation (BPMN) [Object Management Group BPMN Technical Committee 2011] has been introduced as a modeling language for business processes with graphical notation. The processes modeled in BPMN can not be executed directly, but they can be translated to WS-BPEL for execution. In the recent years, it has been widely adopted as a modeling language for business processes, since there is no formalization for BPMN accepted by standards committee, different interpretations could be possible for some of its concepts [Hofstede *et al.* 2010]. Even though the BPMN has become more mature and expressive in the recent versions, but it still lacks clear semantics for some of its constructs, for example ad-hoc sub processes.

In addition to the above, there also exists standards for other approaches to model business processes such as activity diagrams of Unified Modeling Language (UML) [OMG 2007] and Event driven Process Chains (EPCs) [Scheer 1998]. UML activity diagrams are not meant to be executed directly and they don't have any formalization accepted by the OMG UML standing committee [Hofstede *et al.* 2010], even though formal semantics for UML activity diagrams were defined in [Eshuis 2002].

1.3 Why Formal Models?

Formal methods is a technique to model complex systems as mathematical entities. The use of formal methods is strongly advocated by many researchers [Bowen & Stavridou 1993, M.Clarke *et al.* 1999, van der Aalst *et al.* 2003] as a way of increasing confidence in building practical and complex systems, as the usage of formal models leaves no scope for ambiguity.

In general business processes involve many stakeholder right from the domain experts to process modeler with varied technical backgrounds. Hence usage graphical languages to make the processes easily understood by different stakeholders is a common practice in business process modeling. Furthermore, business process models can be quite complex in nature, and hence there should not be any scope for many interpretations of the same scenario. Lack of formal semantics for some of the business process languages has resulted in different implementation by different vendors. Therefore the usage of a formal language for specification of complex scenarios will eliminate the scope for ambiguity and will guarantee that there will not be any chance for alternative interpretations.

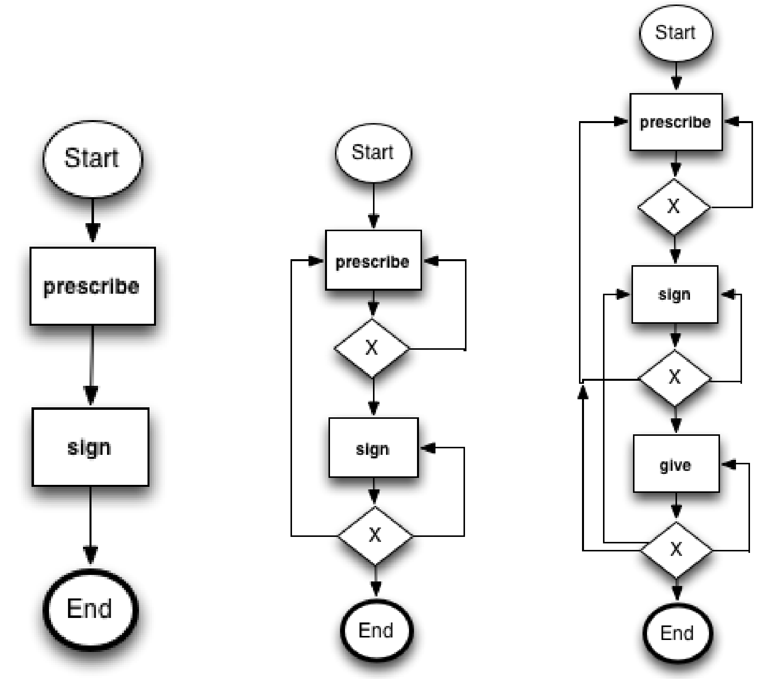
Usage of formal models for specification of business processes has another advantage of using analysis techniques to analyze processes. Since business processes can be complex, it is always advantageous to detect errors at the design stage itself, instead of correcting them after deploying the processes. Moreover, formal models can be used to guarantee certain properties (such as deadlock freeness etc) on business processes, which can be used to analyze them. Now a days, model checking and verification techniques have been developed to a large extent. Usage of formal models for business processes can make use of these model checking and verification techniques to reason about the properties on processes and to provide suitable guidance to the process modeler at the design time.

1.4 Motivation for Declarative Models

There were quite large number of workflow and business processes management systems developed in the past decade and they have been quite successful in providing support to users for the enactment of their processes. However their applicability is still limited to specific sectors like insurance and banking. Current business process technology is pretty good in supporting well-structured business processes with well-defined set of tasks, showing little or variations in their possible execution sequences [Reichert & Dadam 1997, van der Aalst *et al.* 2003, van der Aalst *et al.* 2009]. Traditional business process systems aim at computing a specific algorithm, carrying out an exact set of operations to achieve a fixed goal.

In contrast, the exact sequences of operations needed to fulfill a business process/-workflow may not be always possible to foresee in highly complex and rapidly changing environments [Strong & Miller 1995, Reichert & Dadam 1997], such as healthcare and case management domains. The processes in those domains exhibit a lot of uncertainty, unexpected and ad-hoc behavior. In case management and healthcare domains, the end users like case workers, doctors/nurses will have better knowledge than the process modelers regarding how to deal with un-expected behavior. In case of traditional business processes, any behavior that is not foreseen by the process modelers can not be realized by the process instances at the time of enactment. In those domains, traditional business processes technology did not make considerable impact, as they exhibit too rigid behavior, on contrary healthcare and case management domains require high degree of flexibility.

Declarative process models have been suggested by several research groups as a good approach to handle such ad-hoc nature by describing control flow implicitly and there by offering greater flexibility to the end uses. A key difference between declarative and imperative process languages is that the control flow for the first kind is defined *implicitly* as a set of constraints or rules, and for the latter is defined *explicitly*, e.g. as a flow diagram or a sequence of state changing commands. There is a long tradition for using declarative logic based languages to schedule transactions in the database community, see e.g. [Fernandes *et al.* 1997]. Several researchers have noted [Davulcu *et al.* 1998, Senkul *et al.* 2002, Singh *et al.* 1995, Bussler & Jablonski 1994, van der Aalst *et al.* 2009, van der Aalst & Pesic 2006a, Pesic 2008] that it could be an advantage to use a declarative approach to achieve more flexible process descriptions in other areas, in particular for the specification of case management workflow and ad hoc business processes. The increased flexibility is obtained in two ways: Firstly, since it is often complex to explicitly model all possible ways of fulfilling the requirements of a workflow, imperative descriptions easily lead to over-constrained control flows. In the declarative approach any execution fulfilling the constraints of the workflow is allowed, thereby leaving maximal flexibility in the execution. Secondly, adding a new constraint to an imperative process description often requires that the process code is completely rewritten, while the declarative approach just requires the extra constraint to be added. In other words, declarative models provide flexibility for the execution at run time and with respect to changes to the process.



(a) Flow chart 1 (b) Flow chart 2 (c) Flow chart 3

Figure 1.2: Give medicine example in Flow chart

As a simple motivating example, consider a hospital workflow extracted from a real-life study of paper-based oncology workflow at Danish hospitals [Lyng *et al.* 2008, Mukkamala *et al.* 2008]. As a start, we assume two events, *prescribe* and *sign*, representing a doctor adding a prescription of medicine to the patient record and signing it respectively. We assume the constraints stating that the doctor must sign after having added a prescription of medicine to the patient record and not to sign an empty record. A naive imperative process description may simply put the two actions in sequence, *prescribe;sign*, which allows the doctor to first prescribe medicine and then sign the record as shown in the figure 1.2-(a). In this way the possibilities of adding several prescriptions before or after signing and signing multiple times are lost, even if they are perfectly legal according to the constraints. The most general imperative description should start with the prescribe event, followed by loops allowing either sign or prescribe events and only allow termination after a sign event as shown in the figure 1.2-(b). If the execution continues forever, it must be enforced that every prescription is eventually followed by a sign event.

With respect to the second type of flexibility, consider adding a new event *give*, representing a nurse giving the medicine to the patient, and the rule that a nurse must give medicine to the patient if it is prescribed by the doctor, but not before it has been signed. For the most general imperative description we should add the ability to execute the *give* event within the loop after the first *sign* event and not allow to terminate the flow if we have had a *prescribe* event without a subsequent *give* event as shown in the flowchart 1.2-(c).

The main point of this example is, that we in many cases may want to allow any execution that satisfy the given requirements, but not to constrain ourselves to a specific way of fulfilling the requirements. In order to explain the differences between imperative and declarative approaches, we will make use of the figure 1.3 from [van der Aalst & Pesic 2006a, Pesic 2008], where the behavior exhibited by the procedural and declarative modeling languages is compared.

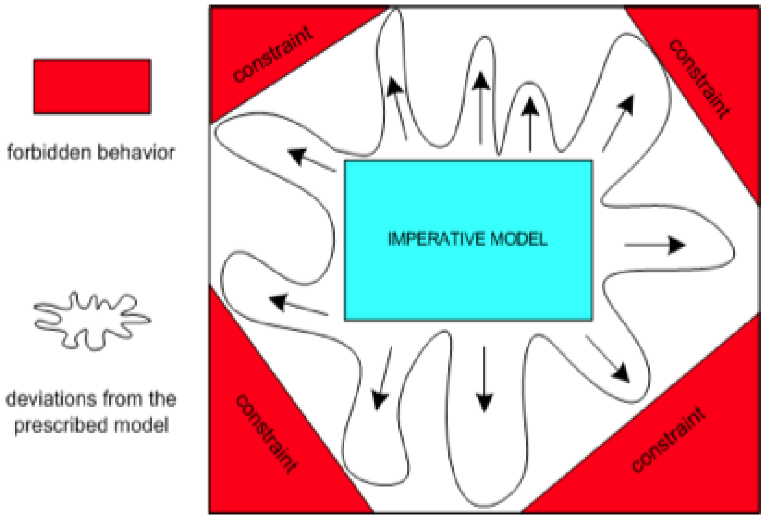
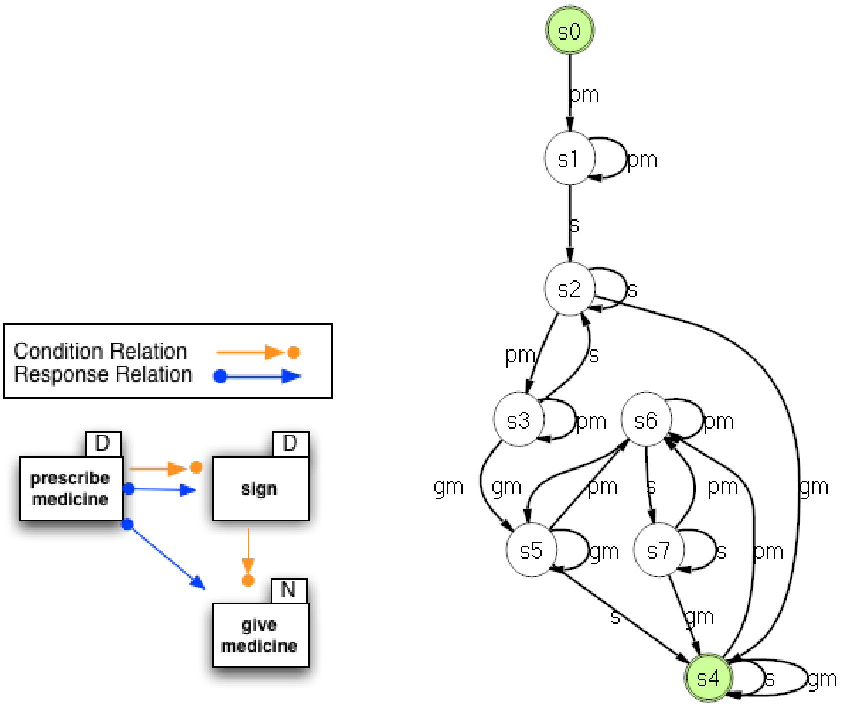


Figure 1.3: Declarative verses Imperative Approaches [van der Aalst & Pesic 2006a]

Imperative languages start specifying models from *inside out* style i.e specifying the control flow explicitly to model the behavior that we want to have in the process. The imperative models focus on specifying *how* the requirements should be fulfilled, where as the declarative models focus on specifying *what* should be fulfilled [van der Aalst & Pesic 2006a, van der Aalst *et al.* 2009], by offering all the possible behavior and using constraints to eliminate the behavior we don't want to happen in the process as shown in the figure 1.3. In the imperative models one may tend to over specify process as the control flow has to be specified explicitly, where as declarative models tend to under specify as the control flow is implicitly specified, there by leaving more options to the end users.



(a) Model (b) Mapped to labeled transition system

Figure 1.4: Give medicine example in DCR Graphs

The above mentioned hospital workflow is modeled using declarative modeling approaches as shown in the figure 1.4-(a), where we have used our formal model DCR Graphs to model the workflow. The model contains the same three events *prescribe medicine* (pm), *sign* (s) and *give medicine* (gm), moreover events can be executed any number of times in any order until unless they are constrained by the relations. The condition relation from *prescribe medicine* to *sign* specifies that *prescribe medicine* must have been done at least once before executing *sign*. Similarly the response relation between *prescribe medicine* and *give medicine* specify that the *give medicine* should be executed at least once after executing *prescribe medicine*, but it does not stop from executing *give medicine* many times. The behavior offered by the model can be seen in the figure 1.4-(b), where the execution semantics are mapped to labeled transition system. Since events can be executed any number of times in declarative models, there will not be any well defined explicit termination, but on the other hand they have a notion of acceptance i.e when they are allowed to stop. The green color states in the figure 1.4-(b) represent accepting states, where all the constraints are satisfied.

One can easily observe that declarative models offer more choices to the end users, by under-specification of the process. In case management and healthcare domains, end users like case workers, doctors/nurses will have better knowledge regarding how to deal with un-expected behavior than the process modelers. Hence, we strongly argue that by using minimal specification in declarative models, you can leave more flexibility to end users of the process.

1.5 Thesis Statement

Having discussed background and motivation of research problem, we will now discuss the research goal of the thesis in this section. This PhD dissertation is part of the TrustCare project and therefore we will first describe the overall goals and key hypothesis of TrustCare project, then we will proceed to define the research goal for the thesis.

1.5.1 TrustCare Project

Trustworthy Pervasive Healthcare Services (TrustCare[1]) project is a strategic and interdisciplinary research effort aimed at innovation of effective and trustworthy it-support for pervasive healthcare services in collaboration with the industrial partner, as well as innovation in research across areas in experimental and theoretical research in computer science [Hildebrandt 2008]. The key research partners in TrustCare project are 1) I T University of Copenhagen 2) Department of Computer Science, Copenhagen University 3) Resultmaker A/S, a Danish IT provider for workflow management systems, which has been quite successful in providing workflow solutions to Danish public sector for the last 12 years, using their patented declarative workflow management system Online Consultant.

The key hypothesis of the TrustCare project is that the patented workflow model of the Resultmaker Online Consultant can be extended to provide both trustworthy and useful it-support for interacting and dynamically changing healthcare services, by formalizing and extending the underlying process model using techniques obtained from theoretical research in domain-specific languages, process models, and type-theory, and integrating this work with experimental research in state-of-the art user-interfaces for pervasive healthcare services rooted in the activity based computing paradigm. The synergy between the development of the Online Consultant and the research in experimental and theoretical computer science is described in the figure 1.5.

[1] The TrustCare `(www.trustcare.eu)` project is funded by Danish Strategic Research Council vide grant # 2106-07-001

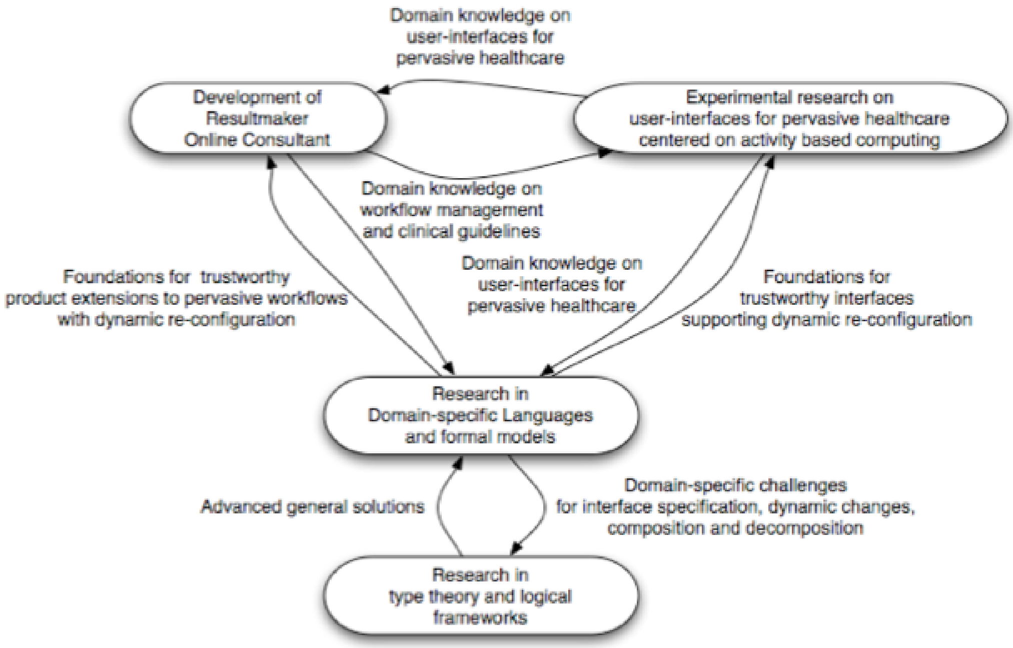

Domain knowledge on
user-interfaces for
pervasive healthcare
Development of
Resultmaker
Online Consultant
Experimental research on
user-interfaces for pervasive healthcare
centered on activity based computing
Domain knowledge on
workflow management
and clinical guidelines
Foundations for trustworthy
product extensions to pervasive workflows
with dynamic re-configuration
Domain knowledge on
user-interfaces for
pervasive healthcare
Foundations for
trustworthy interfaces
supporting dynamic re-configuration
Research in
Domain-specific Languages
and formal models
Advanced general solutions
Domain-specific challenges
for interface specification, dynamic changes,
composition and decomposition
Research in
type theory and logical
frameworks

Figure 1.5: TrustCare project research methodology

The developers at Resultmaker and the experimental research in user-interfaces for pervasive healthcare centered on activity based computing will cross-fertilize each other by providing respectively domain knowledge on workflow management systems and clinical guidelines to the research on activity based computing and domain knowledge on pervasive user- interfaces to the development of workflow management. Likewise, the two groups will provide domain knowledge and the motivation for new features such as dynamic re-configuration and awareness of changes to the group researching in domain specific languages and formal models, which in return will provide the foundations for trustworthy foundations of the On-line Consultant and activity based computing paradigm supporting the proposed extensions. The three groups will thus interact in cycles between the identification of challenges and the need for new features in the product development and prototyping of user-interfaces to development of domain-specific languages and models providing a trustworthy foundation and back to integration of the models and the features into the product and prototypes. Finally, the research in type theory and logical frameworks will be fed by the research in domain-specific languages and models with domain-specific challenges (motivated by the suggested product extensions) and return advanced general solutions to the problem.

1.5.2 Research Goal

Since this PhD project is part of TrustCare project and its research goal is guided by the overall goals and research methodologies of the TrustCare project. As explained in the key hypothesis of the TrustCare project, one of the key challenges in the TrustCare project is to formalize the workflow model of Resultmaker Online Consultant, as it has no formal semantics, but only has a commercial workflow management implementation.

Aligned with the overall focus of the TrustCare project, we will now formulate the goal of the thesis as follows,

> The research goal is to show that it is possible to formalize core primitives of Resultmaker declarative workflow model and further develop it as a comprehensive formal model for specification and execution of workflows based on declarative modeling. The formal model should allow safe distribution of workflows based on a model-driven approach and analysis based on formal verification of processes using model checking.

In order to explain the concrete requirements of the research goal in a better manner, we will further split the research goal into three research questions as follows.

1. *What are the formal semantical models suitable for describing flexible workflow processes for healthcare and other dynamic services?*

   Our research goal as part of this question is to provide formal semantics to the key primitives of the Resultmaker declarative workflow model and further develop it as a comprehensive formal model that is suitable for specification and execution flexible workflows with a key focus on healthcare, case management and other dynamic sectors. We will use the Resultmaker workflow model as a starting point for our goal of developing a comprehensive formal model on declarative modeling primitives, since their workflow method has been proven to be flexible and successful in the Danish public sector.

   Furthermore, our focus is to provide formal semantics to their declarative workflow, but we are not concerned with how these formal semantics could be implemented by their commercial workflow management system and how much flexible will it be then compared to the other existing workflow management systems, for example based on user evaluations. However, we intend to develop a prototype workflow management based on the formal model developed in the PhD thesis to prove that our formal model can be easily implemented by a commercial workflow management system to offer flexible workflows based on declarative modeling. Further, we will also model some use cases from healthcare and case management domains to show the practicality and adequacy of our formal model.

2. *How should one describe interfaces, contracts and interactions for declarative workflows to allow safe distribution?*

   As part of this research question, we intent to study the *distributed synthesis problem*: Given a global model and some formal description of how the model should be distributed, can we synthesize a set of local processes with respect to this distribution which are consistent to the global model? Here our focus is to study about how to distribute a declarative workflow based on top-down model-driven approach, as a global specification into a set of communicating local processes such that the local processes still keep their declarative nature. Furthermore, the goal of the distribution of the global specification should be safe, in the sense that the behavior exhibited by the local processes should be consistent with the behavior exhibited by the global process.

3. *What are the suitable model checking and verification techniques for enhancing trustworthiness in declarative workflows?*

   A drawback of the declarative approach is that the implicit definition of the control flow makes the processes less easily perceived by the users. For example if the users want to know what are the next possible events to execute, one has to solve the set constraints to compute the next possible events.

   We interpret the meaning of *trustworthiness* in the context of the declarative business processes that the process will exhibit the behavior that the user has modeled. Hence in order to enhance the trustworthiness in the declarative processes, one could use formal verification techniques to analyse the processes and guarantee that certain properties will hold. As part of this research question, we would like to explore formal verification techniques that can be applied to the declarative processes.

The research goal of the thesis will be achieved by solving the above research questions. Furthermore, the ideas and concepts developed in the thesis are presented periodically in the workshops of *Interest Group for Processes and IT* [Hildebrandt 2010], which is a forum consisting of Danish IT vendors for workflow management systems, public organizations and researchers. The next section gives a brief overview of the thesis and how these questions have been addressed.

1.6 Thesis Outline

We will now provide a brief outline and structure of the remainder of this thesis. First we will state the list of publications that are published as part of knowledge dissemination in the PhD project, then we will give a brief overview of the rest of the chapters by quoting which publications have been covered in the chapters.

1.6.1 List of Publications

The following papers have been peer reviewed and published at various conference or prestigious workshops associated with conferences.

(1) Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Janus Boris Tøth. The Resultmaker Online Consultant: From Declarative Workflow Management in Prac-

tice to LTL. In Proceedings of First International Workshop on Dynamic and Declarative Business Processes (DDBP 2008).

(2) Karen Marie Lyng, Thomas T. Hildebrandt, and Raghava Rao Mukkamala. From Paper Based Clinical Practice Guidelines to Declarative Workflow Management. In proceedings of 2nd Inter- national Workshop on Process-oriented information systems in healthcare (ProHealth 2008).

(3) Thomas Hildebrandt and Raghava Rao Mukkamala. Distributed dynamic condition response structures. In Proceedings of International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10), Paphos, Cyprus, March 2010.

(4) Raghava Rao Mukkamala and Thomas Hildebrandt. From Dynamic Condition Response Structures to Buchi Automata. In proceedings of 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010).

(5) Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In Kohei Honda and Alan Mycroft, editors, PLACES, volume 69 of EPTCS, pages 59Ð73, 2010.

(6) Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. Nested Dynamic Condition Response Graphs. In Proceedings of Fundamentals of Software Engineering (FSEN), April 2011.

(7) Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. Designing a Cross-organizational Case Management System using Dynamic Condition Response Graphs. In Proceedings of IEEE International EDOC Conference, 2011.

(8) Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. Safe Distribution of Declarative Processes. In 9th International Conference on Software Engineering and Formal Methods (SEFM) 2011, 2011.

(9) Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. Declarative Modelling and Safe Distribution of Healthcare Workflows. In International Symposium on Foundations of Health Information Engineering and Systems, Johannesburg, South Africa, August 2011.

(10) Søren Debois, Thomas Hildebrandt, Raghava Rao Mukkamala, Francesco Zanitti. Towards a Programming Language for Declarative Event-based Context-sensitive Reactive Services. Nordic Workshop on Programming Theory. Västerås, Sweden. October, 2011.

(11) Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. Declarative Modelling and Safe Distribution of Healthcare Workflows. In LNCS Post proceedings of International Symposium on Foundations of Health Information Engineering and Systems, January, 2012.

1.6.2 Chapters Outline

In this section we will give a brief outline of the chapters of the thesis and also mention which papers listed above contribute to the chapters.

- Chapter 2: Background

  This chapter will introduce background and motivation for our formal model developed in the thesis. First it will introduce our first attempt to formalize Resultmaker's declarative workflow model *Process Matrix* using Linear Temporal Logic [Pnueli 1977]. Then we will introduce the case study conducted in Danish hospitals regarding for lung cancer treatment, which will be used as one of the running example for the rest of the thesis. This part of the chapter covers the publications (1) and (2) mentioned above.Later we will give a brief introduction to another declarative process model *Declare* [van der Aalst *et al.* 2010a, van der Aalst & Pesic 2006b, van der Aalst & Pesic 2006a], from which our formal derives some motivation. Finally, we will introduce base formalism behind our formal model, *Event Structures* [Winskel 1986] and explain key primitives of labeled event structures.

- Chapter 3: Dynamic Condition Response Graphs

  We will introduce our formal model Dynamic Condition Response Graphs(DCR Graphs) in this chapter. First, we will describe how we have generalized Event Structures to define the semantics of DCR Graphs, then we will introduce the key primitives and operational semantics of DCR Graphs. The execution semantics for finite runs are mapped to labeled transition system and for infinite runs, where as the semantics for infinite runs have been mapped to Büchi automata. Graphical notation for modeling DCR Graphs along with the runtime notation will also be introduced at the end of the chapter. This chapter covers the work published in papers (3), (4) and (5) from the list mentioned in the previous section.

- Chapter 4: Dynamic Condition Response Graphs - Extensions

  Some important extensions to DCR Graphs developed in the thesis will be introduced here. First we will extend DCR Graphs to allow for modeling of nested sub-graphs, Nested Dynamic Condition Response Graphs. Further we extend the Nested Dynamic Condition Response Graphs with multi-instance subprocesses to model the replicated behavior in declarative processes. Finally we add a basic support for data for DCR Graphs, by considering data as global store of shared variables. This chapter covers the work published in the papers (6) and (7).

- Chapter 5: Distribution of DCR Graphs

  In this chapter we will introduce a technique safe distribution of DCR Graphs as a set of communicating local graphs to represent local behavior. First we will introduce and define the notion of projection and composition on DCR Graphs, then define the notion networks of DCR Graphs. We will also prove that the distribution is safe in the sense that the behavior exhibited by the local graphs is consistent with the behavior exhibited by the global graph. Further we also extend the distribution technique to the nested DCR Graphs and distribute the healthcare example which was introduced in the previous chapters. This chapter covers work published in the papers (8), (9) and (11).

- Chapter 6: Formal Verification, Tools and Implementation

  In this chapter, we introduce the notion of safety and liveness properties on DCR Graphs and further describe how to verify these properties using a model checking tool. As part of formal verification, we will describe how to encode DCR Graphs into PROMELA [Spin 2007] code and verify safety and liveness properties using SPIN [Spin 2008] model checking tool. We will also describe briefly our experience in using ZING [Microsoft-Research 2010] model checker to verify safety properties on DCR Graphs. Finally, we will a brief description of prototype tools for DCR Graphs implemented as part the thesis.

- Chapter 7: Conclusion and Future Work

  This chapter will conclude the results achieved in the thesis and also provides a detailed section explaining about the future work on DCR Graphs.

Chapter 2
Background

This chapter provides a brief introduction to the formalisms and industrial process models that served as motivation behind the development of our formal model Dynamic Condition Response Graphs (DCR Graphs). First of all, section 2.1 describes about the process model employed by our research industrial partner Resultmaker A/S, namely Resultmaker Online Consultant (ROC). Later in the section 2.2, we will describe very briefly about the *Declare* framework [van der Aalst *et al.* 2010a] and its declarative process languages (DecSerFlow [van der Aalst & Pesic 2006b], ConDec [van der Aalst & Pesic 2006a]). Finally, we will give a introduction to Event Structures [Winskel 1986] in the section 2.3 and explain why we have chosen Event Structures to base our formalism DCR Graphs.

2.1 Resultmaker Online Consultant - A Declarative Workflow

In this section, we describe the process model employed in the Resultmaker Online Consultant (ROC) workflow management system as an example of a declarative workflow language used in practice. The ROC workflow management system has evolved from Resultmaker's industrial experiences obtained during the process of authoring solutions for the Danish public sector, and has been used successfully since several years in Denmark and other European countries. It is based on a shared data architecture and electronic forms (updating the shared data) as the key basic activity. Hereto comes activities for connecting to external systems, invitation to participants and digital signatures and other features. The process model employed in ROC is called *Process Matrix*, which is a patented[1] declarative process model developed by Resultmaker.

The key primitives of Process Matrix will be introduced briefly in the later sections and then we will further describe the how these key primitives are formalized using Linear Temporal Logic (LTL) [Pnueli 1977] in line with the approach proposed by van der Aalst and Pesic in DecSerFlow [van der Aalst & Pesic 2006b] and ConDec [van der Aalst & Pesic 2006a]. This work is done as one of the very first steps of Trustworthy Pervasive Healthcare Services (TrustCare[2]) [Hildebrandt 2008] research project. TrustCare is a strategic and interdisciplinary research effort aimed at innovation of effective and trustworthy it-support for pervasive healthcare services by combining research in formal process models, logic, domain specific languages, and pervasive

[1] US Patent # 6,895,573

[2] This project is supported by the Danish Research Agency through grant #2106-07-0019.

user interfaces with the Resultmaker's industrial experience on workflow managements, with cross-fertilization of experimental and theoretical research in computer science. As part of the project, the primary goal is to develop formal foundations of trustworthy and declarative flexible workflows with a key focus on the health care sector. Further, the work on formalization of ROC [Mukkamala *et al.* 2008, Lyng *et al.* 2008] has been published at workshops affiliated to BPM-2008 and EDOC-2008 conferences and received good feedback.

In the subsequent sections we will introduce the ROC workflow architecture and it's key components, in particular the declarative primitives of the ROC process model, referred to as the *Process Matrix* and describe how we have formalized the key primitives [Mukkamala *et al.* 2008]. Later, we will describe a field study of Oncology workflow conducted in Danish hospitals [Lyng *et al.* 2008] and also demonstrate how the oncology workflow can be modeled in ROC.

2.1.1 Resultmaker Online Consultant - Formalization

The key primitives of the ROC Process Matrix are *sequential* and *logical* predecessor relations between activities, and along with *activity conditions* and *dependency expressions* for each activity. Sequential predecessor imposes precedence among activities. If an activity `A` is a sequential predecessor for the activity `B`, then it informally means that activity `A` must be executed before `B` can be executed. Note that by default, any activity can be executed any number of times. On the other hand, If `A` is declared as a *logical* predecessor of `B`, then it means that it is a sequential predecessor with the additional constraint saying that `B` must be re-executed eventually after any re-execution of `A`. A prototypical example of logical predecessor could be to have a logical predecessor between activities `A` and `B`, when `A` is an activity representing filling out a loan/grant application and `B` is an activity of evaluating or signing it . Activity conditions and dependency expressions refer to values of variables in the shared data store and are dynamically evaluated after each step of the workflow. An activity condition determines if an activity is currently included in the workflow instance (i.e. it is active) and a change in a dependency expression determines that an activity must be re-executed. Activity conditions facilitate reuse of a single process description for different purposes with different variants: One just adds a new Boolean variable to the shared data store and use it to toggle the inclusion or exclusion of activities. Dependency expressions allow for a description of logical dependency similar to the logical predecessor constraint, but are based on changes in data rather than re-executions of activities and thus allow declaring a more fine-grained dependency based on data values. In the example of filling out a grant application, for example, one may use a dependency expression to declare that the signature activity has to be re-executed if the data in the budget is changed, but not if the name of the project is changed, even though both values are entered in the grant application form.

ROC is a user-centric workflow management system based on a shared data store and so-called *eForms* as its principal activities. An eForm is a web based

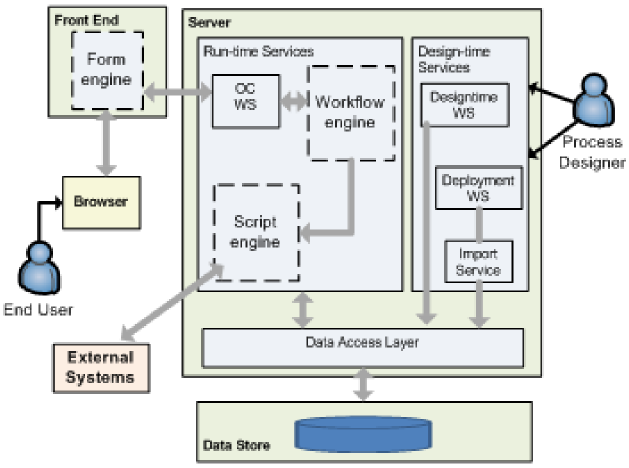


Figure 2.1: The Online Consultant Architecture.

questionnaire presented to the users of the system by the front end Form engine. The fields in the eForms are mapped to variables in the shared data store.

Fig. 2.1 shows the overall architecture of ROC. The Run-time services constitute components that execute a ROC process instance, while the Design-time services constitute e.g. tools for process description and design of eForms. ROC has its own eForm designer tool, but also supports forms developed in Microsoft InfoPath.

2.1.1.1 Process Modeling Primitives

In this section, we describe the key process modeling primitives of ROC.

2.1.1.1.1 Activities

Primarily, ROC has four pre-defined activity types.

1. **eForm Activity:** It is the principal activity of ROC and the data filled in by the users in the eForms will be available to all activities of the workflow instance through the shared data store. eForms are appended to the activities and each activity can contain only one eForm. At run-time when an eForm activity is executed, the corresponding eForm will be displayed to the users for human interaction. If any of the variables on which an eForm activity *A* depends on, is changed by another activity, while the form is being displayed (and edited) by the user, the activity *A* will be skipped when the form is attempt to submit by the user and the user will be notified. In this way eForm activities are guaranteed to run atomically and in isolation.

2. **Invitation Activity:** This type of activity attaches a role to an external user (identified by an email address) and sends him an invitation link to the process instance via email notification.

3. **Signing Activity:** In order to provide authentication for the data filled in by the users, the ROC uses Signing Activity. The user data on eForms will be digitally signed by using XML digital signatures syntax [D. Eastlake 2002] and user's digital certificates. A single signing activity supports signing of data from multiple eForms.

4. **External Activity:** Via a general script engine it is possible to connect to any external system, e.g. for automated tasks.

In our effort to formalize key primitives of ROC, we have only considered eForm activities.

2.1.1.1.2 Control Flow Primitives

ROC contains the following control flow primitives which controls the execution of process instances.

1. **Activity Condition**: Every activity in ROC has an attached activity condition, which is a Boolean expression that reference variables from the shared data store. Activities are included in the workflow instance for execution only if their activity conditions evaluates to true, on the other hand they will be skipped from the list of activities stacked for execution.

   The boolean variables used in activity conditions are referred to as *purposes*. The reason for this terminology is that, activity conditions makes it easy to reuse a process description for a different purpose in a different variant: One just adds a new purpose variable and use it in activity conditions to toggle the inclusion of relevant and exclusion of irrelevant activities. Since activity conditions refer to data values from shared data store, they will be evaluated after execution of each activity, so the inclusion of an activity in the workflow can be changed within in the lifetime of the workflow instance. As described below, changing an activity from non-active to active may influence the state of other activities that logically depend on the activity.

2. **Sequential Predecessors**: If an activity `A` is declared to be a sequential predecessor of activity `B`, then in any process instance `A` must be executed before `B` can be executed. However, the sequential predecessor has only effect if the predecessor activity `A` is included in the workflow instance as per its activity condition. That is, if the activity condition for `A` is false at certain point of time, then activity `B` can be executed even if `A` is a sequential predecessor of `B` with a status of non-executed. At a later point of time, if the activity `A` becomes part of the workflow instance (because the activity condition for activity `A` changes from false to true) after `B` got executed, it will not not have any effect on the execution status of activity B.

3. **Logical Predecessors**: If an activity `A` is declared to be a *logical* predecessor of activity `B`, then `A` is a sequential predecessor of `B`, but in addition, if activity

`A` gets re-executed, reset, or becomes part of the workflow after activity `B` has been executed and then if the activity `B` is active at that time, then activity B is also reset, and thus must be re-executed at a later time (unless it stays inactive for the rest of the instance lifetime, i.e its activity condition continuously evaluates to false). Note that activity resets in this way can propagate through a chain of (currently active) logical predecessors. As also mentioned in the introduction the *Process Matrix* model includes an additional advanced feature called *dependency expressions*.

A dependency expression is a set of expressions attached to an activity. Like activity conditions, dependency expressions can also contain references to variables in the shared store. However, where an activity condition evaluates to a Boolean value, a dependency expression can evaluate to any value, and any change in the value of a dependency expression associated to an activity will reset the activity status to non-executed.

2.1.1.1.3 Additional Primitives

In addition to activities and control flow constructs described above, the ROC also have transactions and resources as explained below. But we have not considered transactions in the work of ROC formalization.

a) **Transactions:** A ROC transaction groups a set of activities to be executed in transaction mode. The ROC transactions differ from standard transactional semantics in the way that they are neither long running nor be rolled back. Instead, as also found in web-service orchestration languages such as WS-BPEL, they can have a compensating logic to be executed in case a transaction has to be aborted. Transactions can be either signed or unsigned. Signed transactions involves signing the data using digital certificates by single/multiple parties containing many eForms.

b) **Resources/Roles:** ROC has a simple resource model that uses Roles to define allowed behaviour of different users within the system. Each Role is assigned an access right for each activity of a workflow. The possible access rights are Read (R), Write (W) and Denied (D). The Read access is the default access right that allows a user with the particular role to see the data of an activity. Write access right allows the user to execute an activity and also to input and submit data for that activity. A Denied access right has the effect of making the activity invisible to the user. As for transactions, we will leave the formalization of Roles for future work.

   Activities are executed by default at least once, but possibly many times in a process instance. The ROC runtime state records whether an activity has been executed or not. If an activity has state *executed*, its state can be reset to *not executed* under certain circumstances described above.

	Activities	Roles			Predecessors	Activity Condition
		App	CW	Mgr		
1	Application	W	R	R		
2	Register Customer Info	W	W	W		
3	Approval 1	D	W	R	$\overset{*}{1}$,2	
4	Approval 2	D	R	W	$\overset{*}{1}$,2	$\neg Rich$
5	Payment	R	W	R	$\overset{*}{3}, \overset{*}{4}$	$\neg Hurry \wedge Accept$
6	Express Payment	R	W	R	$\overset{*}{3}, \overset{*}{4}$	$Hurry \wedge Accept$
7	Rejection	R	W	R	$\overset{*}{3}, \overset{*}{4}$	$\neg Accept$
8	Archive	D	W	R	$\overset{*}{5}, \overset{*}{6}, \overset{*}{7}$	

Table 2.1: Loan application Process Matrix

2.1.1.2 The Process Matrix

There is yet no formal graphical notation for ROC workflow processes. However, there is a guideline for how to identify and specify activities, roles/actors and constraints in a tabular format. This table is referred to as the *Process Matrix*, which is also used as name for the process model. Practical experience has shown that the guideline and the Process Matrix have been useful to extract process descriptions from domain experts.

Below we describe a small fictive example of a loan application process represented by the Process Matrix shown in table. 2.1 . Each row of the matrix represents an activity of the process: Filling in the application (Application), Registering customer information(Register Customer Info), Approval of the application (Approval 1 and 2), Payment, Express Payment, Rejection and Archive. The columns are separated in 3 parts: The first set of columns describes the access rights for the different roles (*Applicant* (App), the *Case Worker* (CW) and *Manager* (Mgr) in the figure). The Roles columns indicate that the applicant can fill out applications, but the case worker and manager can only read the content of the application. Everyone can register customer information, but only the case worker can perform approval 1 and only the manager can perform approval 2, and both approval steps are invisible to the applicant. The remaining actions can only be performed by the case worker - they can be read by the manager and applicant, except for the archiving which is invisible to the applicant.

The next column describes the predecessor constraints, where we indicate by a $*$ that the predecessor is a logical predecessor. That is, activity Approval 1 has activity Register Customer Info as sequential predecessor and activity application as logical predecessor. Thus, the customer may at any time re-submit the basic info (e.g. address and phone number) without causing a re-execution of the approval activity. However, it the application is changed the approval must also be carried out again. (If only changes in the amount given in the Application activity should cause Approval to be re-executed, one could make the application a sequential predecessor of the Approval activities, but add the amount of the loan as a dependency expression to the Approval activities). Finally, the last row describes the activity condition. For instance, the condition $Hurry \wedge Accept$ of activity Express Payment indicates that the Boolean values $Hurry$ and $Accept$ in the shared data store must both be set to true for this activity to be included in the flow. To fit the table within one column of the paper we have left out a column stating which eForm is attached to an activity, and which values in the shared data store are accessed and changed by the eForm: The Application form changes the variables $Rich$ and $Hurry$, and the Approval forms toggle the $Accept$ variable. Concretely, in the online example of the loan application process the purposes *Rich* and *Hurry* are set by radio buttons in the eForm attached to the Application activity in step 1, and the purpose *Accept* is toggled in the eForms attached to Approval 1 and Approval 2.

The Activity Conditions in the last column depend on the purposes *Rich*, *Hurry* and *Accept*. A rich applicant only needs an approval from the case worker, while a poor applicant also needs an approval from the manager in the bank. If the purpose *Hurry* is set to true, the application is treated as an express payment. The result is that the Express payment activity (step 6) is included and not the Payment activity (step 5). Conversely, if the purpose *Hurry* is set to false, the the (normal) Payment activity in step 5 is included and not the express payment activity. Both payment activities require the purpose *Accept* to be true.

2.1.1.3 Process Execution

In table. 2.2, we show a possible state of the system during an instance of the workflow where a poor applicant applies for a non-express loan. The purpose *Hurry* is set to false thus the activity Express Payment is excluded. The activity condition for all other activities except activity Express Payment is set to true and they are included for execution, i.e. the activity Approval 2 is included because the purpose *Rich* evaluates false. The activities Application, Register Customer Info, Approval 2 have already been executed and their activity status is thus executed. The activity Approval 1 is ready for execution, but it has not started executing. Note that the activities Payment and Rejection can not be started because of their predecessors, but only one of them will be executed in future as the value of purpose *Accept* makes the other activity to be excluded. The activity Archive will be executed eventually after all its predecessors, as it does not have any purposes attached to it. As mentioned above, activity conditions will be re-evaluated after execution of each activity which makes the dynamic inclusion or exclusion of activities possible at runtime.

Note that the registration of customer information can be done either before or after the application, and can be redone arbitrarily often without affecting any of the

	Activities	Activity Condition	Activty Status
1	Application	true	executed
2	Register Customer Info	true	executed
3	Approval 1	true	can start
4	Approval 2	true ($\neg Rich$)	executed
5	Payment	true	can not start (wait for $\{3\}$)
6	Express Payment	false($\neg Hurry$)	inactive ($\neg Hurry$)
7	Rejection	true	can not start (wait for $\{3\} \wedge \neg Accept$)
8	Archive	true	can not start (wait for $(\{3\} \wedge \{4\}) \vee (\{3\} \wedge \{7\})$)

Table 2.2: The Process Matrix at Run Time.

other steps.

2.1.2 Formalization using Linear Temporal Logic

In this section we provide formalizations of the key primitives of the Online Consultant (sec. 2.1.1.1) and process matrix described in (sec 2.1.1.2) in terms of Linear time Temporal Logic (LTL) [Pnueli 1977, Sistla *et al.* 1983] formulas. First we briefly recall LTL and the approach in [van der Aalst & Pesic 2006b, van der Aalst & Pesic 2006a].

2.1.2.1 Executable LTL for Workflow

LTL is a temporal logic extending propositional logic to infinite sequences of states. This is done using the temporal modal operators $\mathbf{O}P$ (in the next state of the sequence formula P holds), $\Box P$ (in the current and all of the following states of the sequence formula P holds), $\Diamond P$ (in the current or at least one of the following states of the sequence formula P holds), and $Q\ \mathbf{U}\ P$ (in the current or at least one of the following states of the sequence formula P holds and formula Q holds in all states *until* that state is reached).

LTL has been extensively used as property language [Dwyer *et al.* 1998] for automatic verification of reactive systems, also referred to as *model checking* [M.Clarke *et al.* 1999]. The basic principle of model checking is to use an automatic tool to check if a system, usually described by an automaton, satisfies a property specified in a property language, which is often a temporal logic. In this case one can say that the system is a *model* of the property.

The key idea of the paradigm shift proposed in [van der Aalst & Pesic 2006b] is to turn this around and use the declarative, temporal logic language to provide the system (workflow) definition. The system is then defined as a formula that characterizes the valid completed sequences of activities, e.g. that in a completed instance execution a certain activity must always occur before some other activity.[3] In acknowledgement to the fact that LTL formulas may be too difficult to understand for process designers, the authors in [van der Aalst & Pesic 2006b] propose to use so-called *constraint template formulas*, also referred to as policies or business rules. These templates are further equipped with a graphical notation.

It is worth noting, that a similar paradigm shift was in fact also proposed by Gabbay in [Gabbay 1987] where he suggests to use LTL formulas as execution language for interactive systems. Moreover, Gabbay showed that one could ease the description of systems by using LTL extended with past time modalities by proving that any LTL formula with past time modalities can be rewritten to an equivalent (but in the worst case exponentially longer [Laroussinie *et al.* 2002]) LTL formula with only future time modalities. We exploit the use of past time modalities below to give more succinct formalizations of the activity resets in ROC.

It is important to recall, that the difference between using a declarative language as opposed to an imperative language is on the *ease* and *flexibility* of expression and not on expressiveness: Any LTL formula can be automatically translated to an equivalent finite automaton over infinte sequences and vice versa [Sistla *et al.* 1983]. The point made in [van der Aalst & Pesic 2006b, Gabbay 1987] is that one may use this correspondence to let the workflow engine construct an automaton from the declarative LTL description that can be used for execution of the process.

As described in the previous section the Process Matrix employed in ROC is in fact an example of a declarative workflow language used in practice. Our aim is to give a translation from the Process Matrix model to LTL, which translates any Process Matrix process M into an LTL formula $[[M]]$ such that the sequences of states for which $[[M]]$ is true is exactly the sequences of states that constitute valid executions of the process M. Concretely, our formalization is defined as extensions to the LTL template formulas given in [van der Aalst & Pesic 2006b, van der Aalst & Pesic 2006a]. As in [van der Aalst & Pesic 2006b, van der Aalst & Pesic 2006a] we assume a discrete time model where any step between two consecutive states in the sequence corresponds to the execution of one activity in the workflow, and we deal with the fact that workflow executions are finite and LTL is interpreted over infinite sequences by using the standard stutter extension, assuming that the finite workflow executions are terminated by an infinite sequence of steps with no change in the state. The basic propositional formulas we employ will be Boolean formulas over propositions on the state space and the current activity. In particular, the proposition $(\mathbf{act} == A)$ is true in a state if the last executed activity is A.

A basic example of an LTL template in the DecSerFlow language is the constraint template $existence(A : activity)$ formalized as $\Diamond(\mathbf{act} == A)$ in LTL. It simply states that there exists a step in which activity A is carried out.

An example of a so-called *relation formula* [van der Aalst & Pesic 2006b] is the constraint $precedence(A : activity, B : activity)$ which states that an activity B is

[3] Note that a partial execution sequence need not satisfy the formula, as long as it is possible to complete the sequence in a way that makes the formula satisfied.

preceded by an activity A, i.e. the activity B can not be executed before activity A has been executed. This template formula uses the existence template as a sub formula and is expressed in LTL as

$$existence(B) \implies (!(\mathbf{act} == B)\ \mathbf{U}\ (\mathbf{act} == A))$$

where ! denote the the Boolean negation. Reading the formula, it expresses that if there exists a state in the sequence in which B is carried out then there exists a state in the sequence in which A is carried out for which B is not carried out in any of the preceding states. This is equivalent to the intended property that the activity B can not be executed before activity A has been executed.

Another example of a relation formula is the constraint $response(A : activity, B : activity)$ which expresses that whenever the activity A is executed then B must also be executed after it. This formula is expressed in LTL as

$$\Box((\mathbf{act} == A) \implies existence(B))$$

From the response and precedence templates one may build composite relation templates, such as the template $succession(A : activity, B : activity)$ expressed in LTL simply as a conjunction of the two templates:

$$response(A, B) \wedge precedence(A, B)$$

The formula expresses that every execution of activity A must be followed by an execution of B and any execution of B must be preceded by an execution of A. One may have already noticed similarities with the primitives in the Process Matrix. In the following section we can see that the Process Matrix primitives can indeed be formalized similarly to the templates given above, but with some interesting variations due to the use of activity and dependency conditions. We do not consider the roles nor dependency expressions.

2.1.2.2 From the Process Matrix to LTL

To define the translation from the Process Matrix model to LTL we describe how the individual primitives can be expressed as templates in LTL. The formalization of a Process Matrix workflow M will then be an LTL formula $[[M]]$ which is a set of formulas in conjunction obtained by instantiating the templates according to the entries in the Process Matrix. Our aim is that $[[M]]$ is true exactly for the sequences of states that constitute valid executions of the process M. However, we leave for future work to evaluate the correctness of the formalization.

In the following we assume a Process Matrix workflow M. We let A and B range over activities in M and write $actcon(A)$ for the activity condition specified in the Process Matrix M for an activity A.

The first formula used for the formalization is then the LTL formula $act\_include(A : activity)$ given by

$$\Box(\mathbf{O}(\mathbf{act} == A) \implies actcon(A))$$

It expresses that an activity A can only be executed in the next step if it is included in the present, i.e. its activity condition is true. The formula $act\_include(A)$ is then included in the conjunction in $[[M]]$ for every activity A in M.[4]

To formalize the remaining ingredients we define a few templates used as sub formulas. The first such template is $act\_including(A, B) = (\mathbf{act} == A) \wedge actcon(B)$ which expresses that activity A is executed and at the same time the activity B is included in the process (because the acitivty condition for B is true).

The second template is $existence\_act\_including(A, B) = \Diamond act\_including(A, B)$ which extends the existence template for DecSerFlow to express that an activity A is eventually executed and at the same time the activity B is included in the process.

We now go on to formalize the control flow primitives of the Process Matrix.

2.1.2.2.1 Sequential Predecessor

The sequential predecessor constraint is similar to the precedence formula in DecSerFlow described above, except for the use of the activity condition in the Process Matrix. We define the constraint template $sequential\_predecessor(A : activity, B : activity)$ stating that A is a sequential predecessor of B by the LTL formula $existence\_act\_including(B, A) \implies (!act\_including(B, A)\ \mathbf{U}\ (\mathbf{act} == A))$. Let $A <_M B$ denote that A is a sequential predecessor of B in M. We then include the formula $sequential\_predecessor(A, B)$ in the conjunction $[[M]]$ for any pair $A <_M B$.

2.1.2.2.2 Activity Reset

To formalize the logical predecessor constraint, we need to formalize the somewhat complex handling of *activity resets* in ROC. We want to define a template $reset(A)$ which expresses that the activity A is being reset in the current state. Here we exploit the past time modality *Since* written as $Q\ \mathbf{S}\ P$ and the past time modality $\mathbf{Y}P$. The *Since* modality is the dual of the until modality and is true if in the current or at least one of the *preceeding* states the formula P holds and formula Q holds in all states *since* that state. The past time modality $\mathbf{Y}P$ is true if P holds "Yesterday", i.e. in the previous state. As described in [Gabbay 1987] we can translate the formalization including past time modalities into a pure present and future time formula.

Let $A \overset{*}{<}_M B$ denote that A is a logical predecessor of B in M. If there is a chain of logical predecessors $A_0 \overset{*}{<}_M A_1 \overset{*}{<}_M \ldots\ldots \overset{*}{<}_M A_k$, for which $actcon(A_i)$ is true for $i \in \{0, \ldots, k\}$, i.e. the activities A_i are all included in this state, and the first activity A_0 is executed or changes from not-included in the previous state to included in this state, then the activity A_k will be reset in the Process Matrix. To formalize this, first define the template $included(A : activity) = \mathbf{Y}!actcon(A) \wedge actcon(A)$ and define $chain(A_0, A) = \{[A_0, A_1, \ldots, A_k] \mid A_0 \overset{*}{<}_M A_1 \overset{*}{<}_M \ldots\ldots \overset{*}{<}_M A_k = A\}$, i.e the set of all chains of logical predecessors with A_0 as first and A as the last activity. Then we define the template $resetchain(A) = \bigvee_{B \in M, c \in chain(B,A)} (\bigwedge_{A' \in c} actcon(A') \wedge (included(B) \vee (\mathbf{act} == B))$.

[4] We also include the formula $\bigwedge_{A \in M} !(\mathbf{act} == A)$ in the conjunction stating that no activities are carried out before the initial state.

Finally, we define the template $reset(A) =!(\textbf{act} == A)\ \textbf{S}\ resetchain(A)$, which we will use below.

2.1.2.2.3 Logical Predecessor

Logical Predecessor is a strengthening of the Sequential Predecessor constraint. The template $reset(A)$ allows us to formalize the template $logical\_predecessor(A : activity, B : activity)$ in LTL as $sequential\_predecessor(A, B) \wedge \Box\big(reset(A) \implies sequential\_predecessor(A : activity, B : activity)\big)$ We then include the formula $logical\_predecessor(A, B)$ in the conjunction $[[M]]$ for any pair $A \overset{*}{<}_M B$.

2.1.2.2.4 Activity Execution

The final part of the formalization, is to express when an activity should be executed. We use the template $executed(A : activity) =!reset(A)\ \textbf{S}\ (\textbf{act} == A)$, i.e. using the template $reset(A)$ and the since modality to describe that an activity has status executed if there exist a state in the past where it is executed and it has not been reset since. The activity execution formula can then finally be formalized as

$$(\Diamond\Box executed(A)) \vee (\Diamond\Box !actcon(A))$$

which is included in the conjunction $[[M]]$ for every activity A in M. The formula expresses that either the activity A has status executed continuously in some future state, or it is excluded from the process. (Recall that we interpret LTL over infinite sequences and assume the execution sequences of ROC to be terminated by an infinite sequence of states with no change)

2.1.2.3 Process Matrix

Based on the defintions of primitives explained in the previous paragraphs, we now define the Process Matrix Workflow M and its equivalent LTL formula $[[M]]$. The Process Matrix is 6-turple

$$M = (A_M,\ <_M \subseteq A_M \times A_M,\ \overset{*}{<}_M \subseteq A_M \times A_M,\ V,\ F_M : A_M \to P(V))$$

where

1. A_M is the set of activities
2. $<_M \subseteq A_M \times A_M$ is the sequential predecessor relation
3. $\overset{*}{<}_M \subseteq A_M \times A_M$ is the logical predecessor relation
4. V is a finite set of variables
5. $F_M : A_M \to P(V)$ provides the set of variables the form may modify

The LTL formula for a Process Matrix M will then be

$$
\begin{aligned}
[[M]] = \quad & \textstyle\bigwedge_{A \in A_M} (act\_include(A)) \wedge executed(A)) \\
\wedge \quad & \textstyle\bigwedge_{A,B \in A_M,\ A <_M B} sequential\_predecessor(A, B) \\
\wedge \quad & \textstyle\bigwedge_{A,B \in A_M,\ A \overset{*}{<}_M B} logical\_predecessor(A, B) \\
\wedge \quad & (act == init) \wedge \ \mathbf{O}\Box(act \neq init) \\
\wedge \quad & \textstyle\bigwedge_{A \in A_M,\ V - F_M(A_M),\ x \in Val} (\Box(\mathbf{act} == A) \implies (Y(v = x) \Leftrightarrow (v = x)))
\end{aligned}
$$

where

1. $\bigwedge_{A \in A_M} (act\_include(A)) \wedge executed(A))$ represets set of included and executed activities from workfow instance

2. $\bigwedge_{A,B \in A_M,\ A <_M B} sequential\_predecessor(A, B)$ contains set of sequential predecessor constriants

3. $\bigwedge_{A,B \in A_M,\ A \overset{*}{<}_M B} logical\_predecessor(A, B)$ contains set of logical predecessors constraints.

4. $(act == init)$ represents the initial state of the workflow.

5. $\bigwedge_{A \in A_M,\ V - form(A_M),\ x \in Val} (\Box(\mathbf{act} == A) \implies (Y(v = x) \Leftrightarrow (v = x)))$ represents state of all variables which are not part of the current activity and their state remains the same during the execution of current activity.

This concludes our work on formalization of the Process Matrix. In the next section, we will describe a case study from healthcare sector which was conducted in Danish hospitals on the oncology treatment.

2.1.3 Case Study: Healthcare Workflow

It has been known for quite a while that there is a need for making clinical working practices safer, as too many errors happen causing suffering or even death of patients [Kohn *et al.* 2000]. Due to the complexity, the high mobility and ephemerality of the daily clinical work [Bardram & Bossen 2005, Bødker & Christiansen 2004] safer working practises will require better coordination, efficient collaboration and not least fulfilment of up to date clinical practice guidelines (CPG) [Davis & Taylor-Vaisey 1997, Grol & Grimshaw 2003].

One way of supporting this is by the use of of IT based clinical decision support and better linkages in and among IT-systems [Bates *et al.* 2001]. Indeed, according to [Mulyar *et al.* 2007, Lenz & Reichert 2007] on of the best options for improvement in clinical work seems to be IT supported clinical processes based on CPGÕs. However, the use of IT based CPGÕs is challenging in several ways. Firstly, due to continuous development of new knowledge within the medical domain the mean survival time of clinical guidelines is short, approximately 2 years [Shojania *et al.* 2007]. Secondly, there is a need for guidelines to be flexible and adaptable to the individual patient [Quaglini *et al.* 2001].Thirdly, no coherent theoretical framework of health professional and organiza-tional behaviour and behaviour change has yet been established [Grimshaw *et al.* 2004]. Finally, it is a serious challenge that health professionals currently tend not to follow clinical guidelines [Cabana *et al.* 1999]. One of the reasons for this could be that clinical guidelines are not embedded in the clinical work processes and the technology available in the clinical setting today. Oncology clinics are an example of a clinical speciality for which it is known that there does exists a high number of CPGs that are followed to a certain degree by the health professionals. For this reason we found it interesting to perform a series of field studies in oncology clinics, to examine enablers and obstacles for use of IT-supported clinical guidelines. The field studies are presented in Sec. 2.1.3.1 below. Based on the field studies and our examination, we then proceeded to investigate in Sec. 2.1.4.1 how the current paper based workflows could be supported using a commercial declarative workflow management system, which relates to the CIGDec approach of Pesic and van der Aalst [van der Aalst & Pesic 2006b]. We believe that the resulting model rather naturally extends the paper based flowchart table used at the hospitals, and in particular avoids the introduction of complex cyclic control flow graphs and over specification as also pointed out in [van der Aalst & Pesic 2006b]

2.1.3.1 Field study - usage of CPGs in Danish oncology clinics

2.1.3.1.1 Method

Observations were made on three Danish oncology clinics by two observers. Four days of observation were made at each clinic. Besides observations, access to all clinical guidance material was granted. All the clinics were specialized within oncology; two of them were university clinics. The focus of the observation study was on the use of CPGs as defined by Field and Lohr [Field & Lohr 1992]: Clinical practice guidelines are systematically developed statements to assist practitioner decisions about appropriate health actions for specific clinical circumstance. We especially looked at the work of nurses, doctors and pharmacists in relation to chemotherapeutical treatment of patients.

2.1.3.1.2 Overall treatment processes and guidance documents

Patients are referred to the clinics with a diagnosis of cancer. By the first visit in the outpatient clinic the patient is informed about pros and cons of chemotherapy by a doctor, and an overall patient plan for oncological treatment is outlined. In subsequent visits chemotherapy is given, in between visits to the outpatient clinic monitoring of side effects to chemotherapy are done by laboratory tests. The chemotherapeutic treatment is based on a number of different types of guidance documents and diagrams depicted in Figure 2.2. The basis of the treatment is given in a standard treatment protocol or a research protocol, which constitute the CPG. The protocols are written in a narrative form with a description of the current knowledge of treatment of the disease in case as well as a thorough description of the drugs to be used. The size of a research protocol is app. 60-80 pages and a standard treatment protocol is app. 30-40 pages. Protocols are generally developed in cooperation between several oncology departments, frequently with a pharmaceutical company as a main sponsor and actor. Research protocols are often multinational. Based on the protocols local practice guidelines (also referred to as standard treatment plans) are made as well as a treatment overview, in daily speech referred to as the *noughts and crosses* diagram. The noughts and crosses diagram describes the whole pathway including medical treatment as well as examinations during several months. There will often be deviations from the original plan due to side effects to treatment, other medical problems or resource problems in the hospital.The flow of each chemotherapeutic treatment session is guided by the so-called patient flowchart, which also records the state of the treatment session. Below we will describe the workflow resulting from the flowchart in more detail; this will be the focus of the remaining part of the section.

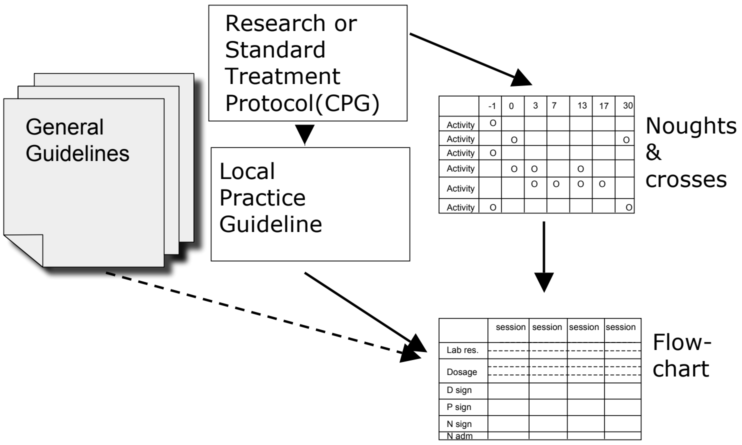


Figure 2.2: Overview of the relation between research protocols/standard treatment plans, local practice guidelines (standard plans) and flow charts. General guidelines are use at the hospital, containing issues like the treatment of diabetes.

2.1.3.2 Current workflow for chemotherapy treatment sessions

Fig. 2.2 shows an overview of the workflow which is reiterated in every chemotherapeutic treatment session. In the flowchart the basic information about the patient is registered, including the latest lab results as well as height, weight of the patient. Based on this information and the patient history of any major adverse effects, the doctor calculates the therapeutic doses of chemotherapy, documents it on the flowchart and signs it. The flowchart is transferred from the doctor to the controlling pharmacist (who can be situated near by in the clinic or far away in the pharmacy) where it functions as a prescription from the doctor. The controlling pharmacist controls the doctors dosage calculation and writes the information in a working slip that is used for the pharmacy assistant who is doing the preparation of the drug(s) in case. During preparation the quantity of all products as well as batch numbers are registered in the working slip, finally the working slip is signed by the pharmacy assistant, and the product - usually a drip bottle or a pump with a content and patient information note stuck to it, is referred to the controlling pharmacist for check out. When the controlling pharmacist has checked that the produced drug mixture and patient information note matches the flowchart and the working slip, the pharmacist put small green ticks on each item in the flowchart and finally signs it. Subsequently the flowchart and the product is referred to the treatment rooms, where the responsible nurse together with another authorized person (nurse or doctor) checks that the product and flowchart matches, both regarding content and patient information. The responsible nurse then signs the flowchart and the medicine is administered to the patient. In parallel to this the nurse will administer adjuvant medicine like anti-emetics, cortisol and other drugs that are prescribed in the local practice guidelines. The nurse registers the medication in the Medicine Order and Administration (MOA) IT system that currently is being implemented in all the oncology departments.

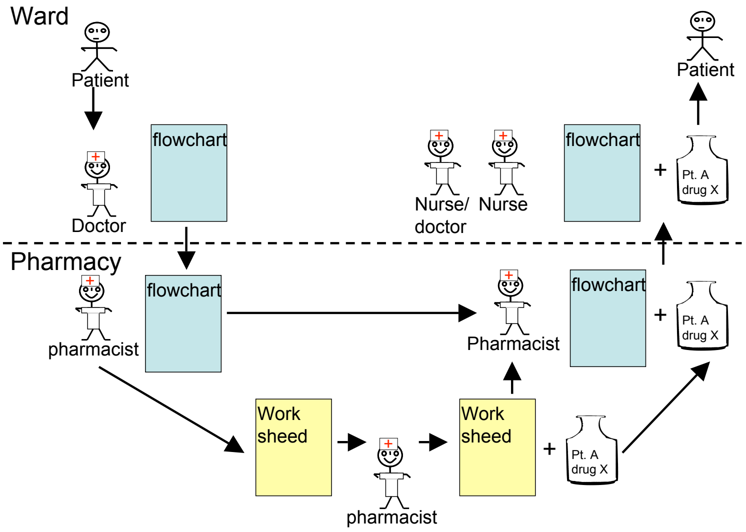


Figure 2.3: Oncologic workflow in relation to chemotherapeutic treatment of patient.

2.1.4 Preliminary conclusion to the case study

Several characteristics of the work were elucidated in the case study:

- There are several professional actors involved in even rather simple workflows like the ones we studied (they are all involved in more than one workflow at the same time).

- The flow is guided by the flowchart, which is simply a table with a column to which the Doctor and Chemist add information and/or a signature, thereby capturing the state of the session.

- The workflow is distributed: the doctor and nurse, pharmacist, and pharmacy assistant are physically located in different places at the hospital and the current paper used for controlling the workflow is physically transferred by a porter or nurse (or faxed) between the different actors.

- Only the actor currently possessing the flowchart knows its state. Much time was used waiting for and controlling the status of the former process step, to be able to plan own work.

- There are a number of check-points. If a check fails (e.g. the Chemist or Nurse doubts the validity of the current state, the previous actors are asked to verify the state and possibly redo a calculation.

- Exceptional events like the medicine getting too old (e.g. if it is not transferred to the treatment rooms and approved within 24 hours) also led to recurrence of activities.

- Only the state (information) and the actors are implicit in the flowchart. The ordering of events (i.e. transfer of the flow chart between actors), handling of exceptions and recurrence/validation of calculations are implicit.

In our observations we found several potential enablers and obstacles to digitalization of the process support, which have been collected in Fig. 2.4 below.

We believe that IT based process support has a potential in relation to chemotherapeutic treatment of cancer patients. It is though important to be aware that such a change in the clinical work is not just a question of giving access to the right applications. Access to the right equipment as well as integrations of it-systems is mandatory. Also the organisational workflows have to be analysed and maybe changed. This demands managerial support. More work has to be done to understand the organisational and social implications. To obtain knowledge about organisational and social implications it is important to establish carefully planned experiments with process support in clinical settings. In this case study, we concentrate on how the workflow of a single chemotherapeutic treatment session may be supported by a workflow management system, and in particular how the workflow can be described as an executable process. A central issue is how to make the implicit ordering of events (and the additional verifications and possibly recurrences of events) explicit. One option is to use an imperative flow graph based notation such as Petri Net or BPMN. However, it would include arrows for capturing the control flow (including cycles for the verification and recurrence of events), which would differ radically from the notation used in the current case study based setting. As suggested by van der Aalst and Pesic in [van der Aalst & Pesic 2006b] one can avoid introducing the explicit control flow as a complex flow graph by instead using a declarative notation such as the CIGDec model. Following this idea, we will investigate below how to specify the treatment session in a commercial declarative workflow management system, the Resultmaker Online Consultant.

Enablers	Obstacles
The nurses do a lot of walking between treatment rooms and pharmaceutical preparation rooms to obtain status on the workflow. An up to time status on preceeding process steps would make it easier for the actors down stream to plan work.	Feeling of competence. "I have been here for a hundred years, so I know what to do, and I know the procedures" guidance are not sought for.
Many patients had to follow more than one CPG, due to co-morbidity or adverse effects of treatment	Oral culture problems are preferably discussed with peers, even rather fact based ones.
Meeting legal demands: In the current situation, the pharmacist is lacking a copy of the prescription, which is a legal demand.	No clinical managerial pressure. It is not expected than professionals look things up in the existing sources (Paper or IT-based). There is no control (no count on hits)
It was clear from our observations that CPGs and standard treatment plans was more vividly used if they were embedded in the work processes. This could be in the form of documentation templates, automated order forms or decision algorithms.	Rigid work flows that has been founded using low-tech information technology like paper
Many new-commers, as they are more active users of CPGs than those that had been in the job for a longer period. So in departments with a high turn around of employees process support will be more sought for.	Lack of integration between process support and all the clinical information systems, among which some are still not digitalised.
Experience among clinicians that guidelines are hard to find especially IT based ones "I get 35 hits in a search for resuscitation".	Lack of access to computers, with low response time and single sign on to (all) the clinical IT-systems

Figure 2.4: Enablers and obstacles for digitalized clinical process support.

2.1.4.1 Treatment Workflow in ROC

As explained in the previous section, the ROC uses so-called eForms as its principal activities and allows one to declare the sequential constraints and dynamically included verification steps (and implied recurrences of activities) as found in the oncology treatment workflow using so-called sequential and logical predecessor constraints and a notion of activity conditions. There is yet no formal graphical notation for the ROC processes, but there is a guideline for how to identify and specify activities, roles/actors and constraints in a table of a specific form jointly with the users. This table is referred to as the Process Matrix (PM), which is also used as name for the process model.

Table 2.5 below shows an example of a PM (simplified to preserve space) for the Oncology workflow presented in the previous section. Each row of the matrix represents an activity of the Oncology workflow. The columns are separated in 3 parts: The first set of columns describes the access rights for the different roles: Doctor (D), Nurse-I (N1), Nurse-II (N2), Controlling Pharmacist (CP), Pharmacist assistant (PA). The next set of columns describes (sequential and logical) predecessor constraints. The last set of columns describes activity conditions.

2.1.4.1.1 Activities and execution.

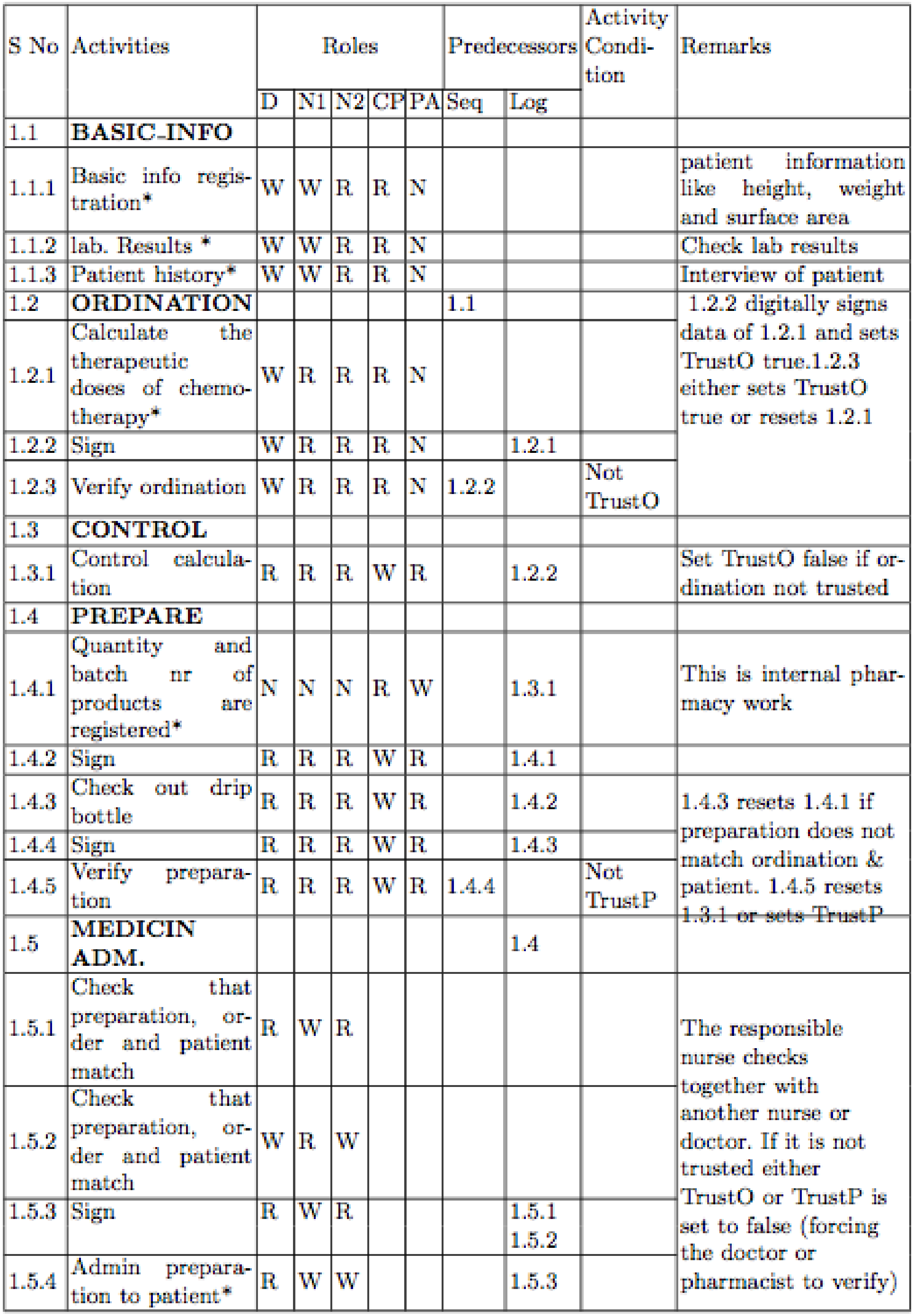
The notion of an activity in ROC is like in any other workflow language, which means an activity is atomic and corresponds to a logical unit of work. Activities are executed in parallel by default and they can be executed any number of times, unless constrained as described below. The state of the ROC records whether an activity has been executed or not. If an activity has been executed, then that activity will have status executed. Its state can be reset under certain circumstances explained in Control Flow Primitives sub section. We say that the flow has state complete at any point where all activities (currently included in the flow) have state executed.

As we have discussed in the previous section, ROC contains pre-defined activity type, *eForm Activity*. The eForms are web questionnaires that have graphical user interface elements displayable in a web browser. The fields on the eForms are mapped to variables in the shared data store and the data filled in by the users will be available to all activities of the workflow instance. The eForms are appended to ROC activities in process definitions and at run-time when an eForm activity is executed, the corresponding eForm will be displayed to the user for human interaction. All activities in the example, except signing activities, are eForm activities.

In order to provide authentication for the data filled in by the users, the ROC uses Signing Activity. The user data on eForms will be digitally signed by using XML digital signatures syntax and users digital identity certificates. A single signing activity supports signing of data from multiple eForms. In the example all the activities named Sign are signing activities.

The ROC supports a simple resource model using Role-based access rights to define permissions on the activities to different users of the system. The possible access rights are Read (R), Write (W), Denied (N) and the default access right on activities is Read access. The Read access right allows a user with the particular role to see the data of an activity, where as Write access right allows the user to execute an activity and also to input and submit data for that activity. A Denied access right is the same as making an activity invisible to the user, i.e. the user does not see it as part of the flow. In the example we have used the denied access right to shield the Pharmacist assistant from the rest of the workflow.

Every activity in the ROC has a logical activity condition. An activity condition is a Boolean expression that can reference the variables from the shared data store. If an activity condition is evaluated to be true, the activity is included in the workflow,



S No	Activities	Roles					Predecessors		Activity Condition	Remarks
		D	N1	N2	CP	PA	Seq	Log		
1.1	**BASIC_INFO**									
1.1.1	Basic info registration*	W	W	R	R	N				patient information like height, weight and surface area
1.1.2	lab. Results *	W	W	R	R	N				Check lab results
1.1.3	Patient history*	W	W	R	R	N				Interview of patient
1.2	**ORDINATION**						1.1			1.2.2 digitally signs data of 1.2.1 and sets TrustO true.1.2.3 either sets TrustO true or resets 1.2.1
1.2.1	Calculate the therapeutic doses of chemotherapy*	W	R	R	R	N				
1.2.2	Sign	W	R	R	R	N		1.2.1		
1.2.3	Verify ordination	W	R	R	R	N	1.2.2		Not TrustO	
1.3	**CONTROL**									
1.3.1	Control calculation	R	R	R	W	R		1.2.2		Set TrustO false if ordination not trusted
1.4	**PREPARE**									
1.4.1	Quantity and batch nr of products are registered*	N	N	N	R	W		1.3.1		This is internal pharmacy work
1.4.2	Sign	R	R	R	W	R		1.4.1		
1.4.3	Check out drip bottle	R	R	R	W	R		1.4.2		1.4.3 resets 1.4.1 if preparation does not match ordination & patient. 1.4.5 resets 1.3.1 or sets TrustP
1.4.4	Sign	R	R	R	W	R		1.4.3		
1.4.5	Verify preparation	R	R	R	W	R	1.4.4		Not TrustP	
1.5	**MEDICIN ADM.**							1.4		
1.5.1	Check that preparation, order and patient match	R	W	R						The responsible nurse checks together with another nurse or doctor. If it is not trusted either TrustO or TrustP is set to false (forcing the doctor or pharmacist to verify)
1.5.2	Check that preparation, order and patient match	W	R	W						
1.5.3	Sign	R	W	R				1.5.1 1.5.2		
1.5.4	Admin preparation to patient*	R	W	W				1.5.3		

Figure 2.5: Information marked with * could be transferred from or registered automatically in another hospital information system (HIS) W= write, R = read, N = denied access.

otherwise the activity will be skipped. Activity Conditions in ROC workflow model are re-evaluated whenever necessary, so the inclusion of an activity can be changed during the lifetime of the workflow instance. If the activity condition changes to false during the execution of an activity (e.g. when a user is filling in an eForm), the user will be informed that the activity is no longer part of the flow and no data will be changed. This guarantees atomicity of activities.

In the example we use two Boolean variables TrustO and TrustP to control the inclusion of the verification actions 1.2.3 and 1.4.5 respectively. When the doctor signs the ordination in activity 1.2.2 , TrustO is also set to false, thereby excluding the verification from the flow. However, it may be set to true during activity 1.3.1, 1.51 or 1.5.2. This will force the verification step to be executed and all activities having it as logical predecessor to be reset (see below).

Sequential predecessor constraints are marked in the Predecessor (Seq) column in the example. For instance, Activity 1.2.2 (Sign) is a sequential predecessor of activity 1.2.3 (Verify), capturing that it does not make sense to verify an ordination if it has not been signed. Also, every activity in the group 1.1 is sequential predecessors of every activity in group 1.2 In the example, the verification action 1.2.3 may reset activity 1.2.1 (if the doctor finds out during verification that he needs to recalculate the ordination). This again causes activity 1.2.2 to be reset, since it has activity 1.2.1 as a logical predecessor.

2.1.4.2 Discussion

It is well known that healthcare processes are complex [Drucker 1993] and although much time is used on coordination [Reddy *et al.* 2001] errors happens too frequently [Kohn *et al.* 2000]. The CPGs can support healthcare employees in the process of following best practice consistently [Grol & Grimshaw 2003, Sim *et al.* 2001], but it is also well known that impediments to access relevant guidelines is an obstacle for use [Thorsen & Makela 1999, Feder *et al.* 1999]. Thus it seems obvious to embed CPGs in clinical IT-process support, although the success of such projects has not been convincing [Lenz & Reichert 2007, Ash *et al.* 2004].

In our case study of a rather simple clinical work process we found that the process had an extension in both time and location and several actors were included. Although the process was frequently repeated there were also frequent alterations and recurrences due to returns to previous steps in the workflow. These challenges could be supported in a natural way by the declarative primitives in the ROC workflow management system. Also, the activity conditions allow smooth combination of several sub-workflows. This would be a way of implementing the *noughts and crosses* diagram, which indeed specify for each day which sub workflows are relevant. ROC supports the paradigm of embedded although visible CPGs in clinical IT-systems. Though one have to be aware that IT based business support will lay the grounds for new work processes, so one should not just automate existing paper based work processes [Berg & Toussaint 2003].

2.1.4.2.1 Professions, professionalism and process support.

In the ROC independent roles can be defined for all actors. The rights to read, fill in, and proceed to next step and to change the flow can defined in relation to each role and activity. This can make it possible for the actors to see the status of the process upstream, and thus make the planning of own work easier.

Health professionals are a heterogeneous group, some with little and some with immense experience within a field. Although experience may not totally protect a clinician from committing errors the risk is less and the source of annoyance from detailed guidance by the IT system will be huge. In the ROC focus is on the overall clinical managerial process, for the inexperienced there are links to CPGs outside the ROC.

Nevertheless it will be a cultural challenge for clinicians to have a clinical process system directing the road ahead [Berg *et al.* 2000], as well as it will have impact on the training and socialization of new comers to the field [Mimnagh & Murphy. 2004]. The communication culture in the healthcare sector is profoundly oral [Coiera 2006]. We observed several examples of clinicians discussing factual topics to which the reply only would be a few clicks away. The cultural element will always be a challenge when implementing new technology, especially when it fundamentally changes the work processes [Orlikowski & Gash 1994].

2.1.5 Conclusion

Initially, the work on formalization of ROC was done as part of PhD candidate's 1st year Industrial PhD project on behalf of Resultmaker A/S as a pre-project to the current doctoral thesis work, with a key focus to formalize the primitives of ROC and to develop it as a formal foundations for declarative flexible workflows and to use it as a testbed for research in TrustCare project. But the industrial PhD project was stopped by Resultmaker later due to their financial troubles.

Further, we had also explored using the ROC formalization work to further develop it as a generic formal foundations for declarative flexible workflows, but we have chosen to develop a new formal model instead of using ROC formalization due to the following reasons.

1. ROC is a data-centric declarative workflow management system with a lot of complex primitives such as dependency expressions, transactions. Further, some of the primitives such as dependency expressions are tightly integrated and built over data. But as part of TrustCare project, we had been looking for a more general formal model that is declarative with simple primitives, yet sufficiently expressive, which can be used for both specifications and execution of workflows and business processes, which may not be not necessarily data-centric.

2. The work on the formalization of ROC is based on using LTL as a language for specification of business processes and during our work we had realized that executing process models specified in temporal logics such as LTL is quite

> complex. Further, a master's thesis in our group [Slaats 2009] has explored the challenges of using temporal logics for business process execution, where in it has looked into various ways of generating an automaton from an LTL process specification that can be used for execution. It has explored the two main approaches of generating an automaton from a LTL specification: creating a generalized Büchi automaton [Gerth *et al.* 1996] and creating a Müller automaton [de Jong 1991] and further it has proposed it's own algorithm based on the rewriting of LTL formulae for the unsafe on-the-fly execution. However it has noticed that for more flow-orientated process models, the LTL formulae to describe them will grow very quickly in size and complexity and further the automata generated for these complex LTL specifications is quite huge and time consuming to generate and hence we came to a conclusion that it is not practical to use LTL as modeling language for business processes.

However, we believe that the study of formalization of ROC is a starting point for a valuable cross-fertilization between development of workflow management systems in practice and research in theoretical computer science and motivated us for development of our formal model DCR Graphs. The predecessor primitives of the Process Matrix are similar to the primitives considered by van der Aalst and Pesic in [van der Aalst & Pesic 2006a, van der Aalst & Pesic 2006b] , and quite useful constructs in the domain of business process modeling and hence carried over to our formal model DCR Graphs. Further in our opinion, the use of activity conditions suggests interesting variants of the constraint templates and hence also partially motivated for dynamic inclusion and exclusion relations in our DCR Graphs.

2.2 DECLARE: A Constraint Based Approach For Flexible Workflows

In this section we briefly introduce another important motivation for our formal model: DECLARE [van der Aalst *et al.* 2010a, Pesic *et al.* 2007, van der Aalst *et al.* 2009] and its declarative process languages [van der Aalst & Pesic 2006a, van der Aalst & Pesic 2006b]. In this section, we very briefly introduce DECLARE framework and key primitives of its declarative languages and further explain how it has served as motivating factor for DCR Graphs.

The DECLARE is a system for supporting declarative or loosely-structured process models. The DECLARE has been developed as a constraint-based framework that uses declarative languages expressed in linear temporal logic [Pnueli 1977], for specification and execution of business processes. Even though the DECLARE is a framework for declarative processes, it offers most of the features similar to traditional workflow management systems such as process development tools, verification support, simulation support for model execution, support for adaptive changes and support for mining of already executed processes [Pesic *et al.* 2007]. As opposed to imperative approaches to process modeling, the DECLARE uses declarative modeling paradigm and the difference between the declarative and imperative approaches have been discussed in sec 1.4 of the introduction chapter.

2.2.1 Process Modeling

The DECLARE uses constraints to specify relations between activities/tasks. Like traditional modeling languages (for example BPMN), offering a predefined set of relations between tasks or activities such as sequence, choice, parallelism, and loop, the DECLARE allows for customized constraints templates for specification. The Declare framework supports two very similar declarative languages: ConDec [van der Aalst & Pesic 2006a] and DecSerFlow [van der Aalst & Pesic 2006b]. ConDec is a declarative language for specification of business processes and workflows, where as DecSerFlow is language tailored towards the specification of web services.

In the DECLARE it is possible to define new constraint templates using linear temporal logic (LTL). Basically new templates can be defined using basic LTL operators: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), next ($\bigcirc$). LTL constraints can be assigned with name and graphical notation, so that users are not constrained to know LTL, on the other hand they can use graphical languages to model the processes, without knowing the underlying LTL formulae. The list of predefined constraints in LTL that can be used in DECLARE can be found in [van der Aalst & Pesic 2006b], however, here we will briefly mention about two constraints: precedence and response constraints, which are also motivation for condition and response relations in our formal model. The semantics of those two constraints are given in 2.1.2.1. In fact, we have used the same graphical notation and semantics for the two relations in our formal model DCR Graphs, which will be introduced in chapter 3.

The DECLARE framework supports both *mandatory* and *optional* constraints. The tool enforces the *mandatory* constraints so that an execution can not violate the *mandatory* constraints, where as in case of *optional* constraints, users are allowed to violate them and in such a case the tool only warns about violation of an optional constraint.

2.2.2 Process Execution

The specification of a process in DECLARE is mapped onto a set of LTL formulae, primarily defining the constraints between the activities. From the specification in LTL, an automata is generated using standard techniques [Gerth *et al.* 1996] to generate automata from an LTL formulae. Many algorithms have been developed in the last decades about generating the an automata from LTL specification and DECLARE uses an algorithm that creates finite words automata [M.Clarke *et al.* 1999] from LTL formulas of specification. Once the automata is generated, it is used to support enactment and monitor state of each constraint.

Adopting the runtime instances of process is an important feature of flexible workflow management systems and the DECLARE supports changing the process models during their execution. In DECLARE, it is possible to add, delete activities together with relating constraints and also possible to change the data associated with activities and constraints can be added or deleted and can be made optional during execution of process instances [Pesic *et al.* 2007].

When a model is adopted, it verifies the compliance of these changes and the users will be notified if there are any conflicts with the already executed part of the instance. After the adaptation of the running instances, the modified process model is re-initialized and a new automata will be generated from the modified process instances based on the new set of constraints and finally the already executed part of the instance is replayed on the new automata, to get the updated state.

2.2.3 Conclusion

The DECLARE and its declarative languages ConDec [van der Aalst & Pesic 2006a] and DecSerFlow [van der Aalst & Pesic 2006b] are one of the first few workflow formalisms that made significant impact on the research of finding new ways of modeling for achieving flexibility. Our approach is closely related to the work on ConDec [van der Aalst & Pesic 2006a] and DecSerFlow [van der Aalst & Pesic 2006b]. The crucial difference is that we allow nesting and a few core constraints making it possible to describe the state of a process as a simple marking. ConDec does not address dynamic inclusion/exclusion of activities, but on the other hand allows one to specify any relation expressible within Linear-time Temporal Logic (LTL). This offers much flexibility with respect to specifying execution constraints. In particular the condition and response relations in our formal model are same as precedence and response constraints and hence we have used same graphical notation.

However their approach suffers from problems related to efficiency in executing business processes [van der Aalst *et al.* 2009]. The DECLARE engine has problems in dealing with large workflow specifications because of the complexity of generating automata from the LTL specification.

In the recent years, there has been significant work in DECLARE framework on improving the efficiency of translation from LTL to automata. Especially in the latest work [Westergaard 2011], significant performance has been achieved by exploiting characteristics of LTL formulae originating from a DECLARE specification as they are conjunction of simpler formulae defined by the individual constraints. The approach used by Westergaard in [Westergaard 2011], is by computing automaton product for the individual formulae instead of computing the automaton for the whole LTL specification which is a conjunction of all the formulae. No doubt, the DECLARE has also served as a big motivation for using declarative modeling primitives for our formal model. However as mentioned in the summary 2.1.5 of last section, we have decided to not to use LTL for specification of modeling language for business processes.

2.3 Event Structures

In this section, we briefly introduce the Event Structures [Nielsen *et al.* 1979, Winskel 1986, Winskel 2011, Winskel & Nielsen 1993, Winskel & Nielsen 1995], which serves as the base theory behind our formal model DCR Graphs. First we give a brief introduction to the theory behind event structures and later we introduce some of the basic definitions of event structure with an example and finally we provide concluding remarks stating the reasons for choosing the event structures as the base theory for DCR Graphs.

2.3.1 Introduction

Event Structures can be regarded as a minimal, declarative model for concurrent processes. In a more general setting, event structures can be thought as a model of computational processes and a process can be represented using event structures as a set of event occurrences with an explicit relation to express how events casually depend on others [Winskel 1986]. More precisely, in event structures what is important is the significance of events and how the occurrence of an event depend on the previous occurrence of some other events. To model nondeterminism, event structures have binary conflict relation between the events, expressing how occurrences of some events will rule out the possibility of happening of others events.

The primary motivation for event structures was to develop a theory of concurrency that incorporates insights from both Petrinets [Petri 1980, Petri 1977] and Scott domain of information [Scott 1970, Scott 1976, Scott 1982], by connecting the idea of events with partial orders of information [Nielsen *et al.* 1979]. The relations on events in the event structures bear a close relationship to the Petrinets, where as the configurations and states of an event structure represents the information about what events have occurred and hence determine a Scott domain of information. Due to this dual nature, Event structures stand as intermediary between the theories of Petri nets and denotational semantics and by sharing the ideas from both formalisms, they serve as bridge between the two theories [Winskel 1986].

2.3.2 Event Structures, Configurations

In this section, we will introduce some of the basic definitions of event structures, based on the formal definitions from [Winskel & Nielsen 1993].

Definition 2.3.1. *A* prime event structure *is a 3-tuple* $\mathsf{ES} = (\mathsf{E}, \leq, \#)$ *where*

(i) E *is a (possibly infinite) set of events*

(ii) $\leq \subseteq \mathsf{E} \times \mathsf{E}$ *is the* causality *relation between events which is a partial order*

(iii) $\# \ \subseteq \mathsf{E} \times \mathsf{E}$ *is a binary* conflict *relation between events which is irreflexive and symmetric*

*An events structure (*ES*) must satisfy the conditions that*

1. *causality relation satisfies principle of finite causes*
   $\forall e\downarrow= \{e' \mid e' < e\}$ *is finite for any* $e \in \mathsf{E}$.

2. *conflict relation satisfies the principle of conflict heredity*
   $\forall e, e', e'' \in \mathsf{E}. e\#e' \leq e'' \implies e\#e''$

The condition (1) states that the set of events which are causally depend on an event is finite, where as the axiom on conflict relation (2) expresses that if an event is causally depends on another event, which is in turn conflict with a third event, then even the first event will also be in conflict with the third event. We now define the causal independence (concurrency) of events in an event structure as follows,

Definition 2.3.2. *For an event structure* $\mathsf{ES} = (\mathsf{E}, \leq, \#)$ *the* causal independence *of events is expressed by a derived relation* $co \subseteq \mathsf{E} \times \mathsf{E}$, *such that* $e\ co\ e'$ *iff* $\neg(e \leq e' \vee e' \leq e \vee e\#e')$.

Further, the behavior of an event structure can be described by stating which subsets of events can happen in a possible run of an system representing the event structure and these subsets of events are called *configurations*. Now we formally define a configuration of an event structure as follows.

Definition 2.3.3. *For an event structure* $\mathsf{ES} = (\mathsf{E}, \leq, \#)$, *a* configuration *is a set of events* $\mathcal{C} \subseteq \mathsf{E}$ *satisfying the conditions*

(i) conflict-free*:* $\forall e, e' \in \mathcal{C}. \neg(e\#e')$

(ii) downwards-closed*:* $\forall e \in \mathcal{C}, e' \in \mathsf{E}. e' \leq e \implies e' \in \mathcal{C}$

We further define configurations $\mathcal{D}(\mathsf{ES})$ *as a set of all* configuration $\mathcal{C}$ *and denote* $\mathcal{D}^0(\mathsf{ES})$ *for a set of finite configurations.*

The conflict relation between the events implies that both events can not happen in the same configuration (i), in other words occurrence of one event will exclude the occurrence of the other, where as (ii) condition says that if an event happened, then all the events that casually depend on it must have happened before.

We can infer important relations associated with an event structure from its finite configurations as follows,

Proposition 2.3.1. *For an event structure* $\mathsf{ES} = (\mathsf{E}, \leq, \#)$ *with a set of finite configurations* $\mathcal{D}^0(\mathsf{ES})$,

(i) $e \leq e' \Leftrightarrow \forall \mathcal{C} \in \mathcal{D}^0(\mathsf{ES}).\ e' \in \mathcal{C} \implies e \in \mathcal{C}$

(ii) $e\#e' \Leftrightarrow \forall \mathcal{C} \in \mathcal{D}^0(\mathsf{ES}).\ e \in \mathcal{C} \implies e' \notin \mathcal{C}$

(iii) $e\ co\ e' \Leftrightarrow \exists \mathcal{C}, \mathcal{C}' \in \mathcal{D}^0(\mathsf{ES}).\ (e \in \mathcal{C}) \wedge (e \notin \mathcal{C}') \wedge (e' \in \mathcal{C}') \wedge (e' \notin \mathcal{C}) \wedge (\mathcal{C} \cup \mathcal{C}') \in \mathcal{D}^0(\mathsf{ES})$

Definition 2.3.4. *For an event structure* ES = (E, ≤, #)*, let* $\mathcal{C}, \mathcal{C}'$ *be the configurations, then we can write that*

$$\mathcal{C} \xrightarrow{e} \mathcal{C}' \Leftrightarrow e \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C} \cup \{e\}$$

In an event structure, events can only happen at most once (def 2.3.4) and further they can be perceived as atomic jumps from one configuration to another, like transitions in asynchronous transition systems [Winskel & Nielsen 1993].

Example 2.3.1. *Event structures can exhibit nondeterminism. For example consider an event structure with two events* e_0 *and* e_1 *with a conflict relation between them* (e_0#e_1) *as shown in the figure 2.6, in which* $\{e_0\}, \{e_1\} \in \mathcal{D}^0(\text{ES})$*, but* $\{e_0, e_1\} \notin \mathcal{D}^0(\text{ES})$.

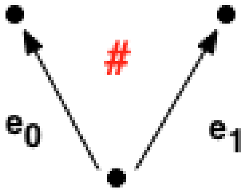


Figure 2.6: Nondeterministic behavior in events structures

Example 2.3.2. *Event structures can exhibit parallelism or concurrency. For example consider an event structure with two events* e_0 *and* e_1 *as shown in the figure 2.7, in which we have configurations* $\emptyset, \{e_0\}, \{e_1\}, \{e_0, e_1\} \in \mathcal{D}^0(\text{ES})$.

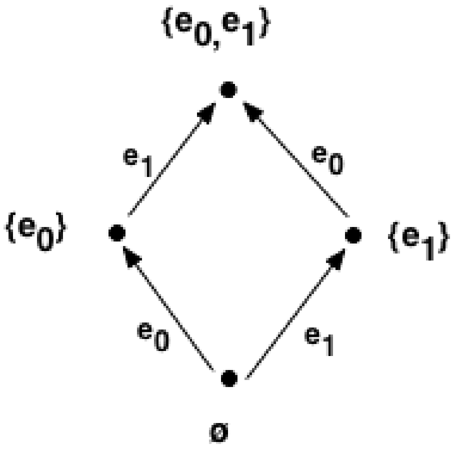


Figure 2.7: Concurrency in events structures

Proposition 2.3.2. *Two events* e_0*,* e_1 *of an event structure are in concurrency relation co, iff there exists configurations* $\emptyset, \{e_0\}, \{e_1\}, \{e_0, e_1\} \in \mathcal{D}^0(\text{ES})$ *as shown in the figuree 2.7 and we will write that as* e_0 *co* e_1.

Often processes needs to perform action multiple times, but events in the event structures can happen only once. In order to model such processes, it would be helpful to add labels to event structures so that each occurrence of an action can be modeled by different event. We do this by extending the definition of event structures with set of labels and a leveling function to define *labeled event structures* as follows,

Definition 2.3.5. *A* labeled event structure *is a tuple* LES $= (\mathrm{ES}, \mathrm{Act}, l)$ *where*

(i) $\mathrm{ES} = (\mathrm{E}, \leq, \#)$ *is an event structre,*

(ii) Act *is the set of actions*

(iii) $l : \mathrm{E} \rightarrow \mathrm{Act}$ *is the* labeling *function mapping events to actions*

A run ρ *of* LES *is a (possibly infinite) sequence of labelled events* $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *such that for all* $i \geq 0$. $\cup_{0 \leq j \leq i} \{e_j\}$ *is a configuration.*

A run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *is* maximal *if any enabled event eventually happen or become in conflict, formally* $\forall e \in \mathrm{E}, i \geq 0 . e \downarrow \subseteq (e_i \downarrow \cup \{e_i\}) \implies \exists j \geq 0.(e \# e_j \vee e = e_j)$.

Let us take a small example to illustrate how labeled event structures can used to model a process that performs actions multiple times.

Example 2.3.3. *Consider a process which exhibit a behavior* $(a; b; c;) + (a \mid b)$*, where we can execute either actions a, b, c sequentially or actions a and b independently.*

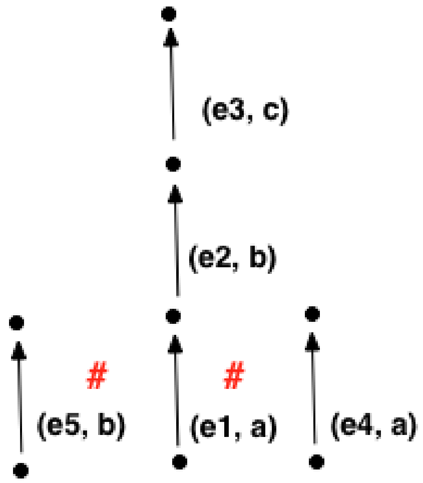


Figure 2.8: Process in labeled events structures

We can model the process in labeled event structures as,
$\mathrm{E} = \{e_1, e_2, e_3, e_4, e_5\}$, $\mathrm{Act} = \{a, b, c\}$, $l = \{(e_1, a), (e_2, b), (e_3, c), (e_4, a), (e_5, b)\}$, $\leq = \{(e_1, e_2), (e_2, e_3)\}$, $\# = \{(e_1, e_4), (e_1, e_5)\}$ *and* $co = \{(e_4, e_5)\}$.

The same process can be shown graphically in 2.8, where we have used arrows or directed arcs to represent causality and hash mark (#) to represent conflict. The set of all configurations will be
$\mathcal{D}^0(\mathrm{LES}) = \{\emptyset, \{(e_1, a)\}, \{(e_1, a), (e_2, b)\}, \{(e_1, a), (e_2, b), (e_3, c)\},$
$\{(e_4, a)\}, \{(e_5, b)\}, \{(e_4, a), (e_5, b)\}\}$

We further model the give medicine healthcare example, which was introduced in sec 1.4 from the case study 2.1.3, in labeled event structures in 2.3.4.

Example 2.3.4. *In this example we consider that the set of events and actions is the same and hence we omit display of action labels in the figure 2.9.*

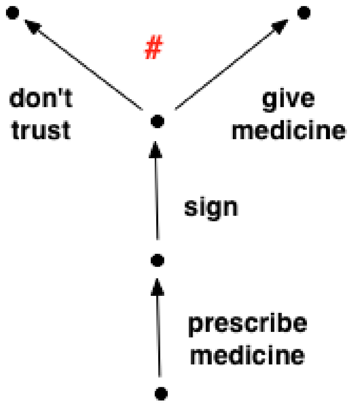


Figure 2.9: Give medicine example in events structures

E = Act = {*prescribe medicine, sign, don't trust, give medicine*} ,
l = {*(prescribe medicine, prescribe medicine), . . .*},
≤ = {*(prescribe medicine, sign), (sign, don't trust), (sign, give medicine)* },
# = {*(don't trust, give medicine)* } ,
$\mathcal{D}^0$(LES) = {∅, {*prescribe medicine*}, {*prescribe medicine, sign*}, {*prescribe medicine, sign, don't trust*}, {*prescribe medicine, sign, give medicine*} }

2.3.3 Conclusion

In this section, we will explain the reasons why we have chosen event structure as base theory behind our formal model DCR Graphs and also talk about short-comings of event structures to use them for specifications and execution of declarative business processes and workflows.

Event structures are suitable for process modeling of declaratives workflows because of the following reasons.

1) Event structures has a strong formal foundation, and it has been developed as a concurrency theory bridging the gap between traditional domain theory and

net based process languages such as Petrinets, by incorporating good insights from both the theories. Further, event structures have been studied by many researchers and used to give semantics for nondeterministic dataflow [Saunders-Evans & Winskel 2007], higher order process languages, CCS and related languages [Winskel 1982, Crafa *et al.* 2007] and a logical framework for reputation systems [Krukow *et al.* 2008].

2) Event structures has causality relation to express partial order among events, which is a more declarative way of specifying the precedence among the events. It also has conflict relation to model nondeterministic behavior among the events in an indigenous way. Moreover events which are not related by causality and conflict are concurrent [prop 2.3.1], so the concurrency has been naturally built into the model. All these characteristics make the event structures as a natural and more suitable choice to use them as formal model for declarative processes.

However, we have noticed that there are certain missing aspects in event structures, in order to able to use them as an formal model for specification execution of workflows.

1) First of all, events in the event structures can only happen once and we feel that it is a limitation to express repeated, possibly infinite behavior. To be more precise, in the example 2.3.4 it should have be possible to execute *give medicine* many times repeatedly, may be after *don't trust* followed by *sign*, but not possible because the events can only be executed once. One may argues that the repeated behavior in event structures could be modeled using labels as shown in the example 2.3.3, but this approach on one hand makes the modeling part complicated and on the other hand it will not be possible to model infinite behavior, for example if we don't know how many times an action should be repeated. So we feel that, lacking of repeated behavior of events, is an important limitation in order to use event structures for modeling of declarative processes.

2) Secondly, it must be possible to specify that only some of the partial (or infinite) computations are acceptable. But event structures has no notion of categorizing some of the configurations/runs as accepting or desirable, which is an important characteristic to model the declarative process models. Moreover, sometimes it is necessary to specify that some of the events are mandatory in a process model, but event structures does not have any such constructs.

3) Finally, we need to be able to describe a distribution of events over the agents /persons/processors. But event structures does not have such notion.

In the next chapter [section 3.1.1], we will describe how we have addressed these limitations by proposing generalizations to the event structures, to develop the formal model for declarative workflows.

2.4 Summary

In this chapter, we have introduced formalisms and process models that have served as motivation for our formal model DCR Graphs. First we have described our previous work on formalization of Resultmaker process model using linear temporal logics in sec 2.1.2 and then we have introduced a healthcare case study in sec 2.1.3. We have explained the drawbacks of our previous approach using LTL for formalization of Resultmaker process model in 2.1.5 and then very briefly introduced DECLARE [van der Aalst *et al.* 2010a, van der Aalst & Pesic 2006b, van der Aalst & Pesic 2006a] tool and its approach in using declarative modeling languages in 2.2.

In the last section 2.3, we have introduced Event structures [Winskel 1986], which is the base theory behind our formal model. We have also explained reasons for choosing Event structures and at the same also pointed out the missing aspects of event structures in order to use them as a formal model for business processes in the concluding section 2.3.3. In the next chapter 3, we will explain how we have generalized event structures to define our formal model and also introduce formal semantics of DCR Graphs.

Chapter 3

Dynamic Condition Response Graphs

In the previous chapter, we have examined the formal models which have served as motivation for our formal model Dynamic Condition Response Graphs (DCR Graphs). This chapter will introduce the formal semantics of DCR Graphs. First we will give a brief motivation for DCR Graphs in sec 3.1 and then we will discuss about a sequence of proposed generalizations to labelled event structures in sec 3.1.1. A brief discussion relating our model to other formalisms will be presented in the section 3.2. We will introduce *condition response event structures* as the first generalization in sec 3.3.1 and then show how the response relation allows us to represent the notion of weak fairness. In sec. 3.3.2 we will introduce the model of DCR Graphs and by extending the model with role and principals we define distributed DCR Graphs in sec 3.3.3 and the execution semantics to are mapped to a labelled transition system. Furthermore, we will further formalize the execution semantics of DCR Graphs for infinite runs by providing a mapping to Büchi-automaton with τ-events in the sec 3.3.4. Then, we will introduce graphical notation and give an healthcare example for DCR Graphs in sec 3.4. Finally, as part of expressibility of DCR Graphs, we encode Büchi-automaton into the DCR Graphs and show that the DCR Graphs are expressive enough to model ω-languages. We will end the chapter by concluding remarks in sec 3.6.

This chapter extends and summarizes the work presented in the two previous short papers [Hildebrandt & Mukkamala 2011, Mukkamala & Hildebrandt 2010] and a journal version [Hildebrandt & Mukkamala 2010]. The paper [Hildebrandt & Mukkamala 2011] introduced condition response event structures and dynamic condition response structures and provided a mapping to finite state machines (ignoring infinite runs), which are essentially DCR Graphs without markings. The paper [Mukkamala & Hildebrandt 2010] provided a mapping from dynamic condition response structures to Büchi automata, but only capturing acceptance for the infinite runs. In [Hildebrandt & Mukkamala 2010], the DCR Graphs graphs were introduced and this paper also characterizes the acceptance of finite runs in the Büchi automata by introducing silent (τ) transitions.

3.1 Motivation

There is a long tradition for using declarative logic based languages to schedule transactions in the database community, see e.g. [Fernandes *et al.* 1997]. Several researchers have noted [Davulcu *et al.* 1998, Senkul *et al.* 2002, Singh *et al.* 1995, Bussler & Jablonski 1994, van der Aalst *et al.* 2009] that it could be an advantage to use a declarative approach to achieve more flexible process descriptions in other areas, in particular for the specification of case management workflow and ad hoc business processes. The increased flexibility is obtained in two ways: Firstly, since it is often complex to explicitly model all possible ways of fulfilling the requirements of a workflow, imperative descriptions easily lead to over-constrained control flows. In the declarative approach any execution fulfilling the constraints of the workflow is allowed, thereby leaving maximal flexibility in the execution. Secondly, adding a new constraint to an imperative process description often requires that the process code is completely rewritten, while the declarative approach just requires the extra constraint to be added. In other words, declarative models provide flexibility for the execution at run time and with respect to changes to the process.

As a simple motivating example, we will again consider the hospital workflow from Danish hospitals [Lyng *et al.* 2008, Mukkamala *et al.* 2008], which has also been used in the previous chapters. As a start, we assume two events, *prescribe* and *sign*, representing a doctor adding a medical prescription to the patient's record and signing it respectively. We assume the constraints stating that the doctor must sign after having added a prescription of medicine to the patient record and not to sign an empty record. A naive imperative process description may simply put the two actions in sequence, *prescribe;sign*, which allows the doctor first to prescribe medicine and then sign the record. In this way, the possibilities of adding several prescriptions before or after signing and signing multiple times are lost, even if they are perfectly legal according to the constraints. The most general imperative description should start with the prescribe event, followed by a loop allowing either sign or prescribe events and only allow termination after a sign event. If the execution continues forever, it must be enforced that every prescription is eventually followed by a sign event.

With respect to the second type of flexibility, consider adding a new event *give*, representing a nurse giving the medicine to the patient, and the rule that a nurse must give medicine to the patient if it is prescribed by the doctor, but not before it has been signed. For the most general imperative description we should add the ability to execute the *give* event within the loop after the first *sign* event and not allow to terminate the flow if we have had a *prescribe* event without a subsequent *give* event. So, we have to change the code of the loop as well as the condition for exiting it.

As discussed in 2.2, van der Aalst and Pesic [van der Aalst & Pesic 2006a, Pesic 2008] propose to use Linear-time Temporal Logic (LTL) as a declarative language for describing the constraints of the workflow. LTL allows for describing a rich set of constraints on the execution flow. However, as stated in sec. 2.2.3, this approach suffers from the fact that the subsequent tools for execution and analysis will refer to the LTL expression (or further compilations to e.g. Büchi automata) and not the graphical notation. Also, the full generality of LTL may lead to a poor execution time.

This motivates researching the problem of finding an expressive declarative process language where both the constraints as well as the run time state can be easily visualized and understood by the end user and also allows an effective execution.

We believe that the declarative process model language of *dynamic condition response graphs* and its graphical representation proposed in this thesis is a promising candidate. Primarily, the model is inspired by and a conservative generalization of the declarative *process matrix* model language [Lyng *et al.* 2008, Mukkamala *et al.* 2008] used by our industrial partner and prime event structures [Nielsen *et al.* 1979, Winskel 1986, Winskel & Nielsen 1993]. The model has also got inspired by the DECLARE [van der Aalst *et al.* 2010a, Pesic *et al.* 2007, van der Aalst *et al.* 2009] and its declarative process languages [van der Aalst & Pesic 2006a, van der Aalst & Pesic 2006b] in respect of the constrained based approach.

3.1.1 DCR Graphs as generalized Event Structures

In this section we will discuss about how distributed dynamic condition response graphs developed as a sequence of three generalizations of prime event structures as shown in the figure 3.1.

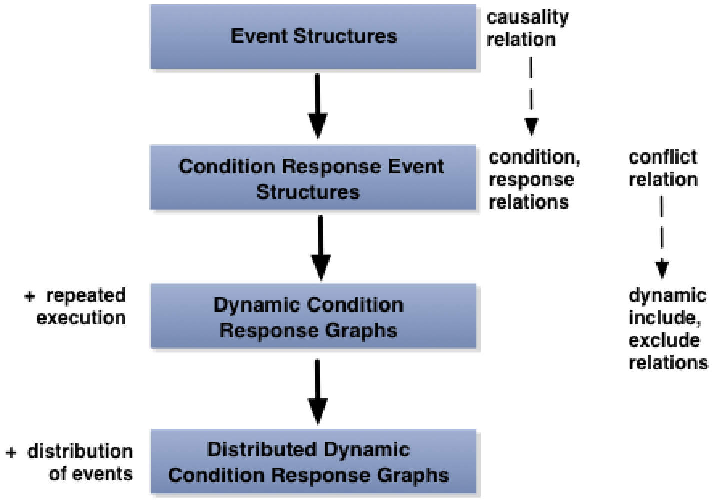

Event Structures
causality relation
Condition Response Event Structures
condition, response relations
conflict relation
+ repeated execution
Dynamic Condition Response Graphs
dynamic include, exclude relations
+ distribution of events
Distributed Dynamic Condition Response Graphs

Figure 3.1: From Event Structures to DCR Graphs overview

The first generalization, named *condition response event structures*, is obtained by adding a set Re of *initially required response events* and by generalizing the causality relation into a *condition* and a *response* relations between events. The *condition* relation imposes precedence among the events, where as the *response* is a kind of follow-up relation. The initially required response events can be regarded as goals that must be fulfilled (or falsified) in order for an execution to be accepting. That is, for any event $e \in \mathrm{Re}$, either e must eventually happen or it must become in conflict with an event that has happened in the past. The response relation in some sense corresponds to the response LTL pattern in [van der Aalst & Pesic 2006a, van der Aalst & Pesic 2006b] as a dual relation to the usual condition relation: If an event b is a response to an event a then b must happen at some point after event a happens or become in conflict. However, note that the response pattern does not allow for conflicts. Operationally, as we will see in the following section, one can think of the event b as being added to the set Re of required responses when a happens.

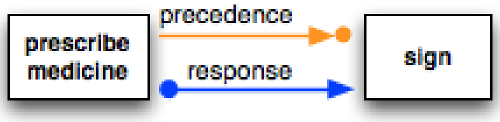


Figure 3.2: Prescribe and Sign Example

Next we generalize *condition response event structures* by allowing each event to happen many times and replacing the symmetric conflict relation by an asymmetric relation which *dynamically* determines which events are included in or excluded from the structure. To allow the graphs to represent intermediate run time state (e.g. like the marking of a Petri Net) we also add sets of *included* (In) and *executed* (Ex) events and refer to the triple of sets of pending responses, included and executed events as the *marking* of the graph. This results in the model of *Dynamic Condition Response Graphs*, short DCR Graphs.

Finally, we reach the model of *Distributed Dynamic Condition Response Graphs* allowing for role based *distribution* by adding a set of *principals* and a set of *roles* assigned to both principals and events, and define that an event can only be executed by a principal assigned one of the roles that were further assigned to the event.

3.2 Related Work

There exists many different approaches to formally specify and enact business processes and also It is not possible to provide a complete overview of related work, especially in the area of business processes and workflows. In this section we give a brief overview of some of the formal approaches and compare our work to them.

In contrast to imperative modeling languages, the authors in [van der Aalst & Pesic 2006a, van der Aalst *et al.* 2009] have proposed *ConDec*, a declarative language for modeling and enacting the dynamic business processes based on Linear Temporal Logic (LTL). In [van der Aalst & Pesic 2006b], the authors have proposed Declarative Service Flow language (DecSerFlow) to specify, enact and monitor service flows, which is a sister language for *ConDec*. Both the languages share the same concepts and are supported in the *Declare* [van der Aalst *et al.* 2009] tool. They specifies *what* should be done, instead of specifying *how* it should be done, there by leaving more flexibility to users. The enactment in both the languages is defined by translating the constraints specified in LTL, into a Buchi automaton and executing the workflow/service by executing the referring Buchi automaton. LTL being a very expressive language, the Declare tool suffers from efficiency problems in executing models with large specification [van der Aalst *et al.* 2009]. Even though our approach is inspired from the *ConDec, DecSerFlow* models [van der Aalst & Pesic 2006a, van der Aalst & Pesic 2006b] in respect of using declarative modeling languages, but our model has fewer primitives than LTL, but more expressible than LTL, as one can encode Büchi automaton into DCR Graphs (as shown in Sec. 3.5) and thus makes it more expressible than LTL.

In [Cicekli & Yildirim 2000, Cicekli & Cicekli 2006] the authors have proposed *Event Calculus* [Kowalski 1992] as a logic-based methodology for specification and execution of workflows. *Event Calculus* [Kowalski 1992] is a logic programming formalism for representing events and their effects in the context of database applications. In their approach, the authors have expressed the basic control flow primitives of workflows as a set of logical formulas and used axioms of *Event Calculus* to specify activity dependency execution rules and agent assignments rules. Their workflow model also supports enactment of concurrent workflow instances and iteration of activities, but does not support specification and verification of global and temporal constraints on workflow activities. Also, their approach is limited to imperative/procedural workflow modeling languages.

On the side of imperative modeling languages, Petri nets has been the major formalism that has been studied and used extensively in the domain of workflows and business processes [Van Der Aalst *et al.* 1997, Verbeek & Aalst 2000, Hee *et al.* 2004, Aalst *et al.* 2011]. Concurrent Transaction Logic (CTR) used in [Davulcu *et al.* 1998] as a language for specifying, analysis, scheduling and verification of workflows. In their framework, the authors have used CTR formulas for expressing the local and global properties of workflows and reasoning about the workflows has been done with the help of proof theory and semantics of logic. In [Senkul *et al.* 2002], the authors have used Concurrent Constraint Transaction Logic (CCTR) which is flavor of CTR integrated with Constraint Logic Programming for scheduling workflows. Like the other logic programming systems, the authors in [Davulcu *et al.* 1998, Senkul *et al.* 2002] have used the proof theory of CTR as run-time environment for enactment of workflows. The CTR approach mainly aims at developing an algorithm for consistency checking and verification of properties of workflows, but again only limited to imperative modeling languages.

In [van der Aalst *et al.* 2009], the authors provided a detailed overview of formalisms related to flexibility, ad-hoc and evolutionary changes. Authors in [Aalst 2001] addressed the problem of the dynamic change bug by computing the change regions of a process based on their structure, where as the dynamic change bug was first introduced by the autors in [Ellis *et al.* 1995]. Further, ADEPT Workflow Management System [Reichert & Dadam 1998, Rinderle *et al.* 2003, Rinderle *et al.* 2004, Reichert *et al.* 2003] offers advanced modeling concepts and features, like temporal constraint management, ad-hoc workflow changes and schema evolution and it has studied the problem related to dynamic changes in the context of workflows, but their approach also limited to imperative workflow modeling.

Further, in [Vanderaalst *et al.* 2005] the authors have strongly advocated case handling is a new paradigm for supporting flexible and knowledge intensive business processes and hence it should be avoided in restricting users in their actions.

In [Adams *et al.* 2006, Adams 2007] authors identified pockets of flexibility that can be selected later in the process as some sort of late binding at run time, where as authors in [Sadiq *et al.* 2001] proposed a similar approach where specification of the change itself integrated in the process.

Another major paradigm in business process modeling is the artifact-centric approach, which strongly argues that data design should be elevated to the same level as control flows for data rich workflows and business processes. In this area, several researchers [Nigam & Caswell 2003, Bhattacharya *et al.* 2007a, Liu *et al.* 2007] have been working with artifact-centric or data-centric workflows. As part of the artifact-centric models, a declarative approach has been taken in the recent years for specifying the life cycles of business entities, using the *Guard- Stage-Milestone* (GSM model) life cycles model [Damaggio *et al.* 2011, Hull *et al.* 2011a, Hull *et al.* 2011b]. The GSM model is a declarative process model for specification of interactions between business entities and its operational semantics are based on rules similar to ECA(Event Condition Action)-like rules from Active database community. In comparison, their main focus is on business artifacts which takes the data-centric view of processes, where as our approach is on the business processes using declarative modeling approaches where the control flow is more explicit than data-centric processes.

3.3 Dynamic Condition Response Graphs

In this section, we will first introduce Condition Response Event Structures and then introduce the formal semantics of DCR Graphs in Sec. 3.3.2. Later, we will extend DCR Graphs with roles and principles and define distributed DCR Graphs. Finally, for infinite runs we will define the execution semantics by mapping to Büchi-automaton in Sec. 3.3.4.

3.3.1 Condition Response Event Structures

As an intermediate step towards dynamic condition response graphs, we generalize lebeled event structures to allow for a notion of *progress* based on a response relation. This model is interesting in itself as an extensional event-based model with progress, abstracting away from the intentional representation of repeated behavior. In particular we show that it allows for an elegant characterization of weakly fair runs of event structures.

First let us recall the definition of a labeled event structure and configurations from the last chapter 2.3.

Definition 3.3.1. *A labeled* event structure (ES) *is a 5-tuple* $E = (\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *where*

(i) E *is a (possibly infinite) set of events*

(ii) Act *is the set of actions*

(iii) $\leq \subseteq \mathsf{E} \times \mathsf{E}$ *is the* causality *relation between events which is a partial order*

(iv) $\# \subseteq \mathsf{E} \times \mathsf{E}$ *is a binary* conflict *relation between events which is irreflexive and symmetric*

(v) $l : \mathsf{E} \rightarrow \mathsf{Act}$ *is the* labeling *function mapping events to actions*

The causality and conflict relations must satisfy the conditions that

1. $\forall e, e', e'' \in \mathsf{E}. e \# e' \leq e'' \implies e \# e''$
2. $\forall e \downarrow = \{e' \mid e' < e\}$ *is finite for any* $e \in \mathsf{E}$.

A configuration *of E is a set* $\mathcal{C} \subseteq \mathsf{E}$ *of events satisfying the conditions*

1. conflict-free*:* $\forall e, e' \in \mathcal{C}. \neg e \# e'$
2. downwards-closed*:* $\forall e \in \mathcal{C}, e' \in \mathsf{E}. e' \leq e \implies e' \in \mathcal{C}$

A run ρ *of E is a (possibly infinite) sequence of labelled events* $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *such that for all* $i \geq 0. \cup_{0 \leq j \leq i} \{e_j\}$ *is a configuration.*

A run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *is* maximal *if any enabled event eventually happen or become in conflict, formally* $\forall e \in \mathsf{E}, i \geq 0. e \downarrow \subseteq (e_i \downarrow \cup \{e_i\}) \implies \exists j \geq 0.(e \# e_j \vee e = e_j)$.

Action names $a \in \mathsf{Act}$ represent the actions the system might perform, an event $e \in \mathsf{E}$ labelled with a represents occurrence of action a during the run of the system. The causality relation $e \leq e'$ means that event e is a prerequisite for the event e' and the conflict relation $e \# e'$ implies that events e and e' both can not happen in the same run, more precisely one excludes the occurrence of the other. The definition of maximal runs follows the definition of weak fairness for concurrency models in [Cheng 1995] and is equivalent to stating that the configuration defined by the events in the run is maximal with respect to inclusion of configurations.

We now generalize labeled event structures to *condition response event structures*, by adding a dual *response* relation $\bullet\!\!\rightarrow$, such that $\{e' \mid e \bullet\!\!\rightarrow e'\}$ is the set of events that must happen (or be in conflict) after the event e has happened for a run to be accepting. The resulting structures, named *condition response event structures*, in this way add the possibility to state progress conditions. The condition relation ($\rightarrow\!\!\bullet$) is same as the causality relation ($\leq$) in labeled event structure imposing precedence among the events. Further, we also introduce a subset of the events Re of *initial* responses, which are events that are initially required eventually to happen (or become in conflict). In this way the structures can represent the state after an event has been executed. As we will see below, it also allows us to capture the notion of maximal runs.

Definition 3.3.2. *A labeled* condition response event structure *(*CRES*) over an alphabet* Act *is a tuple* $(\mathsf{E}, \mathsf{Re}, \mathsf{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \#, l)$ *where* *(i)* $(\mathsf{E}, \mathsf{Act}, \rightarrow\!\bullet, \#, l)$ *is a labelled event structure, referred to as the underlying event structure, where* $\rightarrow\!\bullet$ *is a partial order relation imposing precedence among the events, satisfying the downward-closed condition of configuration in the underlying labeled event structure such that* $\forall e \in \mathcal{C}, e' \in \mathsf{E}. e' \rightarrow\!\bullet\ e \implies e' \in \mathcal{C}$

(ii) $\bullet\!\!\rightarrow\ \subseteq \mathsf{E} \times \mathsf{E}$ is the *response* relation between events, satisfying that $\rightarrow\!\bullet \cup \bullet\!\!\rightarrow$ is acyclic.

(iii) $\mathsf{Re} \subseteq \mathsf{E}$ *is the set of* initial responses.

We define a configuration $\mathcal{C}$ and run ρ of a CRES to be a respectively a configuration and run of the underlying event structure. We define a run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ to be *accepting* if $\forall e \in \mathsf{E}, i \geq 0. e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e \# e_j \vee (i < j \wedge e = e_j)$ and $\forall e \in \mathsf{Re}. \exists j \geq 0.(e \# e_j \vee e = e_j)$. In words, any pending response event must eventually happen or be in conflict. s

A labeled event structure can trivially be regarded as a condition response event structure with empty response relation. This provides an embedding of labeled event structures into condition response event structures which preserves configurations and runs.

Proposition 3.3.1. *The labelled event structure* $\mathsf{ES} = (\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *has the same runs as the condition response event structure* $\mathsf{CRES} = (\mathsf{E}, \emptyset, \mathsf{Act}, \leq, \emptyset, \#, l)$ *for which all runs are accepting.*

Proof. The set of events (E), actions (Act), labeling function (l), conflict relation (#) are same in both the CRES and ES and moreover the causality relation ($\leq$) in ES is same as the condition relation in CRES. Hence the given ES can be regarded as underlying labeled event structure for given CRES, therefore, according to definition 3.3.2 both will have same runs and configurations. Furthermore the set of initial responses (Re) and response relation ($\bullet\!\!\rightarrow$) are empty, hence all the runs in CRES are accepting. □

We can also embed event structures into CRES by considering every condition to be also a response and all events with no conditions to be initial responses. This characterizes the interpretation in [Cheng 1995] where only *maximal* runs are accepting. In other words, the embedding captures the notion of weakly fair execution of event structures.

Proposition 3.3.2. *The labelled event structure* $\mathsf{ES} = (\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *has the same runs and* maximal *runs as respectively the runs and the accepting runs of the condition response event structure* $\mathsf{CRES} = (\mathsf{E}, \{e \mid e\downarrow = \emptyset\}, \mathsf{Act}, \leq, \leq, \#, l)$.

Proof. The CRES and ES have same elements in respect of $\mathsf{E}, \mathsf{Act}, \leq, \#, l$ and hence both of them will have same runs according to definition 3.3.2. Further in order to prove the proposition, we need to prove that any maximal run in ES is also an accepting run in CRES.

According to definition 3.3.1, a run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ in ES is *maximal* if and only if $\forall e \in \mathsf{E}, i \geq 0. e\downarrow \subseteq (e_i\downarrow \cup \{e_i\}) \implies \exists j \geq 0.(e \# e_j \vee e = e_j)$. According to definition 3.3.2, the same run is accepting in CRES if and only if the run satisfies the following conditions.

1. $\forall e \in \mathrm{E}, i \geq 0. e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j)$
   In CRES, $\bullet\!\!\rightarrow = \leq$ and hence $e_i \bullet\!\!\rightarrow e \implies e_i \leq e$. According to definition of causality, we can imply that $\forall e \in \mathrm{E}, i \geq 0. e \downarrow \subseteq (e_i \downarrow \cup \{e_i\})$.
   But the maximal run in ES implies that $\forall e \in \mathrm{E}, i \geq 0. e \downarrow \subseteq (e_i \downarrow \cup \{e_i\}) \implies \exists j \geq 0.(e\#e_j \vee e = e_j)$.
   So we can conclude that $\forall e \in \mathrm{E}, i \geq 0. e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j)$ satisfies in CRES.

2. $\forall e \in \mathrm{Re}. \exists j \geq 0.(e\#e_j \vee e = e_j)$
   In CRES, $\mathrm{Re} = \{e \mid e \downarrow = \emptyset\}$, hence all the events Re are enabled from the beginning of the run. Since the run is maximal in ES, where all enabled events will eventually get executed, we can conclude that $\forall e \in \mathrm{Re}. \exists j \geq 0.(e\#e_j \vee e = e_j)$ will be satisfied in CRES.

Hence the proposition that any maximal run in ES is also an accepting run in CRES holds. □

Now we go further and define formally the event executions in condition response event structures. In order define executions in a formal way, we first define when an event is enabled in a condition response event structure in definition 3.3.3 and also the result of executing an event in condition response event structure in definition 3.3.4.

Definition 3.3.3. *For a labeled* condition response event structure $\mathrm{CRES} = (\mathrm{E}, \mathrm{Re}, \mathrm{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \#, l)$ *with a configuration $\mathcal{C}$, we define that an event e is enabled at a configuration $\mathcal{C}$ written as $\mathcal{C} \vdash e$ if and only if,*

(i) $e \downarrow \in \mathcal{C}$

(ii) $\{e' \mid e'\#e\} \notin \mathcal{C}$

Definition 3.3.4. *For a labeled* condition response event structure $\mathrm{CRES} = (\mathrm{E}, \mathrm{Re}, \mathrm{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \#, l)$ *with a configuration $\mathcal{C}$ and with an enabled event $\mathcal{C} \vdash e$, we define the result of executing e is $\mathcal{C}' = \mathcal{C} \cup e$.*

Having defined when events are enabled for execution and the effect of executing an event, now we finally define a finite execution in condition response event structure and when it is accepting formally as follows.

Definition 3.3.5. *For a labeled* condition response event structure $\mathrm{CRES} = (\mathrm{E}, \mathrm{Re}, \mathrm{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \#, l)$ *with a configuration $\mathcal{C}$, we define an execution to be a finite sequence of tuples $\{(\mathcal{C}_i, e_i, a_i, \mathcal{C}'_i)\}_{i \in [k]}$, each consisting of a configuration, an event, a label and an another configuration such that*

(i) $\mathcal{C} = \mathcal{C}_0$

(ii) $\forall i \in [k].a_i = l(e_i)$

(iii) $\forall i \in [k].\mathcal{C}_i \vdash e_i$

(iv) $\forall i \in [k].\mathcal{C}'_i = \mathcal{C}_i \cup e_i$

(v) $\forall i \in [k-1].\mathcal{C}'_i = \mathcal{C}_{i+1}$

We say that an execution is accepting if $\forall e \in \mathsf{E}, i \geq 0.e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j)$ *and* $\forall e \in \mathsf{Re}.\exists j \geq 0.(e\#e_j \vee e = e_j)$. *In words, any pending response event must eventually happen or be in conflict.*

3.3.2 DCR Graphs - Formal Semantics

We now go on to generalize condition response event structures to dynamic condition response graphs (DCR Graphs). As opposed to event structures, a dynamic condition response graph allows events to be executed multiple times and there are no constraints on the condition and response relations. This allows for finite representations of infinite behavior, but also for introducing deadlocks. Moreover, the conflict relation is generalized to two relations for dynamic exclusion and inclusion of events, which is more appropriate in a model where events can be re-executed and has shown useful in practice as a primitive for skipping events and constraints.

Further, we also add a new relation *milestone* from our later work on the Nested DCR Graphs [Hildebrandt *et al.* 2011c, Hildebrandt *et al.* 2011b], which has been discovered during a case study involving modeling of a workflow from a case management domain, that has been conducted jointly with our industrial partner Exformatics A/S. The *milestone* is also a blocking relation (similar to condition), but *milestone* blocks events based on the events in the set of pending responses (Re).

Being based on only five relations between events (condition, response, include, exclude and milestone) and with the role assignment, the distributed dynamic condition response graphs can be simply visualized as a directed graph with a box for each event as nodes and five different kinds of arrows. In this section, first we formally define a *dynamic condition response graph* (def 3.3.6) and later by adding *roles* and *principals*, we extend it to define *distributed dynamic condition response graph* in the next section (sec 3.3.3).

Definition 3.3.6. *A dynamic condition response graphis a tuple* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!+, \rightarrow\!\!\%, \rightarrow\!\!\diamond, l)$ *where*

(i) E *is the set of events, ranged over by e*

(ii) $\mathsf{M} \in \mathcal{M}(\mathsf{G}) =_{def} \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E})$ *is the marking and* $\mathcal{M}(\mathsf{G})$ *is the set of all markings.*

(iii) Act *is the set of actions*

(iv) $\rightarrow\!\!\bullet\ \subseteq \mathsf{E} \times \mathsf{E}$ *is the condition relation.*

(v) $\bullet\!\!\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$ *is the response relation.*

(vi) $\rightarrow\!+, \rightarrow\!\% \subseteq \mathsf{E} \times \mathsf{E}$ *is the dynamic* include *relation and* exclude *relation, satisfying that* $\forall e \in \mathsf{E}. e \rightarrow\!+ \cap e \rightarrow\!\% = \emptyset$*,*

(vii) $\rightarrow\!\diamond \subseteq \mathsf{E} \times \mathsf{E}$ *is the milestone relation.*

(viii) $l : \mathsf{E} \rightarrow \mathsf{Act}$ *is a labelling function mapping every event to an action.*

The condition (iv) and response (v) relations in *DCR Graphs* are similar to the corresponding relations in CRES, except that now they are not constrained in any way. In particular, we may have cyclic relations.

The marking (ii) $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \in \mathcal{M}(G)$ consists of three sets of events, capturing respectively which events have *previously been executed* (Ex), which events are *pending responses required to be executed or excluded* (Re), and finally which events are currently *included* (In). The set of pending responses Re of DCR Graphs thus plays the same role as the set of initial responses in the CRES.

The *dynamic inclusion/exclusion* (vi) relations $\rightarrow\!+$ and $\rightarrow\!\%$, represented by the (partial map) $\pm : \mathsf{E} \times \mathsf{E} \rightharpoonup \{+, \%\}$, allow events to be included and excluded dynamically in the graph. The intuition is that only the currently included events are considered in evaluating the constraints. This means that if an event a has event b as condition, but the event b is excluded from the graph then it is no longer required for a to happen. Similarly, if event a has the event b as response and if the event b is excluded then it is no longer required to happen for the flow to be acceptable. Formally, the relation $e \rightarrow\!+ e'$ expresses that, whenever event e happens, it will include e' in the graph. On the other hand, $e \rightarrow\!\% e'$ expresses that when e happens it will exclude e' from the graph.

The *milestone* relation (vii) is a blocking relation similar to condition, but it blocks based on events in the pending response set. For example, if an event a has the event b as a milestone ($b \rightarrow\!\diamond a$), then event a is not allowed to execute, if the event b is in the set of pending responses (Re). Similar to condition relation, the milestones are blocking only if they are included in the graph.

Now we go further and define the notion when an event is enabled formally in def 3.3.7. Before doing that, we will give the notation that will be employed in all our later definitions.

Notation 1. *For a set A we write $\mathcal{P}(A)$ for the power set of A. For a binary relation $\rightarrow \subseteq A \times A$ and a subset $\xi \subseteq A$ of A we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{a \in A \mid (\exists a' \in \xi \mid a \rightarrow a')\}$ and the set $\{a \in A \mid (\exists a' \in \xi \mid a' \rightarrow a)\}$ respectively. Also, we write $\rightarrow^{-1}$ for the inverse relation. Finally, for a natural number k we write $[k]$ for the set $\{1, 2, \ldots, k\}$.*

Definition 3.3.7. *For a dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!+, \rightarrow\!\%, \rightarrow\!\diamond, l)$ *with marking* $\mathsf{M} = \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\}$*, we define that an event* $e \in \mathsf{E}$ *is enabled, written as* $\mathsf{M} \vdash_{\mathsf{G}} e$ *if*

i) $e \in \mathsf{In}$

ii) $(\rightarrow\!\bullet\, e \cap \mathsf{In}) \in \mathsf{Ex}$

iii) $(\rightarrow\!\diamond\, e \cap \mathsf{In}) \in \mathsf{E} \setminus \mathsf{Re}$

For an event e to be enabled, first of all, it must be included in the graph (i), further, all the included events which are conditions to the event e must be in the set of executed events (ii) and none of the included events that are milestones for it are in the set of pending responses (iii).

We will now define formally the change to the marking when an enabled event is executed in def 3.3.8. First the event is added to the set of executed events (Ex) and removed from the set of pending responses (Re). Then, all the events that are a response to the event are added to the set of pending responses. Note that if an event is a response to itself, it will remain in the set of pending responses after execution. Similarly, the set of included (In) events is updated by including and excluding events that have include and exclude relation from the executed event. Further an event e' can not be both included and excluded by the same event e, but an event may include/exclude by itself. Also an event may trigger itself as a response and/or can have itself as a condition or a milestone.

Definition 3.3.8. *For a dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!+, \rightarrow\!\%, \rightarrow\!\diamond, l)$ *with marking* $\mathsf{M} = \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\}$ *and with an enabled event* $\mathsf{M} \vdash_{\mathsf{G}} e$*, the result of executing the event* e *will be a dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}', \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!+, \rightarrow\!\%, \rightarrow\!\diamond, l)$*, where* $\mathsf{M}' = \mathsf{M} \oplus_{\mathsf{G}} e = \{\mathsf{Ex}', \mathsf{Re}', \mathsf{In}'\}$ *such that*

i) $\mathsf{Ex}' = \mathsf{Ex} \cup \{e\}$

ii) $\mathsf{Re}' = (\mathsf{Re} \setminus \{e\}) \cup e\,\bullet\!\rightarrow$

iii) $\mathsf{In}' = (\mathsf{In} \cup e\!\rightarrow\!+) \setminus e\!\rightarrow\!\%$

Having defined when events are enabled for execution and the effect of executing an event, we define finite and infinite runs/executions in DCR Graphs and when they are accepting. Intuitively, an execution is accepting if any required, included response in any intermediate marking is eventually executed or excluded. We define a run and an accepting run in DCR Graphs as follows in definition 3.3.9.

Definition 3.3.9. *For a Dynamic Condition Response Graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{L}, l)$ *we define an* execution *of* G *to be a (finite or infinite) sequence of tuples* $\{(\mathsf{M}_i, e_i, a_i, \mathsf{M}_i')\}_{i \in [k]}$ *each consisting of a marking, an event, a label and another marking (the result of executing the event) such that*

i) $\mathsf{M} = \mathsf{M}_0$

ii) $\forall i \in [k].a_i \in l(e_i)$

iii) $\forall i \in [k].\mathsf{M}_i \vdash_{\mathsf{G}} e_i$

iv) $\forall i \in [k].\mathsf{M}_i' = \mathsf{M}_i \oplus_{\mathsf{G}} e_i$

v) $\forall i \in [k-1].\mathsf{M}'_i = \mathsf{M}_{i+1}$.

Further, we say the execution (or a run) is accepting *if* $\forall i \in [k].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$*, where* $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ *and* $\mathsf{M}'_j = (\mathsf{Ex}'_j, \mathsf{In}'_j, \mathsf{Re}'_j)$.

Finally we say that that a marking M' *is reachable in* G *(from the marking* M*) if there exists a finite execution ending in* M' *and let* $\mathcal{M}_{\mathsf{M}\rightarrow^*}(G)$ *denote the set of all reachable markings from* M.

From the semantics defined above, we can construct a labelled transition system with an accepting condition for finite runs as given in def 3.3.10.

Definition 3.3.10. *For a dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ *we define the corresponding labelled transition system* $\mathsf{TS}(\mathsf{G})$ *to be the tuple*

$$(\mathcal{M}(\mathsf{G}), \mathsf{M}, \mathcal{L}_{ts}(\mathsf{G}), \rightarrow)$$

where $\mathcal{L}_{ts}(\mathsf{G}) = \mathsf{E} \times \mathsf{Act}$ *is the set of labels of the transition system,* M *is the initial marking, and* $\rightarrow \subseteq \mathcal{M}(\mathsf{G}) \times \mathcal{L}_{ts}(\mathsf{G}) \times \mathcal{M}(\mathsf{G})$ *is the transition relation defined by* $\mathsf{M} \xrightarrow{(e,a)} \mathsf{M} \oplus_{\mathsf{G}} e$ *if* $\mathsf{M} \vdash_{\mathsf{G}} e$ *and* $a \in l(e)$.

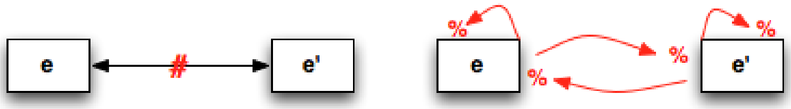
We define a run $a_0, a_1, \ldots$ *of the transition system to be a sequence of labels of a sequence of transitions* $\mathsf{M}_i \xrightarrow{(e_i,a_i)} \mathsf{M}_{i+1}$ *starting from the initial marking. We define a run to be accepting (or completed) if for the underlying sequence of transitions it holds that* $\forall i \geq 0, e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.((e = e_j \vee e \notin \mathsf{In}_{j+1}))$*. In words, a run is accepting/completed if no required response event is continuously included and pending without it happens or become excluded. Finally, we extend the transition relation to a relation between graphs by* $(\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l) \xrightarrow{(e,a)} (\mathsf{E}, \mathsf{M} \oplus_{\mathsf{G}} e, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ *if* $\mathsf{M} \xrightarrow{(e,a)} \mathsf{M} \oplus_{\mathsf{G}} e$.

If one only want to consider finite runs, which is common for workflows, the acceptance condition degenerates to requiring that no pending response is included at the end of the run. This corresponds to defining all states where $\mathsf{Re} \cap \mathsf{In} = \emptyset$ to be accepting states and define the accepting runs to be those ending in an accepting state. If infinite runs are also of interest (as e.g. for reactive systems and the LTL logic) the acceptance condition can be captured by a mapping to a Büchi-automaton with τ-events which we will define in Sec. 3.3.4.

A condition response event structure(CRES) can be represented as a dynamic condition response graph by making every event exclude itself and encode the conflict relation by defining any two conflicting events to mutually exclude each other as shown in figure 3.3.

Proposition 3.3.3. *The condition response event structure* $\mathsf{CRES} = (\mathsf{E}, \mathsf{Re}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \#, l)$ *has the same executions and accepting executions as the dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ *with marking* $\mathsf{M} = \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\}$ *where*

i) $\mathsf{Ex} = \emptyset$, $\mathsf{In} = \mathsf{E}$



(a) # relation in CRES

(b) Encoding of # in DCR Graphs

Figure 3.3: Encoding of conflict from CRES as mutual exclusion in DCR Graphs.

ii) $e \rightarrow\!\% \; e'$ *if* $e = e'$ *or* $e \# e'$ *and undefined otherwise.*

iii) $\rightarrow\!\diamond \; = \emptyset, \; \rightarrow\!+ = \emptyset$

Proof. According to the definition 3.3.5, an execution in CRES is a finite sequence of tuples $\{(\mathcal{C}_i, e_i, a_i, \mathcal{C}'_i)\}_{i \in [k]}$ such that $\mathcal{C} = \mathcal{C}_0$, $\forall i \in [k].a_i = \text{l}(e_i) \wedge \mathcal{C}_i \vdash e_i \wedge \mathcal{C}'_i = \mathcal{C}_i \cup e_i$ and $\forall i \in [k-1].\mathcal{C}'_i = \mathcal{C}_{i+1}$.
For a DCR Graph G, an execution is defined (definition 3.3.9) as a finite/infinite sequence of tuples $\{(\mathrm{M}_i, e_i, a_i, \mathrm{M}'_i)\}_{i \in [k]}$ such that $\mathrm{M} = \mathrm{M}_0$ and $\forall i \in [k].a_i \in \text{l}(e_i) \wedge \mathrm{M}_i \vdash_{\mathrm{G}} e_i \wedge \mathrm{M}'_i = \mathrm{M}_i \oplus_{\mathrm{G}} e_i$ and $\forall i \in [k-1].\mathrm{M}'_i = \mathrm{M}_{i+1}$.
In order to prove the proposition, first we have to prove that any finite execution in CRES is also a valid finite execution in G and then we will prove that any accepting execution in CRES is also accepting in G.

(I) CRES has same executions as G.
Here we use **proof by induction** and show that the proposition is valid for base case and then we will prove it for inductive step, assuming that if proposition is valid for a finite length of execution for $\forall i \in [l]$, then the proposition also valid for for $\forall i \in [l+1]$.
The set of events (E), actions (Act), labeling function (l),initial response set (Re), condition relation ($\rightarrow\!\bullet$), response relation ($\bullet\!\rightarrow$) are same in both CRES and G.

i) Base case (i = 0)
In the base case where i = 0, the execution in CRES is $\{(\mathcal{C}_0, e_0, \text{l}(e_0), \mathcal{C}'_0)\}$ and the tuple $(\mathcal{C}_0, e_0, a_0, \mathcal{C}'_0)$ implies that $\mathcal{C}_0 = \emptyset$, $e_0 \downarrow \; = \emptyset$ and $a_0 = \text{l}(e_0)$.
We have to show that in G for the base case (i=0), execution for event e_0 is valid and possible, that means we have to show that $\{(\mathrm{M}_0, e_0, a_0, \mathrm{M}'_0)\}$ is a valid execution in G.

In M_0, we have $\mathrm{Ex}_0 = \emptyset, \mathrm{Re}_0 = \mathrm{Re}, \mathrm{In}_0 = \mathrm{E}$ and we can imply the following

a) $e_0 \in \mathrm{E} \implies e_0 \in \mathrm{In}_0$

b) $(e_0 \downarrow \; = \emptyset) \implies (\rightarrow\!\bullet\, e_0 = \emptyset) \implies (\rightarrow\!\bullet\, e_0 \cap \mathrm{In}_0) \subseteq \mathrm{Ex}_0$

c) $(\rightarrow\!\diamond \; = \emptyset) \implies (\rightarrow\!\diamond\, e_0 = \emptyset) \subseteq \mathrm{E} \setminus \mathrm{Re}_0$.

Based on above implications, we can conclude that $\mathrm{M}_0 \vdash_{\mathrm{G}} e_0$ and also we know $a_0 = \text{l}(e_0)$ from the CRES.
Therefore $\{(\mathrm{M}_0, e_0, a_0, \mathrm{M}'_0)\}$ is a valid execution in G with $\mathrm{M}'_0 = (\mathrm{Ex} \cup \{e_0\}, \mathrm{Re} \setminus \{e\}) \cup e \bullet\!\!\rightarrow, \mathrm{In} \setminus (\{e_0\} \cup \{e' \mid e_0 \rightarrow\!\% \; e'\}))$.

ii) Inductive step:
Lets us assume that the proposition holds for a fixed length for $\forall i \in [l]$ and we have to show that the proposition also holds for $\forall i \in [l+1]$.
At the position $i = l$, we have the following executions.
CRES: $\{(\mathcal{C}_i, e_i, a_i, \mathcal{C}'_i)\}_{i\in[l]}$ and G: $\{(\mathrm{M}_i, e_i, a_i, \mathrm{M}'_i)\}_{i\in[l]}$.
Since the proposition holds until the position l, the executions in both CRES and G will have label sequences containing exactly the same events and labels and therefore we can imply that the set of executed events in both CRES and G are the same, which means $\mathrm{Ex}_l = \mathcal{C}_l$.
Further, let us say that the tuple CRES at position $i = l{+}1$ is $(\mathcal{C}_{l+1}, e_{l+1}, a_{l+1}, \mathcal{C}'_{l+1})$ and then we have to show that there exists a tuple:$(\mathrm{M}_{i+1}, e_{i+1}, a_{i+1}, \mathrm{M}'_{i+1})$ in G at position $i = l + 1$.
From the tuple in CRES: $(\mathcal{C}_{l+1}, e_{l+1}, a_{l+1}, \mathcal{C}'_{l+1})$, we can imply the following.

C-a) $a_{l+1} = \mathrm{l}(e_{l+1})$,

C-b) $e_{l+1} \downarrow\in \mathcal{C}_{l+1}$,

C-c) $\{e' \mid e' \# e_{l+1}\} \notin \mathcal{C}_{l+1}$,

C-d) $e_{l+1} \notin \mathcal{C}_{l+1}$ as events in CRES can happen only once.

The marking at position $i = l{+}1$ is $\mathrm{M}_{i+1} = (\mathrm{Ex}_{l+1} = \mathrm{Ex}_l \cup \{e_l\}, \mathrm{Re}_{l+1}, \mathrm{In}_{l+1})$.
At the position $i = l + 1$ in G, we can imply the following,

G-a) We have $(\mathcal{C}_{l+1} = \mathcal{C}_l \cup \{e_l\}) \wedge (\mathrm{Ex}_{l+1} = \mathrm{Ex}_l \cup \{e_l\}) \wedge (\mathrm{Ex}_l = \mathcal{C}_l) \implies \mathrm{Ex}_{l+1} = \mathcal{C}_{l+1}$ and with the definition of $\rightarrow\%$ in the proposition, we can imply that $\{e' \mid e' \# e_{l+1}\} \notin \mathcal{C}_{l+1} \wedge e_{l+1} \notin \mathcal{C}_{l+1} \implies (\{e' \mid e' \rightarrow\% e_{l+1}\} \cup \{e_{l+1}\}) \notin \mathrm{Ex}_{l+1}$.
Further $\mathrm{In}_0 = \mathrm{E} \implies e_{l+1} \in \mathrm{In}_0$ and with $(\{e' \mid e' \rightarrow\% e_{l+1}\} \cup \{e_{l+1}\}) \notin \mathrm{Ex}_{l+1}$, we can conclude that $e_{l+1} \in \mathrm{In}_{l+1}$, as the only way to exclude an event in G is by executing itself or any events which have have an exclude relation to it, which has not happen till $i = l{+}1$.

G-b) $(e_{l+1} \downarrow\ \in \mathcal{C}_{l+1}) \implies (\rightarrow\!\bullet\, e_{l+1} = \mathrm{Ex}_l) \implies (\rightarrow\!\bullet\, e_{l+1} \cap \mathrm{In}_{l+1}) \subseteq \mathrm{Ex}_{l+1}$

G-c) $(\rightarrow\diamond\ = \emptyset) \implies (\rightarrow\diamond\, e_l = \emptyset) \subseteq \mathrm{E} \setminus \mathrm{Re}_l$

G-d) From CRES we also know that $a_{l+1} = \mathrm{l}(e_{l+1})$.

From the definition 3.3.7 for enabled event in Gand based on the above implications, we can conclude that $\mathrm{M}_{i+1} \vdash_{\mathrm{G}} e_{l+1}$. Therefore, the tuple: $(\mathrm{M}_{i+1}, e_{i+1}, a_{i+1}, \mathrm{M}'_{i+1})$ exists in G at $i = l{+}1$ position and there by we can conclude that the execution in CRES: $\{(\mathcal{C}_i, e_i, a_i, \mathcal{C}'_i)\}_{i\in[l+1]}$ is same as G: $\{(\mathrm{M}_i, e_i, a_i, \mathrm{M}'_i)\}_{i\in[l+1]}$. Hence we have showed that if the proposition holds for a fixed length $i = l$, then it holds for length $i = l + 1$.

Since we have proved both base case and inductive step, we can conclude that both CRES has same executions as that of G.

(II) CRES has same accepting runs as G
According to definition 3.3.5 a run in CRES is accepting if and only if,

C-a) $\forall e \in \mathsf{Re}.\exists j \geq 0.(e\#e_j \vee e = e_j)$

C-b) $\forall e \in \mathsf{E}, i \geq 0.e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j))$

According to definition 3.3.9 a run in G is accepting if and only if, $\forall i \in [k].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}'_j = (\mathsf{Ex}'_j, \mathsf{In}'_j, \mathsf{Re}'_j)$. Informally, an execution in G is accepting if any required, included response in any intermediate marking is eventually executed or excluded.
To prove the proposition for accepting runs, we again use proof by induction, where we show that the proposition is valid at base case ($i = 0$) is valid and then as part of inductive step we assume that the proposition is valid for a sequence of fixed length $\forall i \in [l]$, then we prove that the proposition is also valid for a sequence of length $\forall i \in [l+1]$.

i) Base case (i = 0)
In the base case where i = 0, the execution in CRES is accepting, hence we have the result: $\forall e \in \mathsf{Re}.\exists j \geq 0.(e\#e_j \vee e = e_j)$
In G at the M_0, we have $\mathsf{Ex}_0 = \emptyset, \mathsf{Re}_0 = \mathsf{Re}, \mathsf{In}_0 = \mathsf{E}$ and we can imply the following

a) $\mathsf{In}_0 = \mathsf{E} \wedge \mathsf{Re}_0 = \mathsf{Re} \implies (\mathsf{Re}_0 \cap \mathsf{In}_0) = \mathsf{Re}$

b) $\forall j \geq 0.e_j\#e \implies \forall j \geq 0.e_j \rightarrow\!\% \; e \implies \forall j \geq 0.e \notin \mathsf{In}'_j$

Using above two results, we can show that
$\forall e \in \mathsf{Re}.\exists j \geq 0.(e\#e_j \vee e = e_j) \implies \forall e \in \mathsf{In}_0 \cap \mathsf{Re}_0.\exists j \geq 0.(e \notin \mathsf{In}'_j \vee e = e_j)$, which concludes that the execution in G at $i = 0$ is also accepting.

ii) Inductive step:
Since the executions in G and CRES are accepting for a finite sequence $\forall i \in [l]$, we have the following conditions satisfied.
G: $\forall i \in [l].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}'_j = (\mathsf{Ex}'_j, \mathsf{In}'_j, \mathsf{Re}'_j)$.

CRES: $\forall e \in \mathsf{E}, l \geq i \geq 0.e_i \bullet\!\!\rightarrow e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j))$
The same can be rewritten using the $\forall i \in [l]$ notation and $e_i \bullet\!\!\rightarrow = \{e \mid e_i \bullet\!\!\rightarrow e\}$ as
CRES: $\forall i \in [l].\big(\forall e \in e_i \bullet\!\!\rightarrow \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j))\big)$

In order to prove that the proposition for inductive step, we have to show that the execution in G is also accepting for sequence $\forall i \in [l+1]$, that means, formally we have to prove that
G: $\forall i \in [l+1].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}'_j = (\mathsf{Ex}'_j, \mathsf{In}'_j, \mathsf{Re}'_j)$.

But we know that execution in G is accepting for $\forall i \in [l]$, so we can rewrite the above statement as
$\forall i \in [l+1].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$

$= \forall i \in [l].\big(\forall e \in \text{In}_i \cap \text{Re}_i.\exists j \geq i.e_j = e \vee e \notin \text{In}'_j)\big) \wedge \big(\forall e \in \text{In}_{l+1} \cap \text{Re}_{l+1}.\exists j \geq l+1.e_j = e \vee e \notin \text{In}'_j)\big)$
As we know that first part of statement is satisfied (as exe is accepting for $\forall i \in [l]$), in order to prove the proposition for the inductive step, we only need to show that $\big(\forall e \in \text{In}_{l+1} \cap \text{Re}_{l+1}.\exists j \geq l+1.e_j = e \vee e \notin \text{In}'_j)\big)$ is satisfied.

i) According to definition for exclude relation ($\rightarrow\%$) in G,
$\forall j \geq 0.e_j \# e \implies \forall j \geq 0.e_j \rightarrow\% \ e.$
Further according to definition 3.3.8,
$\forall j \geq 0.e_j \rightarrow\% \ e \implies \forall j \geq 0.e \notin \text{In}'_j$ and hence
$\forall j \geq 0.e_j \# e \implies \forall j \geq 0.e \notin \text{In}'_j$

ii) Since $(\rightarrow\!+ = \emptyset) \implies (e \rightarrow\!+ = \emptyset)$
$(\text{In}_{l+1} = (\text{In}_l \cup \ e_{l+1} \rightarrow\!+) \setminus \ e_{l+1} \rightarrow\!\%) \implies ((\text{In}_{l+1} = \text{In}_l \setminus \ e_{l+1} \rightarrow\!\%)$ and further we have
$\text{Re}_{l+1} = (\text{Re}_l \setminus \{e_{l+1}\}) \cup \ e_{l+1} \bullet\!\!\rightarrow$
Since $e_{l+1} \rightarrow\% \ e_{l+1}$, we can rewrite $\text{In}_{l+1} \cap \text{Re}_{l+1}$ as
$\text{In}_{l+1} \cap \text{Re}_{l+1} = \big((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow \ \big) \setminus \ e_{l+1} \rightarrow\!\%$
Informally the set $\text{In}_{l+1} \cap \text{Re}_{l+1}$ will contain all the included responses from $\text{In}_l \cap \text{Re}_l$ and the newly added response events ($e_{l+1} \bullet\!\!\rightarrow$) and subtracted with the events excluded by e_{l+1} that is ($e_{l+1} \rightarrow\!\%$).

Since always it is the case that
$\big(((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow) \setminus \ e_{l+1} \rightarrow\!\% \ \big) \subseteq \big((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow \ \big)$, if we prove that the events in $\big((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow \ \big)$ are eventually executed or excluded, then the proposition holds for $\forall i \in [l+1]$.
Hence if we prove that the condition $\big(\forall e \in ((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow).\exists j \geq l+1.e_j = e \vee e \notin \text{In}'_j)\big)$ will be satisfied, then it will be implicitly satisfy the condition, $\big(\forall e \in \text{In}_{l+1} \cap \text{Re}_{l+1}.\exists j \geq l+1.e_j = e \vee e \notin \text{In}'_j)\big)$.

iii) Since the execution in G is accepting for $\forall i \in [l]$, we know that
$\forall i \in [l].\big(\forall e \in \text{In}_i \cap \text{Re}_i.\exists j \geq i.(e_j = e \vee e \notin \text{In}'_j)\big)$.
So in order to prove that $\big(\forall e \in ((\text{In}_l \cap \text{Re}_l) \cup \ e_{l+1} \bullet\!\!\rightarrow).\exists j \geq l+1.e_j = e \vee e \notin \text{In}'_j)\big)$ will be satisfied, we need to show that $\forall e \in e_{l+1} \bullet\!\!\rightarrow .\exists j \geq l+1.(e_j = e \vee e \notin \text{In}'_j)$.
Since the we know that the execution in CRES is accepting for sequence $\forall i \in [l+1]$, we also have the following condition satisfied.
CRES: $\forall i \in [l+1].\big(\forall e \in e_i \bullet\!\!\rightarrow \implies \exists j \geq 0.(e \# e_j \vee (i < j \wedge e = e_j))\big)$
For $i = l+1$ in the above statement, we can imply that the following statement is satisfied.
$\forall e \in e_{l+1} \implies \exists j \geq 0.(e_j \# e \vee (i < j \wedge e = e_j))$.

Using the result $\forall j \geq 0.e_j \# e \implies \forall j \geq 0.e \notin \text{In}'_j$ which is proved above

$\forall e \in e_{l+1} \implies \exists j \geq 0.(e_j \# e \vee (i < j \wedge e = e_j)) \implies \forall e \in e_{l+1} \implies \exists j \geq 0.(e \notin \mathsf{In}'_j \vee (i < j \wedge e = e_j))$
Further, $\forall e \in e_{l+1} \implies \exists j \geq 0.(e \notin \mathsf{In}'_j \vee (i < j \wedge e = e_j)) \implies \forall e \in e_{l+1} \implies \exists j \geq i.(e \notin \mathsf{In}'_j \vee (e = e_j))$, which is the desired result to prove the proposition that execution in G is accepting for $\forall i \in [l+1]$.

Since we have proven the proposition for base case and for inductive step, we can conclude that CRES has same executions that are accepting as those of G.

□

3.3.3 Distributed Dynamic Condition Response Graphs

We now define *distributed dynamic condition response graphs* by adding roles and principals as follows.

Definition 3.3.11. *A distributed dynamic condition response graph is a tuple* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *where*

1. $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{Act}, l)$ *is a dynamic condition response graph,*
2. Roles *is a set of* roles *ranged over by* r*,*
3. P *is a set of* principals *(e.g. persons or processors) ranged over by* p *and*
4. $\mathsf{as} \subseteq (\mathsf{P} \cup \mathsf{Act}) \times \mathsf{Roles}$ *is the role assignment relation to principals and actions.*

For a distributed dynamic condition response graph, the role assignment (4) relation indicates the roles (access rights) assigned to principals and which roles gives right to execute which actions. As an example, assume that $Peter \in \mathsf{P}$ and $Doctor \in \mathsf{Roles}$, then if $Peter$ as $Doctor$ and $sign$ as $Doctor$ then $Peter$ as a doctor can $sign$ as a doctor.

Now we go further and define when an event e is enabled in a distributed dynamic condition response graph by extending the definition of enabled event (def 3.3.7) in DCR Graph. For an event to be enabled in distributed dynamic condition response graph, in addition to the condition that it must have enabled in its DCR Graph, the label for the event must have been assigned to a role and also a principal must be assigned to that role.

Definition 3.3.12. *For a distributed dynamic condition response graph* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *with* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{Act}, l)$*, we define that an event* e *is enabled and write as* $\mathsf{M} \vdash_{\mathsf{DG}} e$*, if* $\mathsf{M} \vdash_{\mathsf{G}} e$*,* p as r *and* a as r*.*

The result of executing an enabled event in distributed dynamic condition response graph will have the same changes as that of executing an enabled event in dynamic condition response graph, as event execution only involves changes to the marking, which are not affected by the roles and principals of a distributed DCR Graph.

Definition 3.3.13. *For a distributed dynamic condition response graph* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *where* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{Act}, l)$*, with* $\mathsf{M} \vdash_{\mathsf{DG}} e$*, executing event* e *in* DG *will have the same effect as that of executing the event* e *in the underlying DCR Graph* G *and in both cases the resulting marking will be the same.*

Now we will define a run in distributed DCR Graph and when it is accepting in as follows,

Definition 3.3.14. *For a distributed dynamic condition response graph* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *where* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{Act}, l)$ *with marking* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$*, we define a run (finite or infinite) to be a sequence of labels* $(e_0, (p_0, a_0, r_0))(e_1, (p_1, a_1, r_1)) \ldots$ *of a sequence of transitions* $\mathsf{M}_i \xrightarrow{(e_i,(p_i,a_i,r_i))} \mathsf{M}_{i+1}$ *for* $i \geq 0$ *starting from initial marking such that* $\mathsf{M}_i \vdash_{\mathsf{DG}} e_i$ *and* $\mathsf{M}_{i+1} = \mathsf{M}_i \oplus_{\mathsf{DG}} e_i$*. We further define that a run to be* accepting *if its underlying DCR Graph* G *is accepting i.e.* $\big(\forall i \geq 0, e \in \mathsf{Re}_i. \exists j \geq i.((e = e_j \vee e \notin \mathsf{In}_{j+1}))\big)$.

Based on the semantics defined above, we can now define labelled transition system semantics for distributed dynamic condition response graph.

Definition 3.3.15. *For a distributed dynamic condition response graph* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *we define the corresponding labelled transition system* $\mathsf{TS}(\mathsf{DG})$ *to be the tuple*

$$(\mathcal{M}(\mathsf{G}), \mathsf{M}, \mathcal{L}_{ts}(\mathsf{G}), \rightarrow)$$

where $\mathcal{L}_{ts}(\mathsf{G}) = \mathsf{E} \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{Roles})$ *is the set of labels of the transition system,* M *is the initial marking, and* $\rightarrow \subseteq \mathcal{M}(\mathsf{G}) \times \mathcal{L}_{ts}(\mathsf{G}) \times \mathcal{M}(\mathsf{G})$ *is the transition relation defined by* $\mathsf{M} \xrightarrow{(e,(p,a,r)} \mathsf{M} \oplus_{\mathsf{DG}} e$ *if* $\mathsf{M} \vdash_{\mathsf{DG}} e$ *and* $\mathsf{M} \xrightarrow{(e,a)} \mathsf{M} \oplus_{\mathsf{G}} e$*. The transition system* $\mathsf{TS}(\mathsf{DG})$ *will have same states as that of the underlying dynamic condition response graph G, but with the transitions labels* $\mathsf{E} \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{Roles})$ *instead of* $\mathsf{E} \times \mathsf{Act}$*. We define a run to be (finite or infinite) sequence of labels* $(e_0, (p_0, a_0, r_0)), (e_1, (p_1, a_1, r_1)) \ldots$ *of a sequence of transitions* $\mathsf{M}_i \xrightarrow{(e_i,(p_i,a_i,r_i))} \mathsf{M}_{i+1}$ *starting from the initial marking. We define a run to be accepting if the underlying run of the DCR Graphs is accepting.*

3.3.4 Infinite runs - From DCR Graphs to Büchi-automata

In this section, we show how to characterize the acceptance condition for DCR Graphs by a mapping to the standard model of Büchi-automata. Recall that a Büchi-automaton is a finite state automaton accepting only infinite runs, and only the runs that pass through an accepting state infinitely often. Acceptance of finite runs can be represented in the standard way by introducing a special silent event, e.g. a τ-event, which may be viewed as a delay. If an infinite accepting run contains infinitely many delays it then represent an accepting run containing only a finite number of (real) events. We define a Büchi-automaton with τ-event as follows.

Definition 3.3.16. *A* Büchi-automaton with τ-event *is a tuple* $(S, s, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ *where* S *is the set of* states, $s \in S$ *is the* initial state, Ev_τ *is the set* *of* events *containing the special event* τ, $\rightarrow \subseteq S \times Ev_\tau \times S$ *is the* transition relation, *and* F *is the set of* accepting states. *A (finite or infinite) run is a sequence of labels not containing the* τ *event that can be obtained by removing all* τ *events from a sequence of labels of transitions starting from the initial state. The run is* accepting *if the sequence of transitions passes through an accepting state infinitely often.*

The mapping from DCR Graphs to Büchi-automata is not entirely trivial, since we at any given time may have several pending responses and thus must make sure that all of them are eventually executed or excluded. To make sure we progress, we assume any fixed order of the finite set of events E of the given dynamic condition response graph. For an event $e \in \mathsf{E}$ we write $rank(e)$ for its rank in that order and for a subset of events $\mathsf{E}' \subseteq \mathsf{E}$ we write $min(\mathsf{E}')$ for the event in E' with the minimal rank.

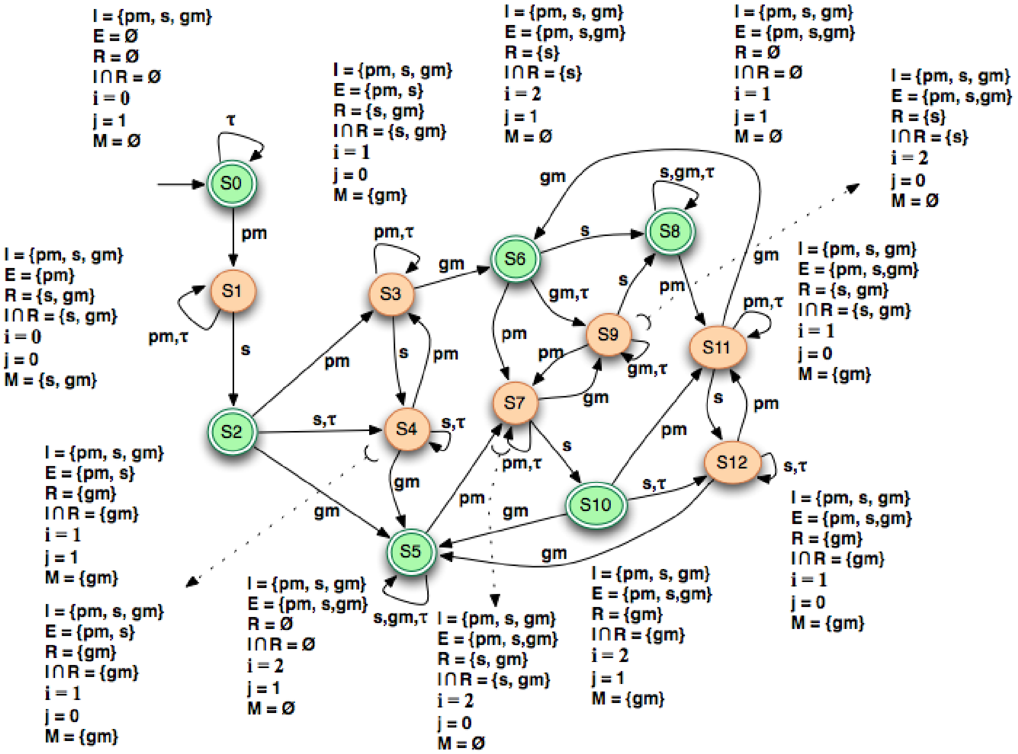


Figure 3.4: The Büchi-automaton for DCR Graph from Fig. 3.6 annotated with state information

Definition 3.3.17. *For a finite distributed dynamic condition response graph* $\mathsf{DG} = (\mathsf{G}, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ *where DCR Graph* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \Longrightarrow, l)$ *with a relation set* $\Longrightarrow\ = \{\rightarrow\bullet, \bullet\rightarrow, \pm, \rightarrow\diamond\}$, $\mathsf{E} = \{e_1, \ldots, e_n\}$ *and* $rank(e_i) = i$*, we define the corresponding Büchi-automaton with* τ*-event to be the tuple* $\mathsf{B}(\mathsf{DG}) = (S, s, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ *where*

- $S = \mathcal{M}(G) \times \{1, \ldots, n\} \times \{0, 1\}$ *is the set of states,*

- $Ev_\tau = (\mathsf{E} \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{Roles})) \cup \{\tau\}$ *is the set of events,*
- $s = (\mathsf{M}, 1, 1)$ *if* $\mathsf{In} \cap \mathsf{Re} = \emptyset$*, and* $s = (\mathsf{M}, 1, 0)$ *otherwise*
- $F = \mathcal{M}(G) \times \{1, \ldots, n\} \times \{1\}$ *is the set of accepting states and*
- $\rightarrow \subseteq S \times Ev_\tau \times S$ *is the transition relation given by two cases, (A) and (B) as follows*

(A) $(\mathsf{M}', i, j) \xrightarrow{\tau} (\mathsf{M}', i, j')$ *where*

(i) $j' = 1$ *if* $\mathsf{In}' \cap \mathsf{Re}' = \emptyset$ *otherwise* $j' = 0$.

and

(B) $(\mathsf{M}', i, j) \xrightarrow{(e,(p,a,r))} (\mathsf{M}'', i', j')$ *where*

(i) $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ *and* $\mathsf{M}'' = (\mathsf{Ex}' \cup \{e\}, \mathsf{Re}'', \mathsf{In}'')$

(ii) $\mathsf{M}' \xrightarrow{(e,(p,a,r))} \mathsf{M}''$ *is a transition of* $\mathsf{TS}(\mathsf{DG})$

(iii) $j' = 1$ *if*

(a) $\mathsf{In}'' \cap \mathsf{Re}'' = \emptyset$ *or*

(b) $min(M_r) \in (\mathsf{In}' \cap \mathsf{Re}' \backslash (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ *or*

(c) $M_r = \emptyset$ *and* $min(\mathsf{In}' \cap \mathsf{Re}') \in (\mathsf{In}' \cap \mathsf{Re}' \backslash (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$

otherwise $j' = 0$.

(iv) $i' = rank(min(M_r))$ *if* $min(M_r) \in (\mathsf{In}' \cap \mathsf{Re}' \backslash (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ *or else*

(v) $i' = rank(min(\mathsf{In}' \cap \mathsf{Re}'))$ *if* $M_r = \emptyset$ *and* $min(\mathsf{In}' \cap \mathsf{Re}') \in (\mathsf{In}' \cap \mathsf{Re}' \backslash (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ *or else*

(vi) $i' = i$ *otherwise.*

for $M_r = \{e \in \mathsf{In}' \cap \mathsf{Re}' \mid rank(e) > i\}$.

The index i is used to make sure that no event stays forever included and in the pending response set without being executed. Finally, the flag j indicates if the state is accepting or not.

Condition (Ai and Biii) defines when a state is accepting. Either there are no included pending responses in the resulting state (Ai) or the included pending response with the minimal rank above the index i was either excluded or executed (B(iii)b). Alternatively, if the set of included pending responses with rank above the index i is empty and the included pending response with the minimal rank is excluded or executed (B(iii)c), then also the resulting state will be accepting. Condition (Biv) records the new rank if the resulting state is accepting according to condition (B(iii)b) and similarly when the state is accepting according to condition (B(iii)c), the condition (Bv) records the new rank.

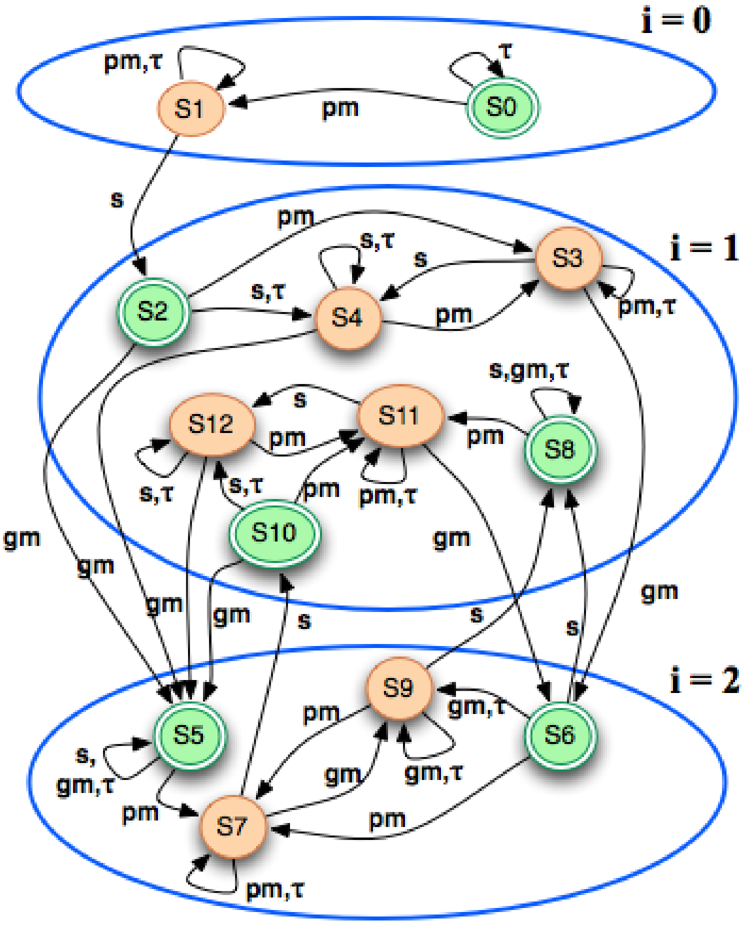


Figure 3.5: The Büchi-automaton with stratified view

To give a simple example of the mapping, let us consider the dynamic condition response graph in Fig. 3.6 and the corresponding Büchi-automaton in Fig. 3.4.

The key point to note is that the automaton enters an accepting state if there is no pending responses, or if the pending response which is the minimal ranked event according to the index i is executed or excluded. State $S7$ and $S11$ illustrate the use of the rank: Both states have the two events **s** (having rank 1) and **gm** as pending responses. In state $S7$ only executing event **s** leads to an accepting state ($S10$). The result of executing event **gm** is to move to state $S9$ which is not accepting. Dually, in state $S11$ only executing event **gm** leads to an accepting state ($S16$). The result of executing event **s** is to move to state $S12$ which is not accepting.

Fig. 3.5 shows a stratified view of the automaton, dividing the state sets according to the rank i in order to emphasize the role of the rank in guaranteeing progress.

3.4 DCR Graphs - Graphical Notation

After introducing the semantics for the DCR Graphs in the previous sections, we are now ready to introduce graphical notation for DCR Graphs with help of small workflow examples from healthcare domain. The examples are from a small oncology healthcare workflow previously identified during a field study at Danish hospitals [Lyng *et al.* 2008].

Let us first consider a small example shown in 3.6 modeled using DCR Graphs. It contains three events: **prescribe medicine** (the doctor calculates and writes the dose for the medicine), **sign** (the doctor certifies the correctness of the calculations) and **give medicine** (the nurse administers medicine to patient). The events are also labelled by the assigned roles (**D** for Doctor and **N** for Nurse).

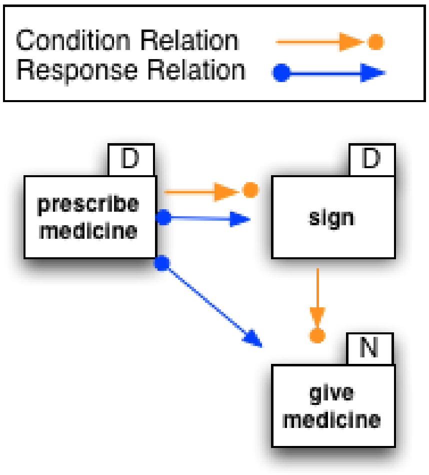


Figure 3.6: Give Medicine Example

The events **prescribe medicine** and **sign** are related by both the condition relation (→•) and the response relation (•→). The condition relation means that the **prescribe medicine** event must happen at least once before the **sign** event. The response relation enforces that, if the **prescribe medicine** event happen, subsequently at some point the **sign** event must happen for the flow to be accepted. Similarly, the response relation between **prescribe medicine** and **give medicine** enforces that, if the **prescribe medicine** event happen, subsequently at some point the **give medicine** event must happen for the flow to be accepted. Finally, the condition relation between **sign** and **give medicine** enforces that the signature event must have happened before the medicine can be given. Note the nurse can give medicine many times, and that the doctor can at any point choose to prescribe new medicine and sign again. (This will not block the nurse from continue to give medicine. The interpretation is that the nurse may have to keep giving medicine according to the previous prescription). The transition system for finite runs for the prescribe medicine example from fig 3.6 is shown in the fig. 3.7.

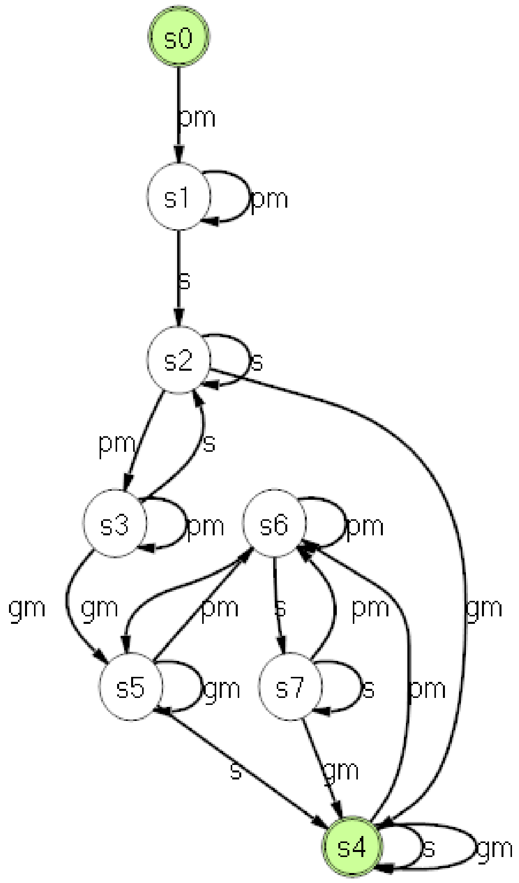


Figure 3.7: Transition system for DCR graph from fig 3.6

The dynamic inclusion and exclusion of events is illustrated by an extension to the scenario (also taken from the real case study): If the nurse distrusts the prescription by the doctor, it should be possible to indicate it, and this action should force either a new prescription followed by a new signature or just a new signature. As long the new signature has not been added, medicine must not be given to the patient.

This scenario can be modeled as shown in Fig. 3.8, where one more action **don't trust** is added. Now, the nurse have a choice to indicate distrust of prescription and thereby avoid **give medicine** until the doctor re-execute **sign** action. Executing the **don't trust** action will exclude **give medicine** and makes the **sign** as pending response. So the only way to execute **give medicine** action is to re-execute **sign** action which will then include **give medicine**. Here the doctor may choose to re-do **prescribe medicine** followed by **sign** actions (new prescription) or simply re-do **sign**.

In Fig. 3.9 below we propose a graphical notation that illustrates the run-time information during two different runs of the extended scenario in Fig. 3.8. We show the events as boxes just as in the graphical notation for the dynamic condition response graph and use three different small icons (Ø, √, !) above the boxes to show if the event is enabled (i.e. not blocked by any conditions), if it has been executed (i.e.

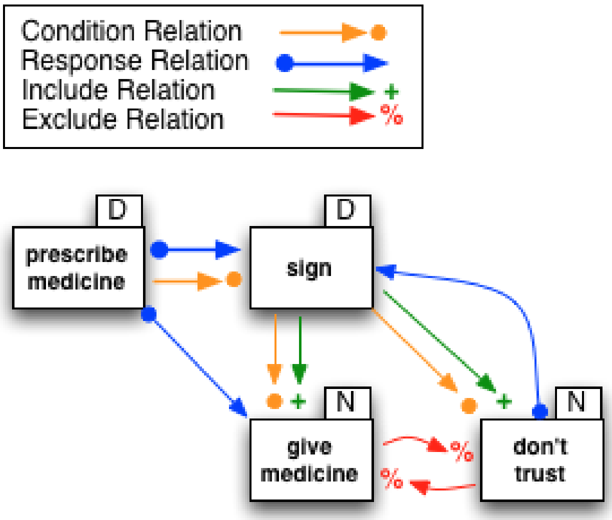


Figure 3.8: Give Medicine Example with Check

included in the set E in the marking), and if it is required as a response (i.e. included in the set R in the marking). We indicate that an event is excluded (i.e. not included in the set I in the marking) by making the box around the event dashed.

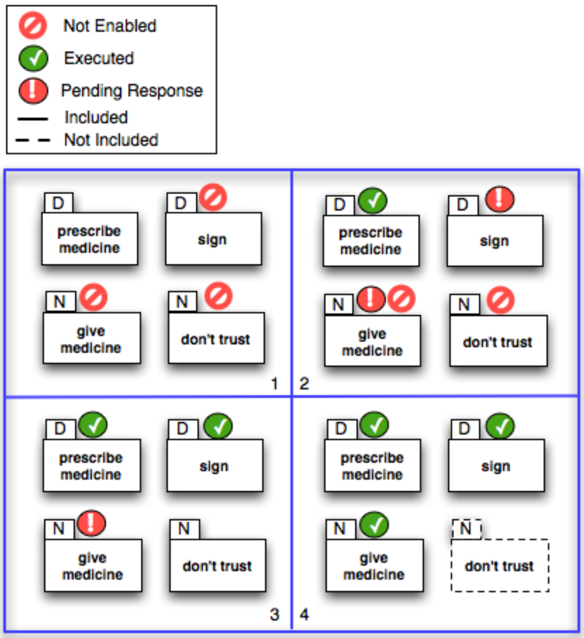


Figure 3.9: Runtime for Give Medicine Example from 3.8

Fig 3.9 shows the four states of a run in the workflow process in Fig. 3.8, starting in the initial state where all events except **prescribe medicine** is blocked. The second state is the result of executing **prescribe medicine**, now showing that **sign** and **give medicine** are required as responses and that **sign** is no longer blocked. The third state is the result of executing the **sign** event, which enables **give medicine** and **don't trust**. Finally, the fourth state is the result of executing the **give medicine** event, excluding the **don't trust** event.

Similarly, Fig. 3.10 shows the six states of a run where the nurse executes **don't trust** in the third step, leading to a different fourth state where **give medicine** is excluded (but still required as response if it gets included again) and **sign** is required as response. The fifth state shows the result of the doctor executing **sign**, which re-includes **give medicine**, which is then executed, leading to the final state where all events have been executed, and **don't trust** is excluded.

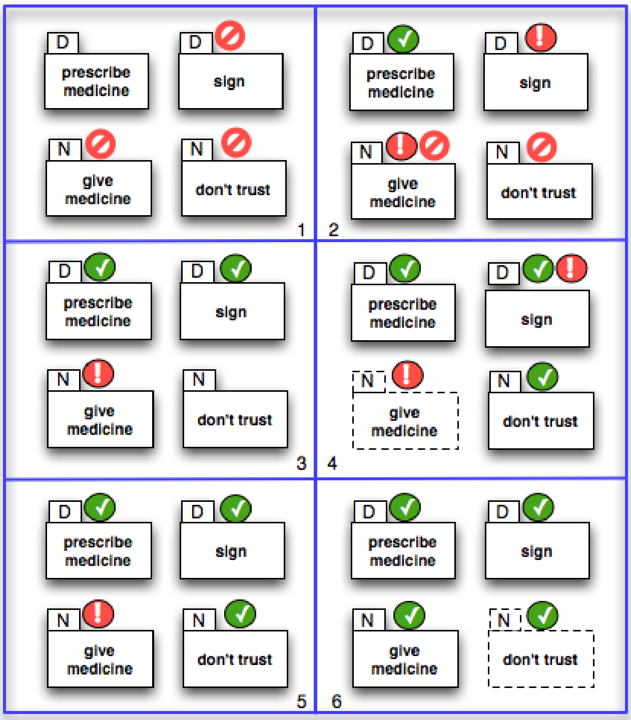


Figure 3.10: Runtime for Give Medicine Example with *Don't trust* from 3.8

In order to model milestone relation, we further extend the healthcare with two more events *receive tests*, to receive test results that were previously ordered (with doctor/nurse roles) and *examine tests*, to examine the receive test results (with doctor role) as shown in the figure 3.11. The intuition is that, if the test results have been received, then the doctor must examine those results before making a prescription. The situation is modeled with a response and condition relation between *receive tests* and *examine tests* and with a milestone relation between *examine tests* and *prescribe medicine*.

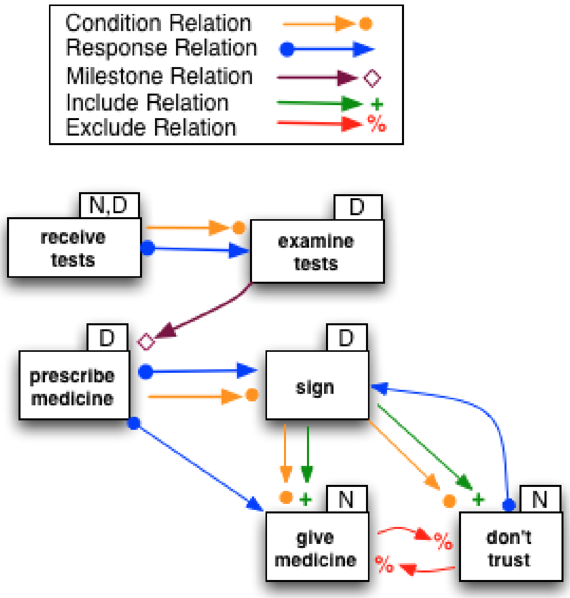


Figure 3.11: Extended Give Medicine Example with milestone relation

Receiving test results (may be in between the prescriptions) will create a pending response on the *examine tests* event and the *prescribe medicine* event will be blocked because of the milestone relation between *examine tests* and *prescribe medicine*, until the doctor examine the results. One may argue that a condition and response relation could have been be used instead of milestone relation in between *examine tests* and *prescribe medicine*, but in that case the *prescribe medicine* will be blocked only for the first time (due to condition relation) and on top of that, the doctor will be compelled to do *prescribe medicine* (due to response relation) afterwards, which may not be necessary e.g. when the test results are good.

Finally, we will show how the above healthcare example can be expressed in the formal definitions of distributed DCR Graphs (def 3.3.11) in the listing 3.1.

Listing 3.1: Formal representation of healthcare example in DCR Graphs

```
Distributed DCR graph DG = (G, Roles, P, as) where

G = (E, M, Act, ⟹, l)

E = Act ={receive tests, examine tests, prescribe medicine, sign, give
    medicine, don't trust}

M = (∅, ∅, E)

l = {(receive tests, receive tests), (examine tests, examine tests), . . .
    . }
```

```
⟹ = {→•, •→, ±, →◇} where

→• ={(receive tests, examine tests), (prescribe medicine, sign), (sign,
     give medicine), (sign, don't trust)}

•→ ={(receive tests, examine tests), (prescribe medicine, sign), (
    prescribe medicine, give medicine), (don't trust, sign)}

→+ ={(sign, give medicine), (sign, don't trust)}

→% ={(don't trust, give medicine), (give medicine, don't trust)}

→◇ ={(examine tests, prescribe medicine)}

Roles ={D, N}

P ={Peter, Rosy}

as ={(Peter, D), (Rosy, N), (receive tests, N), (receive tests, D), (
    examine tests, D), (prescribe medicine, D), (sign, D), (give
    medicine, N), (don't trust, N)}
  .
```

3.5 Expressiveness of DCR Graphs

In this section, we will discuss about expressiveness of DCR Graphs. Despite the simplicity of the the model with five relations, the DCR Graphs model can express all ω-regular languages. In order to show that, we will encode Büchi automaton into DCR Graphs and show that both büchi automaton and DCR Graphs will have same runs and accepting runs.

We will first revisit the definition of non-deterministic Büchi automaton and provide a method to encode it directly into DCR Graphs. Later we will show that Büchi automaton is bisimilar to the encoded DCR Graph by providing a suitable proof by using bisimulation.

3.5.1 Büchi Automaton

In this section we will revisit the definition of nondeterministic Büchi automaton and the language accepted by it.

Definition 3.5.1. *A* nondeterministic Büchi-automaton *is a tuple* $\mathrm{B} = (S, S_0, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ *where*

1. *S is the set of* states *ranged over by s,*

2. *$S_0 \subseteq S$ is the set of* initial states*,*

3. *Ev_τ is an alphabet (the set of names) ranged over by a,*

4. *$\rightarrow \subseteq S \times Ev_\tau \times S$ is the* transition relation*, and*

5. *F is the set of* final *or* accepting *states.*

A run for an infinite word $\sigma = a_0, a_1, a_2, \ldots \in Ev_\tau^\omega$ is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 \in S_0$ and $s_i \xrightarrow{a_i} s_{i+1}$ for $i \geq 0$. A run $s_0, s_1, s_2, \ldots$ is accepting if $s_i \in F$ for $i \geq 0$ infinitely often.

Now we will define the labeled transition relation for Büchi-automaton (B) as follows,

Definition 3.5.2. *For a Büchi automaton* B $= (S, S_0, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$*, we define the corresponding the corresponding labelled transition system* TS(B) *to be a tuple*

$$(\mathcal{P}(S), S_0, \rightarrow_B \subseteq \mathcal{P}(S) \times Ev_\tau \times \mathcal{P}(S), F_N)$$

where S_0 is the set of initial states in B*, $F_N = \{S' \mid S' \in \mathcal{P}(S) \land S' \cap F \neq \emptyset\}$ is the set of final states and the transition relation defined by $\rightarrow_B = \{(S', a, \{s\}) \mid \exists s' \in S'.(s', a, s) \in \rightarrow\}$*

Further we define a run of the transition system $a_0, a_1, a_2, \ldots \in Ev_\tau^\omega$ to be an infinite sequence of labels of transitions $S_i \xrightarrow{(a_i)}_B \{s_{i+1}\}$ starting from the initial state S_0 and a run is accepting if $s_i \in F_N$ infinitely often.

On can observe that the transitions in the labeled transition relation for Büchi automaton TS(B) are from set of states to a singleton state ($S_i \xrightarrow{(a_i)}_B \{s_{i+1}\}$), instead of just from one state to other ($s_i \xrightarrow{(a_i)}_B s_{i+1}$) like in any other labeled transition system. The start state in non-deterministic Büchi automaton is not just one state, but a set of states, and in order to cover the transitions of start state also, we have defined the transitions in TS(B) to be from set of states to a singleton state.

Now we will define the mapping from Büchi automaton to DCR Graphs in the definition 3.5.3. First, every state and every transition of Büchi automaton are mapped to individual events in the corresponding DCR Graph. The events corresponding to the states in Büchi automaton are neither enabled nor executable and they are used to block/unblock the events corresponding to their transitions in Büchi automaton.

In order to map the accepting states of Büchi automaton to corresponding markings in DCR Graph, we add a special event *NAS* with a self condition and with a initial pending response. The event *NAS* will always stay in the pending response set of a marking, as it will never get executed, but by excluding/including the *NAS* event, the marking can be made accepting/non-accepting.

Further, we have added all the transition events to the executed set (Ex) of initial marking in the DCR Graph, as the transition events are the only events, which are executable. The basic intuition behind this, is to have a constant executed set (Ex) in a marking, so that we will get one-to-one correspondence between the states of Büchi automaton to the markings of DCR Graph. Further we are also not interested in the history of execution as it in no way influences the execution of transition events, because all their condition events are only the state events, which will never get executed.

Similarly, the response set is also constant, containing exactly one event *NAS* all the time as the response relation (•→) in the DCR Graph is empty. The only change from marking to marking is in included set (In). We will add all the transition events to the included set of the initial marking. Further all the state events except those are part of start state are also added to included set. Finally the event *NAS* will be added if none of the start states of Büchi automaton are also final states.

Response (•→) and milestone (→⋄) relations are empty in the encoding, where as the condition relation (→•) contains all blocking condition relations. All state events and *NAS* event will contain a self condition relation where as all transition events will have their corresponding state event as a condition. Finally include (→+) and exclude (→%) relations contains mappings to include or exclude a state event or *NAS* event. If a transition (e.g (s, a, s')) in Büchi automaton leading to next state, then the corresponding transition event (e(s,a,s')) will exclude the state event for the leading state (to enables the transitions at the leading state) and also include the state event for the leaving state (to block all transitions at the leaving state). Similarly, the *NAS* event will be excluded if the leading state is one of the accepting states in Büchi automaton, on the other hand if the leading state is not one of the final states, then it will include *NAS*.

Finally, we will add all the transition labels (Ev_τ) to the actions set and the labeling function will contain mapping between transition label (a) and event name (e(s, a, s')). We will not add any labels either for a state event or the *NAS* event, as they will not be executed at any time.

The formal definition of encoding Büchi automaton to a DCR Graph is given below in def 3.5.3.

Definition 3.5.3. *For a Büchi automaton* $\mathsf{B} = (S, S_0, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$*, we define the corresponding DCR Graph to be* $\mathsf{G(B)} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{Act}, l)$ *where*

1. $\mathsf{E} = (\mathsf{E}_s \cup \mathsf{E}_\tau) \uplus \{NAS\}$ *such that*
   - $\mathsf{E}_s = \{e(s) \mid s \in S\}$ *is the set of events for the states in Büchi automaton,*
   - $\mathsf{E}_\tau = \{e(s, a, s') \mid (s, a, s') \in \rightarrow\}$ *is event set for transitions in Büchi automaton,*
2. $\mathsf{M} = \mathsf{M}_B(S_0)$ *where* $\mathsf{M}_B : \mathcal{P}(S) \rightarrow \mathcal{M}(\mathsf{G(B)})$ *and defined as* $\mathsf{M}_B(S') = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *such that*
   - $\mathsf{Ex} = \mathsf{E}_\tau$, $\mathsf{Re} = \{NAS\}$
   - $\mathsf{In} = \begin{cases} \left(\mathsf{E}_\tau \cup \mathsf{E}_s \setminus \{e(s) \mid s \in S'\}\right) \cup \{NAS\} & if\ S' \cap F = \emptyset \\ \mathsf{E}_\tau \cup \mathsf{E}_s \setminus \{e(s) \mid s \in S'\} & if\ S' \cap F \neq \emptyset \end{cases}$
3. $\bullet\!\rightarrow = \emptyset$ *and* $\rightarrow\!\diamond = \emptyset$

4. $\rightarrow\!\bullet = \{(NAS, NAS)\} \cup \{\big(e(s), e(s)\big) \mid s \in S\} \cup$
$\{\big(e(s), e(s, a, s')\big) \mid (s, a, s') \in \rightarrow\}$

5. $\rightarrow\!+ = \{\big(e(s, a, s'), e(s'')\big) \mid e(s, a, s') \in \rightarrow \wedge\ s'' \neq s'\} \cup$
$\{\big(e(s, a, s'), NAS)\big) \mid (s, a, s') \in \rightarrow \wedge\ \ s' \notin F\}$

6. $\rightarrow\!\% = \{\big(e(s, a, s'), e(s')\big) \mid e(s, a, s') \in \rightarrow \wedge\ s \neq s'\} \cup$
$\{\big(e(s, a, s'), NAS)\big) \mid (s, a, s') \in \rightarrow \wedge\ \ s' \in F\}$

7. Act $= Ev_\tau$

8. $l = \{\big(e(s, a, s'), a)\big) \mid e(s, a, s') \in \rightarrow\}$

3.5.2 Encoding of Büchi Automaton into DCR Graphs - Example

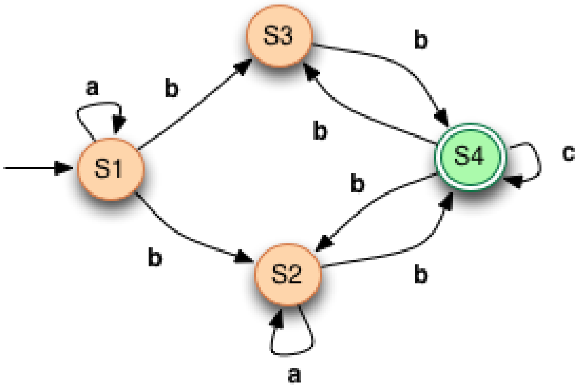


Figure 3.12: Büchi-automaton Example

In this section, we will explain the construction of DCR Graph from the Büchi automaton, by taking an example. Let's take a small example of Büchi automaton as shown in the figure 3.12. The automaton contains 4 states with one initial state (s_1) and one final state (s_4) marked with green color. The automaton shown in the example is nondeterministic, as we can observe nondeterministic transitions at states s_1 and s_4. The automaton will only accepting if in a run, the final state s_4 is visited infinitely often.

The encoded DCR Graph for the Büchi automaton shown in figure 3.12 is shown in the figure 3.13. In the construction, first we will add events (let's call state events) for all the states in the Büchi automaton (B) and they are named after their respective states in the encoded DCR Graph (G(B)). For example, the state s_1 in B, we will get an event $e(s_1)$ in the G(B). All state events will have self condition relation as shown in the figure 3.13, which make them never enabled and executable. The state events (e.g s_1) for the states which are part of start state in B, will be excluded in the initial marking.

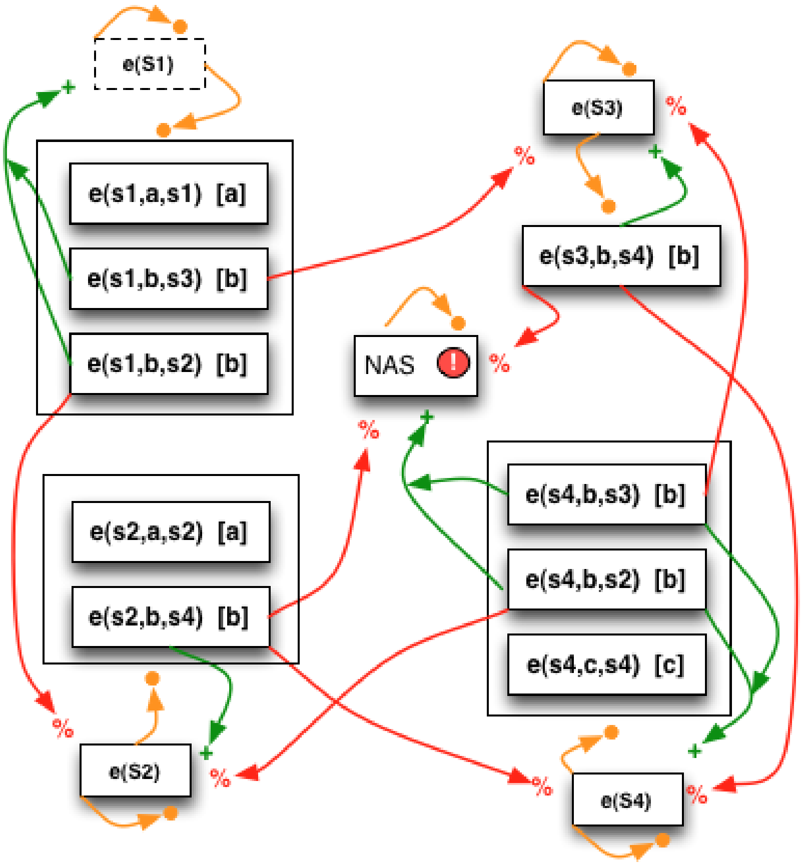


Figure 3.13: DCR Graphfor Büchi-automaton in figure 3.12

Similarly, all the transitions in B will a get event in G(B) (let's call them transition events) and they are named after their respective transitions. For example if we have a transition (s_1, b, s_3) in B, then in the encoded DCR Graph, we will get an event $e((s_1, b, s_3))$. In this way, for non-deterministic transitions in B, we will get deterministic events in G(B) with labels mapped exactly to the transitions in B. For example, we have non-deterministic transition b at state s_1 in B, which will be encoded with events $e((s_1, b, s_3)), e((s_1, b, s_2))$ with their labels as $l(e((s_1, b, s_3))) = b, l(e((s_1, b, s_2))) = b$ respectively. Further all transition events will have their respective state events as conditions. As explained before, the state events can not be executed and hence they act as blocker events, blocking the their transition events and only when a state is excluded in a marking, then all their transition events will be enabled.

Further, for all the transitions that lead to one of the final states in B, the corresponding transition events will exclude the NSA event, to make the resulting marking accepting, on the other hand the transition events will include NSA event if they are leading to a state that is not part of final state. Similarly for the transitions leading to an another state (e.g. b at s_3) in B, their respective transition event $(e(s_3, b, s_4))$

will exclude the leading state event (s_4) in order to enable the transition events at the leading state and include the leaving state event (s_3) to make the transition events blocked at the leaving state.

Finally, we have excluded some of the un-important relations (e.g. include relation from $e(s_1, b, s_3)$ to NSA) in the figure 3.12 to make it more readable. Also, we have used a shorthand notation for the condition relation from state events to their transition events by using a box around transition events and making the condition relation pointing to the box, meaning that the condition relation applies to all the transition events inside the box.

3.5.3 Bisimulation between Büchi and DCR Graph

First we will define the relation between Büchi automaton B and its corresponding DCR Graph G(B) as follows,

Definition 3.5.4. *For a labeled transition system for Büchi automaton* $\mathrm{TS(B)} = (\mathcal{P}(S), S_0, \rightarrow_B \subseteq \mathcal{P}(S) \times Ev_\tau \times \mathcal{P}(S), F_N)$ *where* $\mathrm{B} = (S, S_0, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ *and it's the corresponding DCR Graph to be* $\mathrm{G(B)} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow+, \rightarrow\%, \mathrm{Act}, l)$*, we define the binary relation over* TS(B) *and* TS(G) *as* $\mathcal{R} = \{(\mathcal{S}, \mathrm{M}) \mid \mathcal{S} \in \mathcal{P}(S) \wedge \mathrm{M} = \mathrm{M}_B(\mathcal{S})\}$.

The binary relation $\mathcal{R}$ contains pairs of state form TS(B) and its corresponding marking from G(B), as defined by the function in the encoding.

Proposition 3.5.1. *The labeled transition system* TS(B) *for a Büchi automaton is bisimilar to the labeled transition system* TS(G) *for corresponding DCR Graph.*

Proof. For Büchi automaton $\mathrm{B} = (S, S_0, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$, the labeled transition system (def 3.5.2) is $\mathrm{TS(B)} = (\mathcal{P}(S), S_0, \rightarrow_N \subseteq \mathcal{P}(S) \times Ev_\tau \times \mathcal{P}(S), F_N)$.

For DCR Graph $\mathrm{G} = (\mathrm{E}, \mathrm{M}_0, \mathrm{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\!\diamond, \mathrm{l})$, the corresponding labeled transition system (def 3.3.10) is $\mathrm{TS(G)} = (\mathcal{M}(\mathrm{G}), \mathrm{M}, \mathcal{L}_{ts}(\mathrm{G}), \rightarrow)$ where $\mathcal{L}_{ts}(\mathrm{G}) = \mathrm{E} \times \mathrm{Act}$ is the set of labels of the transition system, M_0 is the initial marking, and $\rightarrow \subseteq \mathcal{M}(\mathrm{G}) \times \mathcal{L}_{ts}(\mathrm{G}) \times \mathcal{M}(\mathrm{G})$ is the transition relation defined by $\mathrm{M} \xrightarrow{(e,a)} \mathrm{M} \oplus_{\mathrm{G}} e$ if $\mathrm{M} \vdash_{\mathrm{G}} e$ and $a \in \mathrm{l}(e)$.

According to def 3.5.4, we have the binary relation $\mathcal{R} = \{(\mathcal{S}, \mathrm{M}) \mid \mathcal{S} \in \mathcal{P}(S) \wedge \mathrm{M} = \mathrm{M}_B(\mathcal{S})\}$ over TS(B) and TS(G). In order to show that TS(B) $\sim$ TS(G), we have to show that

- if $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ in TS(B) then there exists in TS(G) a transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$
- if $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ in TS(G) then there exists in TS(B) a transition $\mathcal{S} \xrightarrow{a} \mathcal{S}'$

We will prove the two directions individually as follows,

(A) If $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ in TS(B) then there exists in TS(G) a transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$

According def 3.5.4, from the binary relation $\mathcal{R}$, we have the corresponding marking M in TS(G) for the state $\mathcal{S}$ from TS(B).

But the state $\mathcal{S} \in \mathcal{P}(S)$ in TS(B) is a set of states in B and hence for $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ in TS(B), there will be a set of transitions $A = \{a \mid (s, a, s') \in\rightarrow \wedge s \in \mathcal{S}\}$ in B.

In order to prove the equivalence we have to show that $\forall a \in A.\exists \mathrm{M} \xrightarrow{(e,a)} \mathrm{M}'$ in TS(G) for some event e and label a.

According to definition for encoding (def 3.5.3), for a transition $s \xrightarrow{a} s'$ in B, the corresponding event in G is $e(s, a, s')$.

According to def 3.3.10 for labeled transition system for DCR Graph, in order to have a transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ in TS(G), we need to show that $\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s')$ and $\mathrm{l}(e(s, a, s')) = a$.

According to definition of labeling function in encoding (def 3.5.3), we already have $\forall a \in A.\mathrm{l}(e(s, a, s')) = a$, hence we only need to show $\forall a \in A.\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s')$.

we can rewrite $\forall a \in A.\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s') = \forall s \in \mathcal{S}.\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s')$ as $e(s, a, s')$ is the corresponding event for $(s, a, s') \in\rightarrow$.

From the encoding definition (def 3.5.3), the marking for state (S) will be $\mathrm{M} = \mathrm{M}_B((S)) = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$ such that

- $\mathrm{Ex} = \mathrm{E}_\tau$, $\mathrm{Re} = \{NAS\}$
- $\mathrm{In} = \begin{cases} \big(\mathrm{E}_\tau \cup \mathrm{E}_s \setminus \{e(s) \mid s \in S'\}\big) \cup \{NAS\} & if\ S' \cap F = \emptyset \\ \mathrm{E}_\tau \cup \mathrm{E}_s \setminus \{e(s) \mid s \in S'\} & if\ S' \cap F \neq \emptyset \end{cases}$

From def 3.3.7, to show that $\forall s \in \mathcal{S}.\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s')$, we need to show that $\forall s \in \mathcal{S}.\big(e(s, a, s') \in \mathrm{In}\big) \wedge \big(\rightarrow\!\diamond e(s, a, s') \cap \mathrm{In} \in \mathrm{E} \setminus \mathrm{Re}\big) \wedge \big(\rightarrow\!\bullet e(s, a, s') \cap \mathrm{In} \in \mathrm{Ex}\big)$ holds.

(i) $\forall s \in \mathcal{S}.e(s, a, s') \in \mathrm{In}$
Since $e(s, a, s') \in \mathrm{E}_\tau \subseteq \mathrm{In}_0$ where In_0 is the included set of initial marking and $\rightarrow\!\% e(s, a, s') = \emptyset$, all the events in E_τ are included in all markings.
Hence we can conclude that $\forall s \in \mathcal{S}.e(s, a, s') \in \mathrm{In}$ holds.

(ii) $\forall s \in \mathcal{S}. \rightarrow\!\diamond e(s, a, s') \cap \mathrm{In} \in \mathrm{E} \setminus \mathrm{Re}$
$(\rightarrow\!\diamond\ = \emptyset) \implies (\rightarrow\!\diamond e(s, a, s') = \emptyset)$.
Hence we can conclude that $\forall s \in \mathcal{S}. \rightarrow\!\diamond e(s, a, s') \cap \mathrm{In} \in \mathrm{E} \setminus \mathrm{Re}$ holds.

(iii) $\forall s \in \mathcal{S}.\ \rightarrow\!\bullet\, e(s, a, s') \cap \mathrm{In} \in \mathrm{Ex}$

From the definition of condition relation in the encoding (def 3.5.3), $\rightarrow\!\bullet\ e(s, a, s') = e(s)$, i.e the only condition events for a transition event $e(s, a, s')$ is its state event $e(s)$.

From the marking (M) for the state $\mathcal{S}$, we can observe that, the events $\{e(s) \mid s \in S'\}$ not included in the included set (In). Hence

$$\big(\{e(s) \mid s \in S'\} \cap \mathrm{In} = \emptyset\big) \implies \forall s \in \mathcal{S}.\ \rightarrow\!\bullet\, e(s, a, s') \cap \mathrm{In} = \emptyset.$$

Therefore we can conclude that $\forall s \in \mathcal{S}.\ \rightarrow\!\bullet\, e(s, a, s') \cap \mathrm{In} \in \mathrm{Ex}$ holds.

From (i), (ii) and (iii), we can conclude that $\forall s \in \mathcal{S}.\big(e(s, a, s') \in \mathrm{In}\big) \wedge \big(\rightarrow\!\diamond\ e(s, a, s') \cap \mathrm{In} \in \mathrm{E} \setminus \mathrm{Re}\big) \wedge \big(\rightarrow\!\bullet\, e(s, a, s') \cap \mathrm{In} \in \mathrm{Ex}\big)$ holds.

Therefore our proposition that If $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ in TS(B) then there exists in TS(G) a transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ is valid.

(B) if $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ in TS(G) then there exists in TS(B) a transition $\mathcal{S} \xrightarrow{a} \mathcal{S}'$

According def 3.5.4, from the binary relation $\mathcal{R}$, we have $\mathcal{S}$ in TS(B) from corresponding marking M in TS(G).

At marking M in TS(G), the transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ implies that there are set of enabled transitions such that $A = \{a \mid \mathrm{M} \xrightarrow{(e,a)} \wedge\ \mathrm{l}(e) = a\}$ for some event e and label a.

In order to prove the equivalence we have to show that $\forall a \in A.\exists s \xrightarrow{a} s' \mid s \in \mathcal{S} \wedge (s, a, s') \in\rightarrow$ in TS(G).

Let's first compute set of all enabled transitions at marking M. According to def 3.3.10 for labeled transition system for DCR Graph, if we have a transition $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ in TS(G), it indicates that $\mathrm{M} \vdash_{\mathrm{G}} e$ and $\mathrm{l}(e) = a$ for some event e and label a.

Further, from def 3.3.7, $\mathrm{M} \vdash_{\mathrm{G}} e$ in G for some event e, indicates that $\big(e \in \mathrm{In}\big) \wedge \big(\rightarrow\!\diamond e \cap \mathrm{In} \in \mathrm{E} \setminus \mathrm{Re}\big) \wedge \big(\rightarrow\!\bullet\, e \cap \mathrm{In} \in \mathrm{Ex}\big)$.

According to definition of encoding (def 3.5.3), the marking $\mathrm{M} = \mathrm{M}_{B}((S)) = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$ such that

- $\mathrm{Ex} = \mathrm{E}_{\tau}$, $\mathrm{Re} = \{NAS\}$
- $\mathrm{In} = \begin{cases} \big(\mathrm{E}_{\tau} \cup \mathrm{E}_{s} \setminus \{e(s) \mid s \in S'\}\big) \cup \{NAS\} & \textit{if } S' \cap F = \emptyset \\ \mathrm{E}_{\tau} \cup \mathrm{E}_{s} \setminus \{e(s) \mid s \in S'\} & \textit{if } S' \cap F \neq \emptyset \end{cases}$

From the above marking, we will compute the set of enabled events $E_e = \{e \mid \mathsf{M} \vdash_{\mathsf{G}} e\}$ and from E_e, we compute the enabled transition by taking label corresponding to those events from labeling function l.

First let us say that $E_e = \emptyset$ and we start filling the set E_e, as we go through the conditions for enabled event one by one as listed below.

(i) $\mathsf{M} \vdash_{\mathsf{G}} e \implies e \in \mathsf{In}$
From the marking M given above, the events currently included in the In set are $\big(\mathsf{E}_\tau \cup \mathsf{E}_s \setminus \{e(s) \mid s \in S'\}\big) \cup \{NAS\}$. So let's say that
$E_e = \big(\mathsf{E}_\tau \cup \mathsf{E}_s \setminus \{e(s) \mid s \in S'\}\big) \cup \{NAS\}$.

(i) $\mathsf{M} \vdash_{\mathsf{G}} e \implies \big(\rightarrow\!\diamond\, e \cap \mathsf{In} \in \mathsf{E} \setminus \mathsf{Re}\big)$
Since $\rightarrow\!\diamond\; = \emptyset$, it does not affect the E_e, therefore it will still be
$E_e = \big(\mathsf{E}_\tau \cup \mathsf{E}_s \setminus \{e(s) \mid s \in S'\}\big) \cup \{NAS\}$.

(i) $\mathsf{M} \vdash_{\mathsf{G}} e \implies \big(\rightarrow\!\bullet\, e \cap \mathsf{In} \in \mathsf{Ex}\big)$
For an enabled event, all its conditions must be in executed (Ex) set.

From the definition of condition relation in the encoding (def 3.5.3),
$\rightarrow\!\bullet\, NAS = \{NSA\} \wedge NSA \notin \mathsf{Ex} \implies NSA \notin E_e$

Similarly $\forall e(s) \in \mathsf{E}_s.\ \rightarrow\!\bullet\, e(s) = \{e(s)\} \wedge e(s) \notin \mathsf{Ex} \implies e(s) \notin E_e$.

After updating E_e for the above the two results, we have $E_e = \mathsf{E}_\tau$.

From the definition of condition relation in the encoding (def 3.5.3),
$\rightarrow\!\bullet\, e(s, a, s') = \{e(s)\}$. The state events $e(s)$ will never get executed as they have self condition ($\rightarrow\!\bullet\, e(s) = \{e(s)\}$), but the only way they can unblock the transition events is by not included in the marking.

From the included set In and executed set Ex in marking M, the set of transition events which are not enabled because their condition events are not in executed set, but included in the marking, $\forall e(s, a, s') \in \mathsf{E}_\tau.\ \rightarrow\!\bullet\, e(s, a, s') \cap \mathsf{In} \nsubseteq \mathsf{Ex} = \mathsf{E}_\tau \setminus \{e(s, a, s') \mid s \in \mathcal{S}\}$.
In order to get the actual enabled events at marking M, we have to remove all these not enabled transition events from the E_e,

$E_e = \mathsf{E}_\tau \setminus \big(\mathsf{E}_\tau \setminus \{e(s, a, s') \mid s \in \mathcal{S}\}\big)$.
$E_e = \{e(s, a, s') \mid s \in \mathcal{S}\}$.

Finally the set of enabled events at M is $\{e(s, a, s') \mid s \in \mathcal{S}\}$ and we have labels for each of these events in the labeling function.

Hence at marking M, we have $\forall s \in \mathcal{S}.\mathsf{M} \vdash_{\mathsf{G}} e(s, a, s') \wedge \mathsf{l}(e(s, a, s')) = a$.

From the encoding definition (def 3.5.3), if we have a transition a in B such that $\{a \mid (s, a, s') \in\rightarrow\}$, then we have a transition event $e(s, a, s')$ with its label $\mathrm{l}(e(s, a, s')) = a$ in G(B).

Therefore $\forall s \in \mathcal{S}.\big(\mathrm{M} \vdash_{\mathrm{G}} e(s, a, s') \wedge \mathrm{l}(e(s, a, s'))\big) = a \implies \exists.\{a \mid (s, a, s') \in\rightarrow \wedge\, s \in \mathcal{S}\}$.

Hence we can conclude that if $\mathrm{M} \xrightarrow{a} \mathrm{M}'$ in TS(G) then there exists in TS(B) a transition $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ holds.

Finally, since we have proved the proposition in both directions, we can conclude that TS(G) and TS(B) are bisimilar, that means TS(G) $\sim$ TS(B).

□

Theorem 3.5.1. *A Büchi automaton* $\mathrm{B} = (S, S_0, Ev_\tau, \rightarrow\subseteq S \times Ev_\tau \times S, F)$ *and its corresponding DCR Graph* $\mathrm{G(B)} = (\mathrm{E}, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathrm{Act}, l)$. *will have same runs and accepting runs.*

Proof. In the proposition 3.5.1, we have proved that the labeled transition system (TS(B)) for Büchi automaton B is bisimilar to labeled transition system (TS(G)) for its corresponding DCR Graph G(B).

Since TS(G) $\sim$ TS(B), we have same transitions and choices at every corresponding state in B and marking in G(B), therefore we can conclude that both B and G(B) will have same runs.

Then, we have to prove that both B and G(B) will have same *accepting* runs.

To prove this, we have to show that accepting runs are same in both directions, and the proof is divided into 2 parts follows,

(A) If a run is accepting in B then it is also accepting in G(B).

From definition for Büchi automaton (def 3.5.1), a run $s_0, s_1, s_2, \ldots$ is accepting if $s_i \in F$ for $i \geq 0$ infinitely often.

In the above run for B, let's us say that, $s_i \xrightarrow{a} s_{i+1}$ is the transition that is visiting a state in F, that means $s_{i+1} \in F$.

Since we have the same runs, for the $s_i \xrightarrow{a} s_{i+1}$ in B, we have a corresponding marking in G(B) such that $\mathrm{M}_i \xrightarrow{(e(s_i, a, s_{i+1}), a)} \mathrm{M}_{i+1}$.

From the definition of exclude relation in the encoding (def 3.5.3), we have
$\forall e(s, a, s') \in\rightarrow \wedge\ \ s' \in F.\ e(s, a, s')\rightarrow\% = \{NAS\}$.

Hence we have $e(s_i, a, s_{i+1})\rightarrow\% = \{NAS\}$ as $s_{i+1} \in F$.

When $\mathsf{M}_i \xrightarrow{(e(s_i,a,s_{i+1}),a)} \mathsf{M}_{i+1}$, it will exclude $\{NAS\}$ from the marking M_{i+1}.

From the definition of the encoding (def 3.5.3), we have $\bullet\!\!\rightarrow = \emptyset$ and $\{NAS\}$ is the only event with pending response.

Therefore in the marking M_{i+1}, the $\mathsf{In} \cap \mathsf{Re} = \emptyset$, making the marking M_{i+1} accepting.

For any transition in B that is visiting the one of the final states, the corresponding marking in G(B) will be accepting as we have exclude relation $\forall e(s, a, s') \in\rightarrow \wedge\ \ s' \in F.\ e(s, a, s')\rightarrow\% = \{NAS\}$.

Since the run in B is accepting, it will visit one of the final states infinitely often, thereby the marking in G(B) will be visiting the state where there are no included pending responses, in other words there is no pending response event in G(B) that stays for ever without being executed or excluded. Therefore the run in G(B) is accepting.

Therefore we can conclude that If a run is accepting in B then it is also accepting in G(B).

(B) If a run is accepting in G(B) then it is also accepting in B.

According to def 3.3.9, a run is DCR Graph is accepting if $\forall i \in [k].\big(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)\big)$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}'_j = (\mathsf{Ex}'_j, \mathsf{In}'_j, \mathsf{Re}'_j)$.

Informally a run in G(B) is accepting if there is no pending response that stays for ever without being executed or excluded.

From the definition of the encoding (def 3.5.3), in G(B), we have $\bullet\!\!\rightarrow = \emptyset$ and $\{NAS\}$ is the only event with pending response.

Therefore an accepting run in G(B) is the one in which the event $\{NAS\}$ is either executed or excluded infinitely often.

Since $\{NAS\}$ can not be executed at all due to it's self condition ($\rightarrow\!\!\bullet\, NAS = \{NAS\}$), the only way a run is accepting in G(B) is by excluding the event NAS infinitely often.

From the definition of exclude relation in the encoding (def 3.5.3), the set of events that can exclude the the event NAS is
$\rightarrow\!\%\, NAS = \{e(s, a, s') \mid (s, a, s') \in\rightarrow \wedge\ \ s' \in F\}$

The transition event $e(s, a, s')$ in G(B) exactly corresponds to a transition (s, a, s') in B and further state s' is visiting one of the final states in B.

Since run in G(B) is accepting, so the event *NAS* will be excluded infinitely often by executing one of the transition event in $\{e(s, a, s') \mid (s, a, s') \in \rightarrow \land \ \ s' \in F\}$, which corresponds to visiting one of the final states infinitely often in B, which satisfies the condition for a run to be accepting in B.

Therefore, we can conclude that, If a run is accepting in G(B) then it is also accepting in B.

Since we have the theorem in both directions, we can conclude that both B and G(B) will have same runs and accepting runs.

□

3.5.4 Conclusion

The *nondeterministic Büchi-automaton* is a kind of automaton that is suited for all accepting ω-regular languages, as it has acceptor for infinite words. Moreover, the equivalence between *nondeterministic Büchi-automaton* and ω-regular languages was proved by McNaughton in 1966 [McNaughton 1966] and therefore *nondeterministic Büchi-automaton* is as expressive as ω-regular languages. Hence *nondeterministic Büchi-automaton* is considered as alternative formalism to describe ω-regular languages.

In this section, we have proved the equivalence between theDCR Graphs and *nondeterministic Büchi-automaton* by providing a straight forward construction from *nondeterministic Büchi-automaton* to a DCR Graph. Moreover, the construction is linear in the sense that the DCR Graph contains number of events equal to total number of transitions plus states and additionally one more event for toggling the accepting condition. Therefore, we conclude that the DCR Graphs is expressive enough to describe ω-regular languages.

3.6 Summary

In this chapter, we have introduced DCR Graphs as a formal model for for a new declarative, event-based workflow process model inspired by the workflow language employed by our industrial partner [Mukkamala *et al.* 2008]. We have demonstrated the use and flexibility of the model on a small example taken from a field study on Danish hospitals [Lyng *et al.* 2008] and proposed a graphical notation for presenting both the process specification and their run-time state.

The model was presented as a sequence of generalizations of the classical model for concurrency of labeled event structures [Winskel 1986]. The first generalization introduced a notion of progress to event structures by replacing the usual causal order by two dual relations, a *condition* relation →• expressing for each event which events it has as preconditions and a *response* relation $\bullet\!\!\rightarrow$ expressing for each event which events that must happen (or be ruled out) after it has happened. We further demonstrated that the resulting model, named *condition response event structures* can express the standard notion of weak concurrency fairness.

The next generalization is to allow for finite representations of infinite behaviours by allowing *multiple execution*, and *dynamic inclusion* and *exclusion* of events, resulting in the model of *dynamic condition response graphs*. Finally, we extended the model to allow distribution of events via roles and presented a graphical notation inspired by related work by van der Aalst et al. [van der Aalst & Pesic 2006a, van der Aalst *et al.* 2009], but extended to include information about the run-time state (e.g. markings).

We have shown that all generalizations conservatively contain the previous model. Moreover, we provide a mapping from dynamic condition response graphs to Büchi-automata characterising the acceptance condition for finite and infinite runs, by introducing a special silent event e.g. τ-event.

One key advantage of the DCR Graphs compared to the related work explored in [van der Aalst & Pesic 2006a, van der Aalst *et al.* 2009, Davulcu *et al.* 1998, Cicekli & Cicekli 2006] is that the latter logics are more complex to visualize and understand by people not trained in logic. Another advantage, illustrated in the given mapping to Büchi-automata and our graphical visualization of the run time state, is that the execution of dynamic condition response graphs can be based on a relatively simple information about the run-time state, which can also be visualized directly as annotations (marking) on the graph.

Finally, we have proved the equivalence between DCR Graphs and *nondeterministic Büchi-automaton*, there by proved that the DCR Graphs expressive enough to describe ω-regular languages.

In the next chapter, we will look into the extensions for DCR Graphs such as nested sub structures for modeling hierarchy, sub processes for modeling of multiple instances and also extend DCR Graphs to support data as a shared global store of variables.

Chapter 4

Dynamic Condition Response Graphs - Extensions

In the previous chapter (chapter 3), we have introduced the basic model and core primitives of DCR Graphs. In this chapter, we will describe the extensions to the DCR Graphs, which will make the formal model more applicable to various real world scenarios and case studies. The first and foremost extension to the DCR Graphs is nested subgraphs which is a standard in most state-of-art modeling notations to model hierarchy. A case study from the Case Management domain will be introduced in section. 4.1.3 and we will demonstrate how we have applied nested DCR Graphs in practice within a project that our industrial partner Exformatics carried out for one of their customers.

Further, in the section 4.2, we will introduce an extension sub-processes to model replicated behavior in the DCR Graphs and then extend it to the nested DCR Graphs. Finally, we will introduce an important extension adding support for data to the DCR Graphs in the section 4.3.

We employ the following notations in this chapter.

Notation: For a set A we write $\mathcal{P}(A)$ for the power set of A. For a binary relation $\rightarrow \subseteq A \times A$ and a subset $\xi \subseteq A$ of A we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{a \in A \mid (\exists a' \in \xi \mid a \rightarrow a')\}$ and the set $\{a \in A \mid (\exists a' \in \xi \mid a' \rightarrow a)\}$ respectively. Also, we write $\rightarrow^{-1}$ for the inverse relation. Finally, for a natural number k we write $[k]$ for the set $\{1, 2, \ldots, k\}$.

4.1 Nested Dynamic Condition Response Graphs

In this section, we describe how to extend the model to allow for *nested sub-graphs*. Initially, the extension was guided by a second case study, in which we have applied the model of *Nested Dynamic Condition Response Graphs* (Nested DCR Graphs) in the design phase of the development of a distributed, inter-organizational case management system.

In the next section (sec 4.1.1), we will introduce the Nested DCR Graphs with the help of oncology healthcare workflow, which was previously identified during a field study at Danish hospitals [Lyng *et al.* 2008]. The formal semantics of Nested DCR Graphs will be given in the sec 4.1.2 and finally in sec 4.1.3, we will describe the case management case study which has motivated the extension of Nested DCR Graphs.

4.1.1 Nested DCR Graphs by Healthcare Workflow Example

In Fig. 4.1, we show the graphical representation of the Nested DCR Graphs formalizing a variant of the oncology workflow studied in [Lyng *et al.* 2008].

As explained in the previous chapter, the boxes denote *activities* (also referred to as events in many places). **Administer medicine** is a *nested* activity having sub activities **give medicine** and **trust**. **Give medicine** is an *atomic* activity, i.e. it has no sub activities and on the other hand, **Trust** is again a nested activity having sub activities **sign nurse 1** and **sign nurse 2**. The activity **medicine preparation** is a nested activity having seven sub activities dealing with the preparation of medicine, where as **manage prescription** is a nested activity with two sub activities. An activity may be either included or excluded, the latter activeties are drawn as a dashed box as e.g. the **edit** and **cancel activities**. Finally, **treatment** is a nested activity containing all other activities as sub activities.

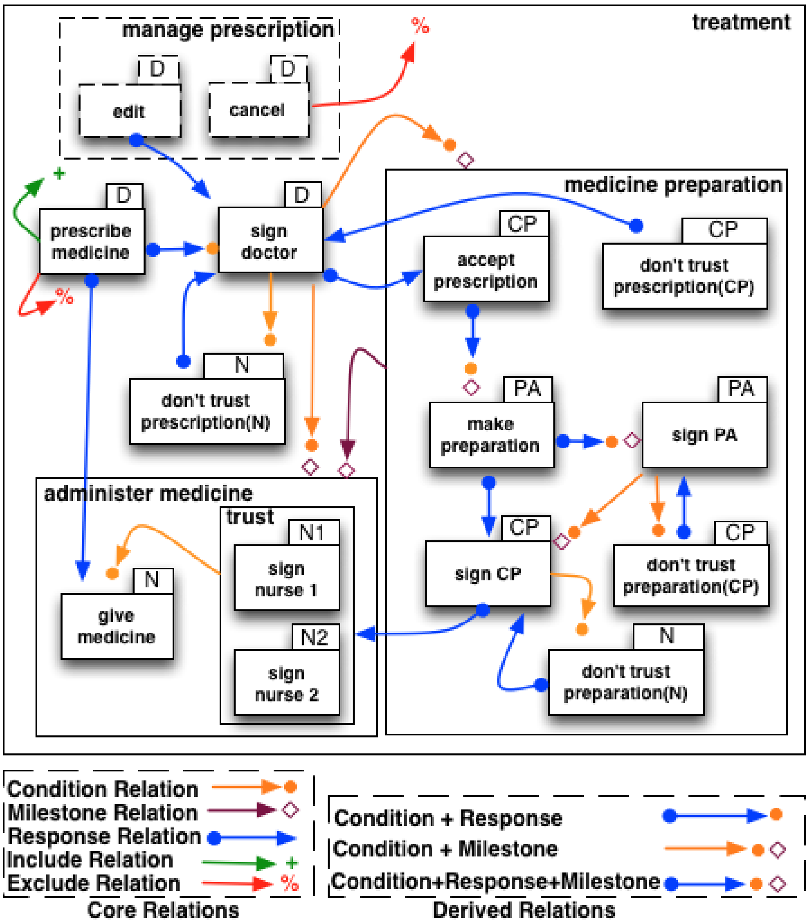


Figure 4.1: Oncology Workflow as a nested DCR Graph

A *run* of the workflow consists of a (possibly infinite) sequence of executions of atomic activities. (A nested activity is considered executed when all its sub activities are executed). An activity can be executed any number of times during a run, as long as the activity is included and the constraints for its execution are satisfied, in which case we say the activity is *enabled*.

As explained in the previous chapter (chapter 3), the constraints and dynamic exclusion and inclusion are expressed as five different core relations between activities represented as arrows in the figure above: The *condition relation*, the *response relation*, the *milestone relation*, the *include relation*, and the *exclude relation*.

The condition relation is represented by an orange arrow with a bullet at the arrow head, e.g. the condition relation from the activity sign doctor to the activity don't trust prescription(N) means that sign doctor must have been executed at least once before the activity don't trust prescription(N) can be executed.

The response relation is represented by a blue arrow with a bullet at its source. E.g. the response relation from the activity prescribe medicine to the activity give medicine means that the latter must be executed (at some point of time) after (any execution of) the activity prescribe medicine. We say that a workflow is in a *completed* state if all such response constraints have been fulfilled (or the required response activity is excluded). However, note that a workflow may be continued from a completed state and change to a non-completed state if an activity is executed that requires another activity as a response or includes an activity which has not been executed since it was last required as a response.

The third core relation used in the example is the *milestone relation* represented as a dark red arrow with a diamond at the arrow head. The milestone relation was introduced in [Hildebrandt *et al.* 2011c] jointly with the ability to nest activities. A relation to and/or from a nested activity simply unfolds to relations between all sub activities. A milestone relation from a nested activity to another activity then in particular means that the entire nested activity must be in a completed state before that activity can be executed. E.g. medicine preparation is a milestone for the activity administer medicine, which means that none of the sub activities of administer medicine can be carried out if any one of the sub activities of medicine preparation is included and has not been executed since it was required as a response.

Further, two activities can be related by any combination of these relations. In the graphical notation we have employed some shorthands, e.g. indicating the combination of a condition and a response relation by and arrow with a bullet in both ends.

Finally, DCR Graphs allow two relations for dynamic exclusion and dynamic inclusion of activities represented as a green arrow with a plus at the arrow head and a red arrow with a percentage at the arrow head respectively. The exclusion relation is used in the example between the cancel activity and the treatment activity. Since all other activities in the workflow are sub activities of the treatment activity, then all activities are excluded if the cancel activity is executed. The inclusion relation is used between the prescribe medicine activity and the manage prescription activity, so when prescribe medicine is executed, the manage prescription will be included.

The run-time state of a nested DCR Graph can be formally represented as a triple (Ex, Re, In) of sets of atomic activities (referred to as the *marking* of the graph). The set Ex is the set of atomic activities that have been executed at least once during the run. The set *Re* is the set of atomic activities that, if included, are required to be executed at least one more time in the future as the result of a response constraint (i.e. they are pending responses). Finally, the set *In* denotes the currently included activities. The set Ex thus may be regarded as a set of completed activities, the set Re as the set of activities on the to-do list and the set In as the activities that are currently relevant for the workflow.

Note that an activity may be completed once and still be on the to-do list, which simply means that it must be executed (completed) again. This makes it very simple to model the situation where an activity needs to be (re)considered as a response to the execution of an activity. In the oncology example this is e.g. the case for the response relation between the don't trust prescription(N) activity (representing that a nurse reports that he/she doesn't trust the prescription) and the sign doctor activity. The effect is that the doctor is asked to reconsider her signature on the prescription. In doing that the doctor may or may not decide to change the prescription, i.e. execute prescribe medicine again.

We indicate the marking graphically by adding a check mark to every atomic activity that has been executed (i.e. is included in the set Ex of the marking), an exclamation mark to every atomic activity which, if included, is required to be executed at least once more in the future (i.e. is included in the set Re), and making a box dashed if the activity is not included (i.e. is not included in the set In of the marking). In Fig. 4.2 we have shown an example marking where prescribe medicine has been executed. This has caused manage prescription and its sub activities edit and cancel to be included, and sign doctor and give medicine to be required as responses, i.e the two activities are included in the set Re of the marking (on the to-do list).

As described in the previous chapter (sec 3.3.2), an activity can be executed if it is enabled. Sign doctor is enabled for execution in the example marking, since its only condition (prescribe medicine) has been executed and it has no milestones. Give medicine on the other hand is not enabled since it has the (nested) activity trust as condition, which means that all sub activities of trust (sign nurse 1 and sign nurse 2) must be executed before give medicine is enabled. Also, both give medicine and trust are sub activities of administer medicine which further has sign doctor as condition and milestone, and medicine preparation as milestone. The condition relation from sign doctor means that the prescription must be signed before the medicine can be administered. The milestone relations means that the medicine can not be given as long as sign doctor or any of the sub activities of medicine preparation is on the to-do list (i.e. in the set Re of pending responses).

With the informal introduction of Nested DCR Graphs using the healthcare workflow, now we will provide a formal definition of Nested DCR Graphs in the next section.

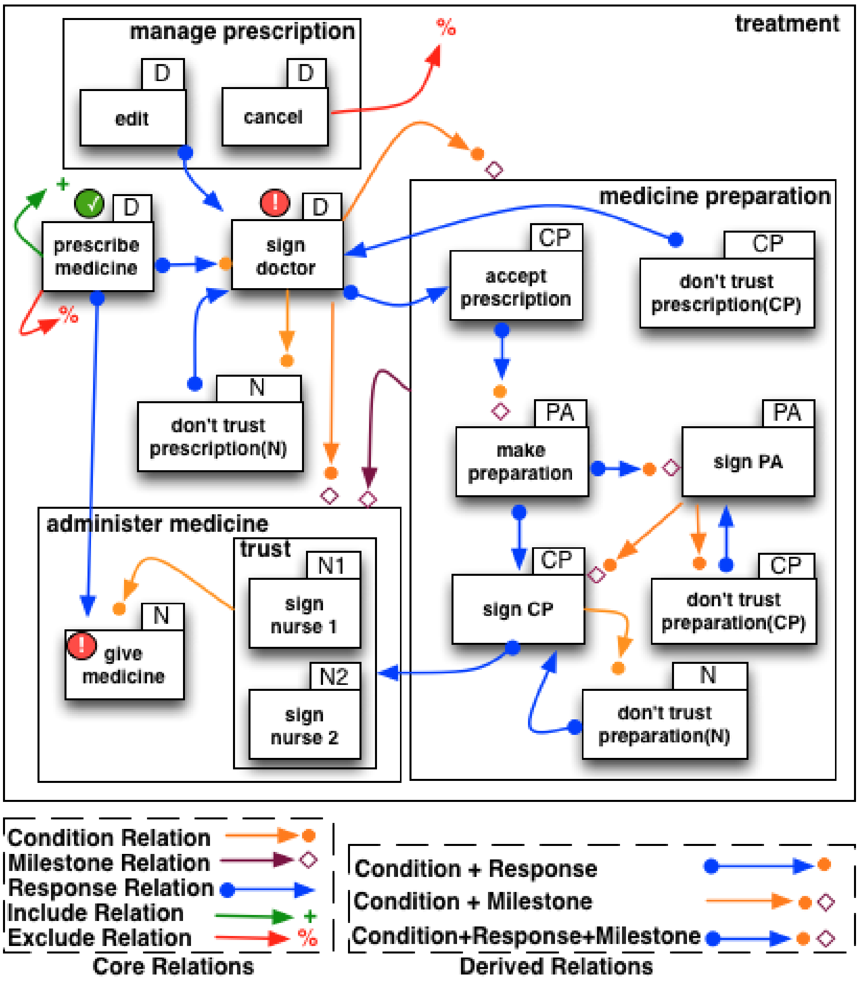


Figure 4.2: Oncology Workflow as a nested DCR Graph with runtime state

4.1.2 Nested DCR Graphs - Formal Semantics

Let us recall the formal definitions of DCR Graphs (sec 3.3.6) and distributed DCR Graphs(sec 3.3.6) from the chapter 3. First we have defined DCR Graphs in definition 3.3.6 and then the model is extended by adding roles and principals to further define distributed DCR Graphs in definition 3.3.6. In the later versions of formalization of DCR Graphs, we have further abstracted away from roles and principals and defined a more general version of DCR Graphs, where labels of events were sets of triples consisting of an action, a role and a principal.

Hence we first give a more general definition of a DCR Graph and then formally define nested dynamic condition response graph as follows.

Definition 4.1.1. *A Dynamic Condition Response Graph (DCR Graph)* G *is a tuple* $(E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$, *where*

(i) E *is the set of* events *(or activities),*

(ii) $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \in \mathcal{M}(\mathsf{G})$ *is the* marking, *for* $\mathcal{M}(\mathsf{G}) =_{def} \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E})$,

(iii) $\rightarrow\!\bullet \subseteq \mathsf{E} \times \mathsf{E}$ *is the* condition *relation,*

(iv) $\bullet\!\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$ *is the* response *relation,*

(v) $\rightarrow\!\diamond \subseteq \mathsf{E} \times \mathsf{E}$ *is the* milestone *relation,*

(vi) $\rightarrow+, \rightarrow\% \subseteq \mathsf{E} \times \mathsf{E}$ *is the dynamic* include *relation and* exclude *relation, satisfying that* $\forall e \in \mathsf{E}.e \rightarrow+ \cap e \rightarrow\% = \emptyset$,

(vii) L *is the set of* labels,

(viii) $l : \mathsf{E} \rightarrow \mathcal{P}(\mathsf{L})$ *is a labeling function mapping events to sets of labels.*

Note that, now each event is mapped to the set of labels (viii), which can consist of name of the event and a role which defines who can execute that event. In our implementation every event can be assigned any number of roles and every user of the system can have multiple roles. A user can then execute an event if she has at least one role that is assigned to the event.

Definition 4.1.2. *A Nested Dynamic Condition Response Graphs (Nested DCR Graph)* G *is a tuple* $(\mathsf{E}, \rhd, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, *where*

(i) $(\mathsf{E}, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ *is a DCR Graph,*

(ii) $\rhd : \mathsf{E} \rightharpoonup \mathsf{E}$ *is a partial function mapping an event to its super-event (if defined),*

(iii) $\mathsf{M} \in \mathcal{P}(\mathsf{atoms}(\mathsf{E})) \times \mathcal{P}(\mathsf{atoms}(\mathsf{E})) \times \mathcal{P}(\mathsf{atoms}(\mathsf{E}))$, *where* $\mathsf{atoms}(\mathsf{E}) = \mathsf{E} \backslash \{e \in \mathsf{E} \mid \exists e' \in \mathsf{E}. \rhd (e') = e\}$ *is the set of atomic events.*

We write $e \rhd e'$ *if* $e' = \rhd^k(e)$ *for* $0 < k$ *and write* $e \unrhd e'$ *if* $e \rhd e'$ *or* $e = e'$, *and* $e \unlhd e'$ *if* $e' \rhd e$ *or* $e = e'$. *We require that the resulting relation,* $\unrhd \subset E \times E$, *referred to as the nesting relation, is a well founded partial order. We also require that the nesting relation is consistent with respect to dynamic inclusion/exclusion in the following sense: If* $e \rhd e'$ *or* $e' \rhd e$ *then* $e \rightarrow+ \cap e' \rightarrow\% = \emptyset$.

We already introduced the graphical notation for Nested DCR Graphs by example in the previous section. The complete formal specification of the example is shown in the listing 4.1. Let us use abbreviations for the event names in the formal specification of example: treatment (**treat**), manage prescription (**man pres**), medicine preparation (**med prep**), administer medicine (**adm med**), trust (**trust**), edit (**edit**), cancel (**canc**), prescribe medicine (**pres med**), sign doctor (**sn doc**), give medicine (**gm**), don't trust prescription(N) (**dt pres N**), sign nurse 1 (**sn N1**), sign nurse 2 (**sn N2**), accept prescription (**acc pres**), don't trust prescription(CP) (**dt pres CP**), make preparation (**mk prep**), sign PA (**sn PA**), sign CP (**sn CP**), don't trust preparation(CP) (**dt prep CP**), don't trust preparation(N) (**dt prep N**).

Listing 4.1: Formal specification of Healthcare Workflow in Nested DCR Graphs.

A Nested DCR Graph $G = (E, \triangleright, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \pm, L, l)$ where

$$\begin{aligned}E = \{&\mathsf{treat}, \mathsf{man\ pres}, \mathsf{med\ prep}, \mathsf{adm\ med}, \mathsf{trust}, \mathsf{edit}, \mathsf{canc}, \mathsf{pres\ med}, \mathsf{sn\ doc}, \mathsf{dt\ pres\ N},\\ &\mathsf{gm}, \mathsf{sn\ N1}, \mathsf{sn\ N2}, \mathsf{acc\ pres}, \mathsf{dt\ pres\ CP}, \mathsf{mk\ prep}, \mathsf{sn\ PA}, \mathsf{sn\ CP}, \mathsf{dt\ prep\ CP}, \mathsf{dt\ prep\ N}\}\end{aligned}$$

$$\begin{aligned}\triangleright = \{&(\mathsf{man\ pres}, \mathsf{treat}), (\mathsf{med\ prep}, \mathsf{treat}), (\mathsf{adm\ med}, \mathsf{treat}), (\mathsf{trust}, \mathsf{treat}),\\ &(\mathsf{pres\ med}, \mathsf{treat}), (\mathsf{sn\ doc}, \mathsf{treat}), (\mathsf{dt\ pres\ N}, \mathsf{treat}), (\mathsf{edit}, \mathsf{man\ pres}),\\ &(\mathsf{canc}, \mathsf{man\ pres}), (\mathsf{acc\ pres}, \mathsf{med\ prep}), (\mathsf{dt\ pres\ CP}, \mathsf{med\ prep}),\\ &(\mathsf{mk\ prep}, \mathsf{med\ prep}), (\mathsf{sn\ PA}, \mathsf{med\ prep}), (\mathsf{sn\ CP}, \mathsf{med\ prep}),\\ &(\mathsf{dt\ prep\ CP}, \mathsf{med\ prep}), (\mathsf{dt\ prep\ N}, \mathsf{med\ prep}), (\mathsf{gm}, \mathsf{adm\ med}),\\ &(\mathsf{trust}, \mathsf{adm\ med}), (\mathsf{sn\ N1}, \mathsf{trust}), (\mathsf{sn\ N2}, \mathsf{trust})\}\end{aligned}$$

$$\begin{aligned}atoms(E) = \{&\mathsf{edit}, \mathsf{canc}, \mathsf{pres\ med}, \mathsf{sn\ doc}, \mathsf{dt\ pres\ N}, \mathsf{gm}, \mathsf{sn\ N1}, \mathsf{sn\ N2}, \mathsf{acc\ pres},\\ &\mathsf{dt\ pres\ CP}, \mathsf{mk\ prep}, \mathsf{sn\ PA}, \mathsf{sn\ CP}, \mathsf{dt\ prep\ CP}, \mathsf{dt\ prep\ N}\}\end{aligned}$$

$$M = (\emptyset, \emptyset, E \setminus \{\mathsf{man\ pres}, \mathsf{edit}, \mathsf{canc}\})$$

$$\begin{aligned}\rightarrow\!\bullet = \{&(\mathsf{pres\ med}, \mathsf{sn\ doc}), (\mathsf{sn\ doc}, \mathsf{med\ prep}), (\mathsf{sn\ doc}, \mathsf{adm\ med}), (\mathsf{sn\ doc}, \mathsf{dt\ pres\ N}),\\ &(\mathsf{acc\ pres}, \mathsf{mk\ prep}), (\mathsf{mk\ prep}, \mathsf{sn\ PA}), (\mathsf{sn\ PA}, \mathsf{sn\ CP}), (\mathsf{sn\ PA}, \mathsf{dt\ prep\ CP}),\\ &(\mathsf{sn\ CP}, \mathsf{dt\ prep\ N}), (\mathsf{trust}, \mathsf{gm})\}\end{aligned}$$

$$\begin{aligned}\bullet\!\rightarrow = \{&(\mathsf{edit}, \mathsf{sn\ doc}), (\mathsf{pres\ med}, \mathsf{gm}), (\mathsf{pres\ med}, \mathsf{sn\ doc}), (\mathsf{sn\ doc}, \mathsf{acc\ pres}),\\ &(\mathsf{dt\ pres\ N}, \mathsf{sn\ doc}), (\mathsf{dt\ pres\ CP}, \mathsf{sn\ doc}), (\mathsf{acc\ pres}, \mathsf{mk\ prep}), (\mathsf{mk\ prep}, \mathsf{sn\ PA}),\\ &(\mathsf{mk\ prep}, \mathsf{sn\ CP}), (\mathsf{dt\ prep\ CP}, \mathsf{sn\ PA}), (\mathsf{dt\ prep\ N}, \mathsf{sn\ CP}), (\mathsf{sn\ CP}, \mathsf{trust})\}\end{aligned}$$

$$\begin{aligned}\rightarrow\!\diamond = \{&(\mathsf{sn\ doc}, \mathsf{med\ prep}), (\mathsf{sn\ doc}, \mathsf{adm\ med}), (\mathsf{acc\ pres}, \mathsf{mk\ prep}), (\mathsf{mk\ prep}, \mathsf{sn\ PA}),\\ &(\mathsf{sn\ PA}, \mathsf{sn\ CP}), (\mathsf{med\ prep}, \mathsf{adm\ med})\}\end{aligned}$$

$$\rightarrow\!+ = \{(\mathsf{pres\ med}, \mathsf{man\ pres})\}$$

$$\rightarrow\!\% = \{(\mathsf{pres\ med}, \mathsf{pres\ med}), (\mathsf{canc}, \mathsf{treat})\}$$

$$\begin{aligned}L = \{&(\mathsf{edit}, \mathsf{D}), (\mathsf{canc}, \mathsf{D}), (\mathsf{pres\ med}, \mathsf{D}), (\mathsf{sn\ doc}, \mathsf{D})\} \cup\\ &\{(\mathsf{dt\ pres\ N}, \mathsf{N}), (\mathsf{gm}, \mathsf{N}), (\mathsf{dt\ prep\ N}, \mathsf{N}), (\mathsf{sn\ N1}, \mathsf{N1}), (\mathsf{sn\ N2}, \mathsf{N2})\} \cup\\ &\{(\mathsf{acc\ pres}, \mathsf{CP}), (\mathsf{dt\ pres\ CP}, \mathsf{CP}), (\mathsf{sn\ CP}, \mathsf{CP}), (\mathsf{dt\ prep\ CP}, \mathsf{CP})\} \cup\\ &\{(\mathsf{mk\ prep}, \mathsf{PA}), (\mathsf{sn\ PA}, \mathsf{PA})\}\end{aligned}$$

$$\begin{aligned}l = \{&(\mathsf{edit}, (\mathsf{edit}, \mathsf{D})), (\mathsf{canc}, (\mathsf{canc}, \mathsf{D})), (\mathsf{pres\ med}, (\mathsf{pres\ med}, \mathsf{D})), (\mathsf{sn\ doc}, (\mathsf{sn\ doc}, \mathsf{D}))\\ &(\mathsf{dt\ pres\ N}, (\mathsf{dt\ pres\ N}, \mathsf{N})), (\mathsf{gm}, (\mathsf{gm}, \mathsf{N})), (\mathsf{dt\ prep\ N}, (\mathsf{dt\ prep\ N}, \mathsf{N})), (\mathsf{sn\ N1},\\ &(\mathsf{sn\ N1}, (\mathsf{sn\ N1}, \mathsf{N1})), (\mathsf{sn\ N2}, (\mathsf{sn\ N2}, \mathsf{N2})), (\mathsf{acc\ pres}, (\mathsf{acc\ pres}, \mathsf{CP})),\\ &(\mathsf{dt\ pres\ CP}, (\mathsf{dt\ pres\ CP}, \mathsf{CP})), (\mathsf{sn\ CP}, (\mathsf{sn\ CP}, \mathsf{CP})), (\mathsf{dt\ prep\ CP}, (\mathsf{dt\ prep\ CP}, \mathsf{CP})),\\ &(\mathsf{mk\ prep}, (\mathsf{mk\ prep}, \mathsf{PA})), (\mathsf{sn\ PA}, (\mathsf{sn\ PA}, \mathsf{PA}))\}\end{aligned}$$

The events are all boxes, e.g. $E = \{\mathsf{treat}, \mathsf{man\ pres}, \mathsf{med\ prep}, \ldots\}$, the nesting relation captures the inclusion of boxes, e.g. $\triangleright(e) = \mathsf{adm\ med}$, if $e \in \{\mathsf{gm}, \mathsf{trust}\}$ and $\triangleright(e) = \mathsf{trust}$, if $e \in \{\mathsf{sn\ N1}, \mathsf{sn\ N2}\}$ and so forth. The initial marking is the triple $M = (\emptyset, \emptyset, E \setminus \{\mathsf{man\ pres}, \mathsf{edit}, \mathsf{canc}\})$, meaning no events have been executed, no events are initially required as responses and all events except the events $\{\mathsf{man\ pres}, \mathsf{edit}, \mathsf{canc}\}$ are included. We take labels as pairs of action names and roles, i.e. the set of labels

L includes e.g. the pairs (**edit**, **D**), (**canc**, **D**), (**gm**, **N**), and (**sn PA**, **PA**). Super events with no role assigned such as **med prep** are assigned the empty set of labels.

To define the execution semantics for Nested DCR Graphs, we first define how to flatten a nested graph to the simpler DCR Graph. Essentially, all relations to and/or from nested events are extended to sub events, and then only the atomic events are preserved.

Definition 4.1.3. *For a Nested DCR Graph* $G = (E, \rhd, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ *define the underlying flat Dynamic Condition Response Graph as*

$$G^\flat = (\mathrm{atoms}(E), M, \rightarrow\!\bullet^\flat, \bullet\!\rightarrow^\flat, \rightarrow\!\diamond^\flat, \rightarrow\!+^\flat, \rightarrow\!\%^\flat, L, l)$$

where $rel^\flat = \unrhd rel \unlhd$ *for some relation* $rel \in \{\rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%\}$.

It is easy to see from the definition that the underlying DCR Graph has at most as many events as the nested graph and that the size of the relations may increase by an order of n^2 where n is the number of atomic events.

The listing 4.2 shows the flattened DCR Graph for the healthcare workflow obtained by flattening the Nested DCR Graph (listing 4.1) according to the definition 4.1.3.

Listing 4.2: Flatten DCR Graph for Healthcare Workflow from listing 4.1.

The underlying flat DCR graph for nested graph G in the listing 4.1

$G^\flat_f = (\mathrm{atoms}(E), M, \rightarrow\!\bullet^\flat, \bullet\!\rightarrow^\flat, \rightarrow\!\diamond^\flat, \rightarrow\!+^\flat, \rightarrow\!\%^\flat, L, l)$ where

atoms(E), M, L, l are same as in listing 4.1

$$\begin{aligned}\rightarrow\!\bullet^\flat = \rightarrow\!\bullet \cup \{&(\textsf{sn doc}, \textsf{acc pres}), (\textsf{sn doc}, \textsf{dt pres CP}), (\textsf{sn doc}, \textsf{mk prep}), (\textsf{sn doc}, \textsf{sn PA}),\\ &(\textsf{sn doc}, \textsf{sn CP}), (\textsf{sn doc}, \textsf{dt prep CP}), (\textsf{sn doc}, \textsf{dt prep N}), (\textsf{sn doc}, \textsf{gm}),\\ &(\textsf{sn doc}, \textsf{sn N1}), (\textsf{sn doc}, \textsf{sn N2}), (\textsf{sn N1}, \textsf{gm}), (\textsf{sn N2}, \textsf{gm})\}\\ \setminus\{&(\textsf{sn doc}, \textsf{med prep}), (\textsf{sn doc}, \textsf{adm med}), (\textsf{trust}, \textsf{gm})\}\end{aligned}$$

$$\bullet\!\rightarrow^\flat = \big(\bullet\!\rightarrow \cup \{(\textsf{sn CP}, \textsf{sn N1}), (\textsf{sn CP}, \textsf{sn N1})\}\big) \setminus \{(\textsf{sn CP}, \textsf{trust})\}$$

$$\begin{aligned}\rightarrow\!\diamond^\flat = \big(\rightarrow\!\diamond \cup \big\{&\{\textsf{sn doc}\} \times \{\textsf{acc pres}, \textsf{dt pres CP}, \textsf{mk prep}, \textsf{sn PA}, \textsf{sn CP}, \textsf{dt prep CP},\\ &\textsf{dt prep N}\}\big\} \cup \big\{\{\textsf{sn doc}\} \times \{\textsf{gm}, \textsf{sn N1}, \textsf{sn N2}\}\big\} \cup \big\{\{\textsf{acc pres}, \textsf{dt pres CP},\\ &\textsf{mk prep}, \textsf{sn PA}, \textsf{sn CP}, \textsf{dt prep CP}, \textsf{dt prep N}\} \times \{\textsf{gm}, \textsf{sn N1}, \textsf{sn N2}\}\big\}\big)\\ &\setminus\{(\textsf{sn doc}, \textsf{med prep}), (\textsf{sn doc}, \textsf{adm med}), (\textsf{med prep}, \textsf{adm med})\}\end{aligned}$$

$$\rightarrow\!+^\flat = \big(\rightarrow\!+ \cup\{(\textsf{pres med}, \textsf{edit}), (\textsf{pres med}, \textsf{canc})\}\big) \setminus \{(\textsf{pres med}, \textsf{man pres})\}$$

$$\rightarrow\!\%^\flat = \big(\rightarrow\!\% \cup\big\{\{\textsf{canc}\} \times \mathrm{atoms}(E)\big\}\big) \setminus \{(\textsf{canc}, \textsf{treat})\}$$

Before defining when an event is enabled in a Nested Dynamic Condition Response Graphs, let us recall the definition of an enabled event in a DCR Graph from definition 3.3.7. It says that an event e of a DCR Graph is enabled when it is included in current marking ($e \in \mathrm{In}$), all the included events that are conditions for it are in the set of executed events (i.e. $(\mathrm{In} \cap \rightarrow\!\bullet e) \subseteq \mathrm{Ex}$) and none of the included events that are milestones for it are in the set of pending response events (i.e. $(\mathrm{In} \cap \rightarrow\!\diamond e) \subseteq E \backslash \mathrm{Re}$).

Further, also recall the definition 3.3.8 from the previous chapter, which defines the change of the marking in a DCR Graph when an enabled event is executed: First the event is added to the set of executed events and removed from the set of pending responses. Then all events that are a response to the event are added to the set of pending responses. Note that if an event is a response to itself, it will remain in the set of pending responses after execution. Similarly, the included events set will updated by adding all the events that are included by the event and by removing all the events that are excluded by the event.

We now define the semantics for Nested DCR Graph by using the corresponding flat graph about when an event is enabled and the result of executing an event in a Nested DCR Graph in the definition 4.1.4.

Definition 4.1.4. *For a Nested Dynamic Condition Response Graphs* $\mathsf{G} = (\mathsf{E}, \rhd, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{L}, l)$*, where* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *we define that* $e \in \mathsf{atoms}(\mathsf{E})$ *is enabled, written* $\mathsf{M} \vdash_{\mathsf{G}} e$*, if* $\mathsf{M} \vdash_{\mathsf{G}^\flat} e$*. Similarly, the result of executing* $\mathsf{M} \oplus_{\mathsf{G}} e$ *same as executing the event in flattened graph and it is defined as:* $\mathsf{M} \oplus_{\mathsf{G}^\flat} e = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \oplus_{\mathsf{G}^\flat} e$*.*

As an example, in the intial marking $\mathsf{M} = (\emptyset, \emptyset, \mathsf{E} \setminus \{\textsf{man pres}, \textsf{edit}, \textsf{canc}\})$ we have that $\mathsf{G} \vdash$ pres med, i.e. the event prescribe medicine is enabled. After executing pres med the new marking $\mathsf{M}' = \mathsf{M} \oplus_{\mathsf{G}}$ pres med $= (\{\textsf{pres med}\}, \{\textsf{sn doc}, \textsf{gm}\}, \mathsf{E} \setminus \{\textsf{pres med}\})$. That is, pres med is added to the set of executed events, and sn doc and gm are added to the set of pending responses, because pres med $\bullet\!\!\rightarrow$ sn doc and pres med $\bullet\!\!\rightarrow$ gm. The event pres med is removed from the set of included events because pres med $\rightarrow\!\%$ pres med. The events $\{\textsf{man pres}, \textsf{edit}, \textsf{canc}\}$ are included since pres med $\rightarrow\!+$ man pres, and the inclusion relation is "flattened" to include also pres med $\rightarrow\!+$ edit and pres med $\rightarrow\!+$ canc.

From the definition of enabling and execution above we can construct a labelled transition semantics for a nested DCR Graphs, with acceptance conditions for finite and infinite computations.

Definition 4.1.5. *For a nested dynamic condition response graph* $\mathsf{G} = (\mathsf{E}, \rhd, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{L}, l)$ *we define the corresponding labelled transition system* $\mathsf{TS}(\mathsf{G})$ *to be the tuple*

$$(\mathcal{M}(\mathsf{G}), \mathsf{M}, \mathcal{L}_{ts}(\mathsf{G}), \rightarrow)$$

where $\mathcal{L}_{ts}(\mathsf{G}) = \mathsf{atoms}(\mathsf{E}) \times \mathsf{L}$ *is the set of labels of the transition system,* M *is the initial marking, and* $\rightarrow \subseteq \mathcal{M}(\mathsf{G}) \times \mathcal{L}_{ts}(\mathsf{G}) \times \mathcal{M}(\mathsf{G})$ *is the transition relation defined by* $\mathsf{M} \xrightarrow{(e,a)} \mathsf{M} \oplus_{\mathsf{G}^\flat} e$ *if* $\mathsf{M} \vdash_{\mathsf{G}^\flat} e$ *and* $a \in l(e)$*.*

We define a run $a_0, a_1, \ldots$ *of the transition system to be a sequence of labels of a sequence of transitions* $\mathsf{M}_i \xrightarrow{(e_i,a_i)} \mathsf{M}_{i+1}$ *starting from the initial marking. We define a run to be accepting (or completed) if for the underlying sequence of transitions it holds that* $\forall i \geq 0, e \in \mathsf{In}_i \cap \mathsf{Re}_i . \exists j \geq i.((e = e_j \vee e \notin \mathsf{In}_{j+1}))$*. In words, a run is accepting/completed if no required response event is continuously included and pending without it happens or become excluded.*

4.1.3 Case Study: Case Management Example In Nested DCR Graphs

In this section we demonstrate how we have applied DCR Graphs in practice within a project that our industrial partner Exformatics carried out for one of their customers. In the process, we have applied DCR Graphs in meetings with Exformatics and the customer to capture the requirements in a declarative way, accompanying the usual UML sequence diagrams and prototype mock-ups. Sequence diagrams typically only describe *examples* of runs, and even if they are extended with loops and conditional flows they do not capture the constraints explicitly.

The customer of the system is *Landsorganisationen i Danmark* (LO), which is the overarching organization for most of the trade unions in Denmark. Their counterpart is *Dansk Arbejdsgiverforening* (DA), which is an overarching organization for most of the Danish employers organizations.

At the top level, the workflow to be supported is that a case worker at the trade union must be able to create a case, e.g. triggered by a complaint by a member of the trade union against her employer. This must be followed up by a meeting arranged by LO and subsequently held between case workers at the trade union, LO and DA. After being created, the case can at any time be managed, e.g. adding or retrieving documents, by case workers at any of the organizations.

Fig. 4.3 shows the graphical representation of a simple DCR Graph capturing these top level requirements of our case study.

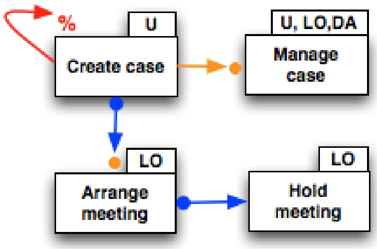


Figure 4.3: Top level requirements of case management as a DCR Graph

Four top-level events were identified, shown as boxes in the graph labelled Create case, Manage case, Arrange meeting and Hold meeting.

For the top-level events we identified the following requirements:

1. A case is created by a union case worker, and only once.

2. The case can be managed at the union, LO and DA after it has been created.

3. After a case is created, LO can and must arrange a meeting between the union case worker, the LO case worker and the DA case worker.

4. After a meeting is arranged it must be held (organized by LO).

The requirements translate to the following DCR Graph role assignments (shown as "ears" on the event boxes) and relations shown as different types of arrows between the events in Fig. 4.3:

1. **Create case** has assigned role U and excludes itself.

2. **Create case** is a condition for **Manage case**, which has assigned role U, LO and DA.

3. **Create case** has **Arrange meeting** as response, which has assigned role LO.

4. **Arrange meeting** has **Create case** as a condition and **Hold meeting** as response, which has assigned role LO.

For example, the **U** on **Create case** indicates that only a case worker at the trade union (U) can create a case, and the **U, LO, DA** on **Manage case** indicate that both the trade union, LO and DA can manage the case.

The arrow **Create case→•Manage case** denotes that **Manage case** has **Create case** as a (pre) *condition*. This simply means that **Create case** must have happened before **Manage case** can happen. Dually, **Arrange meeting** has **Hold meeting** as *response*, denoted by the arrow **Arrange meeting•→Hold meeting** This means that **Hold meeting** *must* eventually happen after **Arrange meeting** happens. Finally, the arrow **Create case →%Create case** denotes that the event **Create case** excludes itself.

In the subsequent meetings, we came to the following additional requirements:

1. (a) To create a case, the case worker should enter meta-data on the case, inform about when he/she is available for participating in a meeting and then submit the case.

   (b) When a case is submitted it may get a local id at the union, but it should also subsequently be assigned a case id in LO.

   (c) When a case is submitted, LO should eventually propose dates.

2. (a) Only after LO has assigned its case id it is possible to manage the case and for LO to propose dates.

   (b) Manage case consists of three possible activities (in any order): editing case meta data, upload documents and download documents. All activities can be performed by LO and DA. Upload and download documents can also be performed by the Union.

3. (a) The meeting should be arranged in agreement between LO and DA: LO should always propose dates first - and then DA should accept, but can also propose new dates. If DA proposes new dates LO should accept, but can also again propose new dates. This could in principle go on forever.

   (b) The union can always update information about when they are available and edit the metadata of the case.

4. (a) No meeting can be held while LO and DA are negotiating on a meeting date. Once a date has been agreed upon a meeting should eventually be held.

These requirements led to the extension of the model allowing *nested* events as formalized in the previous section (sec 4.1.2).

The requirements could then be described by first adding the following additional events to the graph: A new super event Edit (E) which has the sub events: Metadata (E-M) and Dates available (E-D) and is itself a sub event to Create case (CC). The Create case (CC) event has two sub events: Cc (SC) and Assign case Id (ACI). The Manage case (MC) event has two sub events: Edit metadata (EM) and Document (D), which in turn has two sub events: Upload (D-U) and Download (D-D). The Arrange meeting (AM) event has four sub events: Propose dates-LO (PLO), Propose dates-DA (PDA), Accept LO (ALO) and Accept DA (ADA). The Hold meeting (HM) event remains an atomic top-level event.

Subsequently, the relations was adapted to the following (Nested) DCR Graph relations, as shown in Fig 4.4:

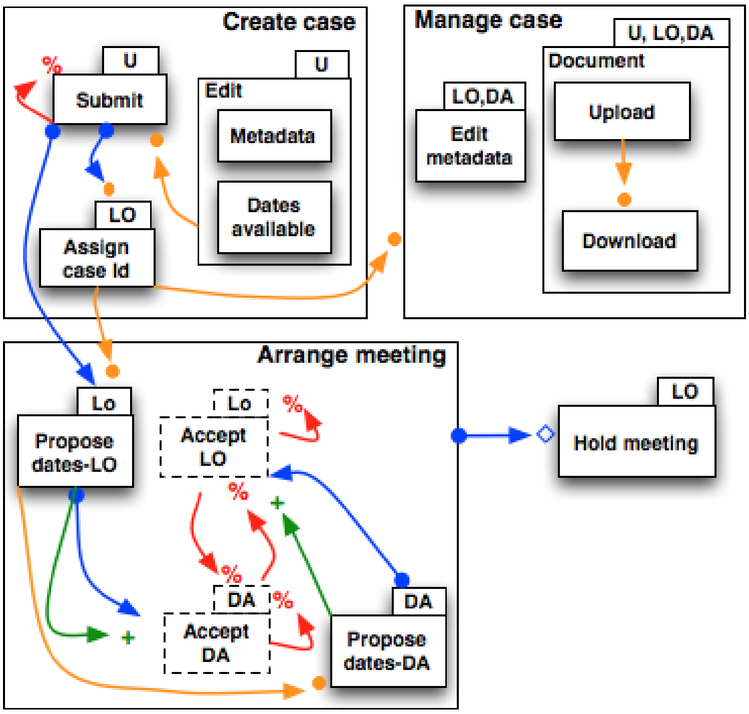


Figure 4.4: Case Handling Process

1. Edit is a condition to Cc and is assigned role U.

2. Within the Create case superevent:

   (a) Cc is a condition to Assign case Id and also requires it as a response.
   (b) Assign case Id is a condition for Manage case (and therefore also all it's sub events).
   (c) Assign case Id is now the condition for Propose dates-LO and Cc requires it as a response.

3. Within the Arrange meeting superevent:
   (a) Arrange meeting still has Hold meeting as response, but is now also required as a milestone for Hold meeting
   (b) Propose dates-LO is a condition for Propose dates-DA
   (c) Propose dates-LO includes Accept DA and requires it as a response
   (d) Propose dates-DA includes Accept LO and requires it as a response
   (e) Accept LO excludes itself and Accept DA
   (f) Accept DA excludes itself and Accept LO

4. Within the Manage case superevent:
   (a) Edit metadata has roles LO and DA assigned to it.
   (b) Upload and Download have been grouped under a superevent Document with roles U, LO and DA assigned to it.
   (c) Upload is a condition for Download.

In Fig. 4.5, 4.6, 4.7, we have illustrated how the execution state of the case-handling process may be visualized using the runtime notation of the DCR Graphs.

The graph in the figure. 4.5 shows the state after a run where the union started by creating a case: they edited meta-data, indicated the dates they were available and submitted. When LO received the case they assigned their own case ID to it. Some time later LO proposed possible dates for a meeting to DA. DA did not agree with these dates and responded by proposing some of their own. In the graph both Accept LO and Accept DA are included and have a pending response because both LO and DA have proposed dates. Because of these pending responses Hold meeting is disabled. Because no files have been uploaded to the document yet, Download is also disabled.

The graph in the figure. 4.6 shows the runtime state after the union has uploaded an agenda for the meetings. Note that, since the union has uploaded a file to the case, Download is now enabled. But at the same time, Accept LO and Accept DA still remain the same as the previous graph, as the proposed dates have not been accepted yet by either LO or DA.

Figure 4.7 shows the graph representing the state after LO has accepted one of the dates proposed by DA. Note that both Accept LO and Accept DA are excluded due to the mutual exclude relation between them. Even though there is a pending response on Accept DA, it is not considered relevant as it is excluded and Hold meeting

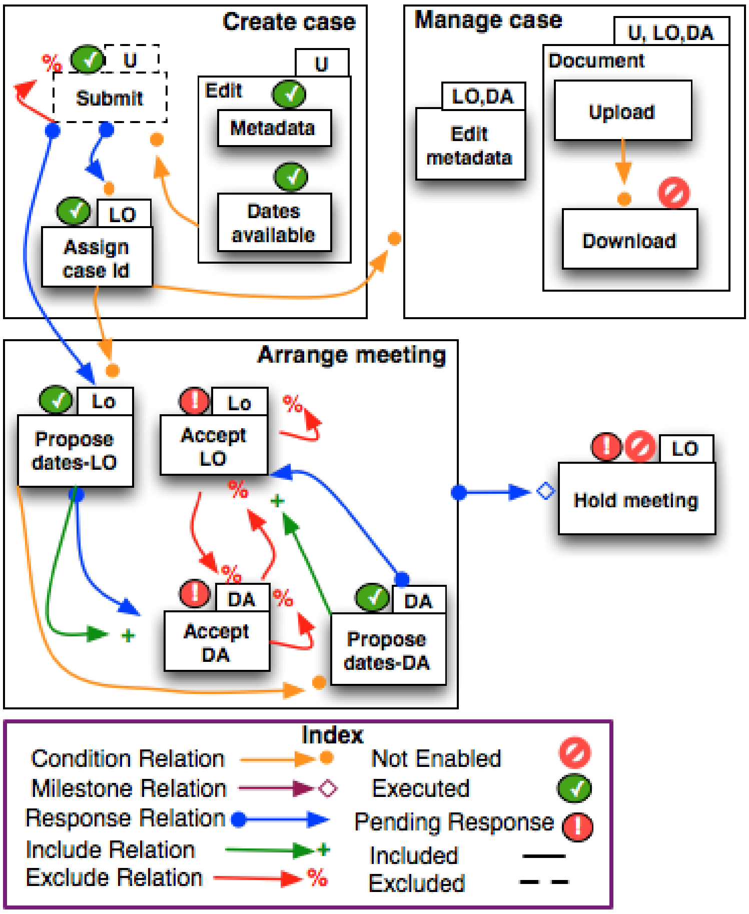


Figure 4.5: Case Handling Process Runtime

has become pending because of the response relation. Continuing by executing **Hold meeting** as LO will cause the graph to reach an accepting state, as there will be no included pending responses.

4.2 Nested DCR Graphs with Sub Processes

In this section, we will introduce an important extension to Nested Dynamic Condition Response Graphs, by name subprocesses to model the replicated behavior in processes.

4.2.1 Formal definition of Nested DCR Graphs with sub processes

First we define a Nested Dynamic Condition Response Graphs with subprocess formally as follows.

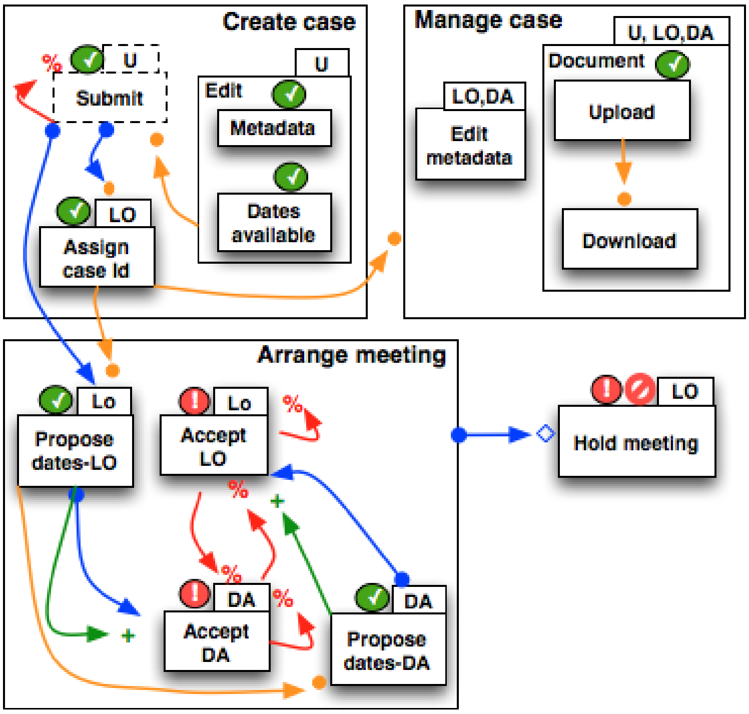


Figure 4.6: Case Handling Process Runtime After Upload Document

Definition 4.2.1. *A Nested Dynamic Condition Response Graphs with Subprocess is a tuple* $G = (E, \triangleright, \mathsf{Sub}, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$, *where*

(i) $\mathsf{Sub} : E \rightarrow \{0, 1\}$ *is a function defining subprocess events that can spawn multiple instances. An event* e *is subprocess event if* $\mathsf{Sub}(e) = 1$.

(ii) $(E, \triangleright, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ *is a nested DCR Graph with only one restriction that* $e \bullet\!\rightarrow e' \implies e \in Scope(e')$ *where*

$$Scope(e') = \begin{cases} \{e'' \mid \exists i \geq 1.\ \triangleright^1(e') = \triangleright^i(e'')\} & \textit{if } \mathsf{Sub}(\triangleright^k(e')) = 1 \textit{ for } k \geq 1 \\ E & \textit{otherwise} \end{cases}$$

With the addition of subprocess events, we impose a restriction on response relation in a DCR Graph, as an instance of a sub process event is created by executing an event which has a response relation to it. The item (ii) says that, response relation to an event whose ancestor/parent is a subprocess event ($\mathsf{Sub}(\triangleright^k(e')) = 1$ for $k \geq 1$), will only be allowed from the descendants of it's parent i.e self/siblings or descendants of them.

In order to explain the semantics of subprocesses in a better way, we will use a revised version of *prescribe medicine* example shown in the figure 4.8. The example contains a *prescribe* event modeling prescription of medicine by the doctor and a nested subprocess event *administer medicine*, which contains *sign*(signing

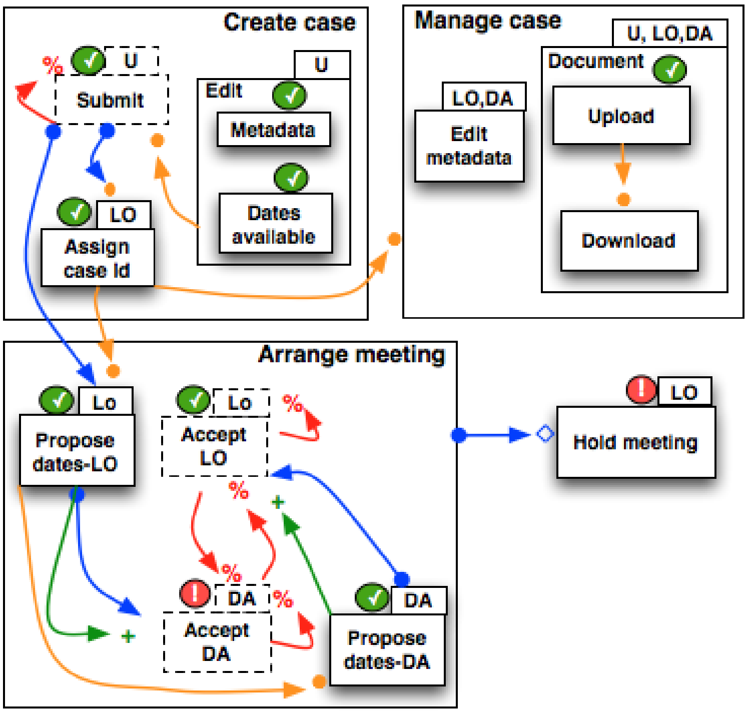


Figure 4.7: Case Handling Process Runtime After Accept Dates

a prescription by doctor), *remove* (canceling a prescription by doctor), *give* (giving medicine to patient by nurse) and *don't trust*(prescription not trusted by the nurse). The basic idea is that the doctor can prescribe any number of prescriptions and each prescription will be administered individually.

Graphically in DCR Graphs, subprocesses will be represented by marking the events with three parallel lines at the bottom (similar to the BPMN representation of multi-instance subprocess activities). Any event either an atomic event or nesting event (an event nested events) can be marked as a sub process event, but only sub process events that are required as responses can be instantiated to spawn new instances, as defined formally in the def 4.2.4. As shown in the example whenever the doctor prescribes a medicine a new instance of sub process *administer medicine* will be added to the process. The formal specification of the prescribe medicine example in Nested DCR Graph with subprocesses is given in the listing 4.3.

Listing 4.3: Formal specification of prescribe medicine example in Nested DCR Graphs with subprocesses.

A Nested DCR Graph with subprocesses $G = (E, \rhd, Sub, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ where

$E = \{\mathsf{prescribe}, \mathsf{administer\ medicine}, \mathsf{sign}, \mathsf{remove}, \mathsf{give}, \mathsf{don't\ trust}\}$

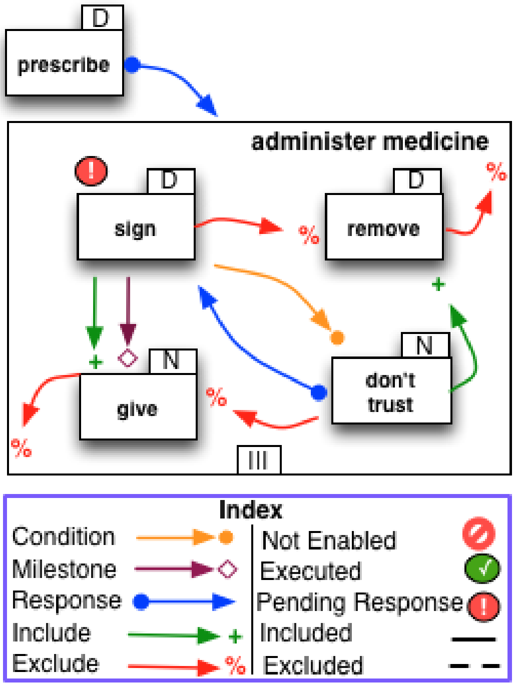


Figure 4.8: Prescribe medicine example with subprocesses

$\triangleright = \{(\text{sign}, \text{administer medicine}), (\text{remove}, \text{administer medicine}),$
$\quad (\text{give}, \text{administer medicine}), (\text{don't trust}, \text{administer medicine})\}$

$\text{Sub} = \{(\text{administer medicine}, 1), (\text{sign}, 0), (\text{remove}, 0), (\text{give}, 0), (\text{don't trust}, 0)\}$

$M = (\emptyset, \{\text{sign}\}, E)$

$\rightarrow\!\bullet = \{(\text{sign}, \text{don't trust})\}$

$\bullet\!\rightarrow = \{(\text{prescribe}, \text{administer medicine}), (\text{don't trust}, \text{sign})\}$

$\rightarrow\!\diamond = \{(\text{sign}, \text{give})\}$

$\rightarrow\!+ = \{(\text{sign}, \text{give}), (\text{don't trust}, \text{remove})\}$

$\rightarrow\!\% = \{(\text{sign}, \text{remove}), (\text{remove}, \text{administer medicine}), (\text{don't trust}, \text{give}),$
$\quad (\text{give}, \text{administer medicine})\}$

$L = \{(\text{prescribe}, D), (\text{sign}, D), (\text{remove}, D), (\text{give}, N), (\text{don't trust}, N)\}$

$l = \{(\text{prescribe}, (\text{prescribe}, D)), (\text{sign}, (\text{sign}, D)), (\text{remove}, ((\text{remove}, D)), (\text{give}, (\text{give}, N))$
$\quad (\text{don't trust}, (\text{don't trust}, N))\}$

4.2.2 Flattening of Nested DCR Graph with sub processes

To define the execution semantics for Nested Dynamic Condition Response Graphs with sub processes, we first define how to flatten a nested graph to the simpler DCR Graph with subprocesses. We define the level of nesting as $level(e) = k$ if $\triangleright^k(e)$ is defined and $\triangleright^{k+1}(e)$ is undefined.

Definition 4.2.2. *For a Nested DCR Graph with sub processes* $G = (E, \triangleright, Sub, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$*, define the underlying flat DCR Graph with subprocesses as*

$$G^\flat = (atoms(E), S_N, M, \rightarrow\bullet^\flat, \bullet\rightarrow^\flat, \rightarrow\diamond^\flat, \rightarrow+^\flat, \rightarrow\%^\flat, L, l)$$

where

1. $S_N : E \rightarrow \mathbb{N}^0$ *is a function mapping events to their subprocess nesting level as such that* $S_N(e) = \sum_{0 \leq k \leq level(e)} Sub(\triangleright^k(e))$

2. $\rightarrow^\flat = \trianglerighteq \rightarrow \trianglelefteq$ *for some relation* $\rightarrow \in \{\rightarrow\bullet, \rightarrow\diamond, \bullet\rightarrow, \rightarrow+, \rightarrow\%\}$

In the flatten graph, we introduced a function $S_N : E \rightarrow \mathbb{N}^0$ to keep track of the subprocess nesting level, which will be 0 for non-subprocess events and for others, it will be a summation of all subprocess flags till its top level parent event. Further, all the relations from and to the nested events will be expanded to include their descendant atomic events in the flattened graph (2).

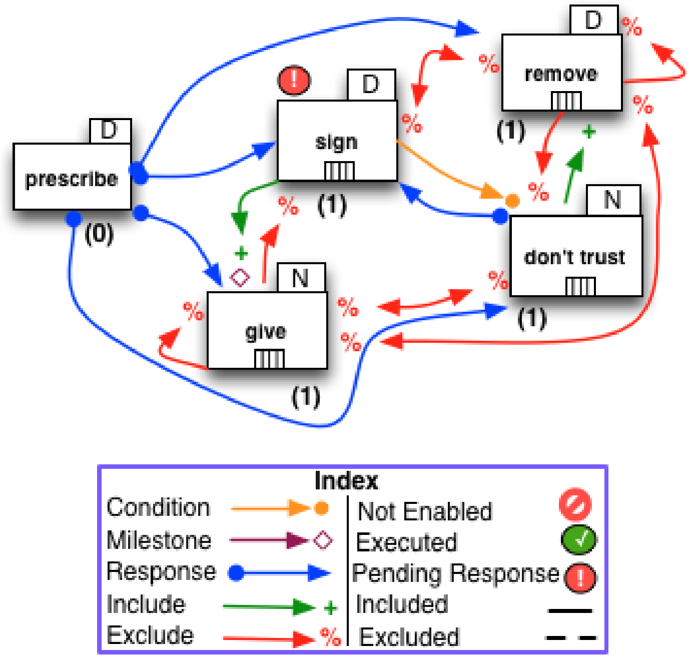


Figure 4.9: Flattened prescribe medicine example

The flattened DCR Graph is shown in the figure 4.9. On can see that all the relations are to the nested event are expanded to their children and also sub process nesting level is marked adjacent to the events. All the events whose subprocess nesting level greater than 0 are sub process events which can spawn new instances. Further listing 4.4 shows specification of flat underlying DCR Graph for the prescribe medicine Nested DCR Graph with sub processes given in the listing 4.3 that is flattened according to the def 4.2.2.

Listing 4.4: Flattened DCR graph for prescribe medicine example

Nested graph from listing 4.3 $G = (E, \rhd, Sub, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$

Underlying Flat DCR graph $G^{\flat} = (atoms(E), S_N, M, \rightarrow\bullet^{\flat}, \bullet\rightarrow^{\flat}, \rightarrow\diamond^{\flat}, \rightarrow+^{\flat}, \rightarrow\%^{\flat}, L, l)$
where

$atoms(E) = E \setminus \{\mathsf{administer\ medicine}\}$

$S_N = \{(\mathsf{prescribe}, 0), (\mathsf{sign}, 1), (\mathsf{remove}, 1), (\mathsf{give}, 1), (\mathsf{don't\ trust}, 1)\}$

$M = (\emptyset, \{\mathsf{sign}\}, E)$

$\rightarrow\bullet^{\flat} = \rightarrow\bullet$

$$\bullet\rightarrow^{\flat} = \bullet\rightarrow \cup \{(\mathsf{prescribe}, \mathsf{sign}), (\mathsf{prescribe}, \mathsf{remove}), (\mathsf{prescribe}, \mathsf{give}), (\mathsf{prescribe}, \mathsf{don't\ trust})\} \setminus \{(\mathsf{prescribe}, \mathsf{administer\ medicine})\}$$

$\rightarrow\diamond^{\flat} = \rightarrow\diamond$

$\rightarrow+^{\flat} = \rightarrow+$

$$\rightarrow\%^{\flat} = \rightarrow\% \cup \{(\mathsf{remove}, \mathsf{sign}), (\mathsf{remove}, \mathsf{remove}), (\mathsf{remove}, \mathsf{give}), (\mathsf{remove}, \mathsf{don't\ trust}), (\mathsf{give}, \mathsf{sign}), (\mathsf{give}, \mathsf{remove}), (\mathsf{give}, \mathsf{give}), (\mathsf{give}, \mathsf{don't\ trust})\} \setminus \{(\mathsf{remove}, \mathsf{administer\ medicine}), (\mathsf{give}, \mathsf{administer\ medicine})\}$$

4.2.3 Execution Sematics of DCR Graphs with Subprocesses

In this section, we go further and formalize in Def. 4.2.3, that an event e of a (flat) DCR Graph (with sub processes) is enabled when it's subprocess Index is 0, all its included condition events are executed, and none of milestones events are pending responses.

Definition 4.2.3. *For a (flat) DCR Graph with sub processes* $G = (E, S_N, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$*, and* $M = (Ex, Re, In)$ *we define that an event* $e \in E$ *is* enabled*, written* $M \vdash_G e$*, if* $S_N(e) = 0 \wedge e \in In \wedge (In\cap \rightarrow\!\bullet\, e \subseteq Ex) \wedge (In\cap \rightarrow\!\diamond\, e \subseteq E \backslash Re)$.

Now we will define execution semantics of an enabled event and describe the changes that will be made to a DCR Graph with sub processes.

Definition 4.2.4. *For a DCR Graph with sub processes* $G = (E, S_N, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$*, where* $M = (Ex, Re, In)$ *the result of executing e is* $G \oplus e = G'$ *where* $G' = (E', S'_N, M', \rightarrow\!\bullet', \bullet\!\rightarrow', \rightarrow\!\diamond', \rightarrow\!+', \rightarrow\!\%', L, l')$ *is a DCR Graph with sub processes such that*

(i) $E' = E \cup \{\mathrm{fresh}(e') \mid e \bullet\!\rightarrow e' \wedge S_N(e') > 0\}$

(ii) $S'_N(e_1) = \begin{cases} S_N(e') - 1 & \text{if } e_1 = \mathrm{fresh}(e') \\ S_N(e_1) & \text{if } e_1 \in E \end{cases}$

(iii) $l'(e_1) = \begin{cases} l(e') & \text{if } e_1 = \mathrm{fresh}(e') \\ l(e_1) & \text{if } e_1 \in E \end{cases}$

(iv) $e_1 \rightarrow' e_2$ *if*

- *(a)* $e_1 \rightarrow e_2$
- *(b) or* $e_i = \mathrm{fresh}(e'_i)$ *for* $i \in \{1, 2\} \wedge e'_1 \rightarrow e'_2$
- *(c) or* $e_1 = \mathrm{fresh}(e'_1) \wedge e'_1 \rightarrow e_2$ *where* $\rightarrow \in \{\rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%\}$
- *(d) or* $e_2 = \mathrm{fresh}(e'_2) \wedge e_1 \rightarrow e'_2$ *where* $\rightarrow \in \{\rightarrow\!\bullet, \rightarrow\!\diamond\}$

(v) $M' = (Ex, Re, In) \oplus e =_{def} \left(Ex \cup \{e\}, Re', In'\right)$ *where*

- *(a)* $Re' = (Re \setminus \{e\}) \cup \{e' \mid e' \in e \bullet\!\rightarrow \wedge S_N(e') = 0\} \cup \{\mathrm{fresh}(e') \in E' \setminus E \mid (e' \in Re)\}$
- *(b)* $In' = (In\cup\, e \rightarrow\!+) \cup \{\mathrm{fresh}(e') \in E' \setminus E \mid (e' \in In)\} \setminus e \rightarrow\!\%$

Def. 4.2.4 defines the changes to DCR Graph with sub processes when an enabled event is executed. The sub process events are instantiated to spawn new instances when they are responses to the executed event *e*.

First a new instance will be created for each subprocess event which is response to event *e* and update them to the events (E) set (i). Further, we use a temporary set (fresh) (which is initially empty for each event execution), to keep track of newly created events, as relations to the subprocess events are copied to the newly created instances with some restrictions. The subprocess nesting index (S_N) of newly created event will be one less than that of its subprocess event as stated in (ii). Finally, the set of labels (L) will remain the same, but the labelling function (l) will be updated by adding mapping of the new instances of subprocess events with the labels of parent subprocess events as shown in (iii).

The second step involves coping of relations from subprocess events to their newly created instances. All the relations in between the subprocess events that both are instantiated as part of event execution e, will be copied to their instances (ivb). Similarly all the relations from a subprocess events pointing to atomic events are copied from the new instances to the respective atomic events (ivc). But only condition and milestone relations pointing from atomic events to subprocess events are copied to the newly created instances of sub process events (ivd).

Finally, the marking will be updated by adding the executed event to the Ex set. The included events set (In) will be updated by including/excluding all the events that are included/excluded by the executing event e and all the new instances are also added, if their parent subprocess event is included (vb). Updates to the pending responses set (Re) are liftle bit different, as subprocess events are instantiated by a response relation. First of all, the executed event (e) will be taken out of the Re set and then all the atomic events which are responses to e are added. But in order to propagate required as a response from subprocess events to their instances, only those new instances whose parent subprocess event carries a initial pending response ($\mathsf{S}_N(e') > 0 \wedge e' \in e\bullet\!\!\rightarrow \wedge \mathsf{Re}$) will be added to Re set.

Lets us use the prescribe medicine example again to explain the execution semantics of DCR Graph with subprocesses. The figure 4.10 shows the prescribe medicine example after the execution of prescribe event. When the *prescribe* event gets executed, an instance of all the subprocess events which are responses for *prescribe* will be created, added to set of events and respective relations are copied to the newly created instances as defined in the def 4.2.4. The formal specification of prescribe medicine example after the execution of prescribe event is given in the listing 4.5.

Listing 4.5: Prescribe medicine example after execution of *prescribe*

The result of executing event **prescribe** is $\mathsf{G} \oplus$ **prescribe** $= \mathsf{G}'$ where

$\mathsf{G} = (\mathsf{E}, \mathsf{S}_N, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, \mathsf{l}) = \mathsf{G}^\flat$ from listing 4.4

$\mathsf{G}' = (\mathsf{E}', \mathsf{S}'_N, \mathsf{M}', \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L}, \mathsf{l}')$

Let's say fresh(**sign**) = **sign**[1], fresh(**remove**) = **remove**[1], fresh(**give**) = **give**[1] and
fresh(**don't trust**) = **don't trust**[1].

$\mathsf{E}' = \mathsf{E} \cup \{$**sign**[1], **remove**[1], **give**[1], **don't trust**[1]$\}$

$\mathsf{S}_N{}' = \mathsf{S}_N \cup \{($**sign**[1], 0), (**remove**[1], 0), (**give**[1], 0), (**don't trust**[1], 0)$\}$

$\mathsf{M}' = (\{$**prescribe**$\}, \{$**sign**, **sign**[1]$\}, \mathsf{E}')$

$\rightarrow\bullet' = \rightarrow\bullet \cup \{($**sign**[1], **don't trust**[1]$)\}$

$\bullet\rightarrow{}' = \bullet\rightarrow \cup \{($**don't trust**[1], **sign**[1]$)\}$ (we copy only responses from subprocess events)

$\rightarrow\diamond{}' = \rightarrow\diamond \cup \{($**sign**[1], **give**[1]$)\}$

$\rightarrow+{}' = \rightarrow+ \cup \{($**sign**[1], **give**[1]$)$, (**don't trust**[1], **remove**[1]$)\}$

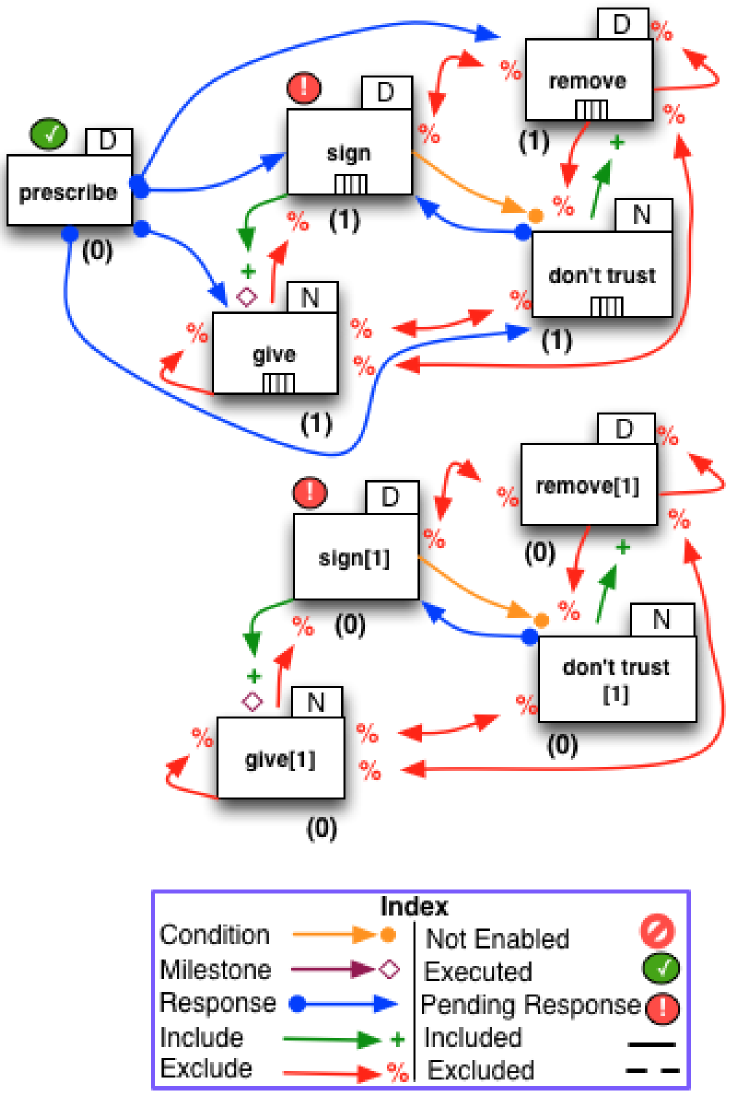


Figure 4.10: Prescribe medicine example with an instance of subprocess

$\rightarrow\%' = \rightarrow\% \cup \{(\mathsf{sign}[1], \mathsf{remove}[1]), (\mathsf{remove}[1], \mathsf{sign}[1]), (\mathsf{remove}[1], \mathsf{remove}[1]),$
$(\mathsf{remove}[1], \mathsf{give}[1]), (\mathsf{remove}[1], \mathsf{don't\ trust}[1]), (\mathsf{don't\ trust}[1], \mathsf{give}[1]),$
$(\mathsf{give}[1], \mathsf{sign}[1]), (\mathsf{give}[1], \mathsf{remove}[1]), (\mathsf{give}[1], \mathsf{give}[1]), (\mathsf{give}[1], \mathsf{don't\ trust}[1])\}$

$l' = l \cup \{(\mathsf{prescribe}[1], (\mathsf{prescribe}, \mathsf{D})), (\mathsf{sign}[1], (\mathsf{sign}, \mathsf{D})), (\mathsf{remove},$
$((\mathsf{remove}, \mathsf{D})), (\mathsf{give}[1], (\mathsf{give}, \mathsf{N})), (\mathsf{don't\ trust}[1], (\mathsf{don't\ trust}, \mathsf{N}))\}$

Definition 4.2.5. *For a Nested DCR Graph with sub processes* $\mathsf{G} = (\mathsf{E}, \triangleright, \mathsf{Sub}, \mathsf{M}, \rightarrow\bullet$ $, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ *with* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *we define that* $e \in \mathsf{atoms}(\mathsf{E})$ *is enabled, written* $\mathsf{M} \vdash_{\mathsf{G}} e$*, if the underlying flat DCR Graph with with sub processes* $\mathsf{M} \vdash_{\mathsf{G}^\flat} e$*.*

Similarly, the result of executing event e, written as $\mathsf{M} \oplus_{\mathsf{G}} e$ *is same as* $\mathsf{M} \oplus_{\mathsf{G}^\flat} e$*.*

4.3 DCR Graphs with Data

In this section we will introduce an another extension to the (Nested) DCR Graphs, data. Here we consider a global data store as a set of variables that are shared among the events of a DCR graph. The variables that can be read and/or assigned by an event are explicitly defined in the model. We propose a simple model where we allow integer as data types for the variables, as other primitive data types can be easily encoded as integers.

The motivation for the extension of data also originated from the PhD candidate's visit to IBM Research, New York as part of stay abroad, to study the relation between DCR Graphs and IBM Research's declarative process model Business Artifacts with Guard-Stage-Milestone (GSM Model) life cycles [Hull *et al.* 2011a].

Definition 4.3.1. *We define integer expressions as,*

$$\mathsf{iexp} ::= \mathbb{Z} \mid \mathsf{intvar} \mid \mathsf{iexp}\ \mathit{intop}\ \mathsf{iexp}$$

where $\mathit{intop} \in \{+, -, *, \%\}$

Definition 4.3.2. *A Boolean expression is defined as*

$$\mathsf{bexp} ::= \top \mid \bot \mid \mathsf{iexp}\ \mathsf{OP}\ \mathsf{iexp}$$

where $\mathsf{OP} \in \{=, <, >\}$*,* $\top$ *is true and* $\bot$ *is false.*

In this extension to DCR Graphs, we add basic support for data as a global of shared integer variables and the variables are modified and read by the events. Further we also introduce notion of Boolean expressions built over the values of variables confirming to syntax mention in the Def. 4.3.2. We also propose the notion of guards (similar to GSM model, but the semantics are not exactly same) mapped over the set of Boolean expressions which act as conditions on the events and relations.

Formally we first define a DCR Graph extended with data as follows and in the next section we will also extend Nested DCR Graphs with data .

Definition 4.3.3. *A Dynamic Condition Response Graph with Data is a tuple* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \mathbb{V}, \mathbb{B}\mathsf{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{L}, l, \mathsf{guard}, \mathsf{ar}, \mathsf{read}, \mathsf{assign})$ *where*

(i) E *is the set of* events*, ranged over by e,*

(ii) $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \sigma) \in \mathcal{M}(G)$ *is the* marking *containing a set of executed events* (Ex)*, a set of pending responses* (Re)*, a set of currently included events* (In) *and current valuation of variables* $(\sigma : \mathbb{V} \rightarrow Int)$*. The markings set* $\mathcal{M}(G) =_{def} \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times Int^{\mathbb{V}}$ *is a set of all markings where* $Int^{\mathbb{V}}$ *is the set of all valuations of variables* $\mathbb{V}$*.*

(iii) $\mathbb{V}$ *is the set of integer* variables *that represents a global data store and ranged over by* v

(iv) $\mathbb{B}$exp *is a set of* Boolean expressions *ranged over by* bexp,

(v) $\rightarrow\!\bullet \subseteq \mathsf{E} \times \mathsf{E}$ *is the condition relation*

(vi) $\rightarrow\!\diamond \subseteq \mathsf{E} \times \mathsf{E}$ *is the milestone relation*

(vii) $\bullet\!\rightarrow \subseteq \mathsf{E} \times \mathbb{B}\mathrm{exp} \times \mathsf{E}$ *is a guarded* response *relation,*

(viii) $\rightarrow\!+ \subseteq \mathsf{E} \times \mathbb{B}\mathrm{exp} \times \mathsf{E}$ *is a guarded* include *relation,*

(ix) $\rightarrow\!\% \subseteq \mathsf{E} \times \mathbb{B}\mathrm{exp} \times \mathsf{E}$ *is a guarded* exclude *relation,*

(x) L *is the set of* labels

(xi) $l : \mathsf{E} \rightarrow \mathcal{P}(\mathsf{L})$ *is a labeling function mapping events to sets of labels.*

(xii) guard : $\mathsf{E} \rightarrow \mathbb{B}\mathrm{exp}$ *is a function mapping events to Boolean expressions.*

(xiii) ar $= \mathsf{E} \rightarrow \mathbb{N}$ *is the arity of events indicating the number of input variables for an event.*

(xiv) read : $\mathsf{E} \rightarrow \mathcal{P}_{fin}(\mathbb{V})$ *is a function specifying the variables that an event can read.*

(xv) assign : $\mathsf{E} \rightarrow (\mathbb{V} \rightharpoonup_{fin} \mathrm{iexp})$ *is a function indicating which variables can an event modify, such that* $\mathrm{assign}(e) =< v_1 = ex_1, .., v_n = ex_n >$ *where* $ex_1, .., ex_n$ *are integer expressions. Further the variables that are part of an expression* $Var(ex_i) \subseteq \mathrm{read}(e) \cup \{\$j \mid 0 \leq j \leq \mathrm{ar}(e)\}$.

An event labelled with an action represents an execution of a (human or automated) task/activity/action in the workflow process and each event can be mapped to more than one label. The marking M (ii) defines the runtime state of DCR Graph and consists of a set capturing which events have *previously been executed* (Ex), which events are *pending responses* (Re), which events are currently included (In) and finally current valuation of variables ($\sigma : \mathbb{V} \rightarrow \mathrm{Int}$) in the global data store ($\mathbb{V}$).

Further, (iv) defines a set of well-formed *Boolean expressions* formed according to syntax defined in Def 4.3.2. Note that a Boolean expression can refer to the values of the data variables (for example $v_1 > 5$) and they are always evaluated in the context of current marking.

Further, the condition (v) and milestone (vi) relations are same as normal DCR Graph, but the response, include and exclude relations (vii - ix) are now guarded with Boolean expressions (which can refer to data variables) attached to them. In case of guarded relations, the relations will have additional constraint saying that the Boolean expression must be true, in order for the relation take an effect. For example, if two events e, e' are related by a guarded response relation ($e\bullet\!\xrightarrow{\mathrm{bexp}} e'$), when the event e get executed the event e' will be added to set of responses (Re) only if the bexp is true. Finally, guard is a function mapping events to Boolean expressions as shown in (xii).

Further, (xiii) defines *arity* of events which specifies the number of input variables that an event can have and for auto events the arity is 0, as auto events can not have any input variables. Similarly *read* (xiv) is a function mapping events to finite set of variables that an event can read values of variables. Moreover *assign* (xv), is a function mapping events to expressions for assigning values to variables and we also specify that variables of an expression should be either part of variables that an event can read or input, to make sure in any case an event can not assign a value to a variable, for which neither the event does have read mapping or part of it's input variables.

4.3.1 Nested DCR Graphs with Data

Now, we will go further and give the formal definition of a Nested Dynamic Condition Response Graph with Data as follows,

Definition 4.3.4. *A Nested Dynamic Condition Response Graph with Data is a tuple* $\mathrm{G} = (\mathrm{E}, \rhd, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$, *where*

(i) $(\mathrm{E}, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$ *is a Dynamic Condition Response Graph with Data and*

(ii) $\rhd : \mathrm{E} \rightharpoonup \mathrm{E}$ *is a partial function mapping an event to its super-event (if defined), and*

(iii) $\mathrm{M} \in \mathcal{P}(\mathrm{atoms(E)}) \times \mathcal{P}(\mathrm{atoms(E)}) \times \mathcal{P}(\mathrm{atoms(E)}) \times Int^{\mathbb{V}}$, *where* $\mathrm{atoms(E)} = \mathrm{E} \backslash \{e \in \mathrm{E} \mid \exists e' \in \mathrm{E}. \rhd (e') = e\}$ *is the set of atomic events.*

(iv) $\mathrm{E_s} \subseteq \mathrm{atoms(E)}$

We write $e \rhd e'$ *if* $e' = \rhd^k(e)$ *for* $0 < k$ *and write* $e \trianglerighteq e'$ *if* $e \rhd e'$ *or* $e = e'$, *and* $e \trianglelefteq e'$ *if* $e' \rhd e$ *or* $e = e'$. *We require that the resulting relation,* $\trianglerighteq \subset E \times E$, *referred to as the nesting relation, is a well founded partial order. We also require that the nesting relation is consistent with respect to dynamic inclusion/exclusion in the following sense: If* $e \rhd e'$ *or* $e' \rhd e$ *then* $e \rightarrow\!+ \cap e' \rightarrow\!\% = \emptyset$.

To define the execution semantics for a Nested Dynamic Condition Response Graph with Data, we first define how to flatten a nested graph to DCR Graph with Data in def 4.3.5. Essentially, all relations to and/or from nested events are extended to sub events, and then only the atomic events are preserved. Further, we define the level of nesting as $level(e) = k$ if $\rhd^k(e)$ is defined and $\rhd^{k+1}(e)$ is undefined.

Definition 4.3.5. *For a Nested Dynamic Condition Response Graph with Data* $\mathrm{G} = (\mathrm{E}, \rhd, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$, *we define the underlying flat DCR Graph with Data as*
$\mathrm{G}^\flat = (\mathrm{atoms(E)}, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet^\flat, \bullet\!\rightarrow^\flat, \rightarrow\!\diamond^\flat, \rightarrow\!+^\flat, \rightarrow\!\%^\flat, \mathrm{L}, l, \mathrm{ar}, \mathrm{guard}^\flat, \mathrm{read}^\flat, \mathrm{assign}^\flat)$ *where*

(i) $rel^{\flat} = \trianglerighteq rel \trianglelefteq$ *for some relation* $rel \in \{\rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%\}$

(ii) $\mathrm{guard}^{\flat}(e) = \bigwedge_{0 \leq k \leq level(e)} \mathrm{guard}(\triangleright^{k}(e)$

(iii) $fun^{\flat} = fun \setminus \{(e, fun(e)) \mid \exists e' \in \mathrm{E}.\ \triangleright (e') = e\} \cup \{(e_i, fun(e_i)) \mid 0 < i \leq k \wedge \triangleright^{k}(e_i) = e\}$ *for* $fun \in \{\mathrm{read}, \mathrm{assign}\}$.

In flattening a nested DCR Graph with data into a DCR Graph with data, all the relations from the super events will be propagated to their decedent events as shown in (i), like in case of nested DCR Graphs (4.1.3). Similarly, Boolean expressions of super events are also propagated to their decedents and therefore a Boolean expression of a nested event (or atomic event) will be in conjunction of all such expressions inherited from its super event (ii). Furthermore, *assign* and *read* function mappings for super events are expanded to their decedents as shown in (iii).

We now define when an event e is enabled in DCR Graph with data in def 4.3.6. An event e is enabled if it is included in current marking ($e \in \mathrm{In}$), all its condition events are executed ($\rightarrow\!\bullet\ (e) \in \mathrm{Ex}$), all its milestone events are not in set of pending responses ($\rightarrow\!\diamond\ (e) \in \mathrm{E}\setminus\mathrm{Re}$) and the Boolean expression assigned to the event should be true when evaluated in the context of current marking ($[\![\mathrm{guard}(e)]\!]_{\mathrm{M}}$).

Definition 4.3.6. *For a Dynamic Condition Response Graph with Data* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$, *and* $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In}, \sigma)$ *we define that an event* $e \in \mathrm{E}$ *is* enabled, *written* $\mathrm{M} \vdash_{\mathrm{G}} e$*, if*

(i) $e \in \mathrm{In}$

(ii) $\rightarrow\!\bullet\ (e) \in \mathrm{Ex}$

(iii) $\rightarrow\!\diamond\ (e) \in \mathrm{E} \setminus \mathrm{Re}$

(iv) $[\![\mathrm{guard}(e)]\!]_{\mathrm{M}}$

We will now define the changes to marking when an enabled event is executed.

Definition 4.3.7. *For a Dynamic Condition Response Graph with Data with data* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \mathbb{V}, \mathbb{B}\mathrm{exp}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$ *with a marking* $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In}, \sigma)$ *and an enabled event* $\mathrm{M} \vdash_{\mathrm{G}} e$ *, the result of executing* e *is* $\mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ *where the updated marking* $\mathrm{M}' = (\mathrm{Ex}', \mathrm{Re}', \mathrm{In}', \mathrm{update}(\sigma, \mathrm{assign}(e)))$ *such that,*

(i) $\mathrm{Ex}' = \mathrm{Ex} \cup \{e\}$

(ii) $\mathrm{Re}' = \mathrm{Re} \setminus \{e\} \cup \{e' \mid e \bullet\!\rightarrow (\mathrm{bexp}, e') \wedge [\![\mathrm{bexp}]\!]_{\mathrm{M}}\}$

(iii) $\mathrm{In}' = \big(\mathrm{In} \cup \{e' \mid e \rightarrow\!+ (\mathrm{bexp}, e') \wedge [\![\mathrm{bexp}]\!]_{\mathrm{M}}\}\big) \setminus \{e' \mid e \rightarrow\!\% (\mathrm{bexp}, e') \wedge [\![\mathrm{bexp}]\!]_{\mathrm{M}}\}$

(iv) $\mathrm{update}(\sigma, \mathrm{assign}(e)) : \mathbb{V} \rightarrow \mathrm{Int}$ *is a function updating data store such that*

$$\sigma'(v) = \begin{cases} \sigma(v) & \textit{if } v \notin dom(\mathrm{assign}(e)) \\ [\![\mathrm{assign}(e)(v)]\!]_{M} & \end{cases}$$

Def. 4.3.7 above then defines the change of the marking when an enabled event is executed: First the event is added to the set of executed events (i). Further the set of pending responses (Re) will be updated by removing the event *e* and adding all the events which are responses for event *e* with guard (Boolean expression) associated with response relation evaluated to true (ii). Similarly the set of included events is updated by adding/removing events which are included/excluded by event *e* with the guard (Boolean expression) associated with relation is evaluated to true (ii). Finally, the current valuation of variable (σ) will be updated with new data values assigned by the event *e* (iv).

Definition 4.3.8. *For a Nested Dynamic Condition Response Graph with Data* $G = (E, \rhd, M, \mathbb{V}, \mathbb{B}exp, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l, \mathrm{guard}, \mathrm{ar}, \mathrm{read}, \mathrm{assign})$, *where* $M = (Ex, Re, In, \sigma)$ *we define that* $e \in \mathrm{atoms}(E)$ *is enabled, written* $M \vdash_G e$, *if the underlying flat DCR Graph with data* $M \vdash_{G^\flat} e$. *Similarly, the result of executing event e, written as* $M \oplus_G e$ *is same as* $M \oplus_{G^\flat} e$.

4.3.2 Healthcare Example in DCR Graphs with Data

In this section, we will use our running example *prescribe medicine* to explain the semantics of DCR Graphs with data.

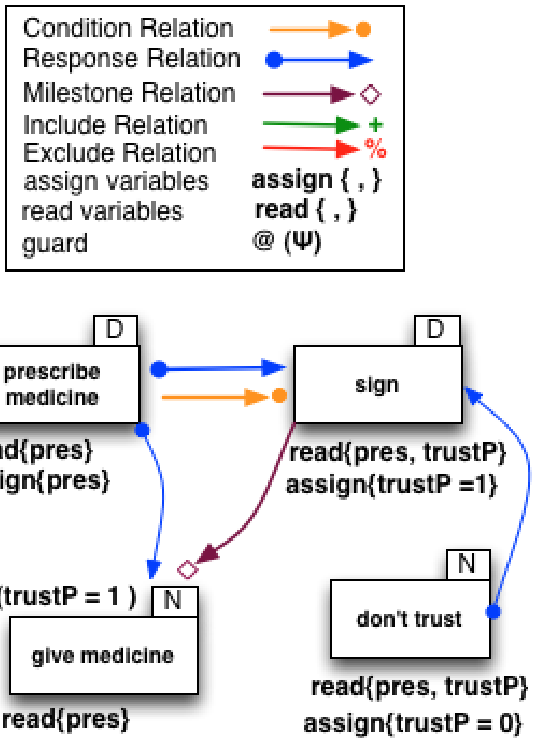


Figure 4.11: Prescribe medicine example in DCR Graphs with data.

Figure 4.11 shows *prescribe medicine* example modeled using DCR Graphs with data, where we have the same four events *prescribe medicine*, *sign*, *give medicine* and *don't trust* and they mean same as what we have discussed in the Sec. 3.4. Now we have variables from a shared data store and the variables that can be read by an event are marked under the events with a function *read*, which maps each event to set of variables. For example the variable *pres* meant for storing the values of prescription and we can notice that all the events in example have access to read the variable *pres*, where as only *prescribe medicine* event can assign a value for the *pres* variable, since the event has the mapping for the variable in the *assign* function. Further, the arity (*ar*) of *prescribe medicine* event is 1 (not shown in the figure), which indicates that the event has one input field, whose value will be assigned to variable *pres*. The mapping between the input fields and variables are defined in the *assign* function (Def. 4.3.3-xv).

Similarly, *trustP* (meaning trust prescription) is a variable that can assigned a value by *sign* and *don't trust* events. When the *sign* event gets executed, it assigns a value 1 to the *trustP* variable. In this case, we have a predefined assignment meaning that whenever *sign* event gets executed, it will always assign 1 to *trustP*. Furthermore the *sign* event always uses predefined assignment, hence there will no input field, which means the arity of *sign* event is 0. As explained before, Boolean expressions ($\mathbb{B}$exp) can be defined over the values of variables and one such expression is *trustP = 1*, which is defined as guard for the event *give medicine* with a syntax *@(trustP = 1)*. The guards for events and relations are always evaluated in the context of the current marking, and the marking in DCR Graphs with data now includes the valuation set of variables in addition to the standard three sets.

Let's consider an execution <*prescribe medicine, sign, don't trust* > where the doctor has prescribed a medicine (by assigning some value to *pres*) and executed the *sign* event, (which automatically assigns a value 1 to *trustP* variable). Further when the nurse executes *don't trust* event, the *trustP* will be assigned to value 0, making the guard *@(trustP = 1)* evaluates to false. In that context, the event *give medicine* will not be enabled because of it's guard *@(trustP = 1)* evaluates to false. Later the doctor may choose to assign a new value for *pres* and then execute *sign* or else he may simply choose to re-execute the *sign* event, making the value of *trustP* to 1, which will make the event *give medicine* enabled.

One may observe in the Def. 4.3.3 that only response ($\bullet\!\!\rightarrow$), include ($\rightarrow\!+$) and exclude ($\rightarrow\!\%$) are defined as guarded relations, but condition ($\rightarrow\!\bullet$) and milestone ($\rightarrow\!\diamond$) relations do not have any guards, as they are blocking relations. In this figure 4.11, we have not shown any guards on the relations, therefore all the guarded relations will have a guard mapped to true ($\top$). In case if there are guards on the relations, then the guards will be evaluated in the current marking before updating the marking for the guarded relation and in case if the guard evaluated to false, then no updates will be applied to the marking for that relation.

4.4 Summary

In this chapter, we have given a conservative extension of DCR Graphs to allow for nested sub-graphs motivated from guided by a case study carried out jointly with our industrial partner in the section 4.1. Later, in section 4.2 we have introduced another important extension multi instance sub processes to model replicated behavior in DCR Graphs. Finally, in the section 4.3, we have added support for data to the DCR Graphs based on the motivation from both the case management case study and from the study of relating DCR Graphs with the IBM Research's declarative workflow business artifacts with guard-stage-milestone life cycles model.

Chapter 5

Distribution of DCR Graphs

In the previous chapter (4), we have introduced several extensions of DCR Graphs and in this chapter we will introduce a technique to distribute DCR Graphs as a set of local components to model local behavior and to guarantee that the behavior in local components is consistent with the global behavior. First we will introduce and define the notion of projection and composition on DCR Graphs in section 5.3, then we will introduce semantics of synchronous distributed execution in sec 5.3.3 by defining the notion of networks of DCR Graphs.

Further we will extend the distribution technique to the *nested* DCR Graphs in the section 5.4 and also we exemplify the distribution technique of nested DCR Graphs using healthcare workflow that was introduced in the case study 2.1.3. Finally, we will also prove the theorems (thm 5.3.1 and thm 5.4.1) for distributed execution of DCR Graphs and *nested* DCR Graphs saying that the behavior in global graph is bisimilar to the behavior in the network of projected graphs.

5.1 Introduction

In general the commercial workflow implementations are based on a centralized workflow manager controlling the execution of the entire, global workflow. However, workflows often span different units or departments within the organization, e.g. the pharmacy and the patient areas, or even cross boundaries of different organizations (e.g. different hospitals). In some situations it may be very relevant to execute the local parts of the workflow on a local (e.g. mobile) device without permanent access to a network, e.g. during preparation of the medicine in the pharmacy. Also, different organizations may want to keep control of their own parts of the workflow and not delegate the management to a central service. This motivates the ability to split the workflow in separate components, each only referring to the activities relevant for the local unit and being manageable independently of the other components.

A model-driven software engineering approach to distributed information systems typically include both *global* models describing the collective behavior of the system being developed and *local* models describing the behavior of the individual peers or components.

The global and local descriptions should be consistent. If the modeling languages have formal semantics and the local model language support composition of individual processes, the consistency can be formally established, which we will refer to as the *consistency problem*: Given a global model and a set of local models, is the behavior of the composition of the local models consistent with the global model? In order to support *top-down* model-driven engineering starting from the global model, one should address the more challenging *distributed synthesis problem*: Given a global model and some formal description of how the model should be distributed, can we synthesize a set of local processes with respect to this distribution which are consistent to the the global model?

In past work, as discussed in related work (sec 5.2), the result of the distributed synthesis have been a network of local processes described in an imperative process model, e.g. as a network of typed pi-calculus processes or a product automaton. The global process description has either been given declaratively, e.g. in some temporal logic, or imperatively, e.g. as a choreography or more generally a transition system.

In this chapter, we address the distributed synthesis problem in a setting where both the global and the local processes are described *declaratively* as DCR Graphs.

To safely distribute a DCR Graph we first define (Def. 5.3.1, Sec. 5.3.1) a new general notion of *projection* of DCR Graphs relative to a subset of labels and events. The key point is to identify the set of events that must be communicated from other processes in the network in order for the state of the local process to stay consistent with the global specification (Prop. 5.3.1-5.3.3, Sec. 5.3.1). To also enable the reverse operation, building global graphs from local graphs, we then define the composition of two DCR Graphs, essentially by gluing joint events. As a sanity check we prove (Prop. 5.3.4, Sec. 5.3.2) that if we have a collection of projections of a DCR Graph that cover the original graph (Def. 5.3.5, Sec. 5.3.2) then the composition yields back the same graph. We then finally proceed to the main technical result, defining networks of synchronously communicating DCR Graphs and stating (in Thm. 5.3.1, Sec. 5.3.3) the correspondence between a global process and a network of communicating DCR Graphs obtained from a covering projection (relying on Prop. 5.3.1-5.3.3). Throughout the paper we exemplify the distribution technique on a simple cross-organizational process identified within a case study (sec 4.1.3) carried out jointly with Exformatics A/S using DCR Graphs for model-driven design and engineering of an inter-organizational case management system.

Further in the Sec. 5.4, we extend the notion of projection on *nested* DCR Graphs and provide the semantics for distributed execution on *nested* DCR Graphs. Further we then proceed to the main technical result on *nested* DCR Graphs, stating (in Thm. 5.4.1) the correspondence between a global process and a network of communicating *nested* DCR Graphs obtained from a covering projection (relying on Prop. 5.4.1).

In this chapter, we will follow the following notation.

Notation: For a set A we write $\mathcal{P}(A)$ for the power set of A. For a binary relation $\rightarrow \subseteq A \times A$ and a subset $\xi \subseteq A$ of A we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{a \in A \mid (\exists a' \in \xi \mid a \rightarrow a')\}$ and the set $\{a \in A \mid (\exists a' \in \xi \mid a' \rightarrow a)\}$ respectively. Also, we write $\rightarrow^{-1}$ for the inverse relation. Finally, for a natural number k we write $[k]$ for the set $\{1, 2, \dots, k\}$.

5.2 Related Work

There are many researchers [van der Aalst 1999a, Kindler *et al.* 2000, ter Hofstede *et al.* 2003, van der Aalst *et al.* 2010b, Aalst & Weske 2001, van der Aalst 2003, Martens 2005] who have explicitly focussed on the problem of verifying the correctness of inter-organizational workflows in the domain of petri nets. In [van der Aalst 1999a], message sequence charts are used to model the interaction between the participant workflows that are modeled using petri nets and the overall workflow is checked for consistency against an interaction structure specified in message sequence charts. In [Kindler *et al.* 2000] Kindler et. al. followed a similar but more formal and concrete approach, where the interaction of different workflows is specified using a set of scenarios given as sequence diagrams and using criteria of local soundness and composition theorem, guaranteed the global soundness of an inter-organizational workflow. The authors in [ter Hofstede *et al.* 2003] proposed *Query Nets* based on predicate/transition petri nets to guarantee global termination, without the need for having the global specification. The work on workflow nets [Aalst & Weske 2001, van der Aalst 2003] use a P2P (Public-To-Private) approach to partition a shared public view of an inter-organizational workflow over its participating entities and projection inheritance is used to generate a private view that is a subclass to the relevant public view, to guarantee the deadlock and livelock freedom. Further a more liberal and a weaker notion than projection inheritance, *accordance* has been used in [van der Aalst *et al.* 2010b] to guarantee the weak termination in the multiparty contracts based on open nets.

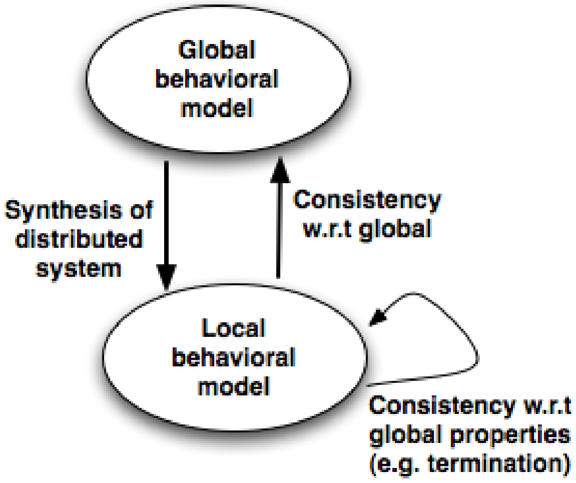

Global behavioral model
Synthesis of distributed system
Consistency w.r.t global
Local behavioral model
Consistency w.r.t global properties (e.g. termination)

Figure 5.1: Key problems studied in related work

Modeling global behavior as a set of conversations among participating services has been studied by many researchers [Fu *et al.* 2004b, Yi & Kochut 2004a, Rinderle *et al.* 2006, Wodtke & Weikum 1997, Bravetti & Zavattaro 2007, Bravetti & Zavattaro 2009] in the area business processes. An approach based on guarded automata studied in [Fu *et al.* 2004b], for the realizability analysis of conversation protocols, whereas the authors in [Yi & Kochut 2004a] used colored petri nets to capture the complex conversations. A framework for calculating and controlled propagation of changes to the process choreographies based on the modifications to partner's private processes has been studied in [Rinderle *et al.* 2006]. Similarly, but using process calculus to model service contracts, Bravetti-Zavattaro proposed conformance notion for service composition in [Bravetti & Zavattaro 2007] and further enhanced their correctness criteria in [Bravetti & Zavattaro 2009] by the notion of strong service compliance.

Researchers [Fdhila & Godart 2009, Nanda *et al.* 2004, Khalaf & Leymann 2006, Mitra *et al.* 2008] in the web services community have been working on web service composition and decentralized process execution using BPEL [OASIS WSBPEL Technical Committee 2007] and other related technologies to model the web services. A technique to partition a composite web service using program analysis was studied in [Nanda *et al.* 2004] and on the similar approach, [Khalaf & Leymann 2006] explored decomposition of a business process modeled in BPEL, primarily focussing on P2P interactions . Using a formal approach based on I/O automata representing the services, the authors in [Mitra *et al.* 2008] have studied the problem of synthesizing a decentralized choreography strategy, that will have optimal overhead of service composition in terms of costs associated with each interaction.

The derivation of descriptions of local components from a global model has been researched for the imperative choreography language WS-CDL in the work on structured communication-centred programming for web services by Carbone, Honda and Yoshida [Carbone *et al.* 2007]. To put it briefly, the work formalizes the core of WS-CDL as the global process calculus and defines a formal theory of end-point projections projecting the global process calculus to abstract descriptions of the behavior of each of the local "end-points" given as pi-calculus processes typed with session types.

A methodology for deriving process descriptions from a business contract formalized in a formal contract language was studied in [Milosevic *et al.* 2006], while [Sadiq *et al.* 2006] proposes an approach to extract a distributed process model from collaborative business process. In[Fdhila *et al.* 2009, Fdhila & Godart 2009], the authors have proposed a technique for the flexible decentralization of a process specification with necessary synchronization between the processing entities using dependency tables, where as the authors in [Dong *et al.* 2000] presented a framework for optimizing the physical distribution of workflow schemas based on the families of communicating flow charts.

In [Castellani *et al.* 1999, Heljanko & Stefanescu 2005, Mukund 2002] foundational work has been made on synthesizing distributed transition systems from global specification for the models of synchronous product and asynchronous automata[Zielonka 1987]. In [Mukund 2002] Mukund categorized structural and behavioral characterizations of the synthesis problem for synchronous and loosely cooperating communication systems based on three different notions of equivalence: state space, language and bisimulation equivalence. Further Castellani et. al. [Castellani *et al.* 1999] characterized when an an arbitrary transition system is isomorphic to its product transition systems with a specified distribution of actions and they have shown that for finite state specifications, a finite state distributed implementation can be synthesized. Complexity results for distributed synthesis problems for the three notions of equivalences were studied in [Heljanko & Stefanescu 2005].

Many commercial and research workflow management systems also support distributed workflow execution and some of them even support ad-hoc changes as well. ADEPT [Reichert & Bauer 2007], Exotica [Mohan *et al.* 1995], ORBWork [Das *et al.* 1996], Rainman [Paul *et al.* 1997] and Newcastle-Nortel [Shrivastava *et al.* 1998] are some of the distributed workflow management systems. A good overview and discussion about distributed workflow management systems can be found in [Reichert *et al.* 2009, Ranno & Shrivastava 1999].

So far the formalisms discussed above are more or less confined to imperative modeling languages such as Petri nets, workflow/open nets and automata based languages. To the best of our knowledge, there exists very few works [Fahland 2007, Montali 2010] that have studied the synthesis problem in declarative modeling languages and none where both the global and local processes are given declaratively. In [Fahland 2007], Fahland has studied synthesizing declarative workflows expressed in DecSerFlow [van der Aalst & Pesic 2006b] by translating to Petri nets. Only a predefined set of DecSerFlow constraints are used in the mapping to the Petri nets patterns, so this approach has a limitation with regards to the extensibility of the DecSerFlow language. On the other hand, in [Montali 2010] Montali has studied the composition of ConDec [van der Aalst & Pesic 2006a] models with respect to conformance with a given choreography, based on the compatibility of the local ConDec models. But his study was limited to only composition, whereas the problem of synthesizing local models from a global model has not been studied.

5.3 DCR Graphs - Projection and Composition

In this section we define projections and compositions of DCR Graphs. In Sec. 5.3.1 below we define the notion of projection of a DCR Graphs, restricting the graph to a subset of the events and labels, and in Sec. 5.3.2 we define the technique for binary composition of two DCR Graphs, to get a global DCR Graph.

5.3.1 Projection

First we define how to project a DCR Graph G with respect to a *projection parameter* $\delta = (\delta_E, \delta_L)$ where $\delta_E \subseteq E$ is a subset of the events of G and $\delta_L \subseteq L$ is a subset of the labels.

Intuitivly, the projection $G_{|\delta}$ contains only those events and relations that are relevant for the execution of events in δ_E and the labeling is restricted to the set δ_L. This includes both the events in δ_E and any other event that can affect the marking, or ability to execute of an event in δ_E through one or more relations. The technical difficulty is to infer the events and relations not in δ_E, referred to as *external events* below, that should be included in the projection because they influence the execution of the workflow restricted to the events in δ_E.

Definition 5.3.1. *If* $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathsf{L}, l)$ *then* $\mathsf{G}_{|\delta} = (\mathsf{E}_{|\delta}, \mathsf{M}_{|\delta}, \rightarrow\!\bullet_{|\delta}, \bullet\!\rightarrow_{|\delta}, \rightarrow\!\diamond_{|\delta}, \rightarrow\!+_{|\delta}, \rightarrow\!\%_{|\delta}, \delta_\mathsf{L}, l_{|\delta})$ *is the projection of* G *with respect to* $\delta \subseteq \mathsf{E}$ *where:*

(i) $\mathsf{E}_{|\delta} = \rightarrow \delta_\mathsf{E}$, *for* $\rightarrow = \bigcup_{c \in C} c$, *and* $C = \{\mathrm{id}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \bullet\!\rightarrow\rightarrow\!\diamond, \rightarrow\!+\rightarrow\!\bullet, \rightarrow\!\%\rightarrow\!\bullet, \rightarrow\!+\rightarrow\!\diamond, \rightarrow\!\%\rightarrow\!\diamond\}$

(ii) $$\text{ł}_{|\delta}(e) = \begin{cases} l(e) \cap \delta_\mathsf{L} & \textit{if } e \in \delta_\mathsf{E} \\ \emptyset & \textit{if } e \in \mathsf{E}_{|\delta} \backslash \delta_\mathsf{E} \end{cases}$$

(iii) $\mathsf{M}_{|\delta} = (\mathsf{Ex}_{|\delta}, \mathsf{Re}_{|\delta}, \mathsf{In}_{|\delta})$ *where:*

(a) $\mathsf{Ex}_{|\delta} = \mathsf{Ex} \cap \mathsf{E}_{|\delta}$

(b) $\mathsf{Re}_{|\delta} = \mathsf{Re} \cap (\delta_\mathsf{E} \cup \rightarrow\!\diamond\ \delta_\mathsf{E})$

(c) $\mathsf{In}_{|\delta} = \mathsf{In} \cap (\delta_\mathsf{E} \cup \rightarrow\!\bullet\ \delta_\mathsf{E} \cup \rightarrow\!\diamond\ \delta_\mathsf{E})$

(iv) $\rightarrow\!\bullet_{|\delta} = \rightarrow\!\bullet \cap ((\rightarrow\!\bullet\ \delta_\mathsf{E}) \times \delta_\mathsf{E})$

(v) $\bullet\!\rightarrow_{|\delta} = \bullet\!\rightarrow \cap ((\bullet\!\rightarrow\rightarrow\!\diamond\ \delta_\mathsf{E}) \times (\rightarrow\!\diamond\ \delta_\mathsf{E})) \cup ((\bullet\!\rightarrow\ \delta_\mathsf{E}) \times \delta_\mathsf{E}))$

(vi) $\rightarrow\!\diamond_{|\delta} = \rightarrow\!\diamond \cap ((\rightarrow\!\diamond\ \delta_\mathsf{E}) \times \delta_\mathsf{E})$

(vii) $\rightarrow\!+_{|\delta} = \rightarrow\!+ \cap \Big(\big((\rightarrow\!+\ \delta_\mathsf{E}) \times \delta_\mathsf{E}\big) \cup \big((\rightarrow\!+\rightarrow\!\bullet\ \delta_\mathsf{E}) \times (\rightarrow\!\bullet\ \delta_\mathsf{E}\big) \cup \big((\rightarrow\!+\rightarrow\!\diamond\ \delta_\mathsf{E}) \times (\rightarrow\!\diamond\ \delta_\mathsf{E}\big)\Big)$

(viii) $\rightarrow\!\%_{|\delta} = \rightarrow\!\% \cap \Big(\big((\rightarrow\!\%\ \delta_\mathsf{E}) \times \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\%\rightarrow\!\bullet\ \delta_\mathsf{E}) \times (\rightarrow\!\bullet\ \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\%\rightarrow\!\diamond\ \delta_\mathsf{E}) \times (\rightarrow\!\diamond\ \delta_\mathsf{E}\big)\Big)$

(i) defines the set of events as the union of the set δ_E of events that we project over, any event that has a direct relation towards an event in δ_E and events that exclude or include an event which is either a condition or a milestone for an event in δ_E. The additional events will be included in the projection without labels, as can be seen from the definition of the labeling function in *(ii)*. This means that the events can not be executed locally. However, when composed in a network containing other processes that can execute these events, their execution will be communicated to the process. For this reason we refer to these events as the (additional) external events of the projection. As proven in Prop. 5.3.1–5.3.3 the communication of the execution of this set of external events in addition to the local events shared by others ensure that the local state of the projection stay consistent with the global state.

Further *(iii)* defines the projection of the marking: The executed events remain the same, but are limited to the events in $\mathsf{E}_{|\delta}$. The responses are restricted to events in δ_E and events that have a milestone relation to an event in δ_E because these are the only responses that will affect the local execution of the projected graph. Note that these events will by definition be events in $E_{|\delta}$ but may be external events. In case of set of included events, we take the actual included status of the events in projection parameter along with the events that are conditions and milestones to the events in projection parameter, as the include status of those events will have an influence on the execution of events in local graph. All other external events of the projected graph are not included in the projected marking regardless of their included status in the marking of the global graph, because their include/exclude status will have no influence on the execution of events in local graph. Finally, *(iv)*, *(v)*, *(vi)*, *(vii)* and *(viii)* state which relations should be included in the projection. For the events in δ_E all incoming relations should be included. Additionally inclusion and exclusion relations to events that are either a condition or a milestone for an event in δ_E are included as well.

To define networks of communicating DCR Graphs and their semantics we use the following extension of a DCR Graph allowing any event to be executed with a special input label (ε). These transitions will only be used for the communication in a network and thus not be visible as user events.

Definition 5.3.2. *For an DCR Graph* $G = (E, \triangleright, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *define* $G^{\varepsilon} = (E, \triangleright, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L \cup \{\varepsilon\}, l^{\varepsilon})$*, where* $l^{\varepsilon} = l(e) \cup \{\varepsilon\}$ *(assuming that* $\varepsilon \notin L$*).*

We are now ready to state the key correspondence between global execution of events and the local execution of events in a projection.

Proposition 5.3.1. *Let* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *be a DCR Graph and* $G_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\delta_E, \delta_L)$*. Then,*
for $e \in \delta_E$ *and* $a \in \delta_L$ *it holds that* $M \vdash_G e \wedge M \oplus_G e = M' \wedge M'_{|\delta} = M''$ *if and only if* $M_{|\delta} \vdash_{G_{|\delta}} e \wedge M_{|\delta} \oplus_{G_{|\delta}} e = M''$*.*

Proof. In order to prove the proposition, we have to show that the proposition in both directions.

(G→P) for $e \in \delta_E$ and $a \in \delta_L$. $M \vdash_G e \wedge M \oplus_G e = M' \wedge M'_{|\delta} = M'' \implies M_{|\delta} \vdash_{G_{|\delta}} e \wedge M_{|\delta} \oplus_{G_{|\delta}} e = M''$.
We will split the proof into 2 steps:

(A) $M \vdash_G e \implies M_{|\delta} \vdash_{G_{|\delta}} e$
From def 3.3.7, we have $M \vdash_G e \implies e \in In \wedge (In \cap \rightarrow\bullet\, e) \subseteq Ex$ and $(In \cap \rightarrow\diamond e) \subseteq E \backslash Re$.

In order to prove that $M_{|\delta} \vdash_{G_{|\delta}} e$, we have to show that
$e \in In_{|\delta} \wedge (In_{|\delta} \cap \rightarrow\bullet_{|\delta} e) \subseteq Ex_{|\delta} \wedge (In_{|\delta} \cap \rightarrow\diamond_{|\delta} e) \subseteq E_{|\delta} \backslash Re_{|\delta}$. We will prove each part individually as follows,

(i) To prove: $e \in \mathsf{In}_{|\delta}$

From def 5.3.1-iiic we have,
$\mathsf{In}_{|\delta} = \mathsf{In} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\bullet\ \delta_{\mathsf{E}} \cup \rightarrow\!\diamond\ \delta_{\mathsf{E}})$ therefore
$e \in \mathsf{In} \wedge e \in \delta_{\mathsf{E}} \implies e \in \mathsf{In}_{|\delta}$.

(ii) To prove: $(\mathsf{In}_{|\delta} \cap \rightarrow\!\bullet_{|\delta} e) \subseteq \mathsf{Ex}_{|\delta}$

$\forall e' \in (\mathsf{In}_{|\delta} \cap \rightarrow\!\bullet_{|\delta} e)$,

(a) $e' \in \mathsf{In}_{|\delta} \implies e' \in \mathsf{In}$

(b) form the def 5.3.1-iv, we have $\rightarrow\!\bullet_{|\delta} = \rightarrow\!\bullet \cap ((\rightarrow\!\bullet\ \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}})$ therefore $e' \in \rightarrow\!\bullet_{|\delta} e \implies e' \in \rightarrow\!\bullet e$.

Using above 2 statements and from $\mathsf{M} \vdash_{\mathsf{G}} e$
$\forall e'.e' \in (\mathsf{In}_{|\delta} \cap \rightarrow\!\bullet_{|\delta} e) \implies e' \in (\mathsf{In} \cap \rightarrow\!\bullet e) \implies e' \in \mathsf{Ex}$,
Further, from def 5.3.1-iiia we have $\mathsf{Ex}_{|\delta} = \mathsf{Ex} \cap \mathsf{E}_{|\delta}$ therefore
$e' \in \mathsf{E}_{|\delta} \wedge e' \in \mathsf{Ex} \implies e' \in \mathsf{Ex}_{|\delta}$
Hence we can conclude that $(\mathsf{In}_{|\delta} \cap \rightarrow\!\bullet_{|\delta} e) \subseteq \mathsf{Ex}_{|\delta}$

(iii) To prove: $(\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta} e) \subseteq \mathsf{E}_{|\delta} \backslash \mathsf{Re}_{|\delta}$

$\forall e' \in (\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta} e)$

(a) $e' \in \mathsf{In}_{|\delta} \implies e' \in \mathsf{In}$

(b) form the def 5.3.1-vi, we have $\rightarrow\!\diamond_{|\delta} = \rightarrow\!\diamond \cap ((\rightarrow\!\diamond\ \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}})$ therefore $e' \in \rightarrow\!\diamond_{|\delta} e \implies e' \in \rightarrow\!\diamond e$.

Using above 2 statements and from $\mathsf{M} \vdash_{\mathsf{G}} e$
$e' \in (\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta} e) \implies e' \in (\mathsf{In} \cap \rightarrow\!\diamond e) \implies e' \in \mathsf{E} \backslash \mathsf{Re} \implies e' \notin \mathsf{Re}$,

According to def 5.3.1 iiib, we have $\mathsf{Re}_{|\delta} = \mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\ \delta_{\mathsf{E}})$. and so $e' \notin \mathsf{Re} \implies e' \notin \mathsf{Re}_{|\delta}$.
Further, $e' \in \mathsf{E}_{|\delta} \wedge e' \notin \mathsf{Re}_{|\delta} \implies e' \in \mathsf{E}_{|\delta} \setminus \mathsf{Re}_{|\delta}$.
Hence we can conclude that $(\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta} e) \subseteq \mathsf{E}_{|\delta} \backslash \mathsf{Re}_{|\delta}$

From (G→P)-A-i, (G→P)-A-ii and (G→P)-A-iii, we have proved that $e \in \mathsf{In}_{|\delta} \wedge (\mathsf{In}_{|\delta} \cap \rightarrow\!\bullet_{|\delta} e) \subseteq \mathsf{Ex}_{|\delta} \wedge (\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta} e) \subseteq \mathsf{E}_{|\delta} \backslash \mathsf{Re}_{|\delta}$ is valid.
Therefore we can conclude that $\mathsf{M} \vdash_{\mathsf{G}} e \implies \mathsf{M}_{|\delta} \vdash_{\mathsf{G}_{|\delta}} e$.

(B) To prove: $\mathsf{M} \oplus_{\mathsf{G}} e = \mathsf{M}' \wedge \mathsf{M}'_{|\delta} = \mathsf{M}'' \implies \mathsf{M}_{|\delta} \oplus_{\mathsf{G}_{|\delta}} e = \mathsf{M}''$

We have $\mathsf{M} \oplus_{\mathsf{G}} e = \mathsf{M}'$ where $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ and $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$

and from the def 3.3.8, we can infer
$\mathsf{Ex}' = \mathsf{Ex} \cup \{e\}, \mathsf{Re}' = (\mathsf{Re} \setminus \{e\}) \cup e\bullet\!\!\rightarrow$, and $\mathsf{In}' = (\mathsf{In} \cup e\rightarrow\!\!+) \setminus e\rightarrow\!\!\%$.

In projected graph, we have $\mathsf{M}_{|\delta} = (\mathsf{Ex}_{|\delta}, \mathsf{Re}_{|\delta}, \mathsf{In}_{|\delta})$, $\mathsf{M}'' = (\mathsf{Ex}''_{|\delta}, \mathsf{Re}''_{|\delta}, \mathsf{In}''_{|\delta})$ and from above result we know that $\mathsf{M}_{|\delta} \vdash_{\mathsf{G}_{|\delta}} e$. Hence we can infer that $\mathsf{Ex}''_{|\delta} = \mathsf{Ex}_{|\delta} \cup \{e\}, \mathsf{Re}''_{|\delta} = (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup e\bullet\!\!\rightarrow_{|\delta}$, and $\mathsf{In}''_{|\delta} = (\mathsf{In}_{|\delta} \cup e\rightarrow\!\!+_{|\delta}) \setminus e\rightarrow\!\!\%_{|\delta}$.

We have to prove that $\mathsf{M}'_{|\delta} = \mathsf{M}''$. In order to prove this equivalence, we will show that $\mathsf{Ex}'_{|\delta} = \mathsf{Ex}''_{|\delta}$, $\mathsf{Re}'_{|\delta} = \mathsf{Re}''_{|\delta}$ and $\mathsf{In}'_{|\delta} = \mathsf{In}''_{|\delta}$ individually as follows,

(i) To prove: $\mathsf{Ex}'_{|\delta} = \mathsf{Ex}''_{|\delta}$

$\mathsf{Ex}'_{|\delta} = (\mathsf{Ex} \cup \{e\}) \cap \mathsf{E}_{|\delta}$ from def 5.3.1-iiia
$= (\mathsf{Ex} \cap \mathsf{E}_{|\delta}) \cup (\{e\} \cap \mathsf{E}_{|\delta})$ distributive law of sets
$= \mathsf{Ex}_{|\delta} \cup \{e\}$ according to def 5.3.1-iiia and $e \in \delta_{\mathsf{E}} \subseteq \mathsf{E}_{|\delta}$.
$= \mathsf{Ex}''_{|\delta}$.
Hence we can conclude that $\mathsf{Ex}'_{|\delta} = \mathsf{Ex}''_{|\delta}$

(ii) To prove: $\mathsf{Re}'_{|\delta} = \mathsf{Re}''_{|\delta}$

According to def 5.3.1-v, the response relation in projected graph is
$\bullet\!\!\rightarrow_{|\delta} = \bullet\!\!\rightarrow \cap((\bullet\!\!\rightarrow\rightarrow\!\!\diamond\ \delta_{\mathsf{E}}) \times (\rightarrow\!\!\diamond\ \delta_{\mathsf{E}})) \cup ((\bullet\!\!\rightarrow\ \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}}))$.
Informally it contains relations which can cause a response on an event which is either included in the set of events in the project parameter (δ_{E}) or in a set of events which are milestones to events in project parameter ($\rightarrow\!\!\diamond\ \delta_{\mathsf{E}}$).
$\bullet\!\!\rightarrow_{|\delta} = \{(e'', e') \mid e'' \bullet\!\!\rightarrow e' \wedge e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\}$ and hence
$e\bullet\!\!\rightarrow_{|\delta} = \{e' \mid e \bullet\!\!\rightarrow e' \wedge e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\}$

$\mathsf{Re}'_{|\delta} = \big((\mathsf{Re} \setminus \{e\}) \cup e\bullet\!\!\rightarrow\ \big) \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})$ from def 5.3.1-iiib
$= \big((\mathsf{Re} \setminus \{e\}) \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\big) \cup \big(\ e\bullet\!\!\rightarrow \cap(\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\big)$ distributive law
$= \big(\mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}}) \setminus (\{e\} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\big) \cup \big(\ e\bullet\!\!\rightarrow \cap(\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\big)$
set intersection distributes over set difference
$= (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup \big(\ e\bullet\!\!\rightarrow \cap(\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\big)$
$= (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup \{e' \mid e \bullet\!\!\rightarrow e' \wedge e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\!\diamond\ \delta_{\mathsf{E}})\}$
$= (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup e\bullet\!\!\rightarrow_{|\delta}$
$= \mathsf{Re}''_{|\delta}$
Hence we can conclude that $\mathsf{Re}'_{|\delta} = \mathsf{Re}''_{|\delta}$.

(iii) To prove: $\mathsf{In}'_{|\delta} = \mathsf{In}''_{|\delta}$

(iii-a) According to def 5.3.1-vii, the include relation in projected graph is
$\rightarrow+_{|\delta} = \rightarrow+ \cap \Bigg(\big((\rightarrow+\ \delta_{\mathrm{E}}) \times \delta_{\mathrm{E}}\big) \cup \big((\rightarrow+\rightarrow\bullet\ \delta_{\mathrm{E}}) \times (\rightarrow\bullet\ \delta_{\mathrm{E}}\big) \cup \big((\rightarrow+\rightarrow\diamond\ \delta_{\mathrm{E}}) \times (\rightarrow\diamond\ \delta_{\mathrm{E}}\big) \Bigg)$
$\rightarrow+_{|\delta} = \{(e'', e') \mid e'' \rightarrow+\ e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\}$
$e\rightarrow+_{|\delta} = \{e' \mid e \rightarrow+\ e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\}$
$e\rightarrow+_{|\delta} = e\rightarrow+ \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$

(iii-b) Similarly, according to def 5.3.1-viii, the exclude relation in projected graph is
$\rightarrow\%_{|\delta} = \rightarrow\% \cap \Bigg(\big((\rightarrow\%\ \delta_{\mathrm{E}}) \times \delta_{\mathrm{E}}\big) \cup \big((\rightarrow\%\rightarrow\bullet\ \delta_{\mathrm{E}}) \times (\rightarrow\bullet\ \delta_{\mathrm{E}}\big) \cup \big((\rightarrow\%\rightarrow\diamond\ \delta_{\mathrm{E}}) \times (\rightarrow\diamond\ \delta_{\mathrm{E}}\big) \Bigg)$
$\rightarrow\%_{|\delta} = \{(e'', e') \mid e'' \rightarrow\%\ e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\}$
$e\rightarrow\%_{|\delta} = \{e' \mid e \rightarrow\%\ e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\}$
$e\rightarrow\%_{|\delta} = e\rightarrow\% \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$

From def 5.3.1-iiic we have: $\mathrm{In}_{|\delta} = \mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$
Hence we can compute the projection of global included set (In') as follows
$\mathrm{In}'_{|\delta} = \mathrm{In}' \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$
But we know that $\mathrm{In}' = (\mathrm{In} \cup e\rightarrow+) \setminus e\rightarrow\%$
$\mathrm{In}'_{|\delta} = \big((\mathrm{In} \cup e\rightarrow+) \setminus e\rightarrow\%\ \big) \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$

$\mathrm{In}'_{|\delta} = \big((\mathrm{In} \cup e\rightarrow+) \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big) \setminus \big(\ e\rightarrow\% \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big)$ set intersection distributes over set difference

$\mathrm{In}'_{|\delta} = \big((\mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big)\ \cup \big(\ e\rightarrow+ \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big) \setminus \big(\ e\rightarrow\% \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big)$ distributive law
Using results (iii-a) and (iii-b), we can rewrite the above statement as
$\mathrm{In}'_{|\delta} = \Big(\big(\mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})\big) \cup e\rightarrow+_{|\delta}\ \Big) \setminus e\rightarrow\%_{|\delta}$
But we know that the marking in projected graph before executing event e is $\mathrm{In}_{|\delta} = \mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet\ \delta_{\mathrm{E}} \cup \rightarrow\diamond\ \delta_{\mathrm{E}})$. Using this fact, we can rewrite the above statement as follows,

$\mathrm{In}'_{|\delta} = (\mathrm{In}_{|\delta} \cup e\rightarrow+_{|\delta}) \setminus e\rightarrow\%_{|\delta}$
$\mathrm{In}'_{|\delta} = \mathrm{In}''_{|\delta}$

Hence we can conclude that $\mathrm{In}'_{|\delta} = \mathrm{In}''_{|\delta}$.

From (G→P)-B-i, (G→P)-B-ii and (G→P)-B-iii, we have proved that $\mathrm{Ex}'_{|\delta} = \mathrm{Ex}''_{|\delta}$, $\mathrm{Re}'_{|\delta} = \mathrm{Re}''_{|\delta}$ and $\mathrm{In}'_{|\delta} = \mathrm{In}''_{|\delta}$. and there by we can conclude that

$\mathrm{M}'_{|\delta} = \mathrm{M}''$.

Since we have proved both parts: ((G→P)-A and (G→P)-B), the proposition $\mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}'' \implies \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}''$ holds.

(P→G) for $e \in \delta_{\mathrm{E}}$ and $a \in \delta_{\mathrm{L}}$. $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'' \implies \mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$
Again, we will split the proof into 2 parts.

(A) $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \implies \mathrm{M} \vdash_{\mathrm{G}} e$
From def 3.3.7, we have $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \implies e \in \mathrm{In}_{|\delta} \wedge (\mathrm{In}_{|\delta} \cap \to\bullet_{|\delta} e) \subseteq \mathrm{Ex}_{|\delta} \wedge (\mathrm{In}_{|\delta} \cap \to\diamond_{|\delta} e) \subseteq \mathrm{E}_{|\delta} \backslash \mathrm{Re}_{|\delta}$

In order to prove that $\mathrm{M} \vdash_{\mathrm{G}} e$, we have to show that
$e \in \mathrm{In} \wedge (\mathrm{In} \cap \to\bullet e) \subseteq \mathrm{Ex}$ and $(\mathrm{In} \cap \to\diamond e) \subseteq \mathrm{E} \backslash \mathrm{Re}$

(i) To prove: $e \in \mathrm{In}$

From def 5.3.1-iiic we have: $\mathrm{In}_{|\delta} = \mathrm{In} \cap (\delta_{\mathrm{E}} \cup \to\bullet\, \delta_{\mathrm{E}} \cup \to\diamond\, \delta_{\mathrm{E}})$
$e \in \mathrm{In}_{|\delta} \wedge \big(\mathrm{In}_{|\delta} = \mathrm{In} \cap (\delta_{\mathrm{E}} \cup \to\bullet\, \delta_{\mathrm{E}} \cup \to\diamond\, \delta_{\mathrm{E}})\big) \implies e \in \mathrm{In}$.

(ii) To prove: $(\mathrm{In} \cap \to\bullet e) \subseteq \mathrm{Ex}$

From def 5.3.1-iv, we have $\to\bullet_{|\delta} = \to\bullet \cap ((\to\bullet\, \delta_{\mathrm{E}}) \times \delta_{\mathrm{E}})$
$\forall e'.e' \in \to\bullet_{|\delta} e \implies (e', e) \in \to\bullet_{|\delta} \implies (e', e) \in \to\bullet \implies e' \in \to\bullet e$ and therefore $\to\bullet_{|\delta} e = \to\bullet e$.
$\forall e'.e' \in (\mathrm{In}_{|\delta} \cap \to\bullet_{|\delta} e) \implies (e' \in \mathrm{In}_{|\delta}) \cap (e' \in \to\bullet_{|\delta} e) \implies (e' \in \mathrm{In}) \cap (e' \in \to\bullet e) \implies e' \in (\mathrm{In} \cap \to\bullet e)$, and hence
$(\mathrm{In}_{|\delta} \cap \to\bullet_{|\delta} e) = (\mathrm{In} \cap \to\bullet e)$.

$(\mathrm{In}_{|\delta} \cap \to\bullet_{|\delta} e) \subseteq \mathrm{Ex}_{|\delta} \implies (\mathrm{In} \cap \to\bullet e) \subseteq \mathrm{Ex}_{|\delta}$.
according to def 5.3.1-iiia : $\mathrm{Ex}_{|\delta} = \mathrm{Ex} \cap \mathrm{E}_{|\delta}$.
Hence $(\mathrm{In} \cap \to\bullet e) \subseteq \mathrm{Ex}_{|\delta} \implies (\mathrm{In} \cap \to\bullet e) \subseteq \mathrm{Ex}$

(iii) To prove: $(\mathrm{In} \cap \to\diamond e) \subseteq \mathrm{E} \backslash \mathrm{Re}$

From def 5.3.1-vi, we have $\to\diamond_{|\delta} = \to\diamond \cap ((\to\diamond\, \delta_{\mathrm{E}}) \times \delta_{\mathrm{E}})$,
$\forall e'.e' \in \to\diamond_{|\delta} e \implies (e', e) \in \to\diamond_{|\delta} \implies (e', e) \in \to\diamond \implies e' \in \to\diamond e$ and therefore $\to\diamond_{|\delta} e = \to\diamond e$.
$\forall e'.e' \in (\mathrm{In}_{|\delta} \cap \to\diamond_{|\delta} e) \implies (e' \in \mathrm{In}_{|\delta}) \cap (e' \in \to\diamond_{|\delta} e) \implies (e' \in \mathrm{In}) \cap (e' \in \to\diamond e) \implies e' \in (\mathrm{In} \cap \to\diamond e)$, and hence
$(\mathrm{In}_{|\delta} \cap \to\diamond_{|\delta} e) = (\mathrm{In} \cap \to\diamond e)$.

$(\mathrm{In}_{|\delta} \cap \to\diamond_{|\delta} e) \subseteq \mathrm{E}_{|\delta} \backslash \mathrm{Re}_{|\delta} \implies (\mathrm{In} \cap \to\diamond e) \subseteq \mathrm{E}_{|\delta} \backslash \mathrm{Re}_{|\delta} \implies \forall e' \in$

$(\mathsf{In} \cap \rightarrow\!\diamond\, e).e' \notin \mathsf{Re}_{|\delta}$.

according to def 5.3.1-iiib : $\mathsf{Re}_{|\delta} = \mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})$,
$\forall e' \in (\mathsf{In} \cap \rightarrow\!\diamond\, e).e' \notin \mathsf{Re}_{|\delta} \implies e' \notin (\mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}}))$. Further, as $e' \rightarrow\!\diamond\, e$, we know that $e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})$. The only way $e' \notin (\mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}}))$ becomes true is when $e' \notin \mathsf{Re}$.

Hence $(\mathsf{In}_{|\delta} \cap \rightarrow\!\diamond_{|\delta}\, e) \subseteq \mathsf{E}_{|\delta} \backslash \mathsf{Re}_{|\delta} \implies (\mathsf{In} \cap \rightarrow\!\diamond\, e) \subseteq \mathsf{E} \backslash \mathsf{Re}$.

Form (P→G)-A-(i), (P→G)-A-(ii) and (P→G)-A-(iii), we can conclude that $\mathsf{M}_{|\delta} \vdash_{\mathsf{G}_{|\delta}} e \implies \mathsf{M} \vdash_{\mathsf{G}} e$.

(B) $\mathsf{M}_{|\delta} \oplus_{\mathsf{G}_{|\delta}} e = \mathsf{M}'' \implies \mathsf{M} \oplus_{\mathsf{G}} e = \mathsf{M}' \wedge \mathsf{M}'_{|\delta} = \mathsf{M}''$
We have $\mathsf{M}_{|\delta} \oplus_{\mathsf{G}_{|\delta}} e = \mathsf{M}''$ in the local graph where $\mathsf{M}_{|\delta} = (\mathsf{Ex}_{|\delta}, \mathsf{Re}_{|\delta}, \mathsf{In}_{|\delta})$, $\mathsf{M}'' = (\mathsf{Ex}''_{|\delta}, \mathsf{Re}''_{|\delta}, \mathsf{In}''_{|\delta})$ and from the def 3.3.8, we can infer
$\mathsf{Ex}''_{|\delta} = \mathsf{Ex}_{|\delta} \cup \{e\}, \mathsf{Re}''_{|\delta} = (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup e \bullet\!\!\rightarrow_{|\delta}$, and $\mathsf{In}''_{|\delta} = (\mathsf{In}_{|\delta} \cup e \rightarrow\!+_{|\delta}) \setminus e \rightarrow\!\%_{|\delta}$.

In main graph, we know $\mathsf{M} \vdash_{\mathsf{G}} e$ where $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ and hence we can workout the new marking as $\mathsf{M} \oplus_{\mathsf{G}} e = \mathsf{M}'$ where $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ with $\mathsf{Ex}' = \mathsf{Ex} \cup \{e\}, \mathsf{Re}' = (\mathsf{Re} \setminus \{e\}) \cup e \bullet\!\!\rightarrow$, and $\mathsf{In}' = (\mathsf{In} \cup e \rightarrow\!+) \setminus e \rightarrow\!\%$.

We have to prove that $\mathsf{M}'' = \mathsf{M}'_{|\delta}$.

(i) To prove: $\mathsf{Ex}''_{|\delta} = \mathsf{Ex}'_{|\delta}$

Let us start with $\mathsf{Ex}''_{|\delta}$
$\mathsf{Ex}''_{|\delta} = \mathsf{Ex}_{|\delta} \cup \{e\}$
$= (\mathsf{Ex} \cap \mathsf{E}_{|\delta}) \cup \{e\}$ from def 5.3.1-iiia
$= (\mathsf{Ex} \cup \{e\}) \cap (\mathsf{E}_{|\delta} \cup \{e\})$
$= \mathsf{Ex}' \cap \mathsf{E}_{|\delta}$
$= \mathsf{Ex}'_{|\delta}$
Hence we can conclude that $\mathsf{Ex}''_{|\delta} = \mathsf{Ex}'_{|\delta}$.

(ii) To prove: $\mathsf{Re}''_{|\delta} = \mathsf{Re}'_{|\delta}$

(a) According to def 5.3.1-v, the response relation in local graph is
$\bullet\!\!\rightarrow_{|\delta} = \bullet\!\!\rightarrow \cap ((\bullet\!\!\rightarrow \rightarrow\!\diamond\, \delta_{\mathsf{E}}) \times (\rightarrow\!\diamond\, \delta_{\mathsf{E}})) \cup ((\bullet\!\!\rightarrow\, \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}}))$.
$\bullet\!\!\rightarrow_{|\delta} = \{(e'', e') \mid e'' \bullet\!\!\rightarrow e' \wedge e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})\}$ and so
$e \bullet\!\!\rightarrow_{|\delta} = \{e' \mid e \bullet\!\!\rightarrow e' \wedge e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})\}$
$e \bullet\!\!\rightarrow_{|\delta} = \{e' \mid e \bullet\!\!\rightarrow e'\} \cap \{e' \mid e' \in (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})\}$
$e \bullet\!\!\rightarrow_{|\delta} = e \bullet\!\!\rightarrow \cap (\delta_{\mathsf{E}} \cup \rightarrow\!\diamond\, \delta_{\mathsf{E}})$

Let us start with $\mathsf{Re}''_{|\delta}$

$\mathsf{Re}''_{|\delta} = (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup e \bullet\!\!\to_{|\delta}$

$= \big((\mathsf{Re} \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})) \setminus \{e\}\big) \cup e \bullet\!\!\to_{|\delta}$ from def 5.3.1-iiib

$= \big((\mathsf{Re} \setminus \{e\}) \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\big) \cup e \bullet\!\!\to_{|\delta}$ (set relative complements)

$= \big((\mathsf{Re} \setminus \{e\}) \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\big) \cup (e \bullet\!\!\to \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}}))$ using (a)

$= \big((\mathsf{Re} \setminus \{e\}) \cup e \bullet\!\!\to\ \big) \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$

$= \mathsf{Re}' \cap (\delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$

$= \mathsf{Re}'_{|\delta}$ according to def 5.3.1-iiib.

Hence we can conclude that $\mathsf{Re}''_{|\delta} = \mathsf{Re}'_{|\delta}$.

(iii) To prove: $\mathsf{In}''_{|\delta} = \mathsf{In}'_{|\delta}$

(a) According to def 5.3.1-vii, the include relation in projected graph is

$\to\!\!+_{|\delta} = \to\!\!+ \cap \Big(\big((\to\!\!+\ \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}}\big) \cup \big((\to\!\!+\to\!\!\bullet\ \delta_{\mathsf{E}}) \times (\to\!\!\bullet\ \delta_{\mathsf{E}}\big) \cup \big((\to\!\!+\to\!\!\diamond\ \delta_{\mathsf{E}}) \times (\to\!\!\diamond\ \delta_{\mathsf{E}})\big) \Big)$

$\to\!\!+_{|\delta} = \{(e'', e') \mid e'' \to\!\!+\ e' \wedge e' \in (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\}$

$e \to\!\!+_{|\delta} = \{e' \mid e \to\!\!+\ e' \wedge e' \in (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\}$

$e \to\!\!+_{|\delta} = e \to\!\!+ \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$.

(b) Similarly, according to def 5.3.1-viii, the exclude relation in projected graph is

$\to\!\!\%_{|\delta} = \to\!\!\% \cap \Big(\big((\to\!\!\%\ \delta_{\mathsf{E}}) \times \delta_{\mathsf{E}}\big) \cup \big((\to\!\!\%\to\!\!\bullet\ \delta_{\mathsf{E}}) \times (\to\!\!\bullet\ \delta_{\mathsf{E}}\big) \cup \big((\to\!\!\%\to\!\!\diamond\ \delta_{\mathsf{E}}) \times (\to\!\!\diamond\ \delta_{\mathsf{E}})\big) \Big)$

$\to\!\!\%_{|\delta} = \{(e'', e') \mid e'' \to\!\!\%\ e' \wedge e' \in (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\}$

$e \to\!\!\%_{|\delta} = \{e' \mid e \to\!\!\%\ e' \wedge e' \in (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\}$

$e \to\!\!\%_{|\delta} = e \to\!\!\% \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$.

Having sub results (a) and (b), let us starts with $\mathsf{In}''_{|\delta}$ and show that it will be equal to the projection over included set from global graph $(\mathsf{In}'_{|\delta})$.

$\mathsf{In}''_{|\delta} = (\mathsf{In}_{|\delta} \cup e \to\!\!+_{|\delta}) \setminus e \to\!\!\%_{|\delta}$

from def 5.3.1-iiic, we have $\mathsf{In}_{|\delta} = \mathsf{In} \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$ hence,

$\mathsf{In}''_{|\delta} = \big((\mathsf{In} \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})) \cup e \to\!\!+_{|\delta}\ \big) \setminus e \to\!\!\%_{|\delta}$.

Again using the results (a) and (b), we can rewrite the above expression as,

$\mathsf{In}''_{|\delta} = \big((\mathsf{In} \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})) \cup (e \to\!\!+ \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}}))\big) \setminus (e \to\!\!\% \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}}))$.

$\mathsf{In}''_{|\delta} = \big((\mathsf{In} \cup e \to\!\!+) \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})\big) \setminus (e \to\!\!\% \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}}))$.

$\mathsf{In}''_{|\delta} = \big((\mathsf{In} \cup e \to\!\!+) \setminus e \to\!\!\%\ \big) \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$.

$\mathsf{In}''_{|\delta} = (\mathsf{In}') \cap (\delta_{\mathsf{E}} \cup \to\!\!\bullet\ \delta_{\mathsf{E}} \cup \to\!\!\diamond\ \delta_{\mathsf{E}})$.

$\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$.
Hence we can conclude that $\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$.

From (P→G)-B-i, (P→G)-B-ii and (P→G)-B-iii, we have proved that $\mathrm{Ex}''_{|\delta} = \mathrm{Ex}'_{|\delta}$, $\mathrm{Re}''_{|\delta} = \mathrm{Re}'_{|\delta}$ and $\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$ and there by we can conclude that $\mathrm{M}'' = \mathrm{M}'_{|\delta}$.

Since we have proved both parts: ((P→G)-A and (P→G)-B), the proposition for $e \in \delta_{\mathrm{E}}$ and $a \in \delta_{\mathrm{L}}$. $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'' \implies \mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$ holds.

Finally, we have proved the proposition in both ways $\big($(G→P) and (P→G)$\big)$, therefore the proposition: for $e \in \delta_{\mathrm{E}}$ and $a \in \delta_{\mathrm{L}}$ it holds that $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$ if and only if $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}''$ holds.

□

Proposition 5.3.2. *Let* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$ *be a DCR Graph and* $\mathrm{G}_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\delta_{\mathrm{E}}, \delta_{\mathrm{L}})$. *Then, for* $e \notin \mathrm{E}_{|\delta}$ *it holds that* $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ *implies* $\mathrm{M}_{|\delta} = \mathrm{M}'_{|\delta}$.

Proof. According to projection definition 5.3.1, $e \notin \mathrm{E}_{|\delta} \implies e \notin \mathrm{G}_{|\delta}$, therefore there will not be any change in the marking. Hence $\mathrm{M}_{|\delta} = \mathrm{M}'_{|\delta}$. □

Proposition 5.3.3. *Let* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$ *be a DCR Graph and* $\mathrm{G}_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\delta_{\mathrm{E}}, \delta_{\mathrm{L}})$. *Then for* $e \in \mathrm{E}_{|\delta}$ *(and* $a \notin \delta_{\mathrm{L}}$*) it holds that* $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ *implies* $\mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'_{|\delta}$.

Proof. The proof for this proposition is more or less similar to proof in the part (P→G)-(B) of proposition 5.3.1 with minor changes.

We have $\mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ where $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$ and $\mathrm{M}' = (\mathrm{Ex}', \mathrm{Re}', \mathrm{In}')$ and from the def 3.3.8, we can infer
$\mathrm{Ex}' = \mathrm{Ex} \cup \{e\}$, $\mathrm{Re}' = (\mathrm{Re} \setminus \{e\}) \cup e\bullet\!\rightarrow$, and $\mathrm{In}' = (\mathrm{In} \cup e\rightarrow\!+) \setminus e\rightarrow\!\%$.

In projected graph, we have marking projected from global graph, according to def 5.3.1 as $\mathrm{M}_{|\delta} = (\mathrm{Ex}_{|\delta}, \mathrm{Re}_{|\delta}, \mathrm{In}_{|\delta})$. The result of executing event e in projected graph will be a marking, let us say $\mathrm{M}''_{|\delta} = \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e$, then we have to prove that $\mathrm{M}''_{|\delta} = \mathrm{M}'_{|\delta}$.

Let us say that $\mathrm{M}''_{|\delta} = (\mathrm{Ex}''_{|\delta}, \mathrm{Re}''_{|\delta}, \mathrm{In}''_{|\delta})$, and since in the projected graph we have $\mathrm{M}''_{|\delta} = \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e$, we can infer from the def 3.3.8, $\mathrm{Ex}''_{|\delta} = \mathrm{Ex}_{|\delta} \cup \{e\}, \mathrm{Re}''_{|\delta} = (\mathrm{Re}_{|\delta} \setminus \{e\}) \cup e\bullet\!\rightarrow_{|\delta}$, and $\mathrm{In}''_{|\delta} = (\mathrm{In}_{|\delta} \cup e\rightarrow\!+_{|\delta}) \setminus e\rightarrow\!\%_{|\delta}$.

In order to prove this equivalence of $\mathrm{M}''_{|\delta} = \mathrm{M}'_{|\delta}$, we will show that $\mathrm{Ex}''_{|\delta} = \mathrm{Ex}'_{|\delta}$, $\mathrm{Re}''_{|\delta} = \mathrm{Re}'_{|\delta}$ and $\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$ individually as follows,

(i) To prove: $\mathsf{Ex}''_{|\delta} = \mathsf{Ex}'_{|\delta}$

Let us start with $\mathsf{Ex}''_{|\delta}$
$\mathsf{Ex}''_{|\delta} = \mathsf{Ex}_{|\delta} \cup \{e\}$
$= (\mathsf{Ex} \cap \mathsf{E}_{|\delta}) \cup \{e\}$ from def 5.3.1-iiia
$= (\mathsf{Ex} \cup \{e\}) \cap (\mathsf{E}_{|\delta} \cup \{e\})$
$= \mathsf{Ex}' \cap \mathsf{E}_{|\delta}$
$= \mathsf{Ex}'_{|\delta}$
Hence we can conclude that $\mathsf{Ex}''_{|\delta} = \mathsf{Ex}'_{|\delta}$.

(ii) To prove: $\mathsf{Re}''_{|\delta} = \mathsf{Re}'_{|\delta}$

(a) According to def 5.3.1-v, the response relation in local graph is
$\bullet\!\!\rightarrow_{|\delta} = \bullet\!\!\rightarrow \cap((\bullet\!\!\rightarrow\rightarrow\!\!\diamond\ \delta_\mathsf{E}) \times (\rightarrow\!\!\diamond\ \delta_\mathsf{E})) \cup ((\bullet\!\!\rightarrow\ \delta_\mathsf{E}) \times \delta_\mathsf{E}))$.
$\bullet\!\!\rightarrow_{|\delta} = \{(e'', e') \mid e''\ \bullet\!\!\rightarrow\ e' \wedge e' \in (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\}$ and so
$e\bullet\!\!\rightarrow_{|\delta} = \{e' \mid e\ \bullet\!\!\rightarrow\ e' \wedge e' \in (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\}$
$e\bullet\!\!\rightarrow_{|\delta} = \{e' \mid e\ \bullet\!\!\rightarrow\ e'\} \cap \{e' \mid e' \in (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\}$
$e\bullet\!\!\rightarrow_{|\delta} = e\bullet\!\!\rightarrow \cap(\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})$

Let us start with $\mathsf{Re}''_{|\delta}$
$\mathsf{Re}''_{|\delta} = (\mathsf{Re}_{|\delta} \setminus \{e\}) \cup e\bullet\!\!\rightarrow_{|\delta}$
$= \big((\mathsf{Re} \cap (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})) \setminus \{e\}\big) \cup e\bullet\!\!\rightarrow_{|\delta}$ from def 5.3.1-iiib
$= \big((\mathsf{Re} \setminus \{e\}) \cap (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\big) \cup e\bullet\!\!\rightarrow_{|\delta}$ (set relative complements)
$= \big((\mathsf{Re} \setminus \{e\}) \cap (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\big) \cup (e\bullet\!\!\rightarrow \cap(\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E}))$ using (a)
$= \big((\mathsf{Re} \setminus \{e\}) \cup e\bullet\!\!\rightarrow\ \big) \cap (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})$
$= \mathsf{Re}' \cap (\delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})$
$= \mathsf{Re}'_{|\delta}$ according to def 5.3.1-iiib.
Hence we can conclude that $\mathsf{Re}''_{|\delta} = \mathsf{Re}'_{|\delta}$.

(iii) To prove: $\mathsf{In}''_{|\delta} = \mathsf{In}'_{|\delta}$

(a) According to def 5.3.1-vii, the include relation in projected graph is
$\rightarrow\!\!+_{|\delta} = \rightarrow\!\!+\ \cap\Big(\big((\rightarrow\!\!+\ \ \delta_\mathsf{E}) \times \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\!+\!\rightarrow\!\!\bullet\ \ \delta_\mathsf{E}) \times (\rightarrow\!\!\bullet\ \ \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\!+\!\rightarrow\!\!\diamond$
$\delta_\mathsf{E}) \times (\rightarrow\!\!\diamond\ \delta_\mathsf{E}\big)\Big)$
$\rightarrow\!\!+_{|\delta} = \{(e'', e') \mid e''\ \rightarrow\!\!+\ e' \wedge e' \in (\delta_\mathsf{E} \cup \rightarrow\!\!\bullet\ \delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\}$
$e\rightarrow\!\!+_{|\delta} = \{e' \mid e\ \rightarrow\!\!+\ e' \wedge e' \in (\delta_\mathsf{E} \cup \rightarrow\!\!\bullet\ \delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})\}$
$e\rightarrow\!\!+_{|\delta} = e\rightarrow\!\!+ \cap(\delta_\mathsf{E} \cup \rightarrow\!\!\bullet\ \delta_\mathsf{E} \cup \rightarrow\!\!\diamond\ \delta_\mathsf{E})$.

(b) Similarly, according to def 5.3.1-viii, the exclude relation in projected graph is
$\rightarrow\!\!\%_{|\delta} = \rightarrow\!\!\% \cap\Big(\big((\rightarrow\!\!\%\ \delta_\mathsf{E}) \times \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\!\%\!\rightarrow\!\!\bullet\ \delta_\mathsf{E}) \times (\rightarrow\!\!\bullet\ \delta_\mathsf{E}\big) \cup \big((\rightarrow\!\!\%\!\rightarrow\!\!\diamond\ \delta_\mathsf{E}) \times (\rightarrow\!\!\diamond$

$\delta_{\mathrm{E}})\big)$
$\rightarrow\%_{|\delta}= \{(e'', e') \mid e'' \rightarrow\% \; e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})\}$
$e \rightarrow\%_{|\delta}= \{e' \mid e \rightarrow\% \; e' \wedge e' \in (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})\}$
$e \rightarrow\%_{|\delta}=e \rightarrow\% \cap(\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})$.

Having sub results (a) and (b), let us starts with $\mathrm{In}''_{|\delta}$ and show that it will be equal to the projection over included set from global graph ($\mathrm{In}'_{|\delta}$).
$\mathrm{In}''_{|\delta} = (\mathrm{In}_{|\delta} \cup \; e \rightarrow+_{|\delta}) \setminus \; e \rightarrow\%_{|\delta}$
from def 5.3.1-iiic, we have $\mathrm{In}_{|\delta} = \mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})$ hence,
$\mathrm{In}''_{|\delta} = \big((\mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})) \cup \; e \rightarrow+_{|\delta} \; \big) \setminus \; e \rightarrow\%_{|\delta}$.
Again using the results (a) and (b), we can rewrite the above expression as,
$\mathrm{In}''_{|\delta} = \big((\mathrm{In} \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})) \cup (e \rightarrow+ \cap(\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}}))\big) \setminus (e \rightarrow\% \cap(\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}}))$.
$\mathrm{In}''_{|\delta} = \big((\mathrm{In} \cup \; e \rightarrow+) \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})\big) \setminus (e \rightarrow\% \cap(\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}}))$.
$\mathrm{In}''_{|\delta} = \big((\mathrm{In} \cup \; e \rightarrow+) \setminus \; e \rightarrow\% \; \big) \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})$.
$\mathrm{In}''_{|\delta} = (\mathrm{In}') \cap (\delta_{\mathrm{E}} \cup \rightarrow\bullet \; \delta_{\mathrm{E}} \cup \rightarrow\diamond \; \delta_{\mathrm{E}})$.
$\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$.
Hence we can conclude that $\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$.

From (i), (ii) and (iii), we have proved that $\mathrm{Ex}''_{|\delta} = \mathrm{Ex}'_{|\delta}$, $\mathrm{Re}''_{|\delta} = \mathrm{Re}'_{|\delta}$ and $\mathrm{In}''_{|\delta} = \mathrm{In}'_{|\delta}$ and there by we can conclude that $\mathrm{M}'' = \mathrm{M}'_{|\delta}$.

Therefore the proposition: for $e \in \mathrm{E}_{|\delta}$ (and $a \notin \delta_{\mathrm{L}}$) it holds that $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ implies $\mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'_{|\delta}$ is proved. □

5.3.2 Composition

Now we define the binary composition of two DCR Graphs. Intuitively, the *composition* of G_1 and G_2 glues together the events that are both in G_1 and G_2.

Definition 5.3.3. *Formally, the composite* $\mathrm{G}_1 \oplus \mathrm{G}_2 = (\mathrm{E}, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathrm{L}, l)$, *where* $\mathrm{G}_i = (\mathrm{E}_i, \mathrm{M}_i, \rightarrow\bullet_i, \bullet\rightarrow_i, \rightarrow\diamond_i, \rightarrow+_i, \rightarrow\%_i, \mathrm{L}_i, l_i)$, $\mathrm{M}_i = (\mathrm{Ex}_i, \mathrm{Re}_i, \mathrm{In}_i)$ *for* $i \in \{1, 2\}$ *and:*

(i) $\mathrm{E} = (\mathrm{E}_1 \cup \mathrm{E}_2)$

(ii) $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$, *where:*

- *(a)* $\mathrm{Ex} = \mathrm{Ex}_1 \cup \mathrm{Ex}_2$
- *(b)* $\mathrm{In} = \mathrm{In}_1 \cup \mathrm{In}_2$
- *(c)* $\mathrm{Re} = \mathrm{Re}_1 \cup \mathrm{Re}_2$

(iii) $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ *for each* $\rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%\}$

(iv) $l(e) = l_1(e) \cup l_2(e)$

(v) $L = L_1 \cup L_2$

(iib) states that events are included, if they're either included in G_1 or G_2. *(iic)* states that the events with pending responses are those events that have a pending response in G_1 or G_2.

Definition 5.3.4. *The composition* $G_1 \oplus G_2$ *is* well-defined *when:*

(i) $\forall (e \in E_1 \cap E_2 \mid (e \in Ex_1 \Leftrightarrow e \in Ex_2)$

(ii) $\forall (e \in E_1 \cap E_2 \mid (e \in In_1 \Leftrightarrow e \in In_2)$

(iii) $\forall (e \in E_1 \cap E_2 \mid (e \in Re_1 \Leftrightarrow e \in Re_2)$

(iv) $\forall (e, e' \in E_1 \cap E_2 \mid \neg((e \rightarrow+_1 e' \wedge e \rightarrow\%_2 e') \vee (e \rightarrow\%_1 e' \wedge e \rightarrow+_2 e')))$

(i) ensures that those events that will be glued together have the same execution marking. *(ii)* ensures that events that will be glued together and in both DCR Graphs belong to either the set of internal events or the set of events that have a condition/milestone relation towards an internal event, have the same inclusion marking. *(iii)* ensures that events that will be glued together and in both DCR Graphs belong to the set of internal events have the same pending response marking. *(iv)* ensures that by composing the two DCR Graphs no event both includes and excludes the same event. If $G_1 \oplus G_2$ is well-defined, then we also say that G_1 and G_2 are *composable* with respect to each other.

Lemma 5.3.1. *If* $(L, \cdot)$ *is a commutative monoid, then the composition operator* $\oplus$ *is commutative.*

Proof. According to definition 5.3.3, most elements of the tuple defining the graph $G = G_1 \oplus G_2$ are constructed from the union of the same elements in G_1 and G_2. For these elements the composition is commutative, because the union operator is commutative. The exception is the labelling function, which is composed through the monoid operator $\cdot$. If the monoid is commutative then the composition is commutative for the labelling function as well. □

Lemma 5.3.2. *The composition operator* $\oplus$ *is associative.*

Proof. According to definition 5.3.3, most elements of the tuple defining the graph $G = G_1 \oplus G_2$ are constructed from the union of the same elements in G_1 and G_2. For these elements the composition is associative, because the union operator is associative. The exception is the labelling function, which is composed through the monoid operator $\cdot$. Because a monoid operator is always associative, the composition is associative for the labelling function as well. □

Definition 5.3.5. *We call a vector* $\Delta = \delta_1 \ldots \delta_k$ *of projection parameters* covering *for some DCR Graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *if:*

1. $\bigcup\limits_{i \in [k]} \delta_{\mathrm{E}i} = \mathrm{E}$ *and*

2. $(\forall a \in \mathrm{L}.\forall e \in \mathrm{E}.a \in l(e) \Rightarrow (\exists i \in [k].e \in \delta_{\mathrm{E}i} \land a \in \delta_{\mathrm{L}i})$

Proposition 5.3.4. *If some vector* $\Delta = \delta_1 \dots \delta_k$ *of projection parameters is covering for some DCR Graph* G *then:* $\bigoplus\limits_{i \in [k]} \mathrm{G}_{|\delta_i} \;=\; \mathrm{G}$

Proof. Since the vector of projection parameters is covering, every event and label is covered in at least one of the projections. Moreover the definition of composition 5.3.3, is defined over union of individual components. Hence when all projections are composed, we will get the same graph and hence $\bigoplus\limits_{i \in [k]} \mathrm{G}_{|\delta_i} \;=\; \mathrm{G}$.

□

5.3.3 Safe Distributed Synchronous Execution of DCR Graphs

In this section we define networks of synchronously communicating DCR Graphs and prove the main technical theorem of the paper stating that a network of synchronously communicating DCR Graphs obtained by projecting a DCR Graph G with respect to a covering set of projection parameters has the same behavior as the original graph G.

We now define networks of DCR Graphs and their distributed execution.

Definition 5.3.6. *A* network of DCR Graphs *is a finite vector of DCR Graphs* $\overline{\mathrm{G}}$ *sometimes written as* $\Pi_{i \in [n]}\mathrm{G}_i$ *or* $\mathrm{G}_0\mathrm{G}_2 \dots \mathrm{G}_{n-1}$. *Assuming* $\mathrm{G}_i = (\mathrm{E}_i, \mathrm{M}_i, \rightarrow\!\bullet_i, \bullet\!\rightarrow_i, \rightarrow\!\diamond_i$ $, \rightarrow\!+_i, \rightarrow\!\%_i, \mathrm{L}_i, l_i)$, *we define the set of events of the network by* $\mathcal{E}(\Pi_{i \in [n]}\mathrm{G}_i) = \cup_{i \in [n]}\mathrm{E}_i$ *and the set of labels of the network by* $\mathcal{L}(\Pi_{i \in [n]}\mathrm{G}_i) = \cup_{i \in [n]}\mathrm{L}_i$ *and we write the network marking as* $\overline{\mathrm{M}} = \Pi_{i \in [n]}\mathrm{M}_i$.

Finally, let $\mathcal{M}(\overline{\mathrm{G}})$ *denote the set of network markings of* $\overline{\mathrm{G}}$.

We now define when an event is locally enabled in one of the components. and the result of executing an event as the same as locally executing the event in all components of the network sharing the event.

Definition 5.3.7. *For a network of DCR Graphs* $\overline{\mathrm{G}} = \Pi_{i \in [n]}\mathrm{G}_i$ *where* $\mathrm{G}_i = (\mathrm{E}_i, \mathrm{M}_i, \rightarrow\!\bullet_i$ $, \bullet\!\rightarrow_i, \rightarrow\!+_i, \rightarrow\!\%_i, \mathrm{L}_i, l_i)$, *an event* $e \in \mathcal{E}(\Pi_{i \in [n]}\mathrm{G}_i)$ *is enabled at a location* i *in the distributed marking* $\overline{\mathrm{M}} = \Pi_{i \in [n]}\mathrm{M}_i$, *written* $\overline{\mathrm{M}} \vdash_{\overline{\mathrm{G}},i} e$, *if it is locally enabled in the ith dynamic condition response graph, i.e.* $e \in \mathrm{E}_i \land \mathrm{M}_i \vdash_{\mathrm{G}_i} e$. *The result of executing an event* $e \in \mathcal{E}(\Pi_{i \in [n]}\mathrm{G}_i)$ *in a marking* $\overline{\mathrm{M}} \oplus_{\overline{\mathrm{G}_i}} e = \overline{\mathrm{M}} = \Pi_{i \in [n]}\mathrm{M}_i$ *is the new marking* $\overline{\mathrm{M}}' = \Pi_{i \in [n]}\mathrm{M}'_i$ *where* $\mathrm{M}'_i = \mathrm{M}_i \oplus_{\mathrm{G}} e$ *if* $e \in \mathrm{E}_i$ *and* $\mathrm{M}'_i = \mathrm{M}_i$ *otherwise.*

Finally, we define executions of networks as follows. An event can be executed if it is locally enabled in a component where it has assigned at least one label.

Definition 5.3.8. *For a network of DCR Graphs* $\overline{G} = \Pi_{i\in[n]} G_i$ *where* $G_i = (E_i, M_i, \rightarrow\!\bullet_i, \bullet\!\!\rightarrow_i, \rightarrow\!\diamond_i, \rightarrow\!+_i, \rightarrow\!\%_i, L_i, l_i)$ *and* $\overline{M} = \Pi_{i\in[n]} M_i$*, we define an* execution *of* $\overline{G}$ *to be a (finite or infinite) sequence of tuples* $\{(h_i, \overline{M}_i, e_i, a_i, \overline{M}'_i)\}_{i\in[k]}$ *each consisting of a* place $h_i \in [n]$*, a network marking, an event, a label and another network marking (the result of executing the event) such that* $\overline{M} = \overline{M}_0$ *and* $\forall i \in [k].a_i \in l_{h_i}(e_i) \wedge \overline{M}_i \vdash_{\overline{G},h_i} e_i \wedge \overline{M}'_i = \overline{M}_i \oplus_{\overline{G}} e_i$ *and* $\forall i \in [k-1].\overline{M}'_i = \overline{M}_{i+1}$*. We say the execution is* accepting *if* $\forall i \in [k], h \in [n].\big(\forall e \in In_{h,i} \cap Re_{h,i}.\exists j \geq i.e_j = e \vee e \notin In'_{h,j})\big)$*, where* $\overline{M}_i = \Pi_{h\in[n]}(Ex_{h,i}, In_{h,i}, Re_{h,i})$ *and* $\overline{M}'_j = \Pi_{h\in[n]}(Ex'_{h,j}, In'_{h,j}, Re'_{h,j})$*.*

Now we will define the transition system for a network of DCR Graphs as follows,

Definition 5.3.9. *For a network of DCR Graphs* $\overline{G} = \Pi_{i\in[n]} G_i$ *where* $G_i = (E_i, M_i, \rightarrow\!\bullet_i, \bullet\!\!\rightarrow_i, \rightarrow\!\diamond_i, \rightarrow\!+_i, \rightarrow\!\%_i, L_i, l_i)$ *and* $\overline{M} = \Pi_{i\in[n]} M_i$*, we define the corresponding labelled transition system* $TS(\overline{G})$ *to be the tuple*

$$(\mathcal{M}(\overline{G}), \overline{M}, \mathcal{L}_{ts}(\overline{G}), \rightarrow_N)$$

where $\mathcal{L}_{ts}(\overline{G}) = \mathcal{E}(\Pi_{i\in[n]} G_i) \times \mathcal{L}(\Pi_{i\in[n]} G_i)$ *is the set of labels of the transition system,* $\overline{M}$ *is the initial marking, and* $\rightarrow_N \subseteq \mathcal{M}(\overline{G}) \times \mathcal{L}_{ts}(\overline{G}) \times \mathcal{M}(\overline{G})$ *is the transition relation defined by* $\overline{M} \xrightarrow{(e,a)}_N \overline{M} \oplus_{\overline{G},i} e$ *if* $\overline{M} \vdash_{\overline{G},i} e$ *and* $a \in l_i(e)$*.*

We define a run $a_0, a_1, \ldots$ *of the transition system to be a sequence of labels of a sequence of transitions* $\overline{M}_i \xrightarrow{(e_i,a_i)} \overline{M}_{i+1}$ *starting from the initial marking. We define a run to be accepting (or completed) if for the underlying sequence of transitions it holds that if* $\forall i \in [k], h \in [n].\big(\forall e \in In_{h,i} \cap Re_{h,i}.\exists j \geq i.e_j = e \vee e \notin In'_{h,j})\big)$*, where* $\overline{M}_i = \Pi_{h\in[n]}(Ex_{h,i}, In_{h,i}, Re_{h,i})$ *and* $\overline{M}'_j = \Pi_{h\in[n]}(Ex'_{h,j}, In'_{h,j}, Re'_{h,j})$*. In words, a run is accepting/completed if no required response event is continuously included and pending without it happens or become excluded.*

Now we define binary relation between a global DCR Graph and a network of projected DCR Graphs as follows in def 5.3.10.

Definition 5.3.10. *For a Dynamic Condition Response Graph* $G = (E, M, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ *and for a covering vector of projection parameters* $\Delta = \delta_1 \ldots \delta_n$*, for a network of projected graphs where* $\overline{G}_\Delta = \Pi_{i\in[n]} G_{|\delta_i}$ *with* $G_{|\delta_i} = (E_{|\delta_i}, M_{|\delta_i}, \rightarrow\!\bullet_{|\delta_i}, \bullet\!\!\rightarrow_{|\delta_i}, \rightarrow\!\diamond_{|\delta_i}, \rightarrow\!+_{|\delta_i}, \rightarrow\!\%_{|\delta_i}, \delta_{L_i}, l_{|\delta_i})$ *and* $\overline{M}_\Delta = \Pi_{i\in[n]} M_{|\delta_i}$ *, we define the binary relation between* $TS(G)$ *and* $TS(\overline{G}_\Delta)$ *as* $\mathcal{R} = \{(M, \Pi_{i\in[n]} M_{|\delta_i}) \mid M \in \mathcal{M}(M)\}$*.*

Theorem 5.3.1. *For a Dynamic Condition Response Graph* G *and a covering vector of projection parameters* $\Delta = \delta_1 \ldots \delta_n$ *it holds that* $TS(G)$ *is bisimilar to* $TS(\overline{G}_\Delta)$*, where* $\overline{G}_\Delta = \Pi_{i\in[n]} G_{|\delta_i}$*. Moreover, a run is accepting in* $TS(G)$ *if and only if the bisimilar run is accepting in* $TS(\overline{G}_\Delta)$*.*

Proof. For DCR Graph $G = (E, M, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$, the corresponding labeled transition system (def 3.3.10) is $TS(G) = (\mathcal{M}(G), M_0, \mathcal{L}_{ts}(G), \rightarrow)$ where $\mathcal{L}_{ts}(G) = E \times L$ is the set of labels of the transition system, M_0 is the initial marking, and $\rightarrow \subseteq \mathcal{M}(G) \times \mathcal{L}_{ts}(G) \times \mathcal{M}(G)$ is the transition relation defined by $M \xrightarrow{(e,a)} M \oplus_G e$ if

$\mathrm{M} \vdash_{\mathrm{G}} e$ and $a \in \mathrm{l}(e)$.

For a network of projected graphs where $\overline{\mathrm{G}}_{\Delta} = \Pi_{i \in [n]} \mathrm{G}_{|\delta_i}$ with $\mathrm{G}_{|\delta_i} = (\mathrm{E}_{|\delta_i}, \mathrm{M}_{|\delta_i}, \rightarrow\!\bullet_{|\delta_i}, \bullet\!\!\rightarrow_{|\delta_i}, \rightarrow\!\diamond_{|\delta_i}, \rightarrow\!+_{|\delta_i}, \rightarrow\!\%_{|\delta_i}, \delta_{L_i}, \mathrm{l}_{|\delta_i})$ and $\overline{\mathrm{M}}_{\Delta} = \Pi_{i \in [n]} \mathrm{M}_{|\delta_i}$, the corresponding label transition system according to def 5.3.9, $\mathrm{TS}(\overline{\mathrm{G}}_{\Delta}) = (\mathcal{M}(\overline{\mathrm{G}}_{\Delta}), \overline{\mathrm{M}_{0\Delta}}, \mathcal{L}_{ts}(\overline{\mathrm{G}}_{\Delta}), \rightarrow_N)$ where $\mathcal{L}_{ts}(\overline{\mathrm{G}}_{\Delta}) = \mathcal{E}(\Pi_{i \in [n]} \mathrm{G}_{|\delta_i}) \times \mathcal{L}(\Pi_{i \in [n]} \mathrm{G}_{|\delta_i})$ is the set of labels of the transition system, $\overline{\mathrm{M}_{0\Delta}}$ is the initial marking, and $\rightarrow_N \subseteq \mathcal{M}(\overline{\mathrm{G}}_{\Delta}) \times \mathcal{L}_{ts}(\overline{\mathrm{G}}_{\Delta}) \times \mathcal{M}(\overline{\mathrm{G}}_{\Delta})$ is the transition relation defined by $\overline{\mathrm{M}}_{\Delta} \xrightarrow{(e,a)}_N \overline{\mathrm{M}}_{\Delta} \oplus_{\overline{\mathrm{G}},i} e$ if $\overline{\mathrm{M}_{\Delta}} \vdash_{\overline{\mathrm{G}}_{\Delta},i} e$ and $a \in \mathrm{l}_{|\delta_i}(e)$.

Here we have to show that $TS(\mathrm{G}) \sim \mathrm{TS}(\overline{\mathrm{G}}_{\Delta})$. In order to show that both label transition systems are bisimilar, we have to prove the equivalence of binary relation $\mathcal{R}$.

According to def 5.3.10, we have the binary relation $\mathcal{R} = \{(\mathrm{M}, \Pi_{i \in [n]} \mathrm{M}_{|\delta_i}) \mid \mathrm{M} \in \mathcal{M}(\mathrm{M})\}$ between $TS(\mathrm{G})$ and $\mathrm{TS}(\overline{\mathrm{G}}_{\Delta})$. In order to show that $TS(\mathrm{G}) \sim \mathrm{TS}(\overline{\mathrm{G}}_{\Delta})$, we have to show the following

(A) if $\mathrm{M} \xrightarrow{e,a} \mathrm{M}'$ in $TS(\mathrm{G})$ then there exists in $\mathrm{TS}(\overline{\mathrm{G}}_{\Delta})$ a transition $\overline{\mathrm{M}}_{\Delta} \xrightarrow{e,a} \overline{\mathrm{M}'_{\Delta}}$.

According to def 3.3.10 on execution of an event in DCR Graph, $\mathrm{M} \xrightarrow{(e,a)} \mathrm{M} \oplus_{\mathrm{G}} e$ if $\mathrm{M} \vdash_{\mathrm{G}} e$ and $a \in \mathrm{l}(e)$.

According to proposition 5.3.1, for $e \in \delta_{\mathrm{E}}$ and $a \in \delta_{\mathrm{L}}$ it holds that $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$ if and only if $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}''$.

Hence for $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$, we will have following changes in the distributed marking,

- According to proposition 5.3.1 for interface events of projections, for all projections $i \in [k]$ where $e \in \delta_{E_i}$ and $a \in \delta_{L_i}$ will have $\mathrm{M}_{|\delta_i} \vdash_{\mathrm{G}_{|\delta_i}} e \wedge \mathrm{M}_{|\delta_i} \oplus_{\mathrm{G}_{|\delta_i}} e = \mathrm{M}'_{|\delta_i}$
- According to proposition 5.3.3 for external events of projections, for all projections $i \in [k]$ where $e \in \mathrm{E}_{|\delta_i}$ (and $a \notin \delta_{L_i}$) will have $\mathrm{M}_{|\delta_i} \oplus_{\mathrm{G}_{|\delta_i}} e = \mathrm{M}'_{|\delta_i}$.
- According to proposition 5.3.2 for projections where the event does not belongs to, for all projections $i \in [k]$ where $e \notin \mathrm{E}_{|\delta_i}$ will have $\mathrm{M}_{|\delta_i} = \mathrm{M}'_{|\delta_i}$.

Based on the above changes in the projections and $\delta_{E_i} \subseteq \mathrm{E}_{|\delta_i}$, the new marking distributed network will be $\overline{\mathrm{M}'_{\Delta}} = \Pi_{i \in [n]} \mathrm{M}'_{|\delta_i}$ where $\mathrm{M}'_{|\delta_i} = \mathrm{M}_{|\delta_i} \oplus_{\mathrm{G}} e$ if $e \in \mathrm{E}_{|\delta_i}$ and $\mathrm{M}'_{|\delta_i} = \mathrm{M}_{|\delta_i}$ otherwise.

The new marking in distributed network $\overline{\mathrm{M}'_{\Delta}} = \Pi_{i \in [n]} \mathrm{M}'_{|\delta_i}$ is same as executing a local event in a projection according to def 5.3.7.

Hence we can conclude that if $\mathrm{M} \xrightarrow{e,a} \mathrm{M}'$ in $TS(\mathrm{G})$ then there exists in $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$ a transition $\overline{\mathrm{M}}_\Delta \xrightarrow{e,a} \overline{\mathrm{M}'}_\Delta$.

(B) if $\overline{\mathrm{M}}_\Delta \xrightarrow{e,a} \overline{\mathrm{M}'}_\Delta$ in $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$ then there exists in $TS(\mathrm{G})$ a transition $\mathrm{M} \xrightarrow{e,a} \mathrm{M}'$.

According to definition of labeled transition system for network of DCR Graphs (def 5.3.9), the $\overline{\mathrm{M}}_\Delta \xrightarrow{(e,a)}_N \overline{\mathrm{M}}_\Delta \oplus_{\overline{\mathrm{G}},i} e$ if $\overline{\mathrm{M}}_\Delta \vdash_{\overline{\mathrm{G}},i} e$ and $a \in \mathrm{l}_i(e)$.

Further according to execution of a event in distributed marking (def 5.3.7), $\overline{\mathrm{M}}_\Delta \vdash_{\overline{\mathrm{G}},i} e$, if there is a locally enabled in the ith dynamic condition response graph, i.e. $e \in \delta_{E_i} \wedge \mathrm{M}_{|\delta_i} \vdash_{\mathrm{G}_{|\delta_i}} e$, $a \in \mathrm{l}_{|\delta_i}(e)$ and result of executing the event in local component $\mathrm{M}'_{|\delta_i} = \mathrm{M}_{|\delta_i} \oplus_{\mathrm{G}_{|\delta_i}} e$.

According to proposition 5.3.1, if we have an event enabled with a label in a projection, then we can have the same event enabled in the global graph i.e. $e \in \delta_\mathrm{E}$ and $a \in \delta_\mathrm{L}$. $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'' \implies \mathrm{M} \vdash_\mathrm{G} e \wedge \mathrm{M} \oplus_\mathrm{G} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$.

Hence $e \in \delta_{E_i} \wedge a \in \mathrm{l}_{|\delta_i}(e) \wedge \mathrm{M}_{|\delta_i} \vdash_{\mathrm{G}_{|\delta_i}} e \wedge \mathrm{M}'_{|\delta_i} = \mathrm{M}_{|\delta_i} \oplus_{\mathrm{G}_{|\delta_i}} e, \implies \mathrm{M} \vdash_\mathrm{G} e \wedge \mathrm{M} \oplus_\mathrm{G} e = \mathrm{M}'$, which is a condition for making a transition $\mathrm{M} \xrightarrow{(e,a)} \mathrm{M}'$.

Therefore we can conclude that if $\overline{\mathrm{M}}_\Delta \xrightarrow{e,a} \overline{\mathrm{M}'}_\Delta$ in $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$ then there exists in $TS(\mathrm{G})$ a transition $\mathrm{M} \xrightarrow{e,a} \mathrm{M}'$.

By proving the equivalence in both directions, we can conclude that $TS(\mathrm{G}) \sim \mathrm{TS}(\overline{\mathrm{G}}_\Delta)$.

We will now prove that a run is accepting in TS(G) if and only if the bisimilar run is accepting in $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$.

Since $\mathrm{TS(G)} \sim \mathrm{TS}(\overline{\mathrm{G}}_\Delta)$, both TS(G) and $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$ will have same runs.

Let's say that a run in $\mathrm{TS}(\overline{\mathrm{G}}_\Delta)$ is accepting. According to def 5.3.9, a run in labelled transition system for network of DCR Graphs is accepting if for the underlying sequence of transitions it holds that if $\forall i \in [k], h \in [n]. \big(\forall e \in \mathrm{In}_{h,i} \cap \mathrm{Re}_{h,i}. \exists j \geq i. e_j = e \vee e \notin \mathrm{In}'_{h,j}\big)\big)$, where $\overline{\mathrm{M}}_i = \Pi_{h\in[n]}(\mathrm{Ex}_{h,i}, \mathrm{In}_{h,i}, \mathrm{Re}_{h,i})$ and $\overline{\mathrm{M}}'_j = \Pi_{h\in[n]}(\mathrm{Ex}'_{h,j}, \mathrm{In}'_{h,j}, \mathrm{Re}'_{h,j})$. In words, a run is accepting/completed if no required response event is continuously included and pending without it happens or become excluded.

In the network of projected graphs, $\forall i \in [k], h \in [n]. \big(\forall e \in \mathrm{In}_{|\delta_{h,i}} \cap \mathrm{Re}_{|\delta_{h,i}}. \exists j \geq i. e_j = e \vee e \notin \mathrm{In}'_{|\delta_{h,i}}\big)\big)$, where $\overline{\mathrm{M}}_{\Delta_i} = \Pi_{h\in[n]}(\mathrm{Ex}_{|\delta_{h,i}}, \mathrm{In}_{|\delta_{h,i}}, \mathrm{Re}_{|\delta_{h,i}})$ and

$\overline{\mathsf{M}}'_{\Delta_j} = \sqcap_{h\in[n]}(\mathsf{Ex}'_{|\delta_{h,i}}, \mathsf{In}'_{|\delta_{h,i}}, \mathsf{Re}'_{|\delta_{h,i}})$.

According to proposition 5.3.1, if there is an enabled event in the local marking with label, then we can find the same transition in global graph and moreover we also have same runs in both TS(G) and TS($\overline{\mathsf{G}}_\Delta$).

Hence $\forall i \in [k], h \in [n].\big(\forall e \in \mathsf{In}_{|\delta_{h,i}} \cap \mathsf{Re}_{|\delta_{h,i}}.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_{|\delta_{h,i}})\big) \implies \forall i \in [k], e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.((e = e_j \vee e \notin \mathsf{In}'_j))$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}_j = (\mathsf{Ex}_j, \mathsf{In}_j, \mathsf{Re}_j)$ in the global graph.

According to definition of LTS for def 3.3.10, a run $a_0, a_1, \ldots$ of the transition system to be a sequence of labels of a sequence of transitions $\mathsf{M}_i \xrightarrow{(e_i,a_i)} \mathsf{M}_{i+1}$ starting from the initial marking and a run to be accepting (or completed) if for the underlying sequence of transitions it holds that $\forall i \geq 0, e \in \mathsf{Re}_i.\exists j \geq i.((e = e_j \vee e \notin \mathsf{In}_{j+1}))$.

Hence we can conclude from $\forall i \in [k], e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.((e = e_j \vee e \notin \mathsf{In}'_j))$, where $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ and $\mathsf{M}_j = (\mathsf{Ex}_j, \mathsf{In}_j, \mathsf{Re}_j)$ that the run is accepting in TS(G).

Therefore a run is accepting in TS(G) if and only if the bisimilar run is accepting in TS($\overline{\mathsf{G}}_\Delta$).

□

5.3.4 Distribution of Case Management Example

Figure 5.2 below shows a modified version of case management example in DCR Graphs taken from the case study described in Sec 4.1.3, primarily focusing on meeting management and abstracting from the other parts of the case study. As explained in the case study, the meeting management example involves three participants: Landsorganisationen i Danmark (LO) (overarching organization for most of the trade unions in Denmark), Dansk Arbejdsgiverforening (DA) (Danish employers organizations) and employees trade union (U).

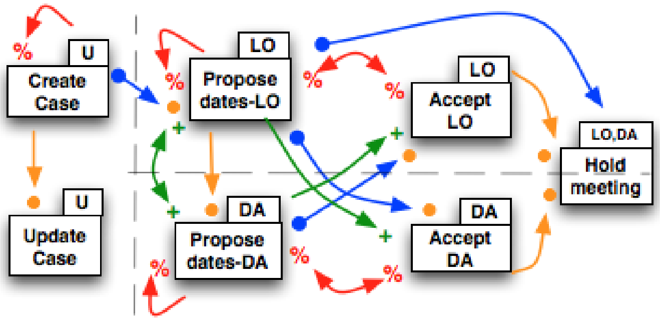
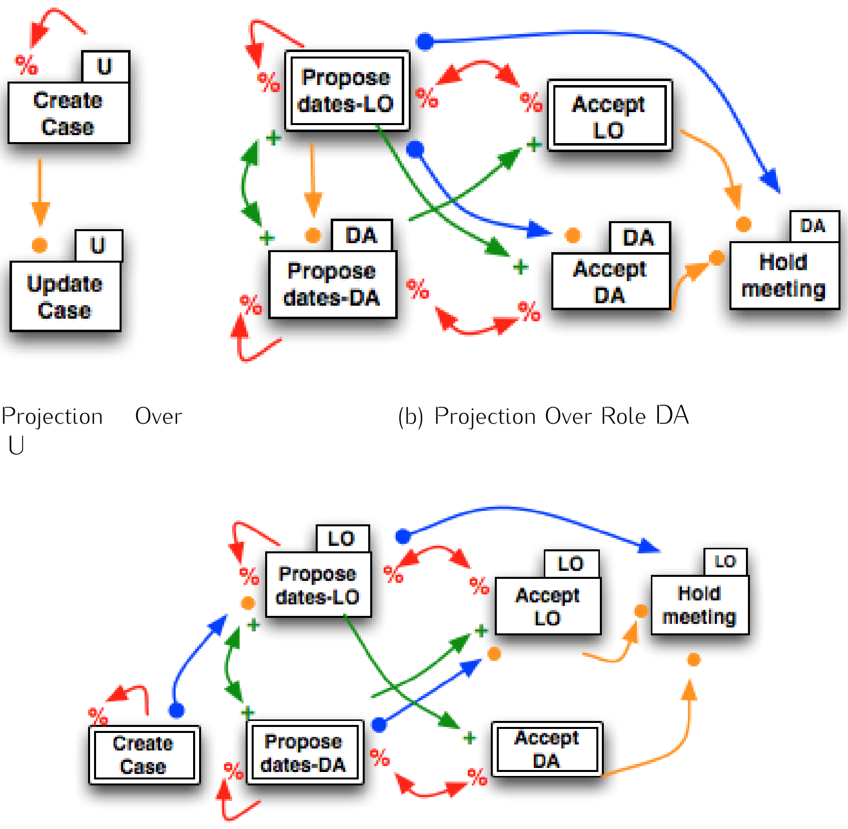


Figure 5.2: Arrange meeting cross-organizational case management example

The DCR Graph shown in the figure 5.2 has 7 events, drawn as boxes with "ears", and captures a process of creating a case, agreeing on meeting dates and holding meetings. The names of the events are written inside the box and the set of actions for each event, representing the roles that can execute the event, is written inside the "ear". That is, the event **Create Case** in the upper left has label **U** and represents the creation of a case by a case manager at a union (role **U**). The rightmost event, **Hold meeting** has two different labels, **LO** and **DA**, representing a meeting held by **LO** and **DA** (the umbrella organization of employers) respectively. The formal specification of global DCR Graph for arrange meeting example (shown in figure 5.2) is given in the listing 5.1.



(a) Projection Over Role U

(b) Projection Over Role DA

(c) Projection Over Role LO

Figure 5.3: Projecting of Arrange Meeting Example Over Roles

Listing 5.1: Formal specification of arrange meeting arrangement example

We will use the following abbreviations for the event names in the example.
Create Case(**Cc**), Update Case(**Uc**), Propose dates-LO(**PdLO**),
Propose dates-DA(**PdDA**), Accept-LO(**ALO**), Accept-DA(**ADA**),
Hold meeting(**Hm**).

DCR Graph $G = (E, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ where
$E = \{\mathbf{Cc}, \mathbf{Uc}, \mathbf{PdLO}, \mathbf{PdDA}, \mathbf{ALO}, \mathbf{ADA}, \mathbf{Hm}\}$
$M = (\emptyset, \emptyset, E)$
$\rightarrow\!\bullet = \{(\mathbf{Cc}, \mathbf{Uc}), (\mathbf{Cc}, \mathbf{PdLO}), (\mathbf{PdLO}, \mathbf{PdDA}), (\mathbf{PdLO}, \mathbf{ADA}), (\mathbf{PdDA}, \mathbf{ALO}),$
$(\mathbf{ALO}, \mathbf{Hm}), (\mathbf{ADA}, \mathbf{Hm})\}$
$\bullet\!\rightarrow = \{(\mathbf{Cc}, \mathbf{PdLO}), (\mathbf{PdLO}, \mathbf{ADA}), (\mathbf{PdDA}, \mathbf{ALO}), (\mathbf{PdLO}, \mathbf{Hm})\}$
$\rightarrow\!\diamond = \emptyset$
$\rightarrow\!+ = \{(\mathbf{PdLO}, \mathbf{PdDA}), (\mathbf{PdDA}, \mathbf{PdLO}), (\mathbf{PdLO}, \mathbf{ADA}), (\mathbf{PdDA}, \mathbf{ALO})\}$
$\rightarrow\!\% = \{(\mathbf{Cc}, \mathbf{Cc}), (\mathbf{PdLO}, \mathbf{PdLO}), (\mathbf{PdDA}, \mathbf{PdDA}), (\mathbf{PdLO}, \mathbf{ALO}),$
$(\mathbf{ALO}, \mathbf{PdLO}), (\mathbf{PdDA}, \mathbf{ADA}), (\mathbf{ADA}, \mathbf{PdDA})\}$

$L = \{(\mathbf{Cc}, U), (\mathbf{Uc}, U), (\mathbf{PdLO}, LO), (\mathbf{PdDA}, DA), (\mathbf{ALO}, LO), (\mathbf{ADA}, DA),$
$(\mathbf{Hm}, LO), (\mathbf{Hm}, DA)\}$
$l = \{(\mathbf{Cc}, (\mathbf{Cc}, U)), (\mathbf{Uc}, (\mathbf{Uc}, U)), (\mathbf{PdLO}, (\mathbf{PdLO}, LO)), (\mathbf{PdDA}, (\mathbf{PdDA}, DA)),$
$(\mathbf{ALO}, (\mathbf{ALO}, LO)), (\mathbf{ADA}, (\mathbf{ADA}, DA)), (\mathbf{Hm}, (\mathbf{Hm}, LO)), (\mathbf{Hm}, (\mathbf{Hm}, DA))\}$

Now we will project the global DCR graph for arrange meeting over participant roles (LO, DA, U) and events belongs to them. As shown in the figure, the projected DCR Graph subgraphs contains both the events internal and interface events as defined ib the definition 5.3.1. The interface events or external events are marked without labels (boxes marked with double lines and no ears). Further all the relations between the events that are necessary for the projected graph (as defined in the definition 5.3.1-iv to 5.3.1-viii) will also be included. The formal specification of the projected graphs for arrange meeting example worked out according to the definition 5.3.1 is given below in the listing 5.2.

Listing 5.2: Formal specification of projected DCR graphs for arrange meeting example

Global DCR graph from the listing 5.1 $G = (E, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$

Projection over role U
projection parameter $\delta = (\delta_E, \delta_L)$ where
$\delta_E = \{\mathbf{Cc}, \mathbf{Uc}\}$ and $\delta_L = \{(\mathbf{Cc}, U), (\mathbf{Uc}, U)\}$

The projected DCR graph over events and labels belonging to U
$G_{|\delta} = (E_{|\delta}, M_{|\delta}, \rightarrow\!\bullet_{|\delta}, \bullet\!\rightarrow_{|\delta}, \rightarrow\!\diamond_{|\delta}, \rightarrow\!+_{|\delta}, \rightarrow\!\%_{|\delta}, \delta_L, l_{|\delta})$ where
$E_{|\delta} = \delta_E = \{\mathbf{Cc}, \mathbf{Uc}\}$

$M_{|\delta} = (Ex_{|\delta}, Re_{|\delta}, In_{|\delta})$ where
$Ex_{|\delta} = Ex \cap E_{|\delta} = \emptyset$
$Re_{|\delta} = Re \cap (\delta_E \cup \rightarrow\!\diamond\ \delta_E) = \emptyset$
$In_{|\delta} = In \cap E_{|\delta} = E_{|\delta}$

$\rightarrow\!\bullet_{|\delta} = \{(\mathbf{Cc}, \mathbf{Uc})\}$
$\bullet\!\rightarrow_{|\delta} = \emptyset$
$\rightarrow\!\diamond_{|\delta} = \emptyset$
$\rightarrow\!+_{|\delta} = \emptyset$
$\rightarrow\!\%_{|\delta} = \{(\mathbf{Cc}, \mathbf{Cc})\}$

$l_{|\delta} = \{(\mathbf{Cc}, (\mathbf{Cc}, U)), (\mathbf{Uc}, (\mathbf{Uc}, U))\}$

Projection over role LO
projection parameter $\delta = (\delta_E, \delta_L)$ where

$\delta_E = \{\mathrm{PdLO}, \mathrm{ALO}, \mathrm{Hm}\}$ and
$\delta_L = \{(\mathrm{PdLO}, \mathrm{LO}), (\mathrm{ALO}, \mathrm{LO}), (\mathrm{Hm}, \mathrm{LO})\}$

The projected DCR graph over events and labels belonging to LO
$G_{|\delta} = (E_{|\delta}, M_{|\delta}, \rightarrow\!\bullet_{|\delta}, \bullet\!\rightarrow_{|\delta}, \rightarrow\!\diamond_{|\delta}, \rightarrow\!+_{|\delta}, \rightarrow\!\%_{|\delta}, \delta_L, l_{|\delta})$ where

$E_{|\delta} = \{\mathrm{PdLO}, \mathrm{ALO}, \mathrm{Hm}, \mathrm{Cc}, \mathrm{PdDA}, \mathrm{ADA}\}$

$M_{|\delta} = (Ex_{|\delta}, Re_{|\delta}, In_{|\delta})$ where
$Ex_{|\delta} = Ex \cap E_{|\delta} = \emptyset$
$Re_{|\delta} = Re \cap (\delta_E \cup \rightarrow\!\diamond\ \delta_E) = \emptyset$
$In_{|\delta} = In \cap E_{|\delta} = E_{|\delta}$

$\rightarrow\!\bullet_{|\delta} = \{(\mathrm{Cc}, \mathrm{PdLO}), (\mathrm{PdDA}, \mathrm{ALO}), (\mathrm{ALO}, \mathrm{Hm}), (\mathrm{ADA}, \mathrm{Hm})\}$
$\bullet\!\rightarrow_{|\delta} = \{(\mathrm{Cc}, \mathrm{PdLO}), (\mathrm{PdDA}, \mathrm{ALO}), (\mathrm{PdLO}, \mathrm{Hm})\}$
$\rightarrow\!\diamond_{|\delta} = \emptyset$
$\rightarrow\!+_{|\delta} = \{(\mathrm{PdLO}, \mathrm{PdDA}), (\mathrm{PdDA}, \mathrm{PdLO}), (\mathrm{PdLO}, \mathrm{ADA}), (\mathrm{PdDA}, \mathrm{ALO})\}$
$\rightarrow\!\%_{|\delta} = \rightarrow\!\%$

$l_{|\delta} = \{(\mathrm{PdLO}, (\mathrm{PdLO}, \mathrm{LO})), (\mathrm{ALO}, (\mathrm{ALO}, \mathrm{LO})), (\mathrm{Hm}, (\mathrm{Hm}, \mathrm{LO}))\}$

Projection over role DA
projection parameter $\delta = (\delta_E, \delta_L)$ where
$\delta_E = \{\mathrm{PdDA}, \mathrm{ADA}, \mathrm{Hm}\}$ and
$\delta_L = \{(\mathrm{PdDA}, \mathrm{DA}), (\mathrm{ADA}, \mathrm{DA}), (\mathrm{Hm}, \mathrm{DA})\}$

The projected DCR graph over events and labels belonging to DA
$G_{|\delta} = (E_{|\delta}, M_{|\delta}, \rightarrow\!\bullet_{|\delta}, \bullet\!\rightarrow_{|\delta}, \rightarrow\!\diamond_{|\delta}, \rightarrow\!+_{|\delta}, \rightarrow\!\%_{|\delta}, \delta_L, l_{|\delta})$ where

$E_{|\delta} = \{\mathrm{PdDA}, \mathrm{ADA}, \mathrm{Hm}, \mathrm{PdLO}, \mathrm{ALO}\}$

$M_{|\delta} = (Ex_{|\delta}, Re_{|\delta}, In_{|\delta})$ where
$Ex_{|\delta} = Ex \cap E_{|\delta} = \emptyset$
$Re_{|\delta} = Re \cap (\delta_E \cup \rightarrow\!\diamond\ \delta_E) = \emptyset$
$In_{|\delta} = In \cap E_{|\delta} = E_{|\delta}$

$\rightarrow\!\bullet_{|\delta} = \{(\mathrm{PdLO}, \mathrm{PdDA}), (\mathrm{PdLO}, \mathrm{ADA}), (\mathrm{ALO}, \mathrm{Hm}), (\mathrm{ADA}, \mathrm{Hm})\}$
$\bullet\!\rightarrow_{|\delta} = \{(\mathrm{PdLO}, \mathrm{ADA}), (\mathrm{PdLO}, \mathrm{Hm})\}$
$\rightarrow\!\diamond_{|\delta} = \emptyset$
$\rightarrow\!+_{|\delta} = \{(\mathrm{PdLO}, \mathrm{PdDA}), (\mathrm{PdDA}, \mathrm{PdLO}), (\mathrm{PdLO}, \mathrm{ADA}), (\mathrm{PdDA}, \mathrm{ALO})\}$
$\rightarrow\!\%_{|\delta} = \{(\mathrm{PdLO}, \mathrm{PdLO}), (\mathrm{PdDA}, \mathrm{PdDA}), (\mathrm{PdLO}, \mathrm{ALO}), (\mathrm{ALO}, \mathrm{PdLO}),$
$(\mathrm{PdDA}, \mathrm{ADA}), (\mathrm{ADA}, \mathrm{PdDA})\}$

$l_{|\delta} = \{(\mathrm{PdDA}, (\mathrm{PdDA}, \mathrm{DA})), (\mathrm{ADA}, (\mathrm{ADA}, \mathrm{DA})), (\mathrm{Hm}, (\mathrm{Hm}, \mathrm{DA}))\}$

Further, we will use the arrange meeting example from figure 5.3 and show how events are executed in distributed setting. We assume the arrange meeting example is projected to a network $G_u^1 \mid\mid G_{da}^1 \mid\mid G_{lo}^1$ of three DCR Graphs as shown in the figure 5.3 and we use abbreviations for the event names as described in listing 5.1.

1. Using *sync step*, *local input*, and *input* we get the transition $G_u^1 \mid\mid G_{da}^1 \mid\mid G_{lo}^1 \xrightarrow{(\mathtt{Cc},\mathrm{U})} G_u^2 \mid\mid G_{da}^1 \mid\mid G_{lo}^2$ capturing the local execution of the event `Cc` labelled

with U in G_u^1 which is communicated synchronously to G_{lo}^1. This updates the markings by adding the event Cc to the set of executed events in both G_u^1 and G_{lo}^1. But since Cc has an exclude relation to itself in both G_u^1 and G_{lo}^1 (see Fig. 5.2(a) and 5.2(c)), the event is also excluded from the set of included events in both markings. Finally, because of the response relation to the event PdLO in G_{lo}^1 (see Fig. 5.2(c)), the event PdLO is added to the set of required responses in the resulting marking G_{lo}^2.

2. We can now execute the event PdLO in the DCR graph G_{lo}^2 concurrently with the event Uc in DCR graph G_u^2.

   As the event Uc is only local to G_u^2 we get by using *local step* the transition $G_u^2 \parallel G_{da}^1 \parallel G_{lo}^2 \xrightarrow{(\mathsf{Uc},\mathsf{U})} G_u^3 \parallel G_{da}^1 \parallel G_{lo}^2$ that only updates the marking of G_u^2.

   In addition to being local to G_{lo}^2, the event PdLO is also external event in graph G_{da}^1, so as in the first step by using *sync step local input*, and *input* we get the transition $G_u^3 \parallel G_{da}^1 \parallel G_{lo}^2 \xrightarrow{(\mathtt{PdLO},\mathsf{LO})} G_u^3 \parallel G_{da}^2 \parallel G_{lo}^3$, where the event PdLO has been added to the executed event set of both the marking of G_{da}^1 and G_{lo}^2. Again, because of the self-exclusion relations, the event PdLO is also excluded from the sets of included events in the two markings, and because of the response relations, the events ADA and Hold meeting are added to the set of pending responses in G_{da}^1 and the event Hold meeting is added to the set of pending responses in G_{lo}^2.

3. In response to the dates proposed by LO, the DA may choose to propose new dates by executing the event PdDA in the graph graph G_{da}^2.

   $G_u^3 \parallel G_{da}^2 \parallel G_{lo}^3 \xrightarrow{(\mathtt{PdDA},\mathsf{DA})} G_u^3 \parallel G_{da}^3 \parallel G_{lo}^4$ This triggers the exclusion of the events PdDA and ADA and the inclusion of the events PdLO and ALO in the markings of both G_{da}^2 and G_{lo}^3. It will also include the event ALO in the required response set in the resulting marking G_{lo}^4.

4. Now LO may choose to accept the new dates proposed by DA by executing the event ALO in the graph graph G_{lo}^4, giving the transition

   $G_u^3 \parallel G_{da}^3 \parallel G_{lo}^4 \xrightarrow{(\mathtt{ALO},\mathsf{LO}))} G_u^3 \parallel G_{da}^4 \parallel G_{lo}^5$. This records the event ALO as executed in markings of both G_{da}^4 and G_{lo}^5 and excludes PdLO in both markings (i.e. it is not possible to propose new dates after acceptance).

5. Since the event ALO is recorded as executed in markings of both G_{da}^4 and G_{lo}^5 and the event ADA is excluded, the hold meeting event Hold meeting will be enabled in both graphs G_{lo}^5 and G_{da}^4. The LO may choose to hold the meeting, giving the transition $G_u^3 \mid \triangleright G_{da}^4 \parallel G_{lo}^5 \xrightarrow{(\text{Hold meeting},\mathsf{LO})} G_u^3 \parallel G_{da}^5 \parallel G_{lo}^6$
   Note that this event is also communicated to DA, added to the set of executed events and removed from the set of pending responses. Since there are no pending responses in any of the local graphs the finite run is in an accepting state.

5.4 Distribution of Nested DCR Graphs

In this section, we define the notion of projection of a nested DCR Graphs, restricting the graph to a subset of the events, and also we define a technique for distributing a nested DCR Graph as a set of local nested DCR Graphs obtained as projections and communicating by notifications of event executions.

5.4.1 Projections

A nested DCR Graph G is projected with respect to a *projection parameter* $\delta = (\delta_E, \delta_L)$, where $\delta_E \subseteq E$ is a subset of the events of G satisfying that $\rhd(\delta_E) \subseteq \delta_E$, i.e. the subset is closed under the super event relation, and $\delta_L \subseteq L$ is a subset of the labels. The intuition is that the graph is restricted to only those events and relations that are relevant for the execution of events in δ_E and the labeling is restricted to the set δ_L. The technical difficulty is to infer the events and relations not in δ_E, referred to as *external events* below, that should be included in the projection because they influence the execution of the workflow restricted to the events in δ_E.

The formal definition of projection for nested DCR Graphs is given in 5.4.1 below. It generalizes the definition of projection introduced in [Hildebrandt *et al.* 2011d] for DCR Graphs to support nesting and milestones.

Definition 5.4.1. *If* $G = (E, \rhd, M, \to\!\bullet, \bullet\!\to, \to\!\diamond, \to\!+, \to\!\%, L, l)$ *then* $G_{|\delta} = (E_{|\delta}, \rhd_{|\delta}, M_{|\delta}, \to\!\bullet_{|\delta}$ $, \bullet\!\to_{|\delta}, \to\!\diamond_{|\delta}, \to\!+_{|\delta}, \to\!\%_{|\delta}, \delta_L, l_{|\delta})$ *is the projection of* G *with respect to* $\delta \subseteq E$ *where:*

(i) $E_{|\delta} = \to \delta_E$, *for* $\to = \bigcup_{c \in C} c$, *and* $C = \{\mathrm{id}, \to\!\bullet^{\flat}, \bullet\!\to^{\flat}, \to\!\diamond^{\flat}, \to\!+^{\flat}, \to\!\%^{\flat}, \bullet\!\to^{\flat}\to\!\diamond^{\flat},$ $\to\!+^{\flat}\to\!\bullet^{\flat}, \to\!\%^{\flat}\to\!\bullet^{\flat}, \to\!+^{\flat}\to\!\diamond^{\flat}, \to\!\%^{\flat}\to\!\diamond^{\flat} \}$

(ii) $\rhd_{|\delta}(e) = \rhd(e)$, *if* $e \in E_{|\delta}$

(iii) $$\text{ł}_{|\delta}(e) = \begin{cases} l(e) \cap \delta_L & \text{if } e \in \delta_E \\ \emptyset & \text{if } e \in E_{|\delta} \backslash \delta_E \end{cases}$$

(iv) $M_{|\delta} = (Ex_{|\delta}, Re_{|\delta}, In_{|\delta})$ *where:*

(a) $Ex_{|\delta} = Ex \cap E_{|\delta}$

(b) $Re_{|\delta} = Re \cap (\delta_E \cup \to\!\diamond^{\flat} \delta_E)$

(c) $In_{|\delta} = In \cap (\delta_E \cup \to\!\bullet^{\flat} \delta_E \cup \to\!\diamond^{\flat} \delta_E)$

(v) $\to\!\bullet_{|\delta} = \to\!\bullet \cap ((\to\!\bullet^{\flat} \delta_E) \times \delta_E)$

(vi) $\bullet\!\to_{|\delta} = \bullet\!\to \cap ((\bullet\!\to^{\flat}\to\!\diamond^{\flat} \delta_E) \times (\to\!\diamond^{\flat} \delta_E)) \cup ((\bullet\!\to^{\flat} \delta_E) \times \delta_E))$

(vii) $\to\!\diamond_{|\delta} = \to\!\diamond \cap ((\to\!\diamond^{\flat} \delta_E) \times \delta_E)$

(viii) $\to\!+_{|\delta} = \to\!+ \ \cap \Big(\big((\to\!+^{\flat} \delta_E) \times \delta_E\big) \cup \big((\to\!+^{\flat}\to\!\bullet^{\flat} \delta_E) \times (\to\!\bullet^{\flat} \delta_E\big) \cup \big((\to\!+^{\flat}\to\!\diamond^{\flat} \delta_E) \times$ $(\to\!\diamond^{\flat} \delta_E\big)\Big)$

(ix) $\rightarrow\%_{|\delta} = \rightarrow\% \cap \Big(\big((\rightarrow\%^{\flat}\delta_{\mathrm{E}}) \times \delta_{\mathrm{E}} \big) \cup \big((\rightarrow\%^{\flat}\rightarrow\bullet^{\flat}\delta_{\mathrm{E}}) \times (\rightarrow\bullet^{\flat}\delta_{\mathrm{E}} \big) \cup \big((\rightarrow\%^{\flat}\rightarrow\diamond^{\flat}\delta_{\mathrm{E}}) \times (\rightarrow\diamond^{\flat}\delta_{\mathrm{E}} \big) \Big)$

(i) defines the set of events in the projection as all events that has a relation pointing to an event in the set δ_{E}, where the relation is either the identity relation (i.e. it is an event in δ_{E}), one of the core relations (flattened) or the relations such as $\bullet\!\rightarrow^{\flat}\rightarrow\diamond^{\flat}$ which includes all events that triggers as a response some event that is a milestone to an event in δ_{E} or the relations that include/exclude conditions and milestones to an event in the set δ_{E}.

Events in $\mathrm{E}_{|\delta} \backslash \delta_{\mathrm{E}}$ are referred to as external events and will be included in the projection without labels, as can be seen from the definition of the labeling function in *(iii)*. As we will formalize below, events without labels can not be executed by a user locally. However, when composed in a network containing other processes that can execute these events, their execution will be communicated to the process.

(iv) defines the projection of the marking: The executed set is simply restricted to the events in $\mathrm{E}_{|\delta}$. Further, the included event set is restricted to events in projection parameter (δ_{E}) plus condition and milestone events to events in projection parameter. Finally the responses are restricted to events in δ_{E} and events that have a milestone relation to an event in δ_{E} because these are the only responses that will affect the local execution of the projected graph. Note that these events will by definition be events in $\mathrm{E}_{|\delta}$ but may be external events.

Finally, *(v)* - *(ix)* state which relations should be included in the projection. For the events in δ_{E} all incoming relations should be included. Additionally response relations to events that are a milestone for an event in δ_{E} are included as well.

To define networks of communicating nested DCR Graphs and their semantics we use the following extension of a nested DCR Graph adding a new label to every event.

Definition 5.4.2. *For an DCR Graph* $\mathrm{G} = (\mathrm{E}, \rhd, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathrm{L}, l)$ *define* $\mathrm{G}^{\varepsilon} = (\mathrm{E}, \rhd, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathrm{L} \cup \{\varepsilon\}, l^{\varepsilon})$*, where* $l^{\varepsilon} = l(e) \cup \{\varepsilon\}$ *(assuming that* $\varepsilon \notin \mathrm{L}$*).*

We are now ready to state the key correspondence between global execution of events and the local execution of events in a projection.

Proposition 5.4.1. *Let* $\mathrm{G} = (\mathrm{E}, \rhd, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathrm{L}, l)$ *be a nested DCR Graph and* $\mathrm{G}_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\delta_{\mathrm{E}}, \delta_{\mathrm{L}})$*. Then and* $\mathrm{G}_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\delta_{\mathrm{E}}, \delta_{\mathrm{L}})$*. Then*

1. *for* $e \in \delta_{\mathrm{E}}$ *and* $a \in \delta_{\mathrm{L}}$ *it holds that* $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}' \wedge \mathrm{M}'_{|\delta} = \mathrm{M}''$ *if and only if* $\mathrm{M}_{|\delta} \vdash_{\mathrm{G}_{|\delta}} e \wedge \mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}''$*,*

2. *for* $e \notin \mathrm{E}_{|\delta}$ *it holds that* $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ *implies* $\mathrm{M}_{|\delta} = \mathrm{M}'_{|\delta}$*,*

3. *for* $e \in \mathrm{E}_{|\delta}$ *(and* $a \notin \delta_{\mathrm{L}}$*) it holds that* $\mathrm{M} \vdash_{\mathrm{G}} e \wedge \mathrm{M} \oplus_{\mathrm{G}} e = \mathrm{M}'$ *implies* $\mathrm{M}_{|\delta} \oplus_{\mathrm{G}_{|\delta}} e = \mathrm{M}'_{|\delta}$*.*

Proof. According to definition when an event is enabled and the result of executing an event for nested DCR Graphs 4.1.4,
an event in nested graph is enabled if it is enabled in the flattened graph. $\mathrm{M} \vdash_{\mathrm{G}} e$, if $\mathrm{M} \vdash_{\mathrm{G}^\flat} e$

Similarly, the result of executing $\mathrm{M} \oplus_{\mathrm{G}} e$ same as executing the event in flattened graph and it is defined as: $\mathrm{M} \oplus_{\mathrm{G}^\flat} e = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In}) \oplus_{\mathrm{G}^\flat} e$.

Moreover the marking M of a nested DCR Graph is same as its flattened DCR Graph.

Hence the above 3 propositions for a nested DCR Graph can be proved based on the similar lines as those propositions for a DCR Graph (propositions 5.3.1, 5.3.2 and 5.3.3).

□

5.4.2 Distributed Execution in Nested DCR Graphs

Intuitively, a vector of projection parameters is covering if every event is included in at least one projection parameter and every label that is assigned to an event occurs at least once together with that event.

Definition 5.4.3. *We call a vector* $\Delta = (\delta_1, \ldots, \delta_k)$ *of projection parameters* covering *for some DCR Graph* $\mathrm{G} = (\mathrm{E}, \rhd, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$ *if:*

1. $\bigcup\limits_{i\in[k]} \delta_{\mathrm{E}i} = \mathrm{E}$ *and*

2. $(\forall a \in \mathrm{L}.\forall e \in \mathrm{E}.a \in l(e) \Rightarrow (\exists i \in [k].e \in \delta_{\mathrm{E}i} \wedge a \in \delta_{\mathrm{L}i})$

The marking of nested DCR Graph is same as marking of its flattened DCR Graph (def 4.1.4) and furthermore the network semantics of DCR Graphs are defined based markings of networks. Therefore the network of nested DCR Graphs is same as the network of DCR Graphs, and hence we use the same definitions on network of DCR Graphs (def 5.3.6, 5.3.7, 5.3.8 and 5.3.9).

We now give the main technical theorem stating that a network of nested DCR Graphs obtained by projecting a nested DCR Graph G with respect to a covering vector of projection parameters has the same behavior as the original graph G. Thm. 5.4.1 below now states the correspondence between a nested DCR Graph and the network of nested DCR Graphs obtained from a covering projection.

Theorem 5.4.1. *For a nested DCR Graph* G *and a covering vector of projection parameters* $\Delta = (\delta_1, \ldots, \delta_n)$ *it holds that* $TS(\mathrm{G})$ *is bisimilar to* $TS(\mathrm{G}_\Delta)$*, where* $\mathrm{G}_\Delta = \Pi_{i\in[n]}\mathrm{G}|_{\delta_i}$. *Moreover, a run is accepting in* $TS(\mathrm{G})$ *if and only if the bisimilar run is accepting in* $TS(\mathrm{G}_\Delta)$.

Proof. The marking of nested DCR Graph is same as marking of its flattened DCR Graph (def 4.1.4). Furthermore the labeled transition system $TS(\mathrm{G})$ for nested DCR Graph (def 4.1.5) is defined in terms of markings, which is same for both nested DCR Graph and its flattened DCR Graph.

The labeled transition system ($TS(G_\Delta)$) for network of projected nested DCR Graphs is same as network of projected DCR Graphs.

Therefore using proposition 5.4.1 and following the theorem 5.3.1 we can easily prove that $TS(G) \sim TS(G_\Delta)$.

Similarly following the theorem 5.3.1, we can also prove that a run is accepting in $TS(G)$ if and only if the bisimilar run is accepting in $TS(G_\Delta)$ as the accepting condition for a run only depends on the markings of nested DCR Graph, which is same as its flattened DCR Graph.

□

The generality of the distribution technique given above allows for fine tuned projections where we select only a few events for a specific role and actor, but in most cases the parameter is likely to be chosen so that the projected graph shows the full responsibilities of a specific role or actor. A set of nested DCR Graphs can be maintained and executed in a distributed fashion, meaning that there is a separate implementation for every graph and that the execution of shared events is communicated between them. Through the distributed execution of projected graphs, nested DCR Graphs can be used as a (declarative) choreography model to the line of work (on typed imperative process models) in [Carbone *et al.* 2007]: The original graph can be seen as the choreography, describing how the system as a whole should function, from which we project multiple end-points for individual roles or actors that can be implemented independently.

5.4.3 Distribution of Healthcare Workflow

In Fig. 5.4 below we show the graphical representation of the nested Dynamic Condition Response Graph formalizing a variant of the oncology workflow studied in [Lyng *et al.* 2008]. In this section we informally describe the formalism and the distribution technique formalized in the previous section using the example workflow.

As explained before, the boxes denote *activities* (also referred to as events in the following sections). **Administer medicine** is a *nested* activity having sub activities **give medicine** and **trust**. **Give medicine** is an *atomic* activity, i.e. it has no sub activities. **Trust** is again a nested activity having sub activities **sign nurse 1** and **sign nurse 2**. Finaly, **medicine preparation** is a nested activity having seven sub activities dealing with the preparation of medicine. An activity may be either included or excluded, the latter are drawn as a dashed box as e.g. the **edit** and **cancel activities**.

A *run* of the workflow consists of a (possibly infinite) sequence of executions of atomic activities. (A nested activity is considered executed when all its sub activities are executed). An activity can be executed any number of times during a run, as long

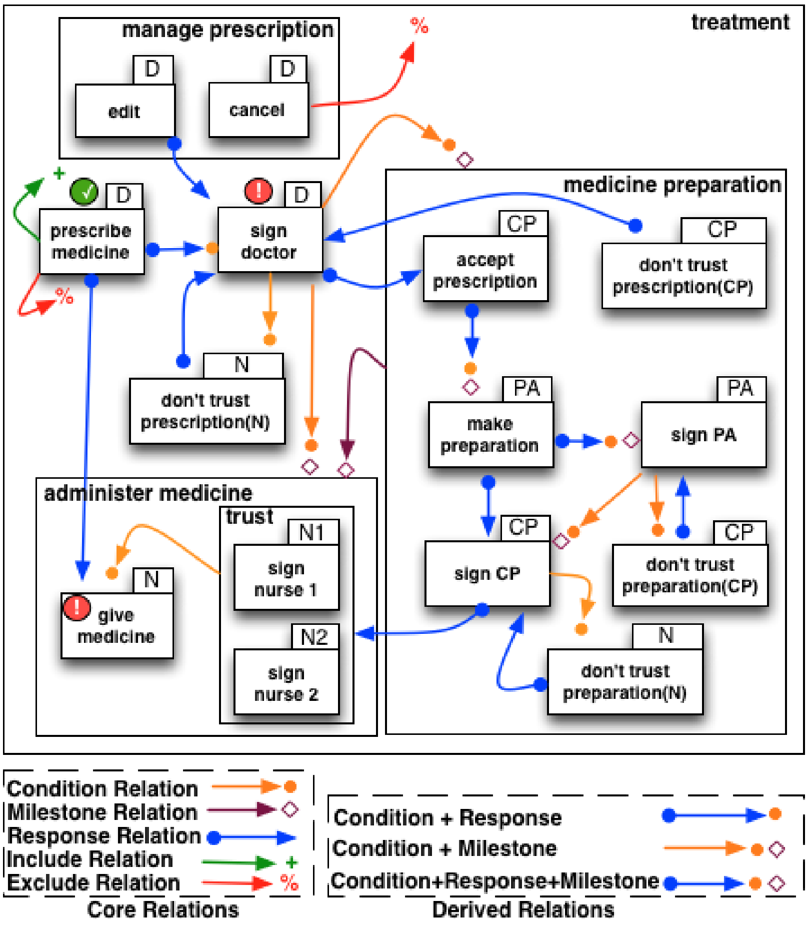


Figure 5.4: Oncology Workflow as a nested DCR Graph

as the activity is included and the constraints for executing it are satisfied, in which case we say the activity is *enabled*.

The constraints and dynamic exclusion and inclusion are expressed as five different core relations between activities represented as arrows in the figure above: The *condition relation*, the *response relation*, the *milestone relation*, the *include relation*, and the *exclude relation*. The condition relation is represented by an orange arrow with a bullet at the arrow head. E.g. the condition relation from the activity **sign doctor** to the activity **don't trust prescription(N)** means that **sign doctor** must have been executed at least once before the activity **don't trust prescription(N)** can be executed.

The response relation is represented by a blue arrow with a bullet at its source. E.g. the response relation from the activity **prescribe medicine** to the activity **give medicine** means that the latter must be executed (at some point) after (any execution of) the activity **prescribe medicine**. We say that a workflow is in a *completed* state if all such response constraints have been fulfilled (or the required response activity is excluded). However, note that a workflow may be continued from a completed state and change to a non-completed state if an activity is executed that requires another response or includes an activity which has not been executed since it was last required as a response. Also note that the response constraint may cause some infinite runs to never pass through a complete state if the executed activities keep triggering new responses.

The third core relation used in the example is the *milestone relation* represented as a dark red arrow with a diamond at the arrow head. The milestone relation was introduced in [Hildebrandt *et al.* 2011c] jointly with the ability to nest activities. A relation to and/or from a nested activity simply unfolds to relations between all sub activities. A milestone relation from a nested activity to another activity then in particular means that the entire nested activity must be in a completed state before that activity can be executed. E.g. **medicine preparation** is a milestone for the activity **administer medicine**, which means that none of the sub activities of administer medicine can be carried out if any one of the sub activities of medicine preparation is included and has not been executed since it was required as a response.

Two activities can be related by any combination of these relations. In the graphical notation we have employed some shorthands, e.g. indicating the combination of a condition and a response relation by and arrow with a bullet in both ends.

Finally, DCR Graphs allow two relations for dynamic exclusion and dynamic inclusion of activities represented as a green arrow with a plus at the arrow head and a red arrow with a minus at the arrow head respectively. The exclusion relation is used in the example between the **cancel** activity and the **treatment** activity. Since all other activities in the workflow are sub activities of the **treatment** activity this means that all activities are excluded if the cancel activity is executed. The inclusion relation is used between the **prescribe medicine** activity and the **manage prescription** activity.

The run-time state of a nested DCR Graph can be formally represented as a pair $(\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$ of sets of atomic activities (referred to as the *marking* of the graph). The set Ex is the set of atomic activities that have been executed at least once during the run. The set Re is the set of atomic activities that, if included, are required to be executed at least one more time in the future as the result of a response constraint (i.e. they are pending responses). Finally, the set In denotes the currently included activities.

The set Ex thus may be regarded as a set of completed activities, the set Re as the set of activities on the to-do list and the set In as the activities that are currently relevant for the workflow.

Note that an activity may be completed once and still be on the to-do list, which simply means that it must be executed (completed) again. This makes it very simple to model the situation where an activity needs to be (re)considered as a response to the execution of an activity. In the oncology example this is e.g. the case for the response relation between the **don't trust prescription(N)** activity (representing that a nurse reports that he doesn't trust the prescription) and the **sign doctor** activity. The effect is that the doctor is asked to reconsider her signature on the prescription. In doing that she may or may not decide to change the prescription, i.e. execute prescribe medicine again.

We indicate the marking graphically by adding a check mark to every atomic activity that has been executed (i.e. is included in the set Ex of the marking), an exclamation mark to every atomic activity which, if included, is required to be executed at least once more in the future (i.e. is included in the set Re), and making a box dashed if the activity is not included (i.e. is not included in the set In of the marking). In Fig. 5.4 we have shown an example marking where prescribe medicine has been executed. This has caused manage prescription and its sub activities edit and cancel to be included, and sign doctor and give medicine to be required as responses, i.e the two activities are included in the set Re of the marking (on the to-do list).

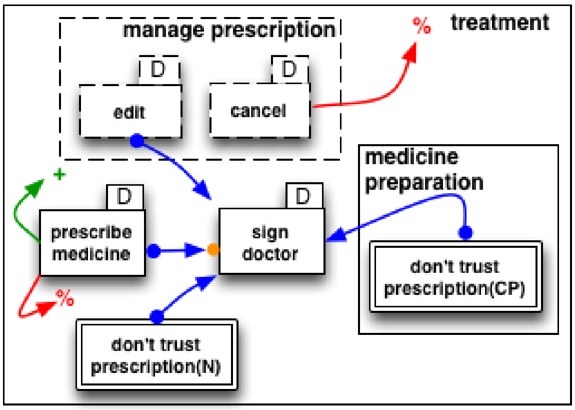


Figure 5.5: Projection over doctor's role (*D*)

As described above, an activity can be executed if it is enabled. Sign doctor is enabled for execution in the example marking, since its only condition (prescribe medicine) has been executed and it has no milestones. Give medicine on the other hand is not enabled since it has the (nested) activity trust as condition, which means that all sub activities of trust (sign nurse 1 and sign nurse 2) must be executed before give medicine is enabled. Also, both give medicine and trust are sub activities of administer medicine which further has sign doctor as condition and milestone, and medicine preparation as milestone. The condition relation from sign doctor means that the prescription must be signed before the medicine can be administered. The milestone relations means that the medicine can not be given as long as sign doctor or any of the sub activities of medicine preparation is on the to-do list (i.e. in the set Re of pending responses).

Every activity should not be available to any user of the workflow system. For this reason the commercial implementation of the workflow management system provided by Resultmaker employs a role based access control, assigning to every atomic activity a finite set of roles and assigning to every role a set of access rights controlling if the activity is invisible or visible to users fulfilling the role. If an activity is visible it is specified wether the role are allowed to execute the activity or not.

Users are either statically (e.g. by login) or dynamically assigned to roles (e.g. by email invitation).

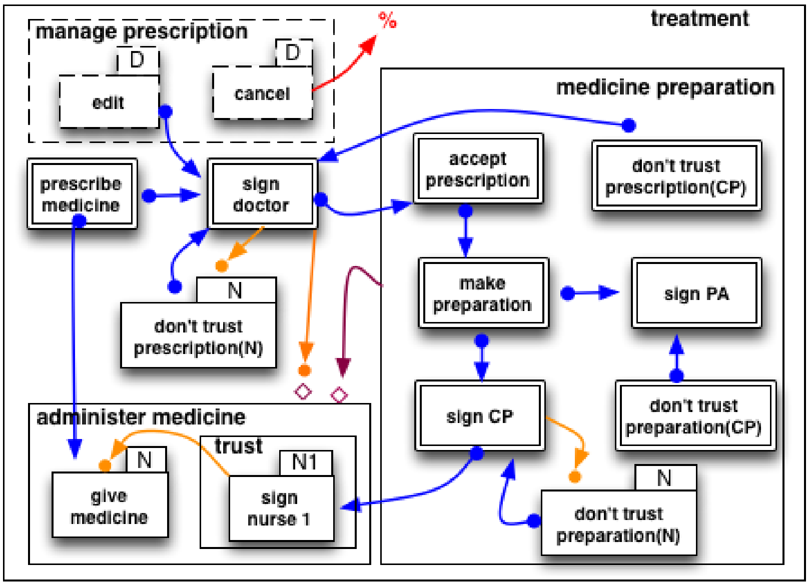


Figure 5.6: Projection over nurse role (*N* and *N1*)

In the formalization presented in previous section, the assigned roles are given as part of the name of the activity. In the graphical representation we have shown the roles within small "ears" on the boxes. In the example workflow we have the following different roles: Doctor (D), Controlling Pharmacist (CP), Pharmacist Assistant (PA) and Nurse (N). Hereto comes roles N1 and N2 which must dynamically be assigned to two different authorized persons (nurses or doctors). This is at present the only way to implement the constraint stating that two different authorized persons must sign the product prepared by the pharmacists before the medicine is administered to the patient. Future work will address less ad hoc ways to handle these kind of constraints between activities referring to the identify of users.

The technique for distributing DCR Graphs introduced in [Hildebrandt *et al.* 2011d] and extended in the present paper is a first step towards supporting this kind of splitting of workflow definitions. Given any division of activities on local units (assigning every activity to at least one unit) it describes how to derive a set of graphs, one for each unit, describing the local part of the workflow. Such a local process, referred to as a *projection* is again a DCR Graph. It includes the activities assigned to the unit but also the relevant *external* activities executed within other units for which an event must be send to the local process when they are executed. An example of a projection relative to the activities assigned the doctor role (D) is given in Fig. 5.5. The diagram shows that the projection also includes the two external activities (indicated as double line boxes) don't trust prescription (N) and don't trust prescription

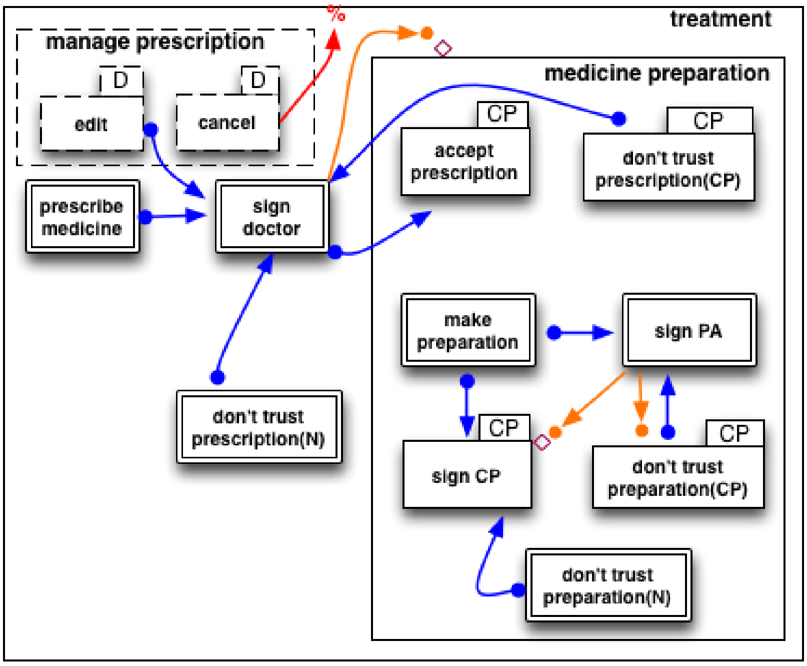


Figure 5.7: Projection over control pharmacist role (*CP*)

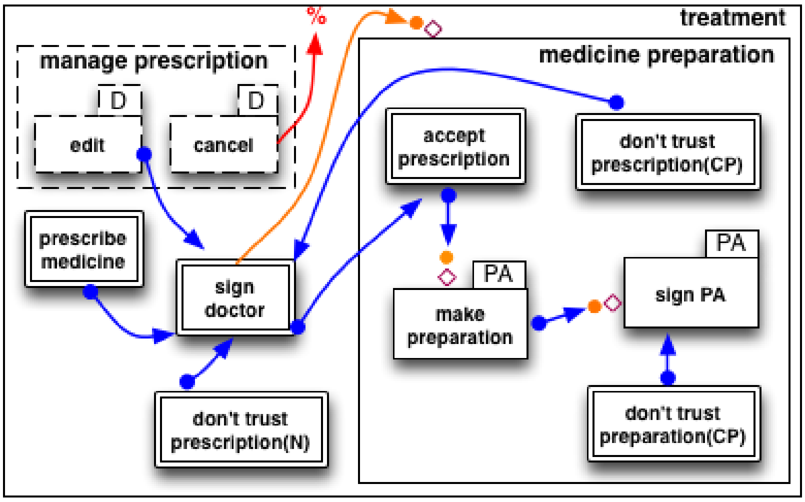


Figure 5.8: Projection over pharmacy assistant role (*PA*)

(CP). These two activities, representing respectively a nurse and a controlling pharmacist reporting that the prescription is not trusted, are the only external activities that may influence the workflow of the doctor by requiring sign doctor as a response.

Similarly, Fig. 5.6,5.7, and 5.8 shows projections corresponding to the nurse, controlling pharmacist, and pharmacist assistant roles. However, if for instance the roles of the controlling pharmacist and the pharmacist assistant are always assigned to the same persons one may instead choose to keep all these activities together in a unit. This can be obtained by simply projecting on all activities assigned either the CP or the PA role.

For instance, Fig. 5.5 shows the projection with respect to the projection parameter (δ_E, δ_L) where δ_E={manage prescription, edit, cancel, prescribe medicine, sign doctor} and δ_L={(edit, D), (cancel, D), (prescribe medicine, D), (sign doctor, D)}. The two events don't trust prescription (N) and don't trust prescription (CP) shown with double line borders are external events included in the projected graph even though they don't appear in the projection parameter. It is interesting to note that the doctor only needs to be aware of these two activities carried out by other participants. In comparison, the projection over the roles for nurses (*N* and *N1*) contains all the events since they may influence (because of the milestone relations) the execution of the events with roles *N* and *N1*. In other words, the doctors can carry out workflows highly independent of the other activities while the nurses are dependent on any event carried out by the other roles.

5.5 Summary

In this chapter, we have given a general technique for distributing a declarative (global) process as a network of synchronously communicating (local) declarative processes and proven the global and distributed execution to be equivalent using the DCR Graphs. Our method is based on *top-down* model-driven approach and addressed the challenging distributed synthesis problem: Given a global model and some formal description of how the model should be distributed, can we synthesize a set of local processes with respect to this distribution which are consistent to the the global model?

In order to safely distribute a DCR Graph, we have defined a general notion of *projection* on the DCR Graphs relative to a subset of labels and events in the section 5.3. Here is the key challange is to identify the set of events that must be communicated from other processes in the network in order for the state of the local process to stay consistent with the global specification. Further, in order to enable the reverse operation, building global graphs from local graphs, we have defined the composition of two DCR Graphs, essentially by gluing joint events. As a sanity check, we then proved that if we have a collection of projections of a DCR Graph that is covering the original graph, then the composition yields back the same graph in Sec. 5.3.2. We then finally proved to the main technical result, defining networks of synchronously communicating DCR Graphs and stating (in Thm. 5.3.1, Sec. 5.3.3) the correspondence between a global process and a network of communicating DCR Graphs obtained from a covering projection. Further, we have exemplified the distribution technique on a simple cross-organizational process identified within a case study (in Sec. 5.3.4) carried out jointly with our industrial partner Exformatics A/S using DCR Graphs for model-driven design and engineering of an inter-organizational case management system.

Moreover, we have extended the safe distribution technique to *Nested* DCR Graphs in Sec. 5.4 by distributing a nested DCR Graph as a set of local nested DCR Graphs obtained as projections and communicating by notifications of event executions. Further, we have also exemplified the distribution technique of *Nested* DCR Graphs on a healthcare workflow identified during a previous field study at Danish hospitals [Lyng *et al.* 2008], which was introduced in Sec. 2.1.3.

Finally, the generality of the distribution technique given in this chapter allows for fine tuned projections where we select only a few events for a specific role and actor, but in most cases the parameter is likely to be chosen so that the projected graph shows the full responsibilities of a specific role or actor. Our distribution technique is quite generic and the strength of distribution lies in the fact that the resulting local components are also DCR Graphs, which keep their declarative nature, therefore they can be further distributed.

Chapter 6

Formal Verification, Tools and Implementation

In this chapter, we will describe the prototype implementation and tools built around the theory of DCR Graphs to demonstrate the usage of our formal model in modeling the business processes and workflows. In addition to this, we will also define safety and liveness properties on the DCR Graphs formally, using the notion of runs and accepting runs defined on markings of the graph. Further, we will also describe a method to encode DCR Graphs and use formal verification methods to verify these properties using SPIN [Holzmann 1997, Holzmann 2004] model checking tool.

First, we will introduce the notion of safety and liveness on DCR Graphs and define corresponding properties in terms markings of a DCR Graph in Sec. 6.2. In Sec. 6.3, first we will give a brief introduction to SPIN tool and its modeling language PROMELA [Spin 2007], then we will describe a method to encode DCR Graphs into PROMELA, to verify safety and liveness properties in SPIN. Further, we will briefly mention our experience with verification of safety properties on DCR Graphs using Zing [Andrews *et al.* 2004, Fournet *et al.* 2004] model checker developed by Microsoft Research. Finally, we will give a overview about tools and implementation that were built around our formal model in the Sec 6.5.

6.1 Related Work

Verification of business processes based on a wide range formal specification models, has been studied in the last couple of decades. First of all, many researchers have studied the problem of formalization and verification of business processes modeled using UML activity diagrams. The authors in [Eshuis 2002, Eshuis & Wieringa 2004] have studied semantics of UML activity diagrams by mapping them to clocked transition system (CTS) [Kesten *et al.* 1996] and explored formal verification of UML diagrams based on their implementation of model checker and also using NuSMV [Cimatti *et al.* 2000] symbolic model checker. Further the authors in [Guelfi *et al.* 2004, Guelfi & Mammar 2005] have given formal semantics for UML timed activity diagrams and translated them to PROMELA [Spin 2007] language to do the formal verification with the help of SPIN [Holzmann 1997, Holzmann 2004, Spin 2008, Ben-Ari 2008] model checker. The work on UML Statechart Diagrams [Latella & Massink 2001, Latella *et al.* 1999] studied the formal verification on behavioral subset of UML state charts using SPIN model checker, where as the PhD thesis [Porres 2001] on Modeling and Analyzing Software Behavior in UML, gave formal semantics to UML statecharts and provided a method to verify them using vUML [Lilius & Paltor 1999] tool. Finally the authors in [Knapp *et al.* 2002] used a different approach and provided verification support for the timed state machines of a UML model, by compiling them into timed automata to verify models using UPPAAL [Larsen *et al.* 1997] model checker.

Petri nets [Reisig 1991, Brauer *et al.* 1987] is one of most widely used formalism for modeling business processes and also there exists a good number of tools for static and reachability analysis on Petri nets. Many researchers have formalized current workflow/business process specification standards such as BPMN [Object Management Group BPMN Technical Committee 2011], BPEL [OASIS WSBPEL Technical Committee 2007] into Petri nets to do formal verification on the business processes, as there exists a good number of tools for static and reachability analysis on Petri nets. The authors in [Dijkman *et al.* 2008] have provided formal semantics for BPMN in terms of Petri nets to do static analysis on them, whereas authors in [Dun *et al.* 2008, Hinz *et al.* 2005] provided an approach to model and verify business processes specified in BPEL by transforming them into ServiceNet, which is a special class of Petri nets. Further, authors in [Narayanan & McIlraith 2002] provided semantics for web service composition in terms of first order logic, which are further encoded into Petri nets to do an automatic verification, on the other hand authors in [Yi & Kochut 2004b] developed a design and verification framework for web services composition based on colored Petri nets. Further, Woflan [van der Aalst 1999b, Verbeek & van der Aalst 2000] is a Petri-net-based tool to analyze the correctness of workflows and business processes specified using Petri-net based formalisms.

Further, automata and process algebras based formalisms have also been used to model business processes. In [Fu *et al.* 2004a], author have explored analysis of interacting BPEL web services by transforming them into a guarded automata with unbounded queues and further converted them into PROMELA code to do verification in SPIN model checker. In an another approach [Diaz *et al.* 2005, Dong *et al.* 2006], web service choreographies and orchestrations have been verified by converting them into timed automata and using UPPAAL [Larsen *et al.* 1997] as the model checker. Many researchers [Karamanolis *et al.* 2000, Salaun *et al.* 2004, Ferrara 2004] used process algebras as a formalism to model web services and business processes and verify them using various model checkers. In [Ferrara 2004], authors have presented a framework based on process algebras for design and verification of services two-way mapping between abstract specifications written using process algebra and web services written in BPEL4WS. Model checking of workflow schemas have been explored in [Karamanolis *et al.* 2000], where the authors used Labelled transition systems for modeling business processes. In [Morimoto 2008], Shoichi Morimoto provided a very good overview and survey of existing approaches for formal verification techniques on business processes.

All the above mentioned approaches have explored formal verification using imperatives models and modeling languages, where as our work focuses on formal verification of business processes modeled using declarative modeling primitives.

SPIN [Holzmann 1997, Holzmann 2004, Vardi & Wolper 1986, Spin 2008, Ben-Ari 2008] is a model checking and verification system that supports verification of properties against asynchronous process models and distributed systems. Many researchers [Havelund *et al.* 1998, Augusto *et al.* 2003, Guelfi *et al.* 2004, Guelfi & Mammar 2005, Janssen *et al.* 1998] have used SPIN tool for formal verification of business processes and services. Authors in [Havelund *et al.* 1998] have used SPIN to formally verify a multi-threaded plan execution programming language for NASA's artificial intelligence-based spacecraft control system that was part of the DEEP SPACE 1 mission to Mars. Further authors in [Janssen *et al.* 1998] have used SPIN to verify business processes modeled in AMBER language as part of TestBed project for business process reengineering where as authors in [Augusto *et al.* 2003] have used both SPIN and STep [Bjørner *et al.* 2000] tools to verify business processes.

Another major paradigm in business process modeling is the artifact-centric approach, which strongly argues that data design should be elevated to the same level as control flows for data rich workflows and business processes. In this area, several researchers [Nigam & Caswell 2003, Bhattacharya *et al.* 2007a, Liu *et al.* 2007] have been working with artifact-centric or data-centric workflows and also efforts has been made to do formal analysis of artifact-centric process models [Gerede *et al.* 2007, Gerede & Su 2007, Bhattacharya *et al.* 2007b, Deutsch *et al.* 2009]. The static analysis and verification work in [Gerede *et al.* 2007, Gerede & Su 2007] focussed on the procedural version of of the artifact-centric workflows, where as the later work [Bhattacharya *et al.* 2007b, Deutsch *et al.* 2009] studied verification on declarative version of artifact-centric models. In comparison, the main focus of verification work on business artifacts is on data-centric view of processes, where as our approach is on verifying the declarative business processes where the control flow is more explicit than data-centric processes.

6.2 Safety and Liveness for DCR Graphs

In this section, we will initiate a study of reasoning about deadlock and liveness in DCR Graphs and formally define properties for them in terms of markings of a DCR Graph. The basic motivation behind defining safety and liveness properties is to use them in the formal verification on DCR Graphs to guarantee the deadlock and livelock freeness as explained further in the section 6.3 and section 6.4.

6.2.1 Executions and Must Executions

First of all, let is recall the definitions of when an event is enabled (def 3.3.7), the result of executing an event (def 3.3.8) and an execution (def 3.3.9) from the chapter 3 for easy readbility, then we will extend these definitions to further define *must* executions.

Below we formalize in definition. 6.2.1 that an event e of a DCR Graph is enabled when it is included in the current marking, all the included events that are conditions for it are in the set of executed event and all the included events that are milestone events for e are not in the set of responses.

Definition 6.2.1. *For a Dynamic Condition Response Graph* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$*, and* $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$ *we define that an event* $e \in \mathrm{E}$ *is* enabled*, written* $\mathrm{M} \vdash_{\mathrm{G}} e$*, if* $e \in \mathrm{In} \wedge (\mathrm{In} \cap \rightarrow\!\bullet\, e) \subseteq \mathrm{Ex}$ *and* $(\mathrm{In} \cap \rightarrow\!\diamond\, e) \subseteq \mathrm{E} \backslash \mathrm{Re}$.

The definition 6.2.2 below then defines the change of the marking when an event e is executed: Firstly, the event e is added to the set of executed events and removed from the set of pending responses. Secondly, all events that are a response to the event e are added to the set of pending responses. Note that if an event is a response to itself, it will remain in the set of pending responses after its execution. Finally, the included events set will be updated by adding/removing all the events that are included/excluded by e.

Definition 6.2.2. *For a Dynamic Condition Response Graph* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$*, where* $\mathrm{M} = (\mathrm{Ex}, \mathrm{Re}, \mathrm{In})$*, event* $\mathrm{M} \vdash_{\mathrm{G}} e$*, we define the result of executing an event* e *as* $(\mathrm{Ex}, \mathrm{Re}, \mathrm{In}) \oplus_{\mathrm{G}} e =_{def} \big(\mathrm{Ex} \cup \{e\}, (\mathrm{Re} \setminus \{e\}) \cup e\!\bullet\!\rightarrow, (\mathrm{In} \cup e\!\rightarrow\!+) \setminus e\!\rightarrow\!\%\big)$.

Having defined when events are enabled for execution and the effect of executing an event we can define finite and infinite executions and when they are accepting. In the definition 6.2.3, we define that an execution in DCR Graphs is a (finite or infinite) sequence of markings and an execution is accepting if and only if, any required, included response in any intermediate marking is eventually executed or excluded.

Definition 6.2.3. *For a Dynamic Condition Response Graph* $\mathrm{G} = (\mathrm{E}, \mathrm{M}, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, \mathrm{L}, l)$ *we define an* **execution** *of* G *to be a (finite or infinite) sequence of tuples* $\{(\mathrm{M}_i, e_i, a_i, \mathrm{M}'_i)\}_{i \in [k]}$ *each consisting of a marking, an event, a label and another marking (the result of executing the event) such that*

i) $\mathrm{M} = \mathrm{M}_0$

ii) $\forall i \in [k].a_i \in l(e_i)$

iii) $\forall i \in [k].\mathrm{M}_i \vdash_{\mathrm{G}} e_i$

iv) $\forall i \in [k].\mathrm{M}'_i = \mathrm{M}_i \oplus_{\mathrm{G}} e_i$

v) $\forall i \in [k-1].\mathrm{M}'_i = \mathrm{M}_{i+1}$.

Further, we say the execution is accepting *if* $\forall i \in [k].\big(\forall e \in \mathrm{In}_i \cap \mathrm{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathrm{In}'_j)\big)$*, where* $\mathrm{M}_i = (\mathrm{Ex}_i, \mathrm{In}_i, \mathrm{Re}_i)$ *and* $\mathrm{M}'_j = (\mathrm{Ex}'_j, \mathrm{In}'_j, \mathrm{Re}'_j)$*. Further we denote the set of all executions and set of all accepting executions by* $\mathrm{exe}_{\mathrm{M}}(\mathrm{G})$ *and* $\mathrm{acc}_{\mathrm{M}}(\mathrm{G})$ *respectively.*

Similarly, we define that a *must* execution is a (finite or infinite) sequence of markings, where only the events that are required as responses are executed at each marking and we further say that a *must* execution is accepting if the included pending responses in any intermediate marking are eventually executed or excluded.

Definition 6.2.4. *For a Dynamic Condition Response Graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define a* **must execution** *of* G *to be a (finite or infinite) sequence of tuples* $\{(M_i, e_i, a_i, M_i')\}_{i\in[k]}$ *each consisting of a marking, an event, a label and another marking such that*

i) $M = M_0$

ii) $\forall i \in [k].a_i \in l(e_i)$

iii) $\forall i \in [k].M_i \vdash_G e_i \wedge e_i \in Re_i$

iv) $\forall i \in [k].M_i' = M_i \oplus_G e_i$

v) $\forall i \in [k-1].M_i' = M_{i+1}$.

Further, we say the **must execution** *is* accepting *if* $\forall i \in [k].\big(\forall e \in In_i \cap Re_i.\exists j \geq i.e_j = e \vee e \notin In_j'\big)\big)$, *where* $M_i = (Ex_i, In_i, Re_i)$ *and* $M_j' = (Ex_j', In_j', Re_j')$. *Further we denote the set of all must executions and set of all accepting must executions by* $\text{mexe}_M(G)$ *and* $\text{macc}_M(G)$ *respectively.*

Finally before defining properties on DCR Graphs, we define that a marking (M′) is reachable from another marking (M), if there exists an finite execution from M to M′, as follows.

Definition 6.2.5. *For a Dynamic Condition Response Graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define that a marking* M' *is reachable in* G *(from the marking* M*) if there exists a finite execution ending in* M' *and let* $\mathcal{M}_{M\rightarrow^*}(G)$ *denote the set of all reachable markings from* M.

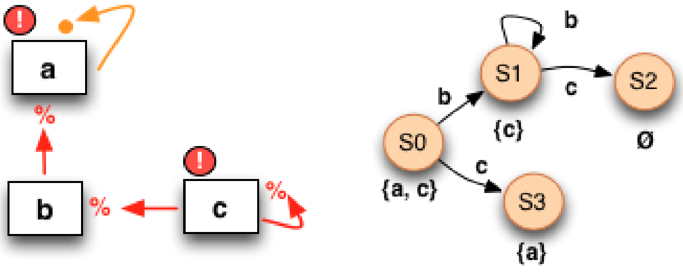
6.2.2 Safety Properties

In this section we introduce and exemplify variations of deadlock freedom as formal safety properties for DCR Graphs. A DCR Graph is said to be *deadlock free* if and only if for any reachable marking, there is either an enabled event or no included required responses.

Definition 6.2.6. *For a dynamic condition response graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define that* G *is* **deadlock free**, *if* $\forall M' = (Ex', In', Re') \in \mathcal{M}_{M\rightarrow^*}.(\exists e \in E.M' \vdash_G e \vee (In' \cap Re' = \emptyset))$.

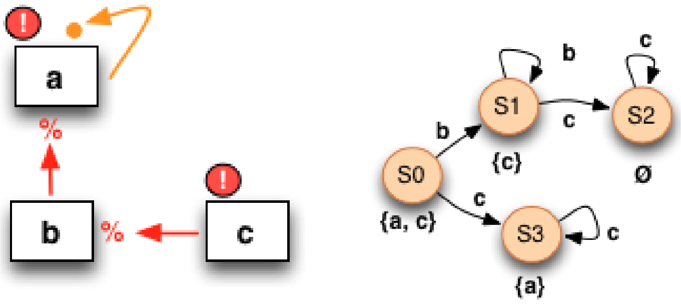
The figure 6.1 shows a DCR Graph and its transitions from different markings, with sets of included pending responses marked under nodes. The graph shown in figure 6.1 is not deadlock free, as we can see at the state $S3$ there is no transition, which indicates that the marking at $S3$ does not have any enabled event, but at the same time it has event a in the included pending responses set, which indicates a deadlock according to the definition 6.2.6.

On the other hand, the DCR Graph shown in figure 6.2 is deadlock free, even though the state $s3$ is non accepting (due to the pending response a), but the marking at $s3$ always has an enabled event c.



(a) non deadlock-free graph (b) state space

Figure 6.1: A non-deadlock free DCR Graph



(a) deadlock-free (b) state space

Figure 6.2: Deadlock free DCR Graph

In definition 6.2.7, we will define that a DCR Graph is *strongly deadlock free* if and only if for any reachable marking there is either an enabled event which is also a required response or no included required responses.

Definition 6.2.7. *For a dynamic condition response graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define that* G *is* **strongly deadlock free**, *if* $\forall M' = (Ex', In', Re') \in \mathcal{M}_{M\rightarrow^*}.(\exists e \in Re'.M' \vdash_G e \vee (In' \cap Re' = \emptyset))$.

One could wonder about why we have defined a stronger notion of deadlock freedom, as it clearly puts more stronger constraint on execution traces. In DCR Graphs, even though an event is enabled at a particular marking, there is no guarantee that the event will be executed as the events are executed by the actors at their own discretion. The only way to specify that an event *must* be executed is by specifying the event as required response. For example, the graph in figure 6.2 is deadlock free and at the state $s3$ the marking contains only one enabled event c, therefore if the user chooses not to execute the event c (of course he is allowed to do that perfectly as the event c is not required as response), then it will lead to deadlock. Hence a *deadlock free* property in a DCR Graph only guarantees that the graph is structurally deadlock free by specification, but it does not guarantee about the situations where an user can create a deadlock by choosing not to execute an enabled event.

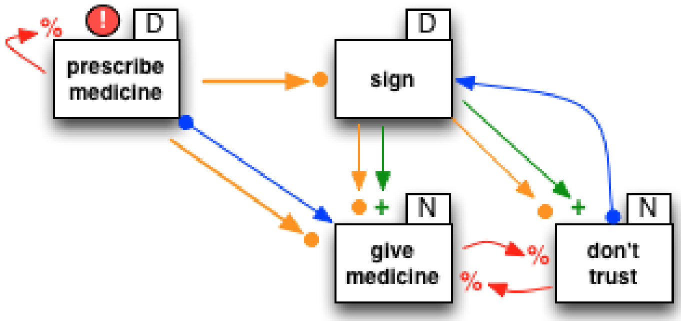


Figure 6.3: Give Medicine example (deadlock free, live, but not strongly deadlock free)

To understand the motivation behind the strongly deadlock free property more clearly, let us consider give medicine (prescribe medicine healthcare workflow) example as shown in the figure 6.3. The example is a slightly modified version of the original version (figure 3.8) and here we have added an self exclude relation on *prescribe medicine* event and removed the response relation between *prescribe medicine* and *sign*, to illustrate the difference between deadlock free and strongly deadlock free properties.

The state space for the give medicine example, generated by one of our prototype tools (described in section 6.5) is shown in the figure 6.4. From the state space for give medicine, one can see that the DCR Graph for give medicine example is deadlock free, as we have enabled transitions at every state (alternatively we have enabled events at every reachable marking), but one can observe that the example can easily end up into deadlock, if the user chooses not to execute an enabled event. More specifically, at the state $S1$ the doctor is not compelled to sign the prescription as *sign* is not a required response at $s1$, hence if the doctor chooses not to sign the prescription then the process will end up in deadlock. Therefore the give medicine example shown in the figure 6.3 is not strongly deadlock free, but the strongly live version of give medicine example (which will be introduced in next section) shown in figure 6.5 is strongly deadlock free.

6.2.3 Liveness Properties

A DCR Graph is said to be *live* if and only if, in every reachable marking, it is always possible to eventually execute or exclude any of the pending responses and thereby continue along an accepting run.

Definition 6.2.8. *For a dynamic condition response graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define that the DCR Graph is* live, *if* $\forall M' \in \mathcal{M}_{M\rightarrow^*}.\mathsf{acc}_{M'}(G) \neq \emptyset$.

The give medicine example shown in the figure 6.3 is live, as on can observe

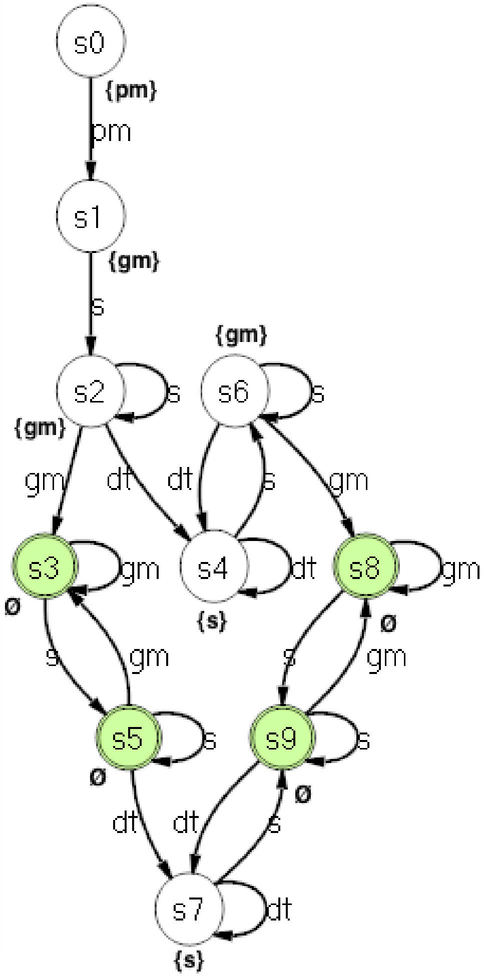


Figure 6.4: State space for Give Medicine example (deadlock free, live, but not strongly deadlock free)

in state space shown in the figure 6.4 that from every reachable marking (or state), there exists a finite execution ending with a marking where there are no included pending responses, there by making the graph as live according to the definition. In other words liveness property on DCR Graphs guarantees that it is possible to reach an accepting state from all reachable markings.

Finally, we say that a DCR Graph is *strongly live* if and only if, from any reachable marking there exists an *accepting must execution* and we define it formally as

Definition 6.2.9. *For a dynamic condition response graph* $G = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ *we define that the DCR Graph is* strongly live, *if* $\forall M' \in \mathcal{M}_{M\rightarrow^*}.\mathrm{macc}_{M'}(G) \neq \emptyset$.

Again, to explain the motivation behind the *strongly live* property, let us refer

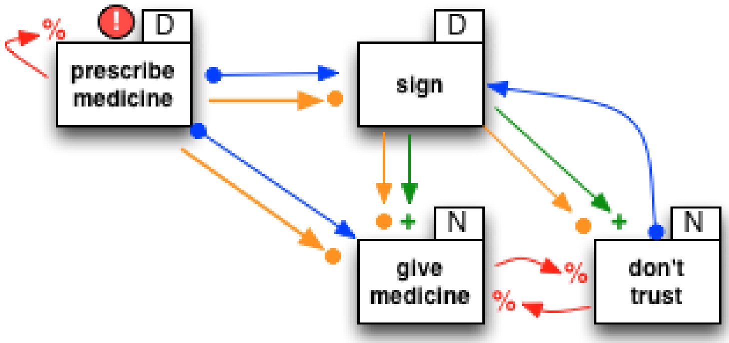


Figure 6.5: Give Medicine example (strongly live)

to the figure 6.3 (and 6.4), where the DCR Graph for give medicine example is live. Even though it is possible to proceed along the accepting run from every reachable marking, but it is not guaranteed if the users choose to execute only the events that are required as responses. For example at the state $s1$ in the figure 6.4, if the doctor does not sign the prescription (as it is not required as response), then it is not possible to proceed along accepting run.

The figure 6.5 shows a strongly live version of give medicine example, where we have added a response relation between *prescribe medicine* and *sign* and the corresponding state space is shown in the figure 6.6. We can observe that, in the revised example, from every reachable marking, there exits a must execution leading to an accepting run just by executing the required as response events at every marking.

In the next section, we will describe a method to verify these safety and liveness properties on DCR Graphs with the help of model checking tools.

6.3 Formal Verification using SPIN

In this section, we will describe about verification of properties on DCR Graphs that were introduced in the previous section. In order to verify safety and Liveness properties on DCR Graphs, we will use SPIN [Holzmann 1997, Holzmann 2004, Vardi & Wolper 1986, Spin 2008, Ben-Ari 2008] model checker which is a well known system for verification of asynchronous process models and distributed systems.

First we will give a very concise introduction to SPIN and its modeling language PROMELA in the next section, then we will describe how to encode DCR Graphs into PROMELA in section 6.3.2 and finally we will show how to verify safety and liveness properties on DCR Graphs in section 6.3.3 and 6.3.4.

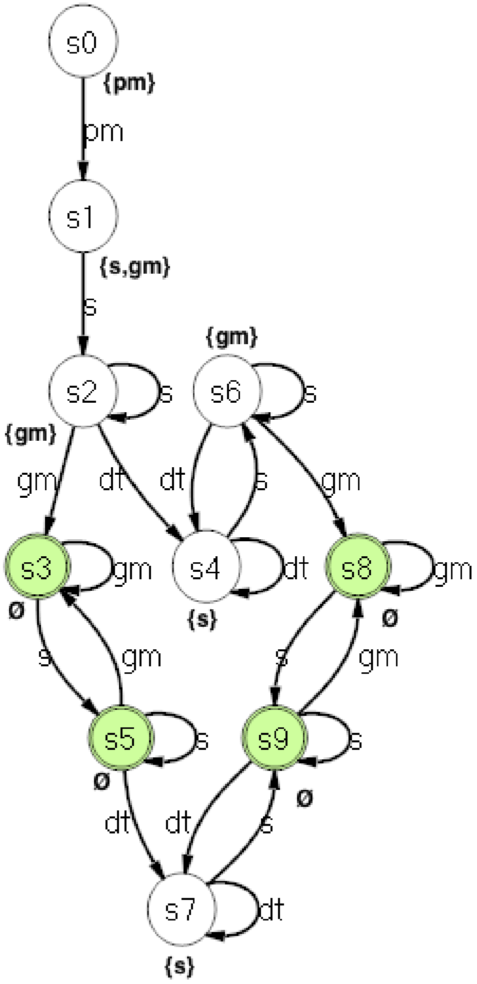


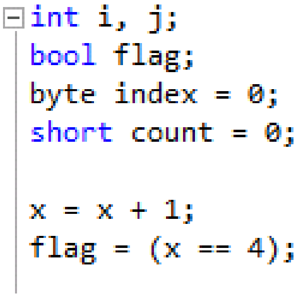
Figure 6.6: State space for Give Medicine example (strongly live)

6.3.1 Brief overview of SPIN and PROMELA lanaguage

In this section, we will give a short description of SPIN [Holzmann 1997, Holzmann 2004, Vardi & Wolper 1986, Spin 2008, Ben-Ari 2008] and its modeling language PROMELA [Spin 2007] and for more details about SPIN tool and PROMELA language reference, we encourage readers to refer to SPIN's homepage [Spin 2008]. SPIN is a model checking and verification system that supports verification of properties against concurrent and distributed processes. The process models or the systems can be encoded using a modeling meta language called PROMELA [Spin 2007], which allows for dynamic creation of concurrent processes and communication between the concurrent processes is handled by either using shared variables or message passing through buffered channels.

In addition to, some of the language constructs available in PROMELA for specification of correctness of the properties, the properties to be verified against the model can also be specified using Linear Temporal Logic (LTL) [Pnueli 1977]. The model and the property can be supplied to SPIN model checker and then it determines whether the given model satisfies the property or not, by performing verification on the state space of the model. In case if the model does not satisfy the property, then SPIN generates an error trace by giving a counter example where the model fails to satisfy the property, which can be used to further debug the model. SPIN can be invoked in different modes, for example, given a model specified in PROMELA, SPIN can either perform random simulations of the execution of the model or it can generate a C program that performs a an exhaustive verification of the state space for the given model.

PROMELA is a meta programming language (with syntax little bit similar to C) containing language constructs for specification of models. In addition to that, it also has certain constructs for specification of non-deterministic behavior and communication via shared variables and buffered channels for modeling distributed and concurrent processes. In this section, we will only briefly describe the constructs that are used in encoding of DCR Graphs into PROMELA.



```
int i, j;
bool flag;
byte index = 0;
short count = 0;

x = x + 1;
flag = (x == 4);
```

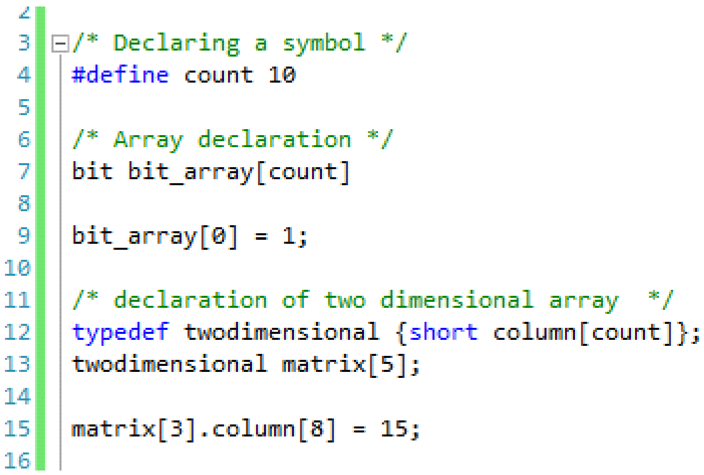
Figure 6.7: Data types and variables in PROMELA

6.3.1.1 Data Types and Variables

The basic data types supported in PROMELA are bit or bool, byte, short, int. All the variables are initialized to 0 and variable assignment is done by using = sign and equality of variables is done by == as, shown in the figure 6.7.

6.3.1.2 Arrays, Type definitions and Macros

The PROMELA language supports macros definition similar to C language and macros can be used to define symbols for the program. For example, in the figure 6.8, at line number 4, we have defined a symbol *count*, whose value is 10. Declaring symbols does not use memory as the preprocessor will replace them with actual values before generating the code for verifier, but they enhance the code readability.



```
2
3  /* Declaring a symbol */
4  #define count 10
5
6  /* Array declaration */
7  bit bit_array[count]
8
9  bit_array[0] = 1;
10
11 /* declaration of two dimensional array  */
12 typedef twodimensional {short column[count]};
13 twodimensional matrix[5];
14
15 matrix[3].column[8] = 15;
16
```

Figure 6.8: Arrays and Type definitions in PROMELA

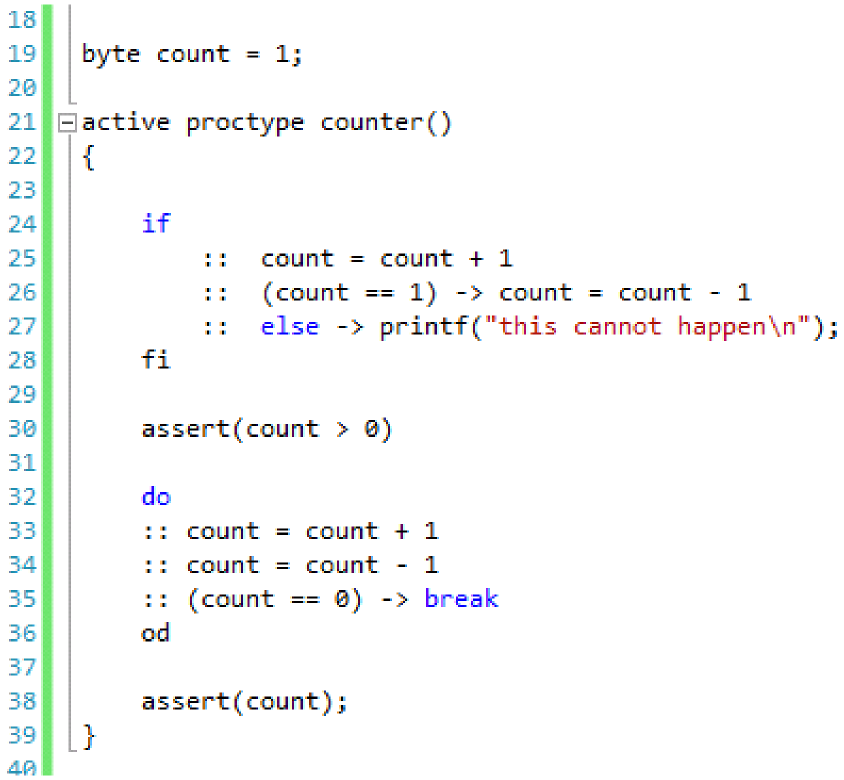
Regarding data structures, PROMELA supports only arrays and *typedef* constructs. Arrays in PROMELA are built-in data structures and they are supported as a sequence of data values of same type, which can be accessed by providing the index indicating the position of the element and array index starts at 0. One such array declaration for bit (or bool) data type can be seen at line number 7 in the figure 6.8.

Further, support for arrays in PROMELA language is limited to one directional arrays and this a big limitation in order to model processes, but on the other hand PROMELA supports *typedef* to construct compound types. We can declare multi dimensional arrays using *typedef* as shown in the code of the figure 6.8, where declaration of a multi dimensional array and assignment of values to its elements is given.

6.3.1.3 Control flow and other constructs

PROMELA supports three different flow constructs *if*, *do* and *goto*, whose semantics are little bit similar to the corresponding constructs in other programming languages, but the flow constructs in PROMELA offer non-deterministic choices in executing one of their flow branches. The sample syntax of *if* statement is shown in the figure 6.9. The alternatives of an if statement starts with double colon (::) and an optional Boolean statement which acts as a guard and then followed by sequence of statements, which will be executed if that alternative is chosen. PROMELA first evaluates all guards for the alternatives and if more than one guard is true, then it chooses an alternative non-deterministically.

For example in the figure 6.9, we can observe that both first alternative (no guard, so true) and second alternative (guard is true) are validated to true and PROMELA will choose of the alternatives non-deterministically. If none of the alternatives are validated to true in a *if* block, then the *else* block will be executed. One may wonder what happens if we don't specify an *else* block and the guards for none of



```

byte count = 1;

active proctype counter()
{

    if
        ::  count = count + 1
        ::  (count == 1) -> count = count - 1
        ::  else -> printf("this cannot happen\n");
    fi

    assert(count > 0)

    do
    :: count = count + 1
    :: count = count - 1
    :: (count == 0) -> break
    od

    assert(count);
}

```

Figure 6.9: Control flow and proctype in PROMELA

the alternatives are validated to true, in such a case process will be get to halt until one of the alternatives for *if* blocks is validated to true.

PROMELA has a *do* construct for repetition, whose syntax is same as *if* construct but with the keyword *do*, as shown in the figure 6.9. The semantics of *do* is similar to *if*, in respect of evaluating guards and executing one of the alternatives non-deterministically. After executing one of the alternatives, the control returns to the starting of the *do* statement and the only way to exit a loop is by using a *break* statement.

The state of a variable can only be modified or inspected by processes. The behavior of a process in PROMELA is defined in a *proctype* declaration as shown in the figure 6.9. The *run* operator can be used to create a new instance of process, where as *active* keyword can be used to instantiate an initial set of processes. Further PROMELA has *assert* statement to specify simple safety properties. An *assert* statement is followed by an Boolean expression and assertion violation will be reported if expression is evaluated to false (or 0). In addition to *assert*, PROMELA also has *progress* and *accept* labels prefixes to specify liveness properties.

Even though PROMELA has other constructs for message passing and sharing values among concurrent processes, we will not describe them here and conclude our discussion with the description of the above constructs, and refer our reader to the

PROMELA language reference [Spin 2007] for further details. As we have covered most important constructs that will be used in the verification of properties on the DCR Graphs, now we proceed to the next section where we will explain encoding of the DCR Graphs into PROMELA language.

6.3.2 Encoding DCR Graphs into PROMELA

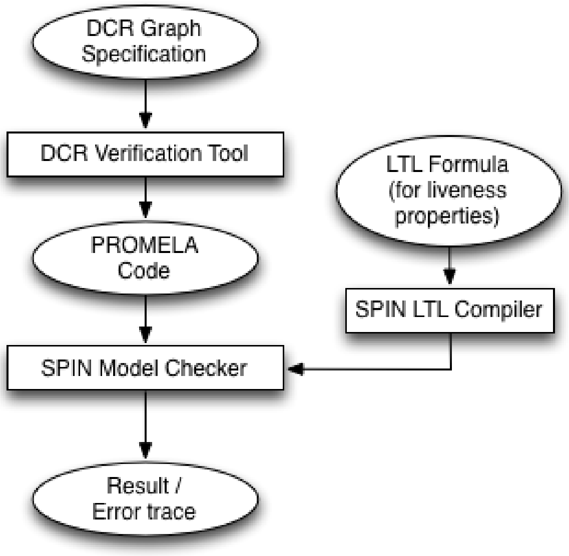


Figure 6.10: Verification of DCR Graphs with SPIN - Overview

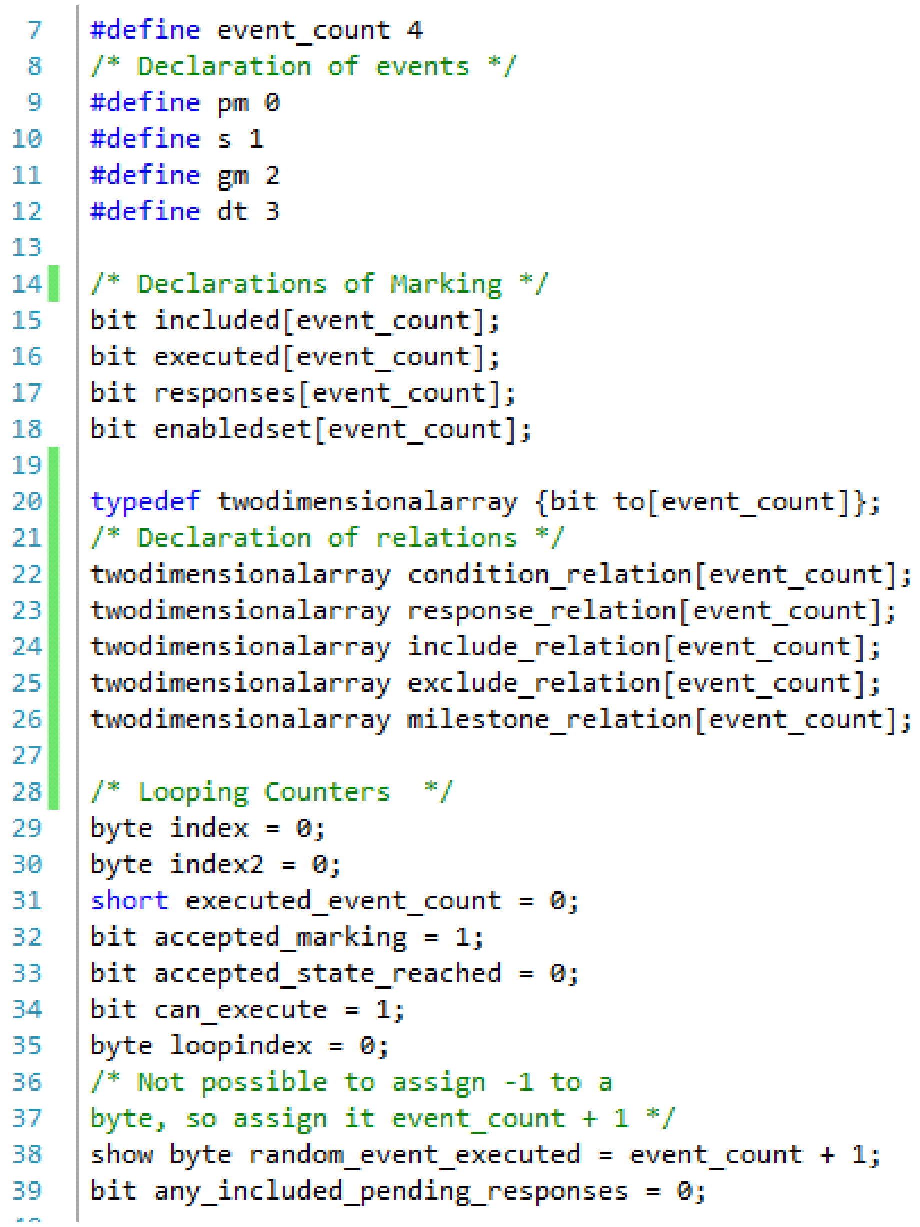
In this section we describe encoding of the DCR Graphs into PROMELA language and the challenges involved in the encoding. First, we will discuss the overall architecture of verification of properties on the DCR Graphs using SPIN as shown in the figure 6.10. We have developed a DCR verification tool which takes a DCR Graph as input in the textual representation using a simple user interface, and the tool will automatically generate the necessary PROMELA code required for verification of safely and liveness properties on DCR Graphs. The SPIN model checker will generate a finite automaton from the PROMELA specification, which will be executed by the verifier for verification of correctness of the properties specified.

In case of verification of liveness properties, we can optionally specify interesting properties expressed in Linear temporal logic and the SPIN LTL compiler will generate a finite automaton call *never claim*, which will be executed together with the finite automaton generated for the PROMELA code. The automaton for *never claim* represents the behavior that is considered as illegal or undesirable with respect to the property specified in LTL. When the verifier searches the state space for the automaton for PROMELA code together with the *never claim* automaton, basically it will look for counter examples where the specified property will be violated. In case if it finds a counter example where the property will be violated, it report an error and necessary trace will be generated for the counter example.

The DCR Graphs have events, relations and markings in the form of sets and hence the basic data structure is lists with event names. Since PROMELA only supports integer data types, so we have chosen to encode event names in the form of numeric constants by assigning numeric values to event names starting with 0. In order to minimize the memory and state space in SPIN, we have used *byte* data type for encoding the event names. The maximum value for a *byte* data type is only 255, therefore it will limit the number of events in a DCR Graph to 255. Of course this will be a limitation for modeling lager DCR Graphs, but in such cases one could easily change it to *short* data type which has a maximum value of 32767. Further we have chosen to use abbreviated event names (for example pm for prescribe medicine) as symbols for the byte constants in order to improve the readability of PROMELA code. Of course this will not affect the memory as symbols will be replaced with their constant values before SPIN generates the code for verifier. The declaration of symbols for abbreviated event names is shown in the figure 6.11 from line 7 to 12.

Further, PROMELA has only *arrays* as the basic data structures, so we have no other go except to use *arrays* to encode the information about events and relations in the DCR Graphs. Encoding of event set is straight forward, as we can encode an event set as an array. But the arrays in PROMELA are of fixed size and the event sets for state management in the DCR Graphs are dynamically changing in the number of events. Hence for this reason, we have decided to use bit arrays of fixed size equal to event count, keep tracking of the existence of an event in a set, by the bit value present at the position of array index equal to numeric value of the event. The events sets required for state management in DCR Graphs are declared as byte arrays as shown in figure 6.11 from line 14 to 18. For example, if we want to indicate that event *give medicine* (with numeric value = 2) is included in the set of *responses*, then we assign the bit value at index = 2 of the *responses* array to 1.

Encoding of relations is a bit complicated as PROMELA does has support for one dimensional array only. Hence we have used *typedef* construct to define two-dimensional array to encode the relations of DCR Graphs. For the encoding of relations, we have followed the same approach as that of encoding event set into byte array. The typedef definition for two dimensional array and declaration of arrays for the relations in DCR Graphs is shown in the figure 6.11 from line 20-26. To encode a relation from one event to other, we have defined a two-dimensional byte array of size (number of events X number of events) in a matrix layout with row index indicating source event numeric value and the column index indicating destination event numeric value of a relation and finally a bit value of located at the cross section of both event indices indicate the existence of a relation. For example, an include relation from *sign* to *give medicine* in figure 6.5 can be encoded in PROMELA as shown at line number 44 in figure 6.12. Finally, the specification for DCR Graph shown in the figure 6.3 encoded into PROMELA as shown in the figure 6.12. As part of the specification, we also write the initial marking, which involves specifying the events initially included in the process, events which are required as initial responses and

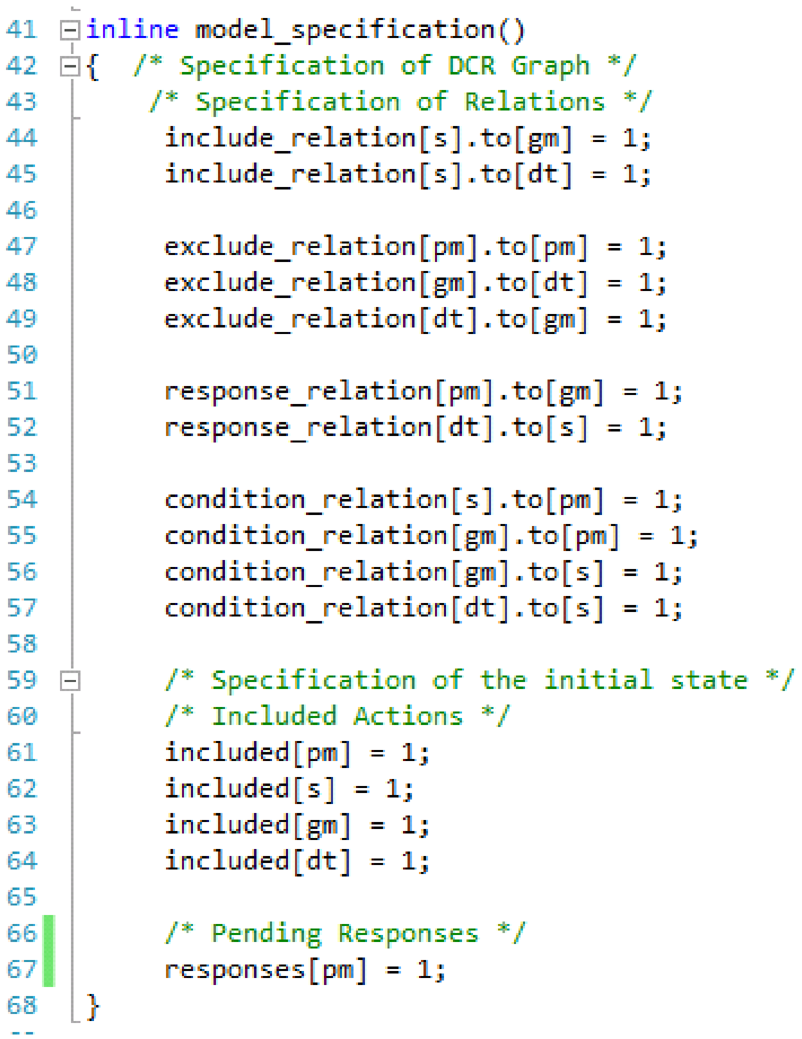


```
#define event_count 4
/* Declaration of events */
#define pm 0
#define s 1
#define gm 2
#define dt 3

/* Declarations of Marking */
bit included[event_count];
bit executed[event_count];
bit responses[event_count];
bit enabledset[event_count];

typedef twodimensionalarray {bit to[event_count]};
/* Declaration of relations */
twodimensionalarray condition_relation[event_count];
twodimensionalarray response_relation[event_count];
twodimensionalarray include_relation[event_count];
twodimensionalarray exclude_relation[event_count];
twodimensionalarray milestone_relation[event_count];

/* Looping Counters  */
byte index = 0;
byte index2 = 0;
short executed_event_count = 0;
bit accepted_marking = 1;
bit accepted_state_reached = 0;
bit can_execute = 1;
byte loopindex = 0;
/* Not possible to assign -1 to a
byte, so assign it event_count + 1 */
show byte random_event_executed = event_count + 1;
bit any_included_pending_responses = 0;
```

Figure 6.11: Variable declarations for DCR Graphs in PROMELA

the set of executed events is always empty as shown from line number 59-65 in the figure 6.12. Note that all data types in PROMELA are initialized to 0 by default, hence all the events which are not explicitly mentioned in the initial marking of the specification are not included in those sets.

Finally, PROMELA does not have procedures or functions to structure the code, but it has *inline* construct which can be used to group a sequence of statements with a given name as shown in the figure 6.12 at line number 41. We will use *inline*



```
inline model_specification()
{  /* Specification of DCR Graph */
    /* Specification of Relations */
     include_relation[s].to[gm] = 1;
     include_relation[s].to[dt] = 1;

     exclude_relation[pm].to[pm] = 1;
     exclude_relation[gm].to[dt] = 1;
     exclude_relation[dt].to[gm] = 1;

     response_relation[pm].to[gm] = 1;
     response_relation[dt].to[s] = 1;

     condition_relation[s].to[pm] = 1;
     condition_relation[gm].to[pm] = 1;
     condition_relation[gm].to[s] = 1;
     condition_relation[dt].to[s] = 1;

     /* Specification of the initial state */
     /* Included Actions */
     included[pm] = 1;
     included[s] = 1;
     included[gm] = 1;
     included[dt] = 1;

     /* Pending Responses */
     responses[pm] = 1;
}
```

Figure 6.12: DCR Graph specification in PROMELA

construct to group the statements related to one logical function through out the PROMELA programs generated from the verification tool.

6.3.3 Verification of Safety Properties

In this section, we will describe how to verify the safety properties: deadlock free (Def. 6.2.6) and strongly deadlock free (Def. 6.2.7) using SPIN tool. For the verification of safety properties, we will the use DCR Graph for the *give medicine* example (shown in figure 6.13), which was introduced in the section 6.2.2 on safety properties on DCR Graphs. The full PROMELA code for verification of a deadlock free property for the *give medicine* example is given in the appendix A.1 and in this section, we will take parts of code to explain the main key aspects.

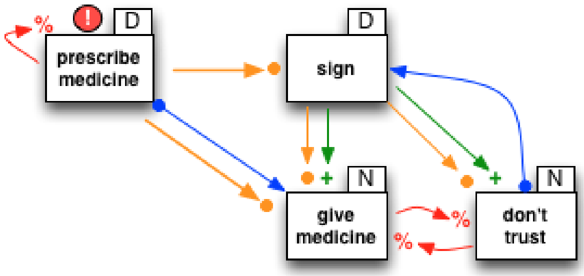
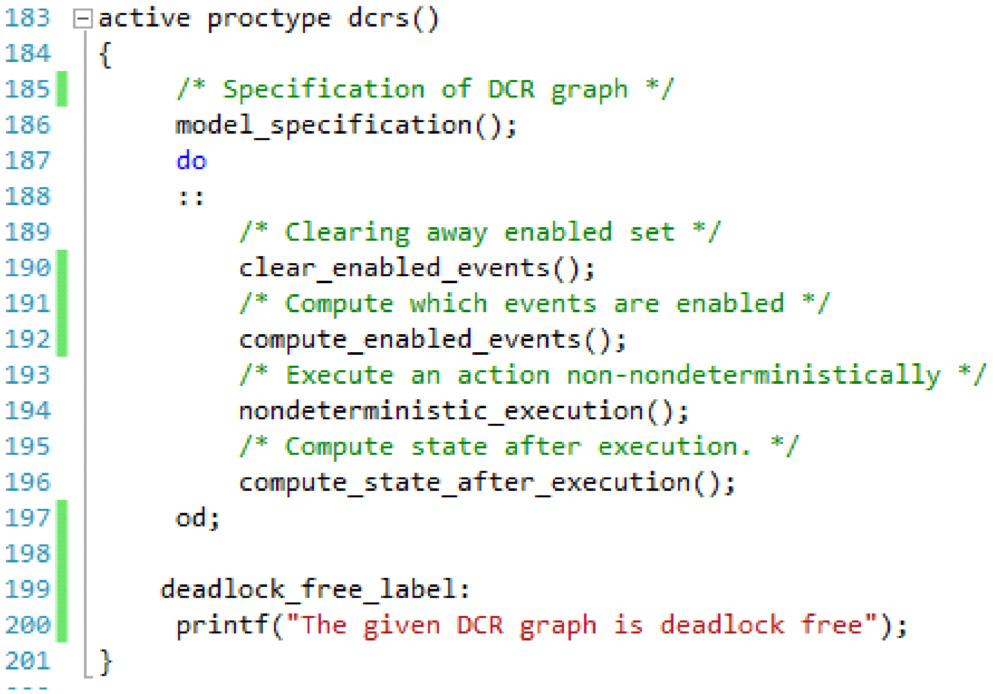


Figure 6.13: Give Medicine example



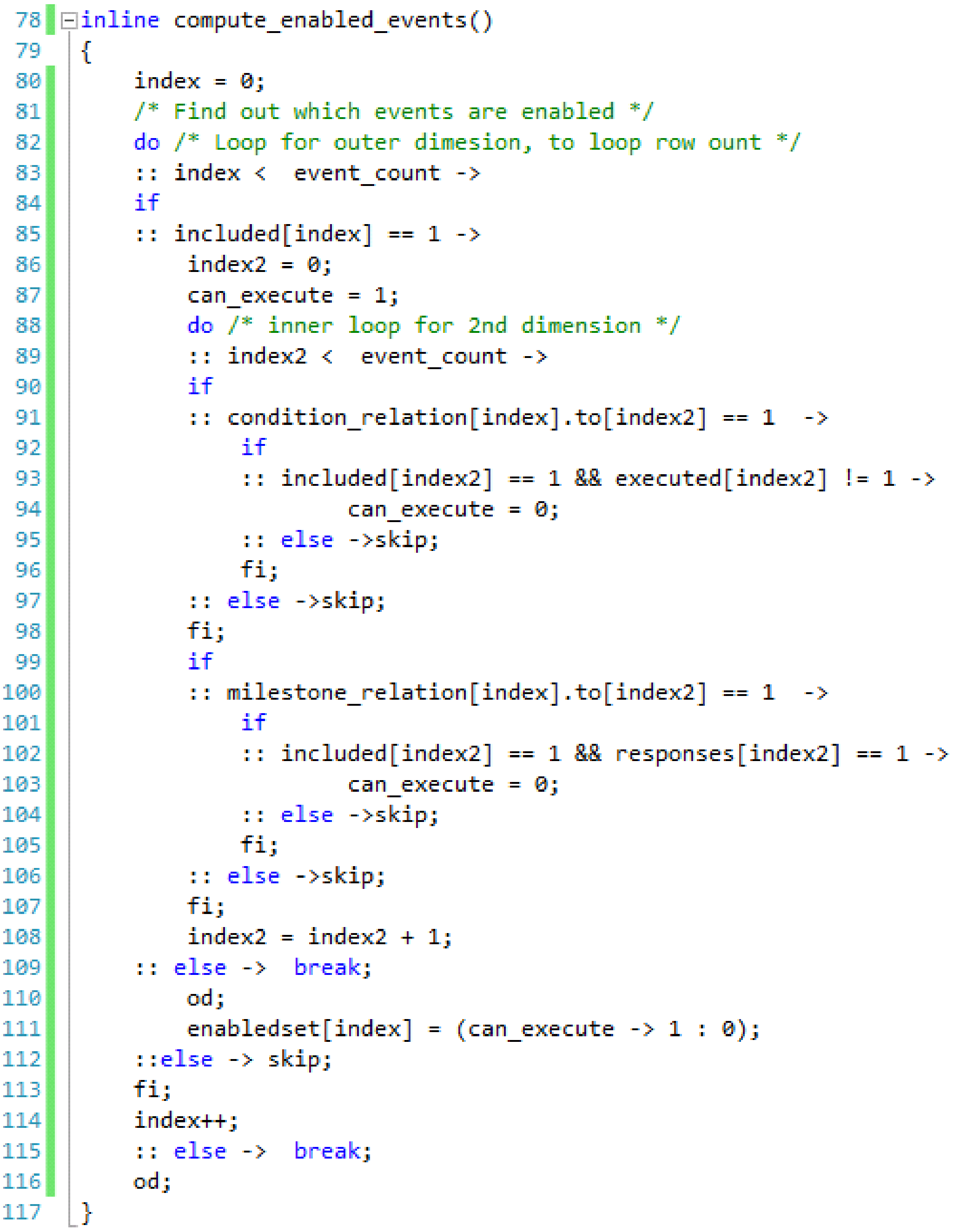
```
active proctype dcrs()
{
     /* Specification of DCR graph */
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which events are enabled */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
         /* Compute state after execution. */
         compute_state_after_execution();
     od;

    deadlock_free_label:
     printf("The given DCR graph is deadlock free");
}
```

Figure 6.14: PROMELA code for main process

6.3.3.1 Verification of deadlock free property

The overview of logic for verification of safety properties on DCR Graphs is shown in the figure 6.14, where it shows the logic for the main process function (*proctype dcrs*), which will instantiated by the SPIN to generate the code for verifier. The *proctype dcrs* contains one main *do* loop and calls to different *inline* code blocks.

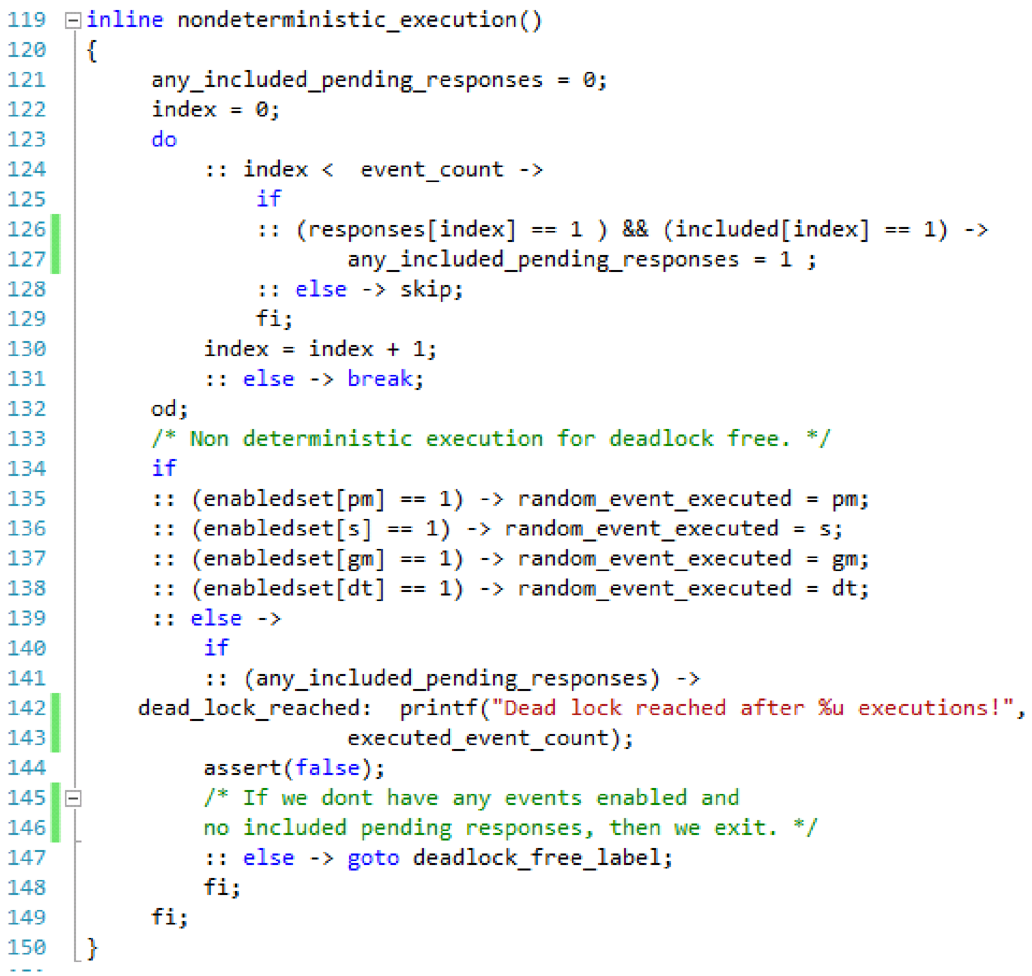
The first *inline* code block is *model_specification()*, which contains the specification of the DCR Graphs in PROMELA as described in the previous section. The next step is to compute the list of enabled events, which has to be computed repeatedly after execution of an event. The figure 6.12 shows the logic of computing enabled



```
78  inline compute_enabled_events()
79  {
80      index = 0;
81      /* Find out which events are enabled */
82      do /* Loop for outer dimesion, to loop row ount */
83      :: index <  event_count ->
84      if
85      :: included[index] == 1 ->
86          index2 = 0;
87          can_execute = 1;
88          do /* inner loop for 2nd dimension */
89          :: index2 <  event_count ->
90          if
91          :: condition_relation[index].to[index2] == 1  ->
92              if
93              :: included[index2] == 1 && executed[index2] != 1 ->
94                      can_execute = 0;
95              :: else ->skip;
96              fi;
97          :: else ->skip;
98          fi;
99          if
100         :: milestone_relation[index].to[index2] == 1  ->
101             if
102             :: included[index2] == 1 && responses[index2] == 1 ->
103                     can_execute = 0;
104             :: else ->skip;
105             fi;
106         :: else ->skip;
107         fi;
108         index2 = index2 + 1;
109     :: else ->  break;
110         od;
111         enabledset[index] = (can_execute -> 1 : 0);
112     ::else -> skip;
113     fi;
114     index++;
115     :: else ->  break;
116     od;
117 }
```

Figure 6.15: Computing enabled events in PROMELA code

events for a given DCR Graph, where we loop through the list of events in the *included* array and for each event in the *included* array, we will find out whether all its condition events included in the current marking are executed or not (line: 90-98 in fig 6.15). Similarly, we also check whether all included milestone events are part



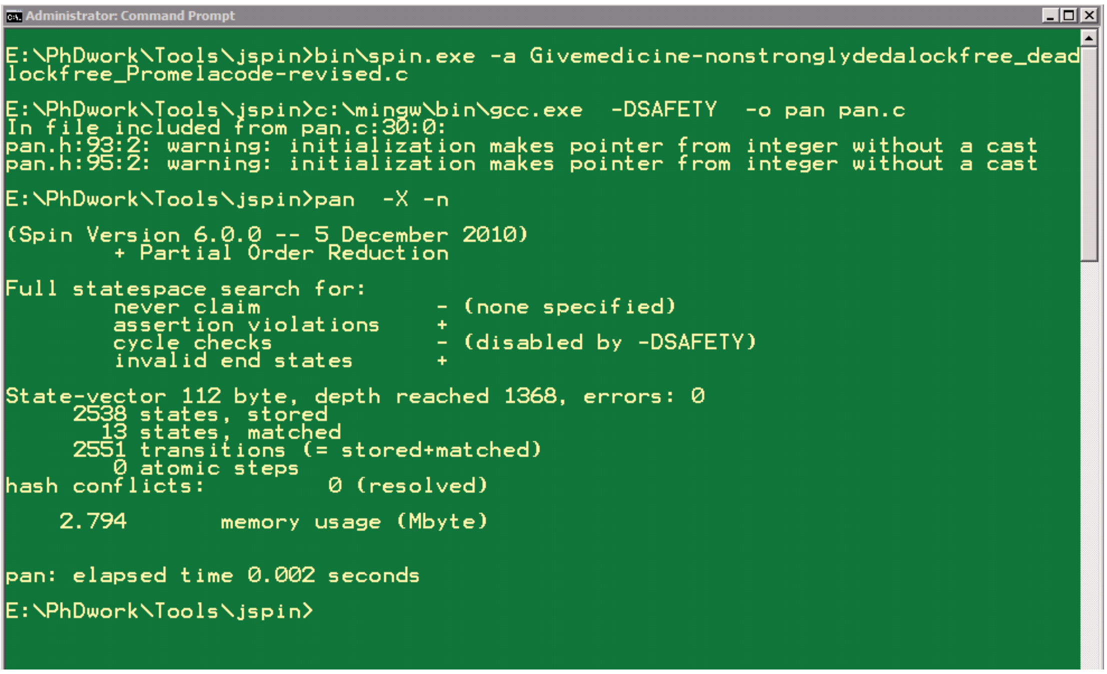
```
119 inline nondeterministic_execution()
120 {
121      any_included_pending_responses = 0;
122      index = 0;
123      do
124          :: index <  event_count ->
125              if
126              :: (responses[index] == 1 ) && (included[index] == 1) ->
127                     any_included_pending_responses = 1 ;
128              :: else -> skip;
129              fi;
130          index = index + 1;
131          :: else -> break;
132      od;
133      /* Non deterministic execution for deadlock free. */
134      if
135      :: (enabledset[pm] == 1) -> random_event_executed = pm;
136      :: (enabledset[s] == 1) -> random_event_executed = s;
137      :: (enabledset[gm] == 1) -> random_event_executed = gm;
138      :: (enabledset[dt] == 1) -> random_event_executed = dt;
139      :: else ->
140          if
141          :: (any_included_pending_responses) ->
142     dead_lock_reached:  printf("Dead lock reached after %u executions!",
143                     executed_event_count);
144          assert(false);
145          /* If we dont have any events enabled and
146          no included pending responses, then we exit. */
147          :: else -> goto deadlock_free_label;
148          fi;
149      fi;
150 }
```

Figure 6.16: Non deterministic execution in verification of deadlock free property

of the *responses* array. Finally, the *enabledset* will be updated with status of events enabled. Before computing the enabled events, the inline block *clear_enabled_events* will be called to clear the bit values of the events enabled.

The next and most important part is *nondeterministic_execution()* inline block, which contains the code for executing one of the enabled events from the *enabledset* as shown in the figure 6.16. First, we will calculate if there are any included pending responses in the current marking and then based on the status bit of the events in the *enabledset*, we will generate options to execute an event. In our formal verification, the execution of an event is nothing but assigning the numeric value of the event selected for execution to a variable called *random_event_executed*. As shown in the figure 6.16 from line 134-139, different alternatives for *if* block will be generated assigning a particular event to *random_event_executed* variable with a guard based on the status of the bit value in the *enabledset*. During verification of the model, the SPIN will evaluate these guards and short list the alternatives for which guards are evaluated to true and then it will execute one of the alternatives non-deterministically. In case if none of the alternatives statements are enabled for execution, which indicates a state where none of the events are available for execution, then the SPIN will execute the statements following *else* option, where it leads to 2 alternative statements.

If there are any included pending response events in the process, then it will lead to a deadlock situation according to Def. 6.2.6 and the verification will be forced to stop and raise an error by executing the *assert* statement with value false. On the other hand, if there are no included pending responses, then the marking is accepting and hence the program will jump to *deadlock_free_label*, which is defined at the end of the program and there by the program terminates. If there are enabled events in every marking, then the *else* block will never gets executed and the *do* loop will continue forever without breaking out, but SPIN is intelligent enough to trace the cycles of the states and then it terminates after inspecting all the states for the automaton.

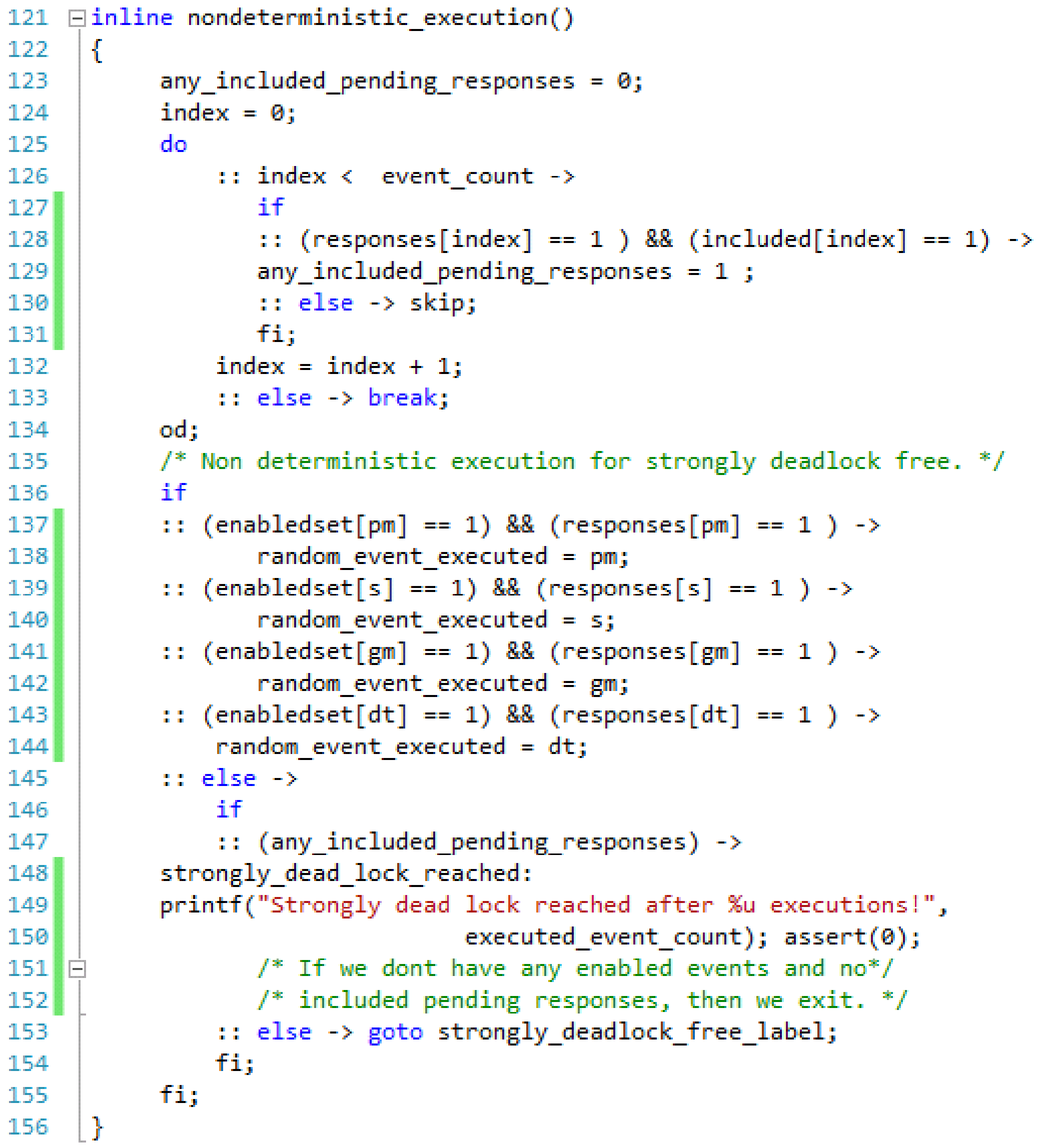


```
E:\PhDwork\Tools\jspin>bin\spin.exe -a Givemedicine-nonstronglydedalockfree_dead
lockfree_Promelacode-revised.c

E:\PhDwork\Tools\jspin>c:\mingw\bin\gcc.exe  -DSAFETY  -o pan pan.c
In file included from pan.c:30:0:
pan.h:93:2: warning: initialization makes pointer from integer without a cast
pan.h:95:2: warning: initialization makes pointer from integer without a cast

E:\PhDwork\Tools\jspin>pan  -X -n

(Spin Version 6.0.0 -- 5 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 112 byte, depth reached 1368, errors: 0
     2538 states, stored
       13 states, matched
     2551 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)

    2.794       memory usage (Mbyte)


pan: elapsed time 0.002 seconds

E:\PhDwork\Tools\jspin>
```

Figure 6.17: Verification of deadlock free property in SPIN - Console output

The generated PROMELA code can be verified by using the *spin.exe* in the command prompt and the output generated is shown in the figure 6.17. First, by using the SPIN command with *-a* will generate the verifier code from the PROMELA specification and it will output a set of C files *pan.**, which can be further compiled with C-compiler to produce an executable verifier. The executable verifier (*pan*) can be called with *-X* option to output the results to the command prompt. If the verifier finds any violations of correctness claims, it will report an error, otherwise the claims for the correctness are valid. In this case, our claim that the *give medicine* shown in the figure 6.13 is deadlock free is valid as the verifier fails to find any errors.

6.3.3.2 Verification of strongly deadlock free property

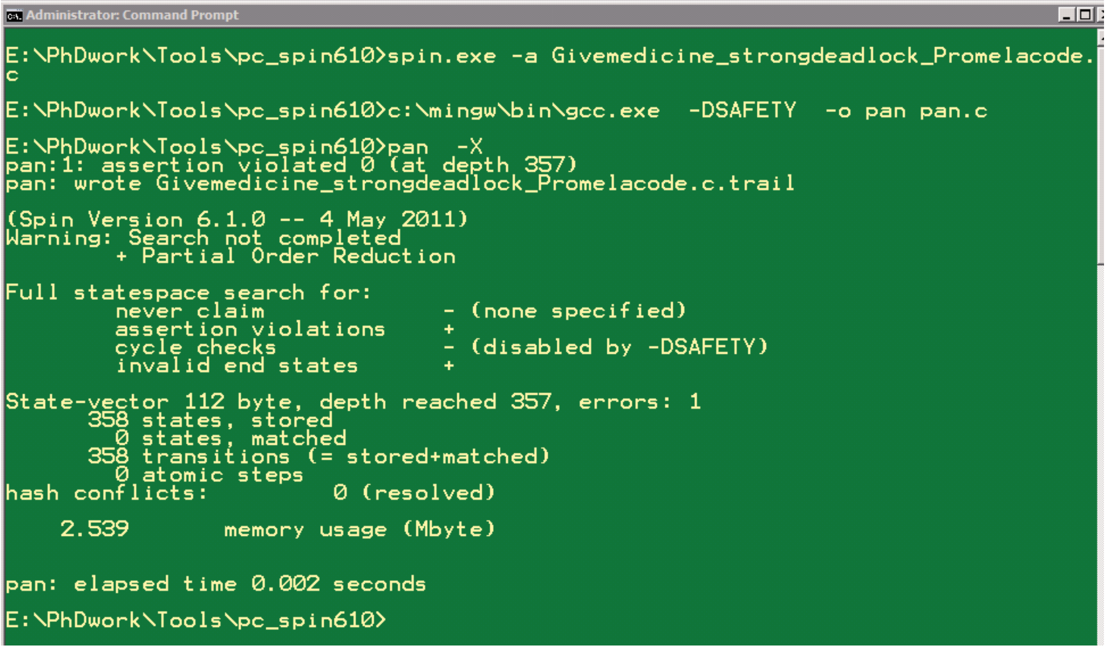


```
inline nondeterministic_execution()
{
     any_included_pending_responses = 0;
     index = 0;
     do
         :: index <  event_count ->
            if
            :: (responses[index] == 1 ) && (included[index] == 1) ->
            any_included_pending_responses = 1 ;
            :: else -> skip;
            fi;
         index = index + 1;
         :: else -> break;
     od;
     /* Non deterministic execution for strongly deadlock free. */
     if
     :: (enabledset[pm] == 1) && (responses[pm] == 1 ) ->
            random_event_executed = pm;
     :: (enabledset[s]  == 1) && (responses[s] == 1 ) ->
            random_event_executed = s;
     :: (enabledset[gm] == 1) && (responses[gm] == 1 ) ->
            random_event_executed = gm;
     :: (enabledset[dt] == 1) && (responses[dt] == 1 ) ->
         random_event_executed = dt;
     :: else ->
         if
         :: (any_included_pending_responses) ->
     strongly_dead_lock_reached:
     printf("Strongly dead lock reached after %u executions!",
                           executed_event_count); assert(0);
            /* If we dont have any enabled events and no*/
            /* included pending responses, then we exit. */
         :: else -> goto strongly_deadlock_free_label;
         fi;
     fi;
}
```

Figure 6.18: Non deterministic execution for strongly deadlock free property

In this section, we will verify the strongly deadlock free property (def 6.2.7) on the same give medicine example (shown in figure 6.13) used in verification of deadlock free property. We have observed that the give medicine example is deadlock free, but it is not strongly deadlock free as per the discussion in sec 6.2.2. Therefore, we will use the same deadlockfree give medicine example and verify whether it is strongly deadlock free or not using the SPIN tool.

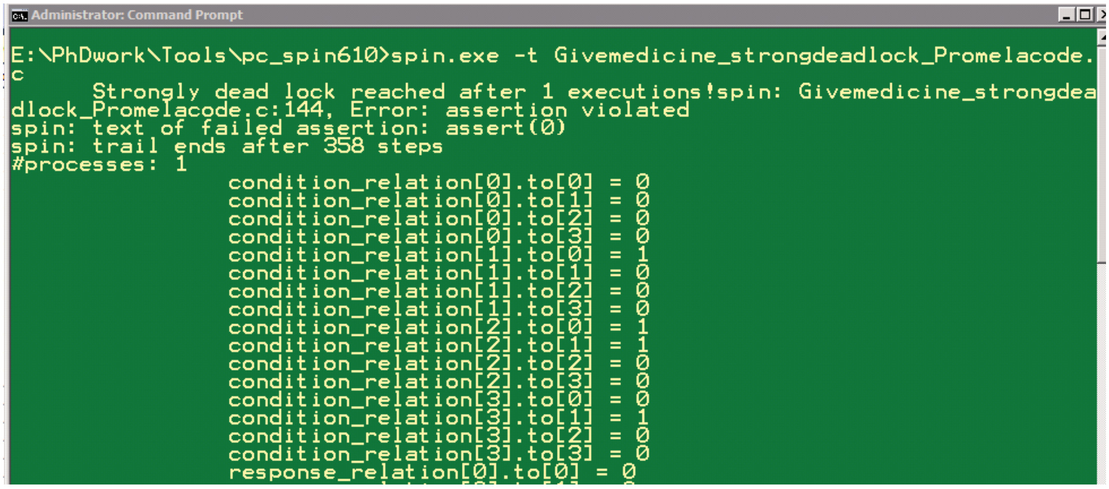
The full version of the PROMELA code generated by the DCR verification tool for the strongly deadlock free property is appended in the appendix A.2, however we will use important parts of the code to explain the key differences. The PROMELA code for strongly deadlock free property is almost same as the code for deadlock free property, except the code in *inline nondeterministic_execution*, which is shown in figure 6.18. One can observe that guards for alternatives (line 136-144) under the *if* block for non deterministic execution generated for strongly deadlock free property, now contains an additional condition saying that the enabled event must also be part of the required response set.



```
E:\PhDwork\Tools\pc_spin610>spin.exe -a Givemedicine_strongdeadlock_Promelacode.
c

E:\PhDwork\Tools\pc_spin610>c:\mingw\bin\gcc.exe  -DSAFETY  -o pan pan.c

E:\PhDwork\Tools\pc_spin610>pan  -X
pan:1: assertion violated 0 (at depth 357)
pan: wrote Givemedicine_strongdeadlock_Promelacode.c.trail

(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 112 byte, depth reached 357, errors: 1
      358 states, stored
        0 states, matched
      358 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)

    2.539       memory usage (Mbyte)


pan: elapsed time 0.002 seconds

E:\PhDwork\Tools\pc_spin610>
```

Figure 6.19: Verification of strongly deadlock free property in SPIN - Console output

The PROMELA code for strongly deadlock free property can be verified in the SPIN using the same commands described above for the deadlock free and we can observe that now the claim for correctness of strongly deadlock free property fails with the console output shown in the figure 6.19. In case the SPIN finds a violation of the claim for correctness of a property, it will generate a error trail by giving the state details where the claim has been violated. One can use *-t* option on the SPIN command to explore the trail and the output of the trail for violation of strongly deadlock free property on give medicine example is shown in the figure 6.20.



```
E:\PhDwork\Tools\pc_spin610>spin.exe -t Givemedicine_strongdeadlock_Promelacode.
c
      Strongly dead lock reached after 1 executions!spin: Givemedicine_strongdea
dlock_Promelacode.c:144, Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 358 steps
#processes: 1
                condition_relation[0].to[0] = 0
                condition_relation[0].to[1] = 0
                condition_relation[0].to[2] = 0
                condition_relation[0].to[3] = 0
                condition_relation[1].to[0] = 1
                condition_relation[1].to[1] = 0
                condition_relation[1].to[2] = 0
                condition_relation[1].to[3] = 0
                condition_relation[2].to[0] = 1
                condition_relation[2].to[1] = 1
                condition_relation[2].to[2] = 0
                condition_relation[2].to[3] = 0
                condition_relation[3].to[0] = 0
                condition_relation[3].to[1] = 1
                condition_relation[3].to[2] = 0
                condition_relation[3].to[3] = 0
                response_relation[0].to[0] = 0
```

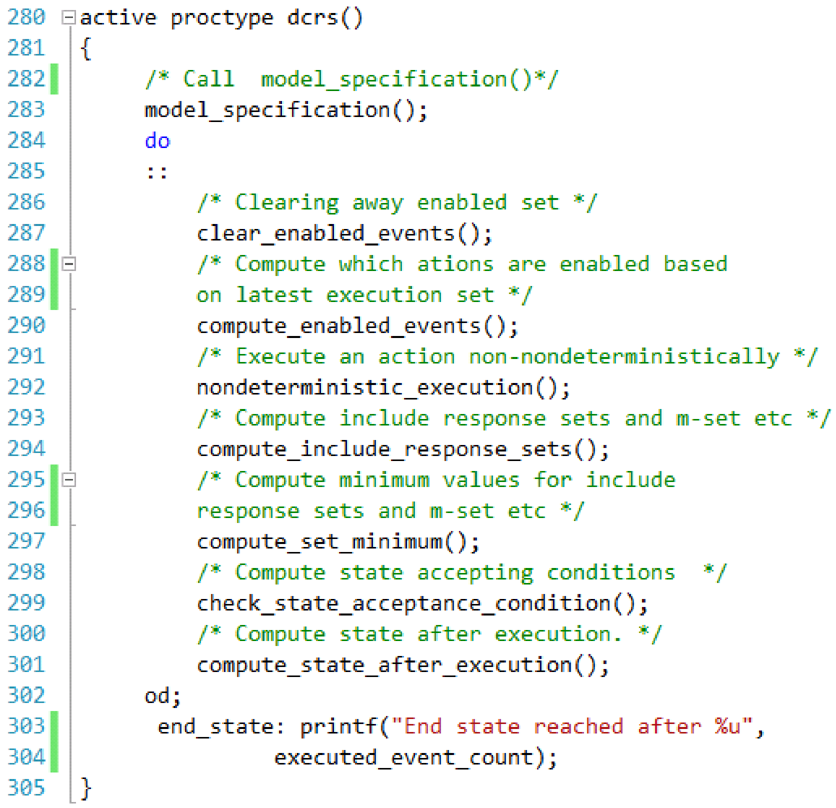
Figure 6.20: Error trail for violation of strongly deadlock free property in SPIN

6.3.4 Verification of Liveness Properties

In this section, we will describe how to check liveness properties on DCR Graphs using the SPIN tool. In contrast to the safety properties that are verified on finite runs, liveness properties are verified on infinite runs and thereby difficult to verify them. But fortunately the SPIN tools has good support for verification of liveness properties by the specifying the correctness properties using Linear Temporal Logic (LTL). In the SPIN [Holzmann 1997], the correctness property specified in LTL will be automatically converted into a Büchi automaton using the technique specified in [Gerth *et al.* 1995]. In order to verify the correctness of the claim, the SPIN uses negation of the specified LTL formulae to generate a *never claim* automaton. It tries to prove the correctness of claim by finding the intersection of the language of the system and the *never claim* is empty. On the other hand, if it finds an execution sequence that matches negated correctness claim, it reports it as error by providing the counter example in the error trail.

Another important challenge is modeling executions of DCR Graph that are accepting. As per the definition 6.2.3, an execution is accepting if any required included response in any intermediate marking are eventually executed or excluded $(\forall i \in [k].(\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathsf{In}'_j)))$. In other words, there should be any included response event left without execution or excluded for a execution to be accepting. In order to model this condition, we will use same formal technique of mapping DCR Graphs to Büchi automata from the section 3.3.4, where the accepting condition for DCR Graphs is characterized by mapping to Büchi automaton (definition 3.3.17). First we will describe the mapping informally and then show how it is encoded into PROMELA code for verification of liveness properties.

In the definition 3.3.17, in order to make sure that no event stays for ever in the included pending response set, we use multiple copies of the state space by adding

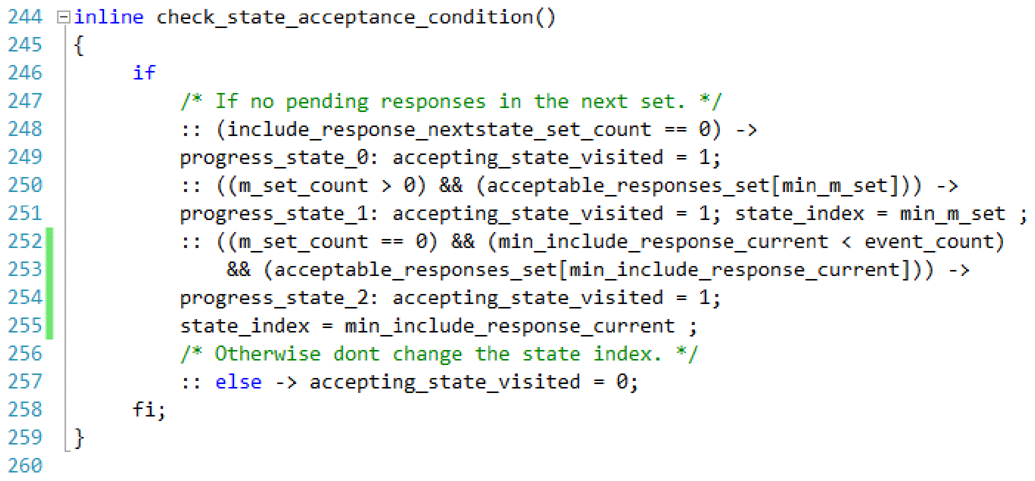


```
active proctype dcrs()
{
     /* Call  model_specification()*/
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which ations are enabled based
         on latest execution set */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
         /* Compute include response sets and m-set etc */
         compute_include_response_sets();
         /* Compute minimum values for include
         response sets and m-set etc */
         compute_set_minimum();
         /* Compute state accepting conditions  */
         check_state_acceptance_condition();
         /* Compute state after execution. */
         compute_state_after_execution();
     od;
      end_state: printf("End state reached after %u",
               executed_event_count);
}
```

Figure 6.21: Specification of global process for liveness properties

a state index to the marking (index i) . All the events in the event set are ranked according to some numerical order and after execution of every event, we will compute a minimum responses set (M_r) containing the included response events whose rank is greater than the state index ($M_r = \{e \in \mathrm{In}' \cap \mathrm{Re}' \mid rank(e) > i\}$). In case M_r is empty, we will use the included pending responses set ($\mathrm{In} \cap \mathrm{Re}$) in place of M_r set and in any case we compute the minimum element of the sets. In case the event executed is same as the minimum element of the responses set (M_r or $\mathrm{In} \cap \mathrm{Re}$), then we mark the state as an accepting state to indicate progress and jump to next copy of the state with state index $i = i + 1$. In this way, we can make sure that no event is left over in included pending responses set with out being executed or excluded.

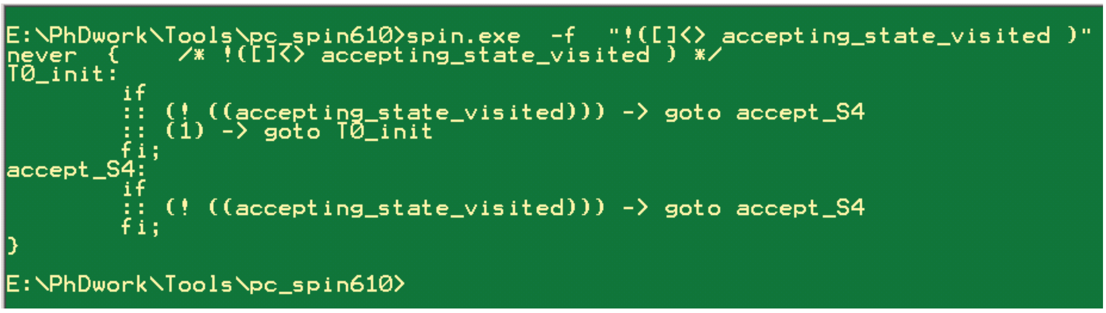
The global process specification in PROMELA for liveness properties is shown in the figure 6.21 and it contains inline code blocks to compute the minimum of M_r set or $\mathrm{In} \cap \mathrm{Re}$ set as explained above. The most important part of verification of liveness properties is the inline block to check whether a marking is accepting or not as shown in the figure 6.22. If the event executed is the minimum of M_r or set of included pending responses or if there are no included responses, then we mark the state as accepting by assigning the variable *accepting_state_visited* to 1 and mark these states with progress labels. In the other case where we don't make any progress, we assign *accepting_state_visited* to 0 to indicate that the the state is non accepting.



```
inline check_state_acceptance_condition()
{
    if
        /* If no pending responses in the next set. */
        :: (include_response_nextstate_set_count == 0) ->
        progress_state_0: accepting_state_visited = 1;
        :: ((m_set_count > 0) && (acceptable_responses_set[min_m_set])) ->
        progress_state_1: accepting_state_visited = 1; state_index = min_m_set ;
        :: ((m_set_count == 0) && (min_include_response_current < event_count)
            && (acceptable_responses_set[min_include_response_current])) ->
        progress_state_2: accepting_state_visited = 1;
        state_index = min_include_response_current ;
        /* Otherwise dont change the state index. */
        :: else -> accepting_state_visited = 0;
    fi;
}
```

Figure 6.22: Computation of accepting marking

Further, we verify the liveness properties by specifying the correctness claim by specifying that [] <> *accepting_state_visited* in LTL and SPIN will generate *never claim* for the negation of the property specified as shown in figure 6.23. Finally, we can save the *never claim* in a separate file and generate the verifier for the model along with the file containing *never claim*, which can be further verified in SPIN as explained in the previous section to verify liveness properties. The full PROMELA code generated by the DCR verification tool for liveness and strongly liveness properties are appended in the appendix A.3 and A.4 respectively.



```
E:\PhDwork\Tools\pc_spin610>spin.exe  -f  "!([]<> accepting_state_visited )"
never  {    /* !([]<> accepting_state_visited ) */
T0_init:
        if
        :: (! ((accepting_state_visited))) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((accepting_state_visited))) -> goto accept_S4
        fi;
}
E:\PhDwork\Tools\pc_spin610>
```

Figure 6.23: SPIN never claim for [] <> *accepting_state_visited*

6.4 Formal Verification using ZING

In this section, we briefly describe our efforts to do formal verification of DCR Graphs using Zing [Andrews *et al.* 2004, Fournet *et al.* 2004] model checker developed by Microsoft Research. Zing is a model checker for concurrent programs that manipulate the heap, by using boundaries that exist in the program. It has a modeling language for expressing concurrent models and a modeling checker for verification programs written in Zing language.

I have come across Zing tool while visiting Microsoft Research India on my stay abroad and explored using Zing as a model checker for verification of properties on DCR Graphs. The basic motivation for exploring Zing tool as against SPIN model checker can be explained as follows.

1. The Zing language has rich set of constructs for modeling programs and processes. Like PROMELA and other the modeling languages, Zing has support for concurrency, message passing communication either through shared memory or buffered queues and also support for modeling non-deterministic behavior. In addition to these, it also supports functions, objects, exceptions, and dynamic memory allocation as the built-in features in the modeling language. The Zing language supports Sets and other complex types, rich flow constructs for branching and iteration, therefore modeling the DCR Graphs in Zing modeling language is more or less straight forward.

2. The Zing Model checker has infrastructure for generating Zing models automatically from common programming languages like VB, C/C++, C#, and MSIL. Since our prototype tools are implemented in C#, we want to explore the possibility of automatic verification of properties on the DCR Graphs by using the Zing compiler inside the prototype tools.

We have modeled few examples of the DCR Graphs in Zing language and verified them in Zing model checker. First of all, modeling the DCR Graphs in Zing language is more or less straight forward as we have expected. We have also noticed that the number of model checker program states (not the DCR markings) are less when compared to SPIN, due to the richness of Zing language.

The main drawback of the Zing model checker is that it only supports verification of safety properties on the models. It does not have support for identifying progress and acceptance cycles on infinite runs and hence liveness properties could not be verified on models. Lack of support for liveness by the Zing limits the usage of the tool for purpose of formal verification of properties on the DCR Graphs. On contrary, SPIN has very good support for verification of liveness properties using LTL that gets translated into Büchi automaton, so we have chosen to use SPIN as the model checker for formal verification of the DCR Graphs. However the give medicine example modeled in Zing language for verification of deadlock property is enclosed in appendix B for more details.

6.5 Prototype Tools

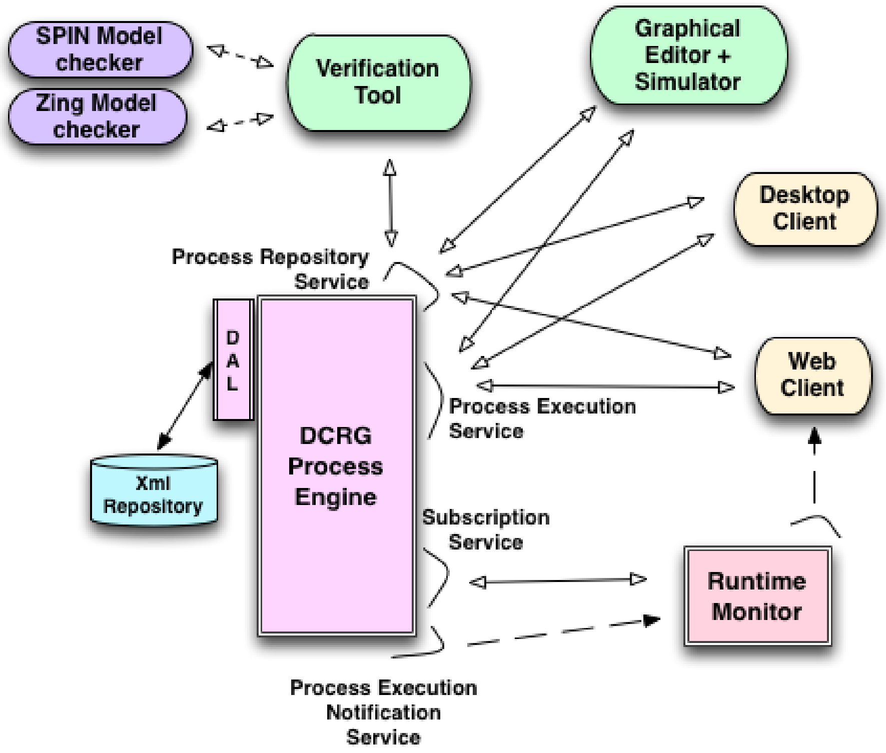
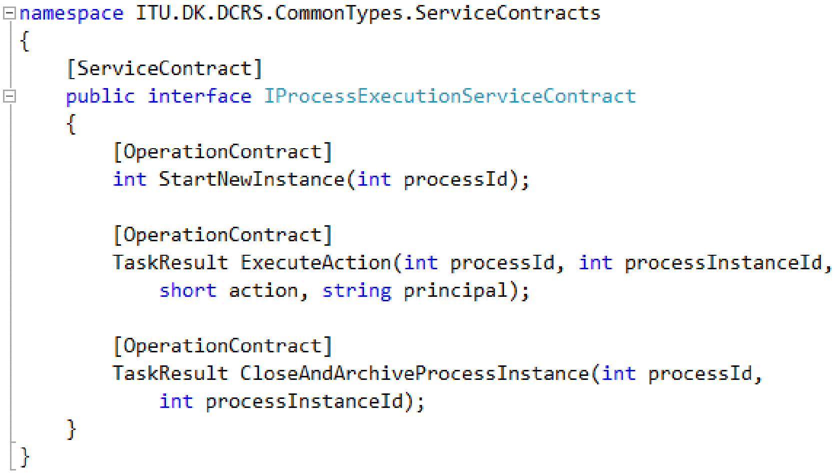


Figure 6.24: Protoype Architecture

To support modeling processes and workflows with the DCR Graphs, making the models available to a wider audience and allow interested parties to experiment with our formal model, we have been developing prototype implementations of various tools for the DCR Graphs. As of now, the tool implementation supports specification and execution of business processes and workflows specified in the DCR Graphs. The recent extensions to the model such as sub-processes, data and distribution of DCR Graphs are not yet supported in the tools, but we do have plans to support them in the implementation in the near future. The over architecture of the prototype tools is shown in the figure 6.24.

Most of the tools in the prototype implementation are written in C# using Microsoft .Net platform, but they have been implemented in service oriented architecture in a flexible manner so that the client applications developed on other platforms will be able to communicate easily. A very brief description of the important components of the prototype implementation with their functionality is explained below.

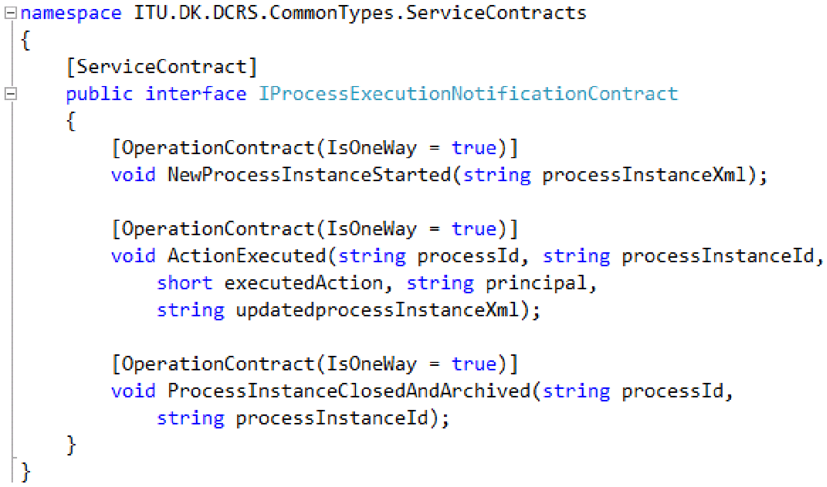
In the following tools, Windows-based Graphical Editor and Web Client are developed by my colleague, but we are including here in the thesis, with a good intension to provide the reader a overall picture of all the tools and the implementation that were built around the DCR Graphs.



```
namespace ITU.DK.DCRS.CommonTypes.ServiceContracts
{
    [ServiceContract]
    public interface IProcessExecutionServiceContract
    {
        [OperationContract]
        int StartNewInstance(int processId);

        [OperationContract]
        TaskResult ExecuteAction(int processId, int processInstanceId,
            short action, string principal);

        [OperationContract]
        TaskResult CloseAndArchiveProcessInstance(int processId,
            int processInstanceId);
    }
}
```

Figure 6.25: Process Execution Service Contract

6.5.1 DCRG Process Engine

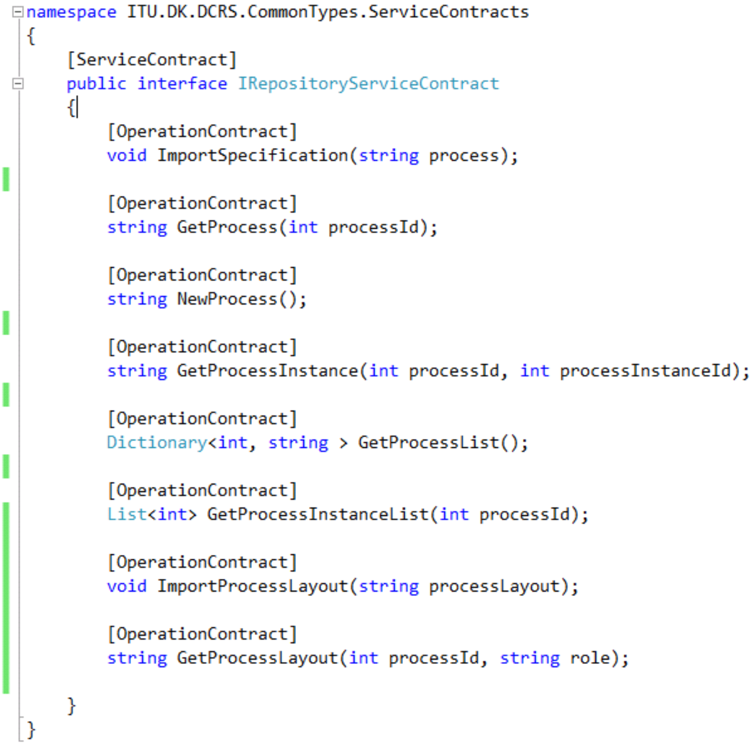


```
namespace ITU.DK.DCRS.CommonTypes.ServiceContracts
{
    [ServiceContract]
    public interface IProcessExecutionNotificationContract
    {
        [OperationContract(IsOneWay = true)]
        void NewProcessInstanceStarted(string processInstanceXml);

        [OperationContract(IsOneWay = true)]
        void ActionExecuted(string processId, string processInstanceId,
            short executedAction, string principal,
            string updatedprocessInstanceXml);

        [OperationContract(IsOneWay = true)]
        void ProcessInstanceClosedAndArchived(string processId,
            string processInstanceId);
    }
}
```

Figure 6.26: Notification Service Contract

It is the core component of the prototype implementation which handles the functionality of executing process instances of the DCR Graphs, based on the requests from various clients. The functionality of process engine has been developed as class library, so that the engine can be hosted in any hosting environment such as a windows or web service. Further, the process engine exposes a service for handling execution requests for process instances through a contract shown in figure 6.25. On the other hand, the process engine does not have any implementation for security, so any client can call the services of process engine without providing any security parameters. Even though it is a limitation, we don't think that it is very crucial as the main purpose of the prototype is to demonstrate the power of our formal model.

Further, the process engine also handles the functionality of subscription and notification of execution of the process instances. Therefore any client that is interested in notifications of execution of a particular process instance, can subscribe to the subscription service and there by receives the notifications from the process engine. This functionality is quite necessary for the clients which implement the functionality of runtime verification on the execution of process instances. The process engine does not have any implementation for runtime verification of process instances, but it provides support for runtime verification clients through the notification services. In order to have a scalable process engine, we think, it is a quite important design choice, not to implement runtime verification functionality as part of the process engine, but to implement as a client functionality. Hence a prototype client implementing runtime verification functionality has been developed separately. The service contract of notification services is shown in the figure 6.26.

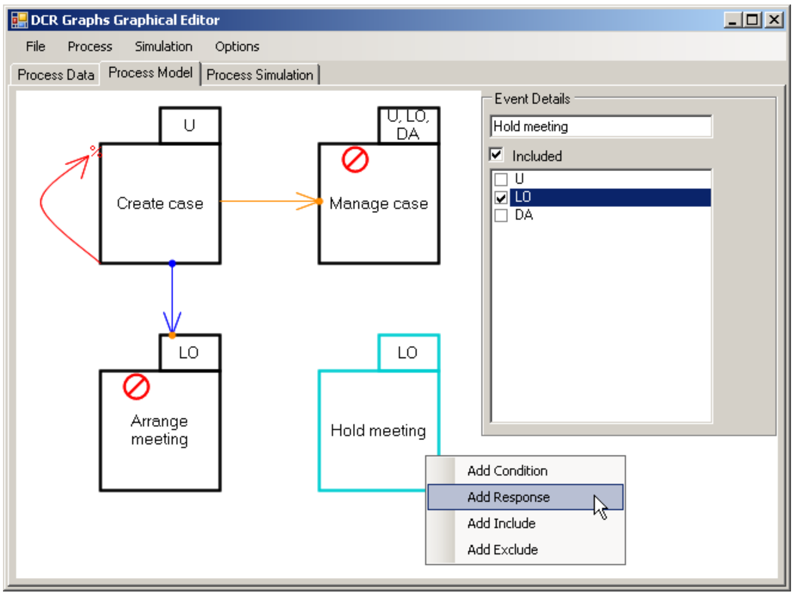


```
namespace ITU.DK.DCRS.CommonTypes.ServiceContracts
{
    [ServiceContract]
    public interface IRepositoryServiceContract
    {
        [OperationContract]
        void ImportSpecification(string process);

        [OperationContract]
        string GetProcess(int processId);

        [OperationContract]
        string NewProcess();

        [OperationContract]
        string GetProcessInstance(int processId, int processInstanceId);

        [OperationContract]
        Dictionary<int, string > GetProcessList();

        [OperationContract]
        List<int> GetProcessInstanceList(int processId);

        [OperationContract]
        void ImportProcessLayout(string processLayout);

        [OperationContract]
        string GetProcessLayout(int processId, string role);

    }
}
```

Figure 6.27: Service Contract implemented by Process Repository

6.5.2 Process Repository

The Process Repository handles the functionality of persisting and supplying the process definitions and instances of the DCR Graphs. It plays a role of persistence or data access layer when the prototype compared to a standard workflow management system or a business process management systems. The functionality of process repository is exposed through Process Repository Service which implements the contract shown in figure 6.27. In the prototype implementation, we have developed a simple process repository based on Xml files stored on a hard disk. However it can be easily replaced by a process repository implemented using a database as store for persisting the process definitions and instances.



Event Details

Hold meeting

☑ Included

□ U
☑ LO
□ DA

Figure 6.28: The Graphical Editor for DCR Graphs

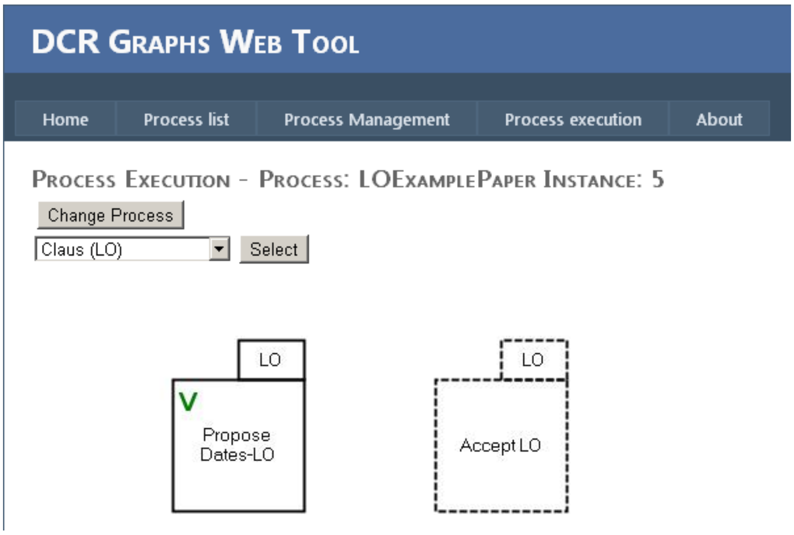
6.5.3 Windows-based Graphical Editor

As part of the prototype implementation, we have also developed a windows-based graphical editor for modeling declarative processes in DCR Graphs as shown in the figure 6.28. The tool uses the graphical notation for DCR Graphs introduced in the section 3.4.

The graphical editor also has support for process simulation by executing a process instance through process engine, so that users can simulate their processes and test them during the specification phase.

6.5.4 Web Client

A platform independent web client has also been developed, which can be used for executing processes modeled in DCR Graphs as shown in figure 6.29. In future, we also aim to support modeling of processes in DCR Graphs through this web interface as well.

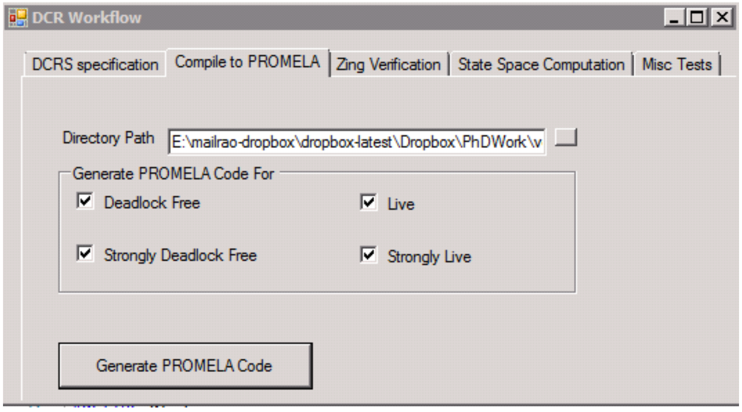


PROCESS EXECUTION - PROCESS: LOEXAMPLEPAPER INSTANCE: 5

Claus (LO)

Figure 6.29: Execution of a DCR Graph in the Web Tool

6.5.5 Model Checking Tool



Directory Path E:\mailrao-dropbox\dropbox-latest\Dropbox\PhDWork\v

Generate PROMELA Code For

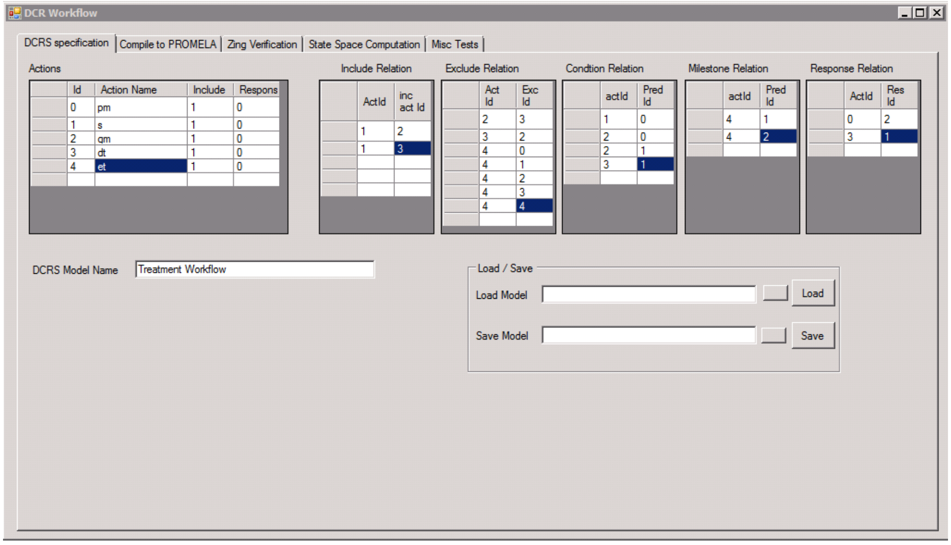
☑ Deadlock Free ☑ Live

☑ Strongly Deadlock Free ☑ Strongly Live

Figure 6.30: Code generation options for Model checkers

In order to do formal verification on the processes modeled in the DCR Graphs, we have used SPIN and ZING model checking tools as explained in the Sec. 6.3 and

Sec. 6.4. In order to generate the code for the model checkers automatically, we have developed a tool, which takes the input of a DCR Graph using a simple graphical user interface as shown in the figure 6.31 and generates the code for the PROMELA language using the options shown in figure 6.30.



Actions

Id	Action Name	Include	Respons
0	pm	1	0
1	s	1	0
2	qm	1	0
3	dt	1	0
4	et	1	0

Include Relation

ActId	inc act Id
1	2
1	3

Exclude Relation

Act Id	Exc Id
2	3
3	2
4	0
4	1
4	2
4	3
4	4

Condtion Relation

actId	Pred Id
1	0
2	0
2	1
3	1

Milestone Relation

actId	Pred Id
4	1
4	2

Response Relation

ActId	Res Id
0	2
3	1

DCRS Model Name: Treatment Workflow

Load / Save

Load Model

Save Model

Figure 6.31: Model Checking Tool for DCR Graphs

6.5.6 Serialization Format for DCR Graphs

Listing 6.1 shows a brief overview of the XML format of the DCR Graphs that is being used in all prototype tools. A single XML format is used to contain information about both the specification and the runtime of a DCR Graph. The resources section of the specification contains information about roles, principals, events and actions, whereas the access controls section contains the mapping of principals and actions to roles. The last part of the specification contains the binary relations between the events. Note that the XML format supports nesting of events and the binary relations in between them and that flattening of nested events and their relations will be done at the beginning of executing a DCR Graph.

The second part of the XML format for a DCR Graph holds the runtime information, which primarily contains the execution trace and information about the current state. The execution trace records the actual sequence of events executed and the current state holds the information about the current marking which contains sets of included, executed and pending response events. In addition to the marking, the current state also holds additional information such as index of state copy, state accepted to support the acceptance condition for infinite computations that were characterized by mapping to Büchi-automata in [Mukkamala & Hildebrandt 2010, Hildebrandt & Mukkamala 2010].

Listing 6.1: Overview of DCR Graph Xml

```
<?xml version="1.0" encoding="utf-8"?>
<dcrg:process xmlns:dcrg="http://itu.dk/trustcare/dcr/2011/">

  <dcrg:specification processId="" modelName="">
      <dcrg:resources>
        <dcrg:roles >.....</dcrg:roles>
        <dcrg:principals >.....</dcrg:principals>
        <dcrg:events >.....</dcrg:events>
        <dcrg:actions >.....</dcrg:actions>
     </dcrg:resources>
     <dcrg:accessControls>
        <dcrg:rolePrincipalAssignments >.....</dcrg:
            rolePrincipalAssignments>
        <dcrg:actionRoleAssignments >.....</dcrg:actionRoleAssignments>
     </dcrg:accessControls>
     <dcrg:constraintSets>
        <dcrg:constraintSet type="condition">...</dcrg:constraintSet>
        <dcrg:constraintSet type="response">...</dcrg:constraintSet>
     </dcrg:constraintSets>
  </dcrg:specification>

  <dcrg:runtime processInstanceId="">
      <dcrg:executionTrace>  </dcrg:executionTrace>
    <dcrg:currentState stateId="">
        <dcrg:eventsIncluded >.....</dcrg:eventsIncluded>
        <dcrg:eventsExecuted >.....</dcrg:eventsExecuted>
        <dcrg:eventsPendingResponses >.....</dcrg:eventsPendingResponses>
        <dcrg:stateAccepting> </dcrg:stateAccepting>
        <dcrg:stateIndex> </dcrg:stateIndex>
        <dcrg:eventsEnabled >.....</dcrg:eventsEnabled>
     </dcrg:currentState>
  </dcrg:runtime>

</dcrg:process>
```

The specification section of the XML document for the case handling process (figure 4.4) introduced in the case management case study (Sec. 4.1.3) is given in listing 6.2.

Listing 6.2: DCRG specification in Xml

```
<dcrg:specification>

<dcrg:resources>
<dcrg:roles>
   <dcrg:role>U</dcrg:role>
   <dcrg:role>LO</dcrg:role>
   <dcrg:role>DA</dcrg:role>
</dcrg:roles>
<dcrg:principals>
   <dcrg:principal>u</dcrg:principal>
   <dcrg:principal>lo</dcrg:principal>
   <dcrg:principal>da</dcrg:principal>
```

```
</dcrg:principals>
<dcrg:events>
    <dcrg:event eventId="0" name="Create case" actionId="Create case">
        <dcrg:event eventId="1" name="Submit" actionId="Submit" />
        <dcrg:event eventId="2" name="Assign case Id" actionId="Assign
            case Id" />
        <dcrg:event eventId="3" name="Edit" actionId="Edit">
                <dcrg:event eventId="4" name="Metadata" actionId="
                    Metadata" />
                <dcrg:event eventId="5" name="Dates available" actionId="
                    Dates available" />
        </dcrg:event>
    </dcrg:event>
    <dcrg:event eventId="6" name="Manage case" actionId="Manage case">
        <dcrg:event eventId="7" name="Edit metadata" actionId="Edit
            metadata" />
        <dcrg:event eventId="8" name="Document" actionId="Document">
                <dcrg:event eventId="9" name="Upload" actionId="Upload"
                    />
                <dcrg:event eventId="10" name="Download" actionId="
                    Download" />
        </dcrg:event>
     </dcrg:event>
     <dcrg:event eventId="11" name="Arrange Meeting" actionId="Submit">
        <dcrg:event eventId="12" name="Propose dates-LO" actionId="
            Propose dates-LO" />
        <dcrg:event eventId="13" name="Accept LO" actionId="Accept LO" />
        <dcrg:event eventId="14" name="Accept DA" actionId="Accept DA" />
        <dcrg:event eventId="15" name="Propose dates-DA" actionId="
            Propose dates-DA" />
     </dcrg:event>
     <dcrg:event eventId="16" name="Hold meeting" actionId="Hold meeting"
          />
</dcrg:events>
<dcrg:actions>
        <dcrg:action actionId="Create case" />
        <dcrg:action actionId="Submit" />
        <dcrg:action actionId="Edit" />
        <dcrg:action actionId="Metadata" />
        <dcrg:action actionId="Dates available" / >....
</dcrg:actions>
</dcrg:resources>
<dcrg:accessControls>
      <dcrg:rolePrincipalAssignments>
        <dcrg:rolePrincipalAssignment role-name="U">
                <principal>u</principal>
        </dcrg:rolePrincipalAssignment>
        <dcrg:rolePrincipalAssignment role-name="LO">
                <principal>lo</principal>
        </dcrg:rolePrincipalAssignment>
      </dcrg:rolePrincipalAssignments>
      <dcrg:actionRoleAssignments>
         <dcrg:actionRoleAssignment actionId="Submit">
                <dcrg:role>U</dcrg:role>
         </dcrg:actionRoleAssignment>
         <dcrg:actionRoleAssignment actionId="Document">
                <dcrg:role>U</dcrg:role>
                <dcrg:role>LO</dcrg:role>
                <dcrg:role>DA</dcrg:role>
         </dcrg:actionRoleAssignment>    ....
       </dcrg:actionRoleAssignments>
</dcrg:accessControls>
<dcrg:constraintSets>
      <dcrg:constraintSet type="condition">
        <dcrg:constraint source="1" target="2" />
        <dcrg:constraint source="3" target="1" / >.....
      </dcrg:constraintSet>
      <dcrg:constraintSet type="response">
        <dcrg:constraint source="1" target="2" / >.....
      </dcrg:constraintSet> .....
</dcrg:constraintSets>

</dcrg:specification>
```

The listing 6.3 shows the runtime information for the case handling process from the figure 4.5.

Listing 6.3: DCRG Runtime in Xml

```
<dcrg:runtime processInstanceId="">
     <dcrg:executionTrace>4,5,4,1,2,12,15</dcrg:executionTrace>
     <dcrg:currentState stateId="S6">
        <dcrg:eventsIncluded>2,4,5,7,9,10,12,13,14,15,16</dcrg:
            eventsIncluded>
        <dcrg:eventsExecuted>1,2,4,5,12,15</dcrg:eventsExecuted>
        <dcrg:eventsPendingResponses>13,14,16</dcrg:
            eventsPendingResponses>
        <dcrg:stateAccepting>0</dcrg:stateAccepting>
        <dcrg:stateIndex>0</dcrg:stateIndex>
        <dcrg:eventsEnabled>1,2,4,5,7,8,12,13,14,15</dcrg:eventsEnabled>
     </dcrg:currentState>
</dcrg:runtime>
```

6.6 Summary

In this chapter, we have introduced the notion of deadlock and livelock freeness on the DCR Graphs and formally defined safety and liveness properties in terms of executions and markings of a DCR Graph in the section 6.2. Since DCR Graphs have a distinction between which events may or must (eventually) happen, we have defined strong variants of safety and liveness representing the situation where only required events are executed.

We then proceeded to give brief introduction to SPIN tool and its language PROMELA and then explained how to encode a DCR Graph process into PROMELA, to do the formal verification of safety and liveness properties using our running example *presrcibe medicine* in the section 6.3. Later, we briefly mentioned about our experience in using ZING model checker for verification of safety properties on the DCR Graphs. Finally, we provided a brief overview of the prototype tools and implementation built around the formal model of DCR Graphs.

Chapter 7

Conclusion and Future Work

In this chapter, we will first conclude the work developed in the thesis and then provide a list of claims for achieving the research goal (Sec 1.5.2) specified in the introduction chapter. Later, in the second part, we will describe possible future work and extensions to the thesis.

7.1 Conclusion

In the thesis, we have developed the formal model DCR Graphs for specification and execution of flexible workflows and business processes based on declarative modeling primitives, taking motivation from declarative workflow language employed by our industrial partner Resultmaker A/S [Resultmaker 2008].

In chapter 2, we have introduced the formalisms that have served as background and motivation for our formal model. As part of that, we have described our first attempt to formalize the key primitives of Resultmaker's Online Consultant workflow using Linear Temporal Logic [Pnueli 1977] and also described a case study and our experiences in modeling a healthcare workflow using Resultmaker workflow method. Furthermore, we have briefly described the motivation from the DECLARE [van der Aalst *et al.* 2010a] framework and finally we have provided an introduction to Event Structures [Winskel 1986], which served as base theory behind our formal model.

Furthermore, we have introduced our formal model DCR Graphs in chapter 3 along with execution semantics mapped to labelled transition system for finite runs and to Büchi automata for infinite runs. We have also introduced graphical language for DCR Graphs along with notation to represent runtime state as marking on the graph itself. Regarding expressiveness of DCR Graphs, we have encoded Büchi automaton and proved that DCR Graphs is expressive enough to model all ω-regular languages. The extensions such as nested sub-graphs, multi-instance sub processes and an initial version of data to DCR Graphs have been presented in chapter 4.

We have defined the notion of projection and composition on DCR Graphs to distribute a global DCR Graph as a set of synchronously communicating local graphs in chapter 5. We have proved that the distribution is safe in the sense that the behavior exhibited by the network of local graphs is bisimilar to that of global graph. The distribution technique have also been extended to nested DCR Graphs and we have distributed the healthcare workflow using nested DCR Graphs with the notion of projection. Our distribution method is quite generic and the strength of distribution lies in the fact that the resulting local components are also DCR Graphs, which keep their declarative nature.

Finally, we have defined safety and liveness properties on DCR Graphs in chapter 6 and explained a method to encode DCR Graphs into PROMELA language and then formally verify the properties using SPIN model checker. We have also briefly introduced the prototype tools build for DCR Graphs in the last chapter.

7.2 Contribution

In the introduction chapter, as part of the thesis statement 1.5, we have stated that the research goal of this thesis is to show that it is possible to formalize the key primitives of Resultmaker declarative model and further develop it as a comprehensive formal model suitable for specification and execution of workflows. Furthermore, we have also stated that the formal model should also allow safe distribution of a global workflow as a set of communicating local components, based on top-down model-driven approach. Finally, we mentioned in the research goal that, we intent to analyze declarative processes by adding support for formal verification with the help of model checking tools.

The thesis has made several contributions, but we list the main contributions as follows.

- We have shown that it is possible to formalize the key primitives of Resultmaker declarative workflow and further developed it as a comprehensive formal model DCR Graphs, which is suitable for specification and execution of workflows based on declarative modeling primitives.

- With 5 core relations, our formal model DCR Graphs is simple but sufficiently expressive enough to model all ω-languages. We have proved that the DCR Graphs is bisimilar to Büchi automata in expressing infinite runs.

- Our formal model allows for an intuitive operational semantics and effective execution expressed by a notion of markings of the graphs. Furthermore we have also developed a graphical language along with runtime notation for the DCR Graphs and the runtime state of a DCR Graph can be simply visualized as a marking on the graph itself. The graphical notation for the DCR Graphs is also quite useful in modeling workflows in DCR Graphs, especially for the people without formal background.

- We have provided a general technique for distributing a declarative global process as set of synchronously communicating local processes, by defining the notion of projection and composition. The generality of our distribution technique allows for fined tuned projections, where one can choose only few events for a specific role or an actor and most importantly the projected local graphs keep their declarative nature as the resulting projections are also DCR Graphs, which can be further distributed.

- In case of distribution of DCR Graphs, we have proved the main theorem that the distribution is safe in the sense that the behavior exhibited by the local

DCR Graphs is consistent with the behavior exhibited by the global DCR Graph. Furthermore the distribution technique has also been extended to nested DCR Graphs.

- We have applied formal verification techniques on declarative business processes specified in DCR Graphs and provided a method to verify properties on DCR Graphs using SPIN model checker. Furthermore we have built a tool that automatically generates verification code from workflows models specified in DCR Graphs and the verification code can be run in SPIN or ZING model checkers, to verify properties on DCR Graphs.

- We have modeled two case studies, one from healthcare and the other from case management domain in DCR Graphs, to show that our formal model is adequate for modeling the workflows from dynamic sectors where flexibility is of paramount importance.

- This thesis is one of the few ones that offers a formal model for business processes based on declarative modeling primitives. We have also build a prototype workflow management engine and tools for DCR Graphs to show that the ideas and concepts developed in the thesis can be easily implemented by a commercial workflow management system.

- The ideas and concepts developed in the thesis will provide a framework for suitable extensions to the Resultmaker declarative workflow model, which could be implemented in the later versions of their product. Furthermore, another industrial partner Exformatics [Exformatics 2009] has already implemented the core primitives of DCR Graphs into their commercial Enterprise Content and Case Management system and they also do have plans to implement our distribution technique developed in the thesis into their commercial tools. It shows that our formal model is both practicable and easily adoptable into commercial workflow management systems.

7.3 Future Work

Besides the need of further extensions to the formal model, this dissertation leaves many open challenges and issues for future research. We will briefly mention some of them in the following section. We will categorize the future work into 2 parts: the first part describes the future work related to extensions to the formal model and the second part describes about relating our work to the other formal models.

7.3.1 Extensions to Formal Model

In this section, we will discuss some of the extensions that we want to add to the formal model to make it more useful and applicable to many practical problems.

7.3.1.1 Time and Exceptions

Temporal constraints are most important to model processes from the real world. As part of the future work, we are planning to add time deadlines to the constraints, for example it will be possible to specify that a response constraint must happen within the specified time interval. Adding temporal constraints will naturally lead us to violations of such constraints. Hence we also need some type exception handling in DCR Graphs coupled with some kind of compensation, that can be applied when an exception happened. We will briefly explain here the motivation and the kind of support for temporal constraints we want to add to the DCR Graphs.

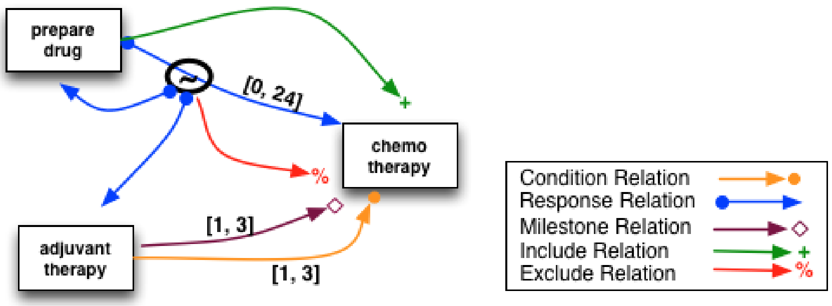


Figure 7.1: Oncology treatment process with temporal constraints

Let us take a small example from Oncology workflow modeled in DCR Graphs as shown in the figure 7.1. In the above example, we have two rules, the first one saying that a drug prepared in the pharmacy must be administered in chemotherapy with in 24 hours. The second rule says that, *adjuvant therapy* must have been performed one to three hours before *chemotherapy*.

The first rule has been modeled with a response constraint with a time interval of *[0-24]* between *prepare drug* and *chemotherapy* events as shown in the figure 7.1. The second rule is modeled with a condition and a milestone relation, both with a time interval of *[1-3]* between *adjuvant therapy* and *chemotherapy*. Further, we have also added an exception handler (shown as tilde sign within a circle) on the response constraint, having an exclude relation to *chemotherapy* event and a response relation to *prepare drug* and *adjuvant therapy* events, as a compensation for the constraint violation.

Naturally the response constraint will be violated if the *chemotherapy* is not performed within 24 hours after the drug is prepared. In such a case the exception handler will be invoked, and it will exclude the *chemotherapy* event and create a pending response on *prepare drug* and *adjuvant therapy*. Until unless the *prepare drug* event is re-executed, the event *chemotherapy* will not be included in the graph and at the same time the pending responses on *prepare drug* and *adjuvant therapy* will make the graph non-accepting. When the *prepare drug* event is re-executed, the *chemotherapy* will be included, but it will be blocked until unless *adjuvant therapy* is re-executed because of the milestone relation. Furthermore, the event *chemotherapy* can only be executed within 1 to 3 hours after executing *adjuvant therapy*, otherwise the *chemotherapy* will be blocked, on the other hand if the *chemotherapy* is not done within 24 hours, the response constraint will be again violated.

One could add support for temporal constraints to DCR Graphs in the similar lines as explained above. Further execution of an event is considered as instantaneous in DCR Graphs, but activities in real world have durations. So one could also consider adding duration to events in DCR Graphs along with the temporal constraints.

7.3.1.2 Programming Language for DCR Graphs with Parametrized Events with data

As part of the extensions to DCR Graphs, another PhD student from our group is working on defining and implementing a new declarative and purely event-based language based on the DCR Graphs, tentatively named *DECoReS*, with a motivation of applying the DCR Graphs to Event-based Context-sensitive Reactive Services. To support the formal semantics of DECoReS, we propose extending the DCR Graphs with *parametrized* events, *automatic* events, time and exception handling.

To provide a brief intuition about the languages and extensions, we exemplify the healthcare process as illustrated by the prescribe medicine healthcare process adapted from [Lyng *et al.* 2008, Hildebrandt *et al.* 2011a]. The process consists of five events: The prescription of medicine and signing of the prescription by a doctor (represented by the events `prescribe` and `sign` respectively), a nurse giving the medicine to the patient (represented by the event `give`, and the nurse indicating that he does not trust the prescription (represented by the event `distrust`) and the doctor removing the prescription, represented by the event `remove`.

```
treatment process{
  doctor may prescribe<$id, $med, $qty> {
    response: administer<$id,$med,$qty>
  }
  administer<$id, $med, $qty> process
  {
    doctor must sign { exclude: remove }
    nurse must give {
      condition: Executed(sign) &
                 not Executed(remove) &
                 not Response(sign)
      exclude: sign, give, distrust, remove
    }
    nurse may distrust {
      response: sign
      include: remove
      exclude: give
    }
```

```
     doctor may remove {
       exclude: sign,give,distrust,remove
     }
   }
 }
```

To capture that every prescription event **prescribe** leads to the possible execution of a "fresh" set of events **sign**, **give**, **distrust** and **remove**, the event **prescribe** will instantiate **administer** sub process to create a fresh or new set of all the four events. Observe that now the events are parameterized with data, which means executing **prescribe** with a particular set of data values, will be create a fresh instance of subprocess events and pass the data values to the newly created instances. Adding parameterized data to events will also bring some changes to semantics of the relations, to enforce constraints between the events with matching data values. Implementation of programming language will also lead to some new extensions to the DCR Graphs.

7.3.1.3 Distribution of DCR Graphs

In the chapter 5, We have given a general technique for distributing a declarative (global) process as a network of synchronously communicating (local) declarative processes and proven that the global and distributed execution to be equivalent. But distributing a global process as a set of asynchronously communicating distributed processes will be a much harder problem to study. As part of future work one may study about distribution technique for the DCR Graphs based on *asynchronous communication* among the distributed processes using buffered queues. This may benefit from researching the true concurrency semantics inherent in DCR Graphs and extend the transition system semantics to include concurrency, e.g. like in [Mukund & Nielsen 1992, Hildebrandt & Sassone 1996]. Further We also planning to study behavioral types describing the interfaces between communicating DCR Graphs, extending the work on session types in [Carbone *et al.* 2007] to a declarative setting. Another PhD student has started working in this direction to apply theory of session types and adapted them from the current imperative models to the declarative DCR Graphs model and thereby to provide a foundation for statically checked communication protocols for a distributed workflow.

As of now, the distribution technique is applicable to basic DCR Graphs and nested DCR Graphs only. One may also extend the distribution technique to the DCR Graphs with present and forthcoming extensions such as sub processes, data, time and exceptions.

7.3.1.4 Dynamic Changes and Adaptive DCR Graphs

The approach for flexibility so far adopted in DCR Graphs can be categorized as flexibility by selection [Heinl *et al.* 1999] or design time flexibility [van der Aalst *et al.* 2009]. In some application scenarios, the flexibility by selection or the design time flexibility may not be sufficient and the processes may have to deal with un expected execution paths.

Dynamic changes are not supported in the current formal model for DCR Graphs, as we have a static constraint set and events, but one could easily extend the formalism to support the dynamic changes. The DCR Graphs allows for an intuitive operational semantics and effective execution expressed by a notion of markings of the graphs, which will be updated after execution of every event. Therefore if new constraints are added at runtime and they will be evaluated in the next execution of an event. Of course, it will lead to certain challenges such as how to deal with conflicts and how to adapt the running instances etc. One may also add new events to the events set at runtime in lines similar to the semantics of sub-processes in DCR Graphs, where a fresh set of subprocess events will be generated and added to events set when a sub-process is instantiated.

7.3.1.5 Formal verification

As of now, the work on formal verification for DCR Graphs is only limited to core model of DCR Graphs. This could be extended to all the present and forthcoming extensions of DCR Graphs, such as nested sub graphs, sub processes, data, time and exceptions. Extending formal verification on DCR Graphs with extensions would be quite challenging. For example adding data domains to the DCR Graphs will cause the state space exploded problem and verification will become quite complex. Probably one could explore the approach used by authors in [Deutsch *et al.* 2009] for verification of data-centric business processes. Further, for formal verification of DCR Graphs with temporal constraints, one could consider using Uppaal [Uppaal-Group 2009] which is model checker for modeling, simulation and verification of real-time system, based on timed automata.

The work on formal verification can be extended to the technique developed in chapter 5 to distribute a DCR Graph as set of local components based on the notion of projection. To verify the distributed DCR Graphs formally, one could consider using SPIN [Holzmann 2004] model checker as it is a well known system for verification of asynchronous process models and distributed systems, with suitable constructs like messages and buffered channels. Moreover, it could be interesting to do formal verification on real world examples taken from the case studies and compare different approaches for verification of declarative processes.

7.3.2 Relating to the other formal models

As part of furture work, one could relate DCR Graphs to various formal models which follow related approaches to model processes and workflows. In this section we will briefly describe some of the models, which follow similar or related approach to our formal model DCR Graphs.

7.3.2.1 Guard-Stage-Milestone Lifecycle model

As part of future work, we want to relate our formal model to IBM Research's declarative process model Business Artifacts with Guard-Stage-Milestone life cycles [Damaggio *et al.* 2011, Hull *et al.* 2011a, Hull *et al.* 2011b]. Business artifacts combine the data aspects and process aspects in a holistic manner and an artifact type contains both an information model and lifecycle model, where information model manages the data for business objects and lifecycle model describes the possible ways the tasks on business objects can be invoked.

As part of the business artifacts, a declarative approach has been taken in the recent years for specifying the life cycles of business entities, using the *Guard- Stage-Milestone* (GSM model) life cycles model. The GSM model is a declarative process model for specification of interactions between business entities and its operational semantics are based on rules similar to ECA(Event Condition Action)-like rules from Active database community.

Our formal model is quite related to the declarative primitives of GSM model and hence as part of PhD stay abroad, the PhD candidate visited IBM Research, New York , to study the relation between DCR Graphs and GSM model.

In this section we will explain briefly our ideas about relating DCR Graphs to GSM model. First, we will explain key primitives of GSM model briefly and then we will describe a method how one can encode some of the primitives of GSM model into the DCR Graphs.

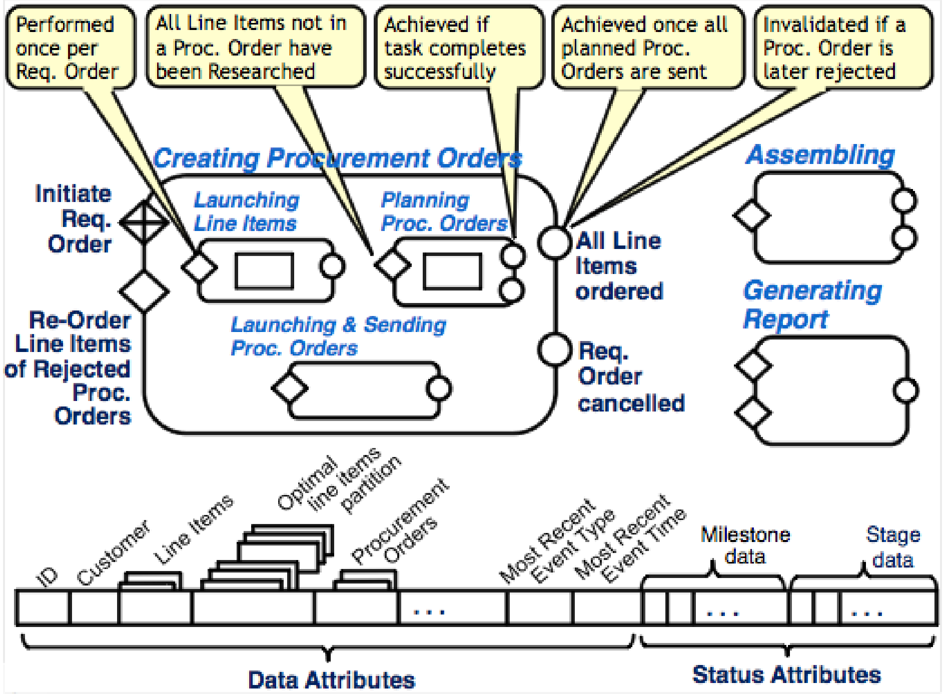

Performed once per Req. Order
All Line Items not in a Proc. Order have been Researched
Achieved if task completes successfully
Achieved once all planned Proc. Orders are sent
Invalidated if a Proc. Order is later rejected
Creating Procurement Orders
Initiate Req. Order
Launching Line Items
Planning Proc. Orders
All Line Items ordered
Assembling
Re-Order Line Items of Rejected Proc. Orders
Launching & Sending Proc. Orders
Req. Order cancelled
Generating Report
ID
Customer
Line Items
Optimal line items partition
Procurement Orders
Most Recent Event Type
Most Recent Event Time
Milestone data
Stage data
...
Data Attributes
Status Attributes

Figure 7.2: Requisition Order in GSM model [Hull *et al.* 2011b]

7.3.2.1.1 GSM Primitives

The full details of GSM model are explained in [Hull *et al.* 2011b] and we will briefly mention the key primitives of GSM model as follows.

1. Information Model: Captures all business relevant data about entities.

2. Milestones: Correspond to business-relevant operational objectives. They can be achieved/invalidated by either triggering events or using conditions over the data attributes or using both. Milestones are owned by Stages.

3. Stages: Correspond to group of activities with hierarchy. The stages own milestones and by executing the activities inside stages (and child stages), the milestone owned by the stage can be achieved. A stage becomes inactive when it's milestone is achieved, and if there are any child stages within a stage whose milestone is achieved, all it's child stages also become inactive. Similarly, if a stage is opened or become active, then all its milestones is invalidated.

4. Guards: They control when a stage becomes active and each guard is associated with a sentry. When a sentry become true, then the stage will be opened.

5. Sentry: A sentry consists of triggering an event type or a data condition or even both. Sentries are some kind of Boolean expressions and they are used as conditions for guards, milestones.

Figure 7.2 shows key components of GSM model, where a process for Requisition Order is modeled. The information model contains both data attributes for business relevant data and status attributes for data of process elements. The rounded rectangles are the stages and the guards are marked as diamonds on the stages. Milestones are marked with circles on the stages. Stages can contain child stages or activities.

Further, GSM model has the notion of GSM Business steps (or B-steps), which focus on updates to a snapshot (i.e., description of current state of a GSM system at a given point of time) when a single incoming event is to be incorporated into it. Basically a B-step primarily concentrates on how the GSM system should react to an incoming event and the focus will be on what stages are opened/closed, and what milestones are achieved/ invalidated.

Moreover, the operational semantics of GSM model are given by Prerequisite-Antecedent-Consequent (PAC) Rules, which specify how to make an update to snapshot in a single B-step. PAC rules impose certain restrictions to avoid inconsistencies or anomaly in the GSM system such as a stage can not opened and closed in a single B-step. Finally GSM model has the notion of events and messages to interact with the external environment.

7.3.2.1.2 Encoding a GSM model into DCR Graphs

In this section, we briefly explain a tentative way of encoding a GSM model into DCR Graphs. The GSM model is data-centric and hence it would appropriate to use

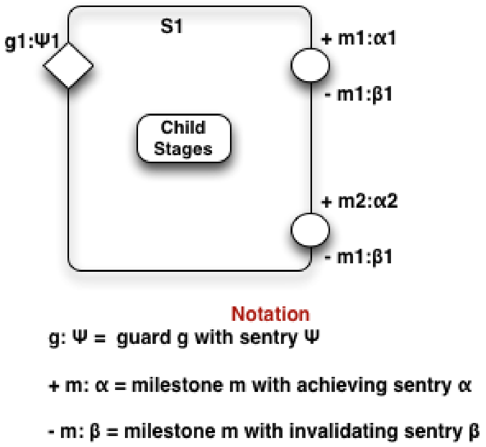

S1
g1:Ψ1
+ m1:α1
- m1:β1
Child Stages
+ m2:α2
- m1:β1
Notation
g: Ψ = guard g with sentry Ψ
+ m: α = milestone m with achieving sentry α
- m: β = milestone m with invalidating sentry β

Figure 7.3: A sample GSM model

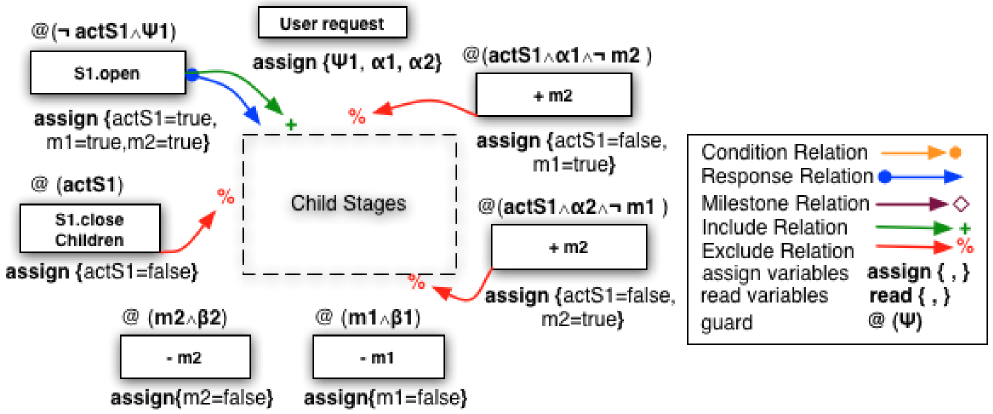


Figure 7.4: DCR Graph for sample GSM model

DCR Graphs with data extension as explained in the Sec. 4.3. The tasks and stages in GSM model can be encoded as the events and nested sub graphs in DCR Graphs. The status variables and sentries in GSM model can be encoded as variables and Boolean expressions.

Figure 7.3 shows a sample GSM model containing one stage (S1) with a guard and two milestones with sentries associated to guards and milestones as shown in the figure. There can be sub stages or tasks as child elements for a stage, but we abstract away from the child elements as they can be encoded in the similar fashion as the stage S1. Further opening and closing of the stages can be modeled by using include/exclude relations in DCR Graphs.

The sample GSM model encoded into DCR Graphs is shown in the figure 7.4.

The status attributes for stages and milestones are encoded explicitly as variables (for e.g. actS1, m1). The opening of a stage, achieving/invalidating of a milestone are modeled as explicit events in the DCR Graph. The sentries associated with guards and milestones in GSM model are modeled as guards (Boolean expression condition) for the events.

7.3.2.1.3 Challenges of Encoding GSM model into DCR Graphs

Even though the GSM model and DCR Graphs have a lot of similarities in declarative aspects, but they differ in some subtle aspects, which will make encoding challenging and we will mention some of the issues briefly here. First of all user request is explicitly modeled in GSM model as some kind of input events to the system, where as in DCR Graphs the user request is quite implicit. Moreover, the execution of events in DCR Graphs are based on the user's choice thereby user driven, where as the execution semantics of GSM model are more like automatic updates to the system state, which be difficult to model. Of course one could add a notion of auto events to DCR Graphs, which can model the behavior of automatic updates to some extent, but in that one must make sure that the semantics of PAC rules (e.g toggle-once principle) are correctly observed. Further explicit interaction with the environment in GSM model is also difficult to model in the DCR Graphs.

However, as part of the future work in the coming months, we are planning to study the relation between DCR Graphs and GSM model formally in the similar lines mention above, which could results in some more extensions to our formal model.

7.3.2.2 Declare model

Declare is a framework for flexible workflows using declarative modeling primitives based on the constraints defined in LTL and it was briefly introduced in background chapter (Sec. 2.2) as one of the motivating formalisms for our model. As part of the future work, we want to relate our formal model with the Declare's declarative language ConDec [van der Aalst & Pesic 2006a] by encoding it's LTL based constraints into DCR Graphs and also encoding the relations of DCR Graphs into ConDec.

Our approach is closely related to the work on ConDec [van der Aalst *et al.* 2009, van der Aalst & Pesic 2006a]. The crucial difference is that we allow nesting and a few core constraints making it possible to describe the state of a process as a simple marking. ConDec does not address dynamic inclusion/exclusion, but allows one to specify any relation expressible within Linear-time Temporal Logic (LTL). This offers much flexibility with respect to specifying execution constraints. In particular the condition and response relations in our model are same as precedence and response constraints in ConDec [van der Aalst & Pesic 2006a] and hence we have used the same graphical notation. Furthermore, we have encoded Büchi automaton in DCR Graphs using a straight forward construction in Sec. 3.5, which shows that DCR Graphs can express all ω-regular languages and thereby more expressive than LTL.

As part of the future work, we are planning to translate constraints from ConDec as a direct encoding into DCR Graphs. Even though most of the constraints from

ConDec can be directly encoded in DCR Graphs using the five core relations, but we think some of constraints from ConDec could be difficult to encode directly. For example, in ConDec the constraint *disjunctive response* says that only one of the choices are to be executed as a response, which can not be expressed directly in DCR Graphs. Therefore we are expecting few extensions to DCR Graphs as part of the work on relating our model to ConDec. On the other hand, expressing dynamic include / exclude relations from DCR Graphs in terms of LTL could be difficult in ConDec.

7.3.2.3 Refinement for Transition Systems

Modal transition systems (MTS) [Larsen & Thomsen 1988, Antonik *et al.* 2008] are basic transition system model supporting stepwise specification and refinement of parallel processes, which can be regarded as label transition systems with *required (must)* and *allowed(may)* transitions, with a consistency condition that all must transitions should be matched directly by a may transition. An over-approximation and an under-approximation of a process can be defined using a MTS simultaneously. A class of MTS without the consistency condition is known as Mixed Transition Systems [Dams *et al.* 1997], which places no restrictions on the relationship between may and must transitions.

Some of the researchers in our group are working on providing a new generalization of MTS as Transition Systems with Responses [Carbone *et al.* 2012] using the labeled transition system of DCR Graphs with response set to define a notion of refinement by taking prescribe medicine healthcare workflow as an example. Study of deadlock and liveness properties in DCR Graphs in relation to Transition Systems with Responses, adding notion of refinement could be part of the future work on DCR Graphs.

Appendix A

PROMELA Code for Verification of Properties

A.1 PROMELA Code for Deadlock Free Property

```
/*
DCRS Example: Givemedicine-nonstronglydedalockfree  PROMELA  language code for model checking in SPIN tool.
Generated on 2012-01-27T00:50:32 by DCRStoPROMELA Compiler.
Developed by Raghava Rao Mukkamala (rao@itu.dk)
*/

#define event_count 4
/* Declaration of events */
#define pm 0
#define s 1
#define gm 2
#define dt 3

/* Declarations of Marking */
bit included[event_count];
bit executed[event_count];
bit responses[event_count];
bit enabledset[event_count];

typedef twodimensionalarray {bit to[event_count]};
/* Declaration of relations */
twodimensionalarray condition_relation[event_count];
twodimensionalarray response_relation[event_count];
twodimensionalarray include_relation[event_count];
twodimensionalarray exclude_relation[event_count];
twodimensionalarray milestone_relation[event_count];

/* Looping Counters  */
byte index = 0;
byte index2 = 0;
short executed_event_count = 0;
bit accepted_marking = 1;
bit accepted_state_reached = 0;
bit can_execute = 1;
byte loopindex = 0;
/* Not possible to assign -1 to a
byte, so assign it event_count + 1 */
show byte random_event_executed = event_count + 1;
bit any_included_pending_responses = 0;

inline model_specification()
{  /* Specification of DCR Graph */
    /* Specification of Relations */
     include_relation[s].to[gm] = 1;
     include_relation[s].to[dt] = 1;

     exclude_relation[pm].to[pm] = 1;
     exclude_relation[gm].to[dt] = 1;
     exclude_relation[dt].to[gm] = 1;

     response_relation[pm].to[gm] = 1;
     response_relation[dt].to[s] = 1;

     condition_relation[s].to[pm] = 1;
     condition_relation[gm].to[pm] = 1;
     condition_relation[gm].to[s] = 1;
     condition_relation[dt].to[s] = 1;

     /* Specification of the initial state */
     /* Included Actions */
     included[pm] = 1;
     included[s] = 1;
     included[gm] = 1;
     included[dt] = 1;

     /* Pending Responses */
     responses[pm] = 1;
}

inline clear_enabled_events()
{
     index = 0;
     do
     :: index <  event_count -> enabledset[index] = 0 ; index = index + 1;
     :: else -> break;
     od;
}

inline compute_enabled_events()
{
    index = 0;
    /* Find out which events are enabled */
    do /* Loop for outer dimesion, to loop row ount */
    :: index <  event_count ->
    if
    :: included[index] == 1 ->
        index2 = 0;
        can_execute = 1;
        do /* inner loop for 2nd dimension */
        :: index2 <  event_count ->
        if
        :: condition_relation[index].to[index2] == 1  ->
            if
            :: included[index2] == 1 && executed[index2] != 1 ->
                    can_execute = 0;
            :: else ->skip;
            fi;
        :: else ->skip;
        fi;
        if
        :: milestone_relation[index].to[index2] == 1  ->
            if
            :: included[index2] == 1 && responses[index2] == 1 ->
                    can_execute = 0;
            :: else ->skip;
            fi;
        :: else ->skip;
        fi;
        index2 = index2 + 1;
    :: else ->  break;
        od;
        enabledset[index] = (can_execute -> 1 : 0);
    ::else -> skip;
    fi;
    index++;
    :: else ->  break;
    od;
}

inline nondeterministic_execution()
{
     any_included_pending_responses = 0;
     index = 0;
     do
         :: index <  event_count ->
             if
             :: (responses[index] == 1 ) && (included[index] == 1) ->
                    any_included_pending_responses = 1 ;
             :: else -> skip;
             fi;
         index = index + 1;
         :: else -> break;
     od;
     /* Non deterministic execution for deadlock free. */
     if
     :: (enabledset[pm] == 1) -> random_event_executed = pm;
     :: (enabledset[s] == 1) -> random_event_executed = s;
     :: (enabledset[gm] == 1) -> random_event_executed = gm;
     :: (enabledset[dt] == 1) -> random_event_executed = dt;
     :: else ->
         if
         :: (any_included_pending_responses) ->
    dead_lock_reached:  printf("Dead lock reached after %u executions!",
                    executed_event_count);
         assert(false);
         /* If we dont have any events enabled and
         no included pending responses, then we exit. */
         :: else -> goto deadlock_free_label;
         fi;
     fi;
}

inline compute_state_after_execution()
{
     /* Update executed actions set*/
     executed[random_event_executed] = 1 ;
     executed_event_count++;
     /* Delete entry from responses set if it is a response to some other action*/
     responses[random_event_executed] = 0 ;
     index = 0;
     do
     :: index <  event_count ->

         /* Include actions which are included by this action in set included */
         if
         :: include_relation[random_event_executed].to[index] == 1 -> included[index] = 1 ;
         :: else -> skip;
         fi;

         /* Exclude actions which are excluded by this action in set included */
         if
         :: exclude_relation[random_event_executed].to[index] == 1 -> included[index] = 0 ;
         :: else -> skip;
         fi;

         /* Include actions which are responses to this action in set responses */
         if
         :: response_relation[random_event_executed].to[index] == 1 -> responses[index] = 1 ;
         :: else -> skip;
         fi;

         index = index + 1;
     :: else -> break;
     od;
}

active proctype dcrs()
{
     /* Specification of DCR graph */
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which events are enabled */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
         /* Compute state after execution. */
         compute_state_after_execution();
     od;

    deadlock_free_label:
     printf("The given DCR graph is deadlock free");
}
```

A.2 PROMELA Code for Strongly Deadlock Free Property

```
/*
DCRS Example: Givemedicine-nonstronglydedalockfree  PROMELA  language code for model checking in SPIN tool.
Generated on 2012-01-28T18:01:59 by DCRStoPROMELA Compiler.
Developed by Raghava Rao Mukkamala (rao@itu.dk)
*/

#define event_count 4
/* Declaration of events */
#define pm 0
#define s 1
#define gm 2
#define dt 3

typedef twodimensionalarray {bit to[event_count]};
/* Declaration of relations */
twodimensionalarray condition_relation[event_count];
twodimensionalarray response_relation[event_count];
twodimensionalarray include_relation[event_count];
twodimensionalarray exclude_relation[event_count];
twodimensionalarray milestone_relation[event_count];
/* Declarations of Marking */
bit included[event_count];
bit executed[event_count];
bit responses[event_count];
bit enabledset[event_count];


/* Looping Counters  */
byte index = 0;
byte index2 = 0;
short executed_event_count = 0;
bit accepted_marking = 1;
bit accepted_state_reached = 0;
bit can_execute = 1;
byte loopindex = 0;
/* Not possible to assign -1 to a byte, so assign it event_count + 1 */
show byte random_event_executed = event_count + 1;
bit any_included_pending_responses = 0;


inline model_specification()
{
/* Specification of DCR Graph */
/* Relations */

     condition_relation[s].to[pm] = 1;
     condition_relation[gm].to[pm] = 1;
     condition_relation[gm].to[s] = 1;
     condition_relation[dt].to[s] = 1;


     response_relation[pm].to[gm] = 1;
     response_relation[dt].to[s] = 1;

     include_relation[s].to[gm] = 1;
     include_relation[s].to[dt] = 1;

     exclude_relation[pm].to[pm] = 1;
     exclude_relation[gm].to[dt] = 1;
     exclude_relation[dt].to[gm] = 1;

     /* Specification of the initial state */
     /* Included Actions */
     included[pm] = 1;
     included[s] = 1;
     included[gm] = 1;
     included[dt] = 1;

     /* Pending Responses */
     responses[pm] = 1;
}

inline clear_enabled_events()
{
     index = 0;
     do
     :: index <  event_count -> enabledset[index] = 0 ; index = index + 1;
     :: else -> break;
     od;
}

inline compute_enabled_events()
{
     index = 0;
     /* Find out which actions are enabled */
     do /* Loop for outer dimesion, to loop row ount */
     :: index <  event_count ->
         if
         :: included[index] == 1 ->
             index2 = 0;
             can_execute = 1;
             do /* inner loop for 2nd dimension */
             :: index2 <  event_count ->
                 if
                 :: condition_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && executed[index2] != 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 if
                 :: milestone_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && responses[index2] == 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 index2 = index2 + 1;
             :: else -> break;
             od;
             enabledset[index] = (can_execute -> 1 : 0);
         ::else -> skip;
         fi;
         index++;
     :: else -> break;
     od;
}

inline nondeterministic_execution()
{
     any_included_pending_responses = 0;
     index = 0;
     do
         :: index <  event_count ->
             if
             :: (responses[index] == 1 ) && (included[index] == 1) ->
                any_included_pending_responses = 1 ;
             :: else -> skip;
             fi;
         index = index + 1;
         :: else -> break;
     od;
     /* Non deterministic execution for strongly deadlock free. */
     if
     :: (enabledset[pm] == 1) && (responses[pm] == 1 ) -> random_event_executed = pm;
     :: (enabledset[s] == 1) && (responses[s] == 1 ) -> random_event_executed = s;
     :: (enabledset[gm] == 1) && (responses[gm] == 1 ) -> random_event_executed = gm;
     :: (enabledset[dt] == 1) && (responses[dt] == 1 ) -> random_event_executed = dt;
     :: else ->
         if
         :: (any_included_pending_responses) ->
      strongly_dead_lock_reached:printf("Strongly dead lock reached after %u executions!",
                           executed_event_count);
         assert(0);
    /* If we dont have any enabled events and no included pending responses, then we exit. */
         :: else -> goto strongly_deadlock_free_label;
         fi;
     fi;
}

inline compute_state_after_execution()
{
     /* Update executed actions set*/
     executed[random_event_executed] = 1 ;
     executed_event_count++;
     /* Delete entry from responses set if it is a response to some other action*/
     responses[random_event_executed] = 0 ;
     index = 0;
     do
     :: index <  event_count ->

         /* Include actions which are included by this action in set included */
         if
         :: include_relation[random_event_executed].to[index] == 1 -> included[index] = 1 ;
         :: else -> skip;
         fi;

         /* Exclude actions which are excluded by this action in set included */
         if
         :: exclude_relation[random_event_executed].to[index] == 1 -> included[index] = 0 ;
         :: else -> skip;
         fi;

         /* Include actions which are responses to this action in set responses */
         if
         :: response_relation[random_event_executed].to[index] == 1 -> responses[index] = 1 ;
         :: else -> skip;
         fi;

         index = index + 1;
     :: else -> break;
     od;
}

active proctype dcrs()
{
     /* Call  model_specification() to assign necessary constraints*/
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which ations are enabled based on latest execution set */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
```

```
        /* Compute state after execution. */
        compute_state_after_execution();
    od;
    strongly_deadlock_free_label: printf("The given DCR graph is strongly deadlock free");
}
```

A.3 PROMELA Code for Liveness Property

```
/*
DCRS Example: Givemedicine  PROMELA  language code for model checking in SPIN tool.
Generated on 2012-01-30T02:22:24 by DCRStoPROMELA Compiler.
Developed by Raghava Rao Mukkamala (rao@itu.dk)
*/

#define event_count 4
/* Declaration of events */
#define pm 0
#define s 1
#define gm 2
#define dt 3

typedef twodimensionalarray {bit to[event_count]};
/* Declaration of relations */
twodimensionalarray condition_relation[event_count];
twodimensionalarray response_relation[event_count];
twodimensionalarray include_relation[event_count];
twodimensionalarray exclude_relation[event_count];
twodimensionalarray milestone_relation[event_count];
/* Declarations of Marking */
bit included[event_count];
bit executed[event_count];
bit responses[event_count];
bit enabledset[event_count];


/* Looping Counters  */
byte index = 0;
byte index2 = 0;
short executed_event_count = 0;
bit accepted_marking = 1;
bit accepted_state_reached = 0;
bit can_execute = 1;
byte loopindex = 0;
/* Not possible to assign -1 to a byte, so assign it event_count + 1 */
show byte random_event_executed = event_count + 1;
bit any_included_pending_responses = 0;


/* New Variables for acceptance over infinite runs. */
byte state_index = 0;
bit include_response_current[event_count];
bit included_actions_nextstate[event_count];
bit pending_responses_nextstate[event_count];
bit include_response_nextstate[event_count];
bit acceptable_responses_set[event_count];
bit m_set[event_count];
byte min_include_response_current;
byte min_m_set;
byte m_set_count = 0;
byte include_response_current_set_count = 0;
byte include_response_nextstate_set_count = 0;
bit accepting_state_visited = 0;


inline model_specification()
{
/* Specification of DCR Graph */
/* Relations */

     condition_relation[s].to[pm] = 1;
     condition_relation[gm].to[pm] = 1;
     condition_relation[gm].to[s] = 1;
     condition_relation[dt].to[s] = 1;
     response_relation[pm].to[gm] = 1;
     response_relation[dt].to[s] = 1;
     response_relation[pm].to[s] = 1;

     include_relation[s].to[gm] = 1;
     include_relation[s].to[dt] = 1;

     exclude_relation[pm].to[pm] = 1;
     exclude_relation[gm].to[dt] = 1;
     exclude_relation[dt].to[gm] = 1;

     /* Specification of the initial state */
     /* Included Actions */
     included[pm] = 1;
     included[s] = 1;
     included[gm] = 1;
     included[dt] = 1;

     /* Pending Responses */
     responses[pm] = 1;
}

inline clear_enabled_events()
{
     index = 0;
     do
     :: index <  event_count -> enabledset[index] = 0 ; index = index + 1;
     :: else -> break;
     od;
}

inline compute_enabled_events()
{
     index = 0;
     /* Find out which actions are enabled */
     do /* Loop for outer dimesion, to loop row ount */
     :: index <  event_count ->
         if
         :: included[index] == 1 ->
             index2 = 0;
             can_execute = 1;
             do /* inner loop for 2nd dimension */
             :: index2 <  event_count ->
                 if
                 :: condition_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && executed[index2] != 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 if
                 :: milestone_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && responses[index2] == 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 index2 = index2 + 1;
             :: else -> break;
             od;
             enabledset[index] = (can_execute -> 1 : 0);
         ::else -> skip;
         fi;
         index++;
     :: else -> break;
     od;
}

inline nondeterministic_execution()
{
     any_included_pending_responses = 0;
     index = 0;
     do
         :: index <  event_count ->
             if
             :: (responses[index] == 1 ) && (included[index] == 1) -> any_included_pending_responses = 1 ;
             :: else -> skip;
             fi;
         index = index + 1;
         :: else -> break;
     od;
     /* Non deterministic execution for strongly Liveness. */
     if
     :: (enabledset[pm] == 1)  -> random_event_executed = pm;
     :: (enabledset[s] == 1)  -> random_event_executed = s;
     :: (enabledset[gm] == 1)  -> random_event_executed = gm;
     :: (enabledset[dt] == 1)  -> random_event_executed = dt;
     :: else ->
         if
         :: (any_included_pending_responses) ->
         strongly_dead_lock_reached:    printf("Dead lock reached after %u executions!",          ↙
    executed_event_count);
         assert(false);
         /* If we dont have any actions enabled and no included pending responses, then we exit. */
         :: else -> goto end_state;
         fi;
     fi;
}

inline compute_include_response_sets()
{
     index = 0;
     do
     :: index <  event_count ->
         /* Update for include_response_current set. */
         include_response_current[index] = ( (included[index] && responses[index]) -> 1: 0 );
         /* Calculation of next state set */
         /* Updating the included_actions_nextstate set */
         if
         :: include_relation[random_event_executed].to[index] -> included_actions_nextstate[index] = 1 ;
         :: exclude_relation[random_event_executed].to[index] -> included_actions_nextstate[index] = 0 ;
         :: else -> included_actions_nextstate[index] = included[index];
         fi;
         /* Updating the pending_responses_nextstate set */
         /* Clear the  pending response for random_event_executed unless it is not included by itself */
         if
         :: response_relation[random_event_executed].to[index] -> pending_responses_nextstate[index] = 1 ;
         :: else -> pending_responses_nextstate[index] = ((random_event_executed == index) -> 0: responses ↙
    [index]);
         fi;
         /* Updating the include_response_nextstate set */
         include_response_nextstate[index] = ( (included_actions_nextstate[index] &&              ↙
    pending_responses_nextstate[index]) -> 1: 0 );
         /* Compute the acceptable_responses_set (I and R \ (I' and R') U (e)) */
         acceptable_responses_set[index] = ( include_response_current[index]  && (!               ↙
    include_response_nextstate[index]) -> 1:0 );
         m_set[index] = ((include_response_current[index] && (index > state_index)) -> 1: 0);
         index = index + 1;
     :: else -> break;
     od;
     /* Add the current random action executed to the acceptable_responses_set to get (I and R \ (I' and R' ↙
    ) U (e))*/
     acceptable_responses_set[random_event_executed] = 1;
}

inline compute_set_minimum()
{
     /* Initially set the min_m_set to highest number as default as 0 is also used as action index */
     min_m_set = event_count;
     min_include_response_current = event_count;
     /* Assign the index to event_count, as we will loop through the array in reverse order to find out min ↙
    . */
     index = event_count;
     m_set_count = 0;
     include_response_current_set_count = 0;
     include_response_nextstate_set_count = 0;
     do
     :: index > 0 ->
         /* min for m_set */
         if
         :: m_set[index -1] -> min_m_set = (index -1);
             m_set_count++;
         :: else -> skip;
         fi;
         /* min for include_response_current set */
         if
         :: include_response_current[index -1] -> min_include_response_current = (index -1);
             include_response_current_set_count++;
         :: else -> skip;
         fi;
         /* Find out how many elements are in the include_response_nextstate set */
         include_response_nextstate_set_count = (include_response_nextstate[index -1] ->                    ↙
    include_response_nextstate_set_count + 1 : include_response_nextstate_set_count);
         index--;
     :: else -> break;
     od;
}

inline check_state_acceptance_condition()
{
     if
         /* If no pending responses in the next set. */
         :: (include_response_nextstate_set_count == 0) ->
         progress_state_0: accepting_state_visited = 1;
         :: ((m_set_count > 0) && (acceptable_responses_set[min_m_set])) ->
         progress_state_1: accepting_state_visited = 1; state_index = min_m_set ;
         :: ((m_set_count == 0) && (min_include_response_current < event_count) &&                          ↙
    (acceptable_responses_set[min_include_response_current])) ->
         progress_state_2: accepting_state_visited = 1; state_index = min_include_response_current ;
         /* Otherwise dont change the state index. */
         :: else -> accepting_state_visited = 0;
     fi;
}

inline compute_state_after_execution()
{
     /* Update executed actions set*/
     executed[random_event_executed] = 1 ;
     executed_event_count++;
     /* Delete entry from responses set if it is a response to some other action*/
     responses[random_event_executed] = 0 ;
     index = 0;
     do
     :: index <  event_count ->
```

```
         /* Include actions which are included by this action in set included */
         if
         :: include_relation[random_event_executed].to[index] == 1 -> included[index] = 1 ;
         :: else -> skip;
         fi;

         /* Exclude actions which are excluded by this action in set included */
         if
         :: exclude_relation[random_event_executed].to[index] == 1 -> included[index] = 0 ;
         :: else -> skip;
         fi;

         /* Include actions which are responses to this action in set responses */
         if
         :: response_relation[random_event_executed].to[index] == 1 -> responses[index] = 1 ;
         :: else -> skip;
         fi;

         index = index + 1;
     :: else -> break;
     od;
}

active proctype dcrs()
{
     /* Call  model_specification() to assign necessary constraints*/
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which ations are enabled based on latest execution set */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
         /* Compute include response sets and m-set etc */
         compute_include_response_sets();
         /* Compute minimum values for include response sets and m-set etc */
         compute_set_minimum();
         /* Compute state accepting conditions  */
         check_state_acceptance_condition();
         /* Compute state after execution. */
         compute_state_after_execution();
     od;
         end_state: printf("End state reached after %u", executed_event_count);
}
```

A.4 PROMELA Code for Strongly Liveness Property

```
/*
DCRS Example: Givemedicine  PROMELA  language code for model checking in SPIN tool.
Generated on 2012-01-30T02:22:24 by DCRStoPROMELA Compiler.
Developed by Raghava Rao Mukkamala (rao@itu.dk)
*/

#define event_count 4
/* Declaration of events */
#define pm 0
#define s 1
#define gm 2
#define dt 3

typedef twodimensionalarray {bit to[event_count]};
/* Declaration of relations */
twodimensionalarray condition_relation[event_count];
twodimensionalarray response_relation[event_count];
twodimensionalarray include_relation[event_count];
twodimensionalarray exclude_relation[event_count];
twodimensionalarray milestone_relation[event_count];
/* Declarations of Marking */
bit included[event_count];
bit executed[event_count];
bit responses[event_count];
bit enabledset[event_count];


/* Looping Counters  */
byte index = 0;
byte index2 = 0;
short executed_event_count = 0;
bit accepted_marking = 1;
bit accepted_state_reached = 0;
bit can_execute = 1;
byte loopindex = 0;
/* Not possible to assign -1 to a byte, so assign it event_count + 1 */
show byte random_event_executed = event_count + 1;
bit any_included_pending_responses = 0;


/* New Variables for acceptance over infinite runs. */
byte state_index = 0;
bit include_response_current[event_count];
bit included_actions_nextstate[event_count];
bit pending_responses_nextstate[event_count];
bit include_response_nextstate[event_count];
bit acceptable_responses_set[event_count];
bit m_set[event_count];
byte min_include_response_current;
byte min_m_set;
byte m_set_count = 0;
byte include_response_current_set_count = 0;
byte include_response_nextstate_set_count = 0;
bit accepting_state_visited = 0;


inline model_specification()
{
/* Specification of DCR Graph */
/* Relations */

     condition_relation[s].to[pm] = 1;
     condition_relation[gm].to[pm] = 1;
     condition_relation[gm].to[s] = 1;
     condition_relation[dt].to[s] = 1;
     response_relation[pm].to[gm] = 1;
     response_relation[dt].to[s] = 1;
     response_relation[pm].to[s] = 1;

     include_relation[s].to[gm] = 1;
     include_relation[s].to[dt] = 1;

     exclude_relation[pm].to[pm] = 1;
     exclude_relation[gm].to[dt] = 1;
     exclude_relation[dt].to[gm] = 1;

     /* Specification of the initial state */
     /* Included Actions */
     included[pm] = 1;
     included[s] = 1;
     included[gm] = 1;
     included[dt] = 1;

     /* Pending Responses */
     responses[pm] = 1;
}

inline clear_enabled_events()
{
     index = 0;
     do
     :: index <  event_count -> enabledset[index] = 0 ; index = index + 1;
     :: else -> break;
     od;
}

inline compute_enabled_events()
{
     index = 0;
     /* Find out which actions are enabled */
     do /* Loop for outer dimesion, to loop row ount */
     :: index <  event_count ->
         if
         :: included[index] == 1 ->
             index2 = 0;
             can_execute = 1;
             do /* inner loop for 2nd dimension */
             :: index2 <  event_count ->
                 if
                 :: condition_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && executed[index2] != 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 if
                 :: milestone_relation[index].to[index2] == 1  ->
                     if
                     :: included[index2] == 1 && responses[index2] == 1 -> can_execute = 0;
                     :: else ->skip;
                     fi;
                 :: else ->skip;
                 fi;
                 index2 = index2 + 1;
             :: else -> break;
             od;
             enabledset[index] = (can_execute -> 1 : 0);
         ::else -> skip;
         fi;
         index++;
     :: else -> break;
     od;
}

inline nondeterministic_execution()
{
     any_included_pending_responses = 0;
     index = 0;
     do
         :: index <  event_count ->
             if
             :: (responses[index] == 1 )
             && (included[index] == 1) ->
             any_included_pending_responses = 1 ;
             :: else -> skip;
             fi;
         index = index + 1;
         :: else -> break;
     od;
     /* Non deterministic execution for strongly Liveness. */
     if
     :: (enabledset[pm] == 1) && (responses[pm] == 1 ) ->
        random_event_executed = pm;
     :: (enabledset[s] == 1) && (responses[s] == 1 ) ->
        random_event_executed = s;
     :: (enabledset[gm] == 1) && (responses[gm] == 1 ) ->
         random_event_executed = gm;
     :: (enabledset[dt] == 1) && (responses[dt] == 1 ) ->
         random_event_executed = dt;
     :: else ->
         if
         :: (any_included_pending_responses) ->
    strongly_dead_lock_reached:
     printf("Dead lock reached after %u executions!",
         executed_event_count);
         assert(false);
         /* If we dont have any actions enabled and
         no included pending responses, then we exit. */
         :: else -> goto end_state;
         fi;
     fi;
}

inline compute_include_response_sets()
{
    index = 0;
    do
    :: index <  event_count ->
        /* Update for include_response_current set. */
        include_response_current[index] =
        ( (included[index] && responses[index]) -> 1: 0 );
        /* Calculation of next state set */
        /* Updating the included_actions_nextstate set */
        if
        :: include_relation[random_event_executed].to[index] ->
        included_actions_nextstate[index] = 1 ;
        :: exclude_relation[random_event_executed].to[index] ->
            included_actions_nextstate[index] = 0 ;
        :: else -> included_actions_nextstate[index] = included[index];
        fi;
        /* Updating the pending_responses_nextstate set */
        /* Clear the  pending response for random_event_executed
        unless it is not included by itself */
        if
        :: response_relation[random_event_executed].to[index] ->
        pending_responses_nextstate[index] = 1 ;
        :: else -> pending_responses_nextstate[index] =
            ((random_event_executed == index) -> 0: responses[index]);
        fi;
        /* Updating the include_response_nextstate set */
        include_response_nextstate[index] =
        ( (included_actions_nextstate[index] && pending_responses_nextstate[index]) -> 1: 0 );
        /* Compute the acceptable_responses_set (I and R \ (I' and R') U (e)) */
        acceptable_responses_set[index] =
        ( include_response_current[index]  && (!include_response_nextstate[index]) -> 1:0 );
        m_set[index] = ((include_response_current[index] && (index > state_index)) -> 1: 0);
        index = index + 1;
    :: else -> break;
    od;
    /* Add the current random action executed to the acceptable_responses_set to get (I and R \ (I' and R') ↙
     U (e))*/
    acceptable_responses_set[random_event_executed] = 1;
}

inline compute_set_minimum()
{
     /* Initially set the min_m_set to highest number as default as 0 is also used as action index */
     min_m_set = event_count;
     min_include_response_current = event_count;
     /* Assign the index to event_count, as we will loop through the array in reverse order to find out min ↙
    . */
     index = event_count;
     m_set_count = 0;
     include_response_current_set_count = 0;
     include_response_nextstate_set_count = 0;
     do
     :: index > 0 ->
         /* min for m_set */
         if
         :: m_set[index -1] -> min_m_set = (index -1);
             m_set_count++;
         :: else -> skip;
         fi;
         /* min for include_response_current set */
         if
         :: include_response_current[index -1] -> min_include_response_current = (index -1);
             include_response_current_set_count++;
         :: else -> skip;
         fi;
         /* Find out how many elements are in the include_response_nextstate set */
         include_response_nextstate_set_count = (include_response_nextstate[index -1] ->              ↙
    include_response_nextstate_set_count + 1 : include_response_nextstate_set_count);
         index--;
     :: else -> break;
     od;
}

inline check_state_acceptance_condition()
{
     if
         /* If no pending responses in the next set. */
         :: (include_response_nextstate_set_count == 0) ->
         progress_state_0: accepting_state_visited = 1;
         :: ((m_set_count > 0) && (acceptable_responses_set[min_m_set])) ->
         progress_state_1: accepting_state_visited = 1; state_index = min_m_set ;
         :: ((m_set_count == 0) && (min_include_response_current < event_count) &&                   ↙
    (acceptable_responses_set[min_include_response_current])) ->
         progress_state_2: accepting_state_visited = 1; state_index = min_include_response_current ;
         /* Otherwise dont change the state index. */
         :: else -> accepting_state_visited = 0;
     fi;
```

```
}

inline compute_state_after_execution()
{
     /* Update executed actions set*/
     executed[random_event_executed] = 1 ;
     executed_event_count++;
     /* Delete entry from responses set if it is a response to some other action*/
     responses[random_event_executed] = 0 ;
     index = 0;
     do
     :: index <  event_count ->

         /* Include actions which are included by this action in set included */
         if
         :: include_relation[random_event_executed].to[index] == 1 -> included[index] = 1 ;
         :: else -> skip;
         fi;

         /* Exclude actions which are excluded by this action in set included */
         if
         :: exclude_relation[random_event_executed].to[index] == 1 -> included[index] = 0 ;
         :: else -> skip;
         fi;

         /* Include actions which are responses to this action in set responses */
         if
         :: response_relation[random_event_executed].to[index] == 1 -> responses[index] = 1 ;
         :: else -> skip;
         fi;

         index = index + 1;
     :: else -> break;
     od;
}

active proctype dcrs()
{
     /* Call  model_specification()*/
     model_specification();
     do
     ::
         /* Clearing away enabled set */
         clear_enabled_events();
         /* Compute which ations are enabled based
         on latest execution set */
         compute_enabled_events();
         /* Execute an action non-nondeterministically */
         nondeterministic_execution();
         /* Compute include response sets and m-set etc */
         compute_include_response_sets();
         /* Compute minimum values for include
         response sets and m-set etc */
         compute_set_minimum();
         /* Compute state accepting conditions  */
         check_state_acceptance_condition();
         /* Compute state after execution. */
         compute_state_after_execution();
     od;
      end_state: printf("End state reached after %u",
               executed_event_count);
}
```

Appendix B

Zing Code for Give Medicine Example

```
/*
DCRS Example: Givemedicine ZING language code for model checking in ZING tool.
Generated on 2010-06-17T14:17:56 by DCRStoPROMELA Compiler.
Developed by Raghava Rao Mukkamala (rao@itu.dk)
*/



// Section for Common Declarations
enum TypeActionsEnum {pm, s, gm, dt };

set TypeActionsSet TypeActionsEnum;

set TypeRelationsSet Relation;

class DCRSMain
{
    // set of actions whose conditions are executed.
    static TypeActionsSet EnabledActionsList;
    // E set
    static TypeActionsSet ExecutedActionsList;
    // I set
    static TypeActionsSet IncludedActionsList;
    // R set
    static TypeActionsSet PendingResponsesList;

    static TypeActionsEnum executingAction;

    static int numberOfExecutions;

    static bool atleast_one_accepting_run;


    activate static void Main()
    {
        // Initialise the sets for the state..
        EnabledActionsList = new TypeActionsSet;

        ExecutedActionsList = new TypeActionsSet;

        IncludedActionsList = new TypeActionsSet;

        PendingResponsesList = new TypeActionsSet;

        // Get the DCRS model specification
        DCRSModel.Initialise();

        while(true)
        {
            ComputeEnabledActions();

            assert( sizeof(EnabledActionsList) > 0, "Dead lock");

            //assert (sizeof(PendingResponsesList) == 0, "accepting state reached");


            executingAction = choose(EnabledActionsList);

            UpdateStatespace(executingAction);

            if(sizeof(PendingResponsesList) == 0)
            {
                atleast_one_accepting_run = false;
            }

            //atleast_one_accepting_run = (sizeof(PendingResponsesList) == 0) ? true :       ↙
    atleast_one_accepting_run ;

            //event ( numberOfExecutions, (numberOfExecutions > 0) );
            //trace ("Number of executions {0}", numberOfExecutions);

        }

        assert((!atleast_one_accepting_run), "There is not even a single accepting run!");

    }


    static void ComputeEnabledActions()
    {

        // start with an assumption that all the included actions are enabled
        EnabledActionsList = CloneActionSets(IncludedActionsList, EnabledActionsList);

        // Here the logic is to iterate through all the  ConditionsSet relations, and find out
        //which actions are to be deleted.
        foreach( Relation relation in DCRSModel.ConditionsSet)
        {
           if (! (relation.Child in ExecutedActionsList))
           {
               EnabledActionsList = EnabledActionsList - relation.Parent ;
           }
        }

        // Here the logic is to iterate through all the  ConditionsSet relations, and find out
        //which actions are to be deleted.
        foreach( Relation relation1 in DCRSModel.StrongConditionsSet)
        {
           if (! (relation1.Child in ExecutedActionsList))  if (relation1.Child in PendingResponsesList)
           {
               EnabledActionsList = EnabledActionsList - relation1.Parent ;
           }
        }





    }

    static void UpdateStatespace(TypeActionsEnum exeAction)
    {

        // Update the counter for numberOfExecutions
        //numberOfExecutions = numberOfExecutions + 1;

        // First update ExecutedActionsList.
        ExecutedActionsList = ExecutedActionsList + exeAction;

        // Update IncludedActionsList with includes and excludes

        // Here the logic is to iterate through all the include relations, and find out
        // which actions are to be included.
        foreach( Relation relation1 in DCRSModel.IncludesSet)
        {
           if (relation1.Parent == exeAction)
           {
               IncludedActionsList = IncludedActionsList + relation1.Child ;
           }
        }

        // Here the logic is to iterate through all the exclude relations, and find out
        // which actions are to be excluded.
        foreach( Relation relation2 in DCRSModel.ExcludesSet)
        {
           if (relation2.Parent == exeAction)
           {
               IncludedActionsList = IncludedActionsList - relation2.Child ;
           }
        }

        // update pending responses ( ie remove exeAction)
        PendingResponsesList = PendingResponsesList - exeAction ;

        foreach( Relation relation3 in DCRSModel.ResponsesSet)
        {
           if (relation3.Parent == exeAction)
           {
               PendingResponsesList = PendingResponsesList + relation3.Child ;
           }
        }

    }

    static TypeActionsSet CloneActionSets(TypeActionsSet source, TypeActionsSet target)
    {
        // Make sure that no elements are left in the set.
        if(sizeof(target) > 0)
        {
            target = new TypeActionsSet;
        }

        // Copy all elements one by one.
        foreach(TypeActionsEnum action in source )
        {
            target = target + action ;
        }

        return target;
    }

};



class DCRSModel
{

    static TypeActionsSet ActionsList;

    static TypeRelationsSet IncludesSet;

    static TypeRelationsSet ExcludesSet;

    static TypeRelationsSet ConditionsSet;

    static TypeRelationsSet StrongConditionsSet;

    static TypeRelationsSet ResponsesSet;

   static void Initialise()
    {
        ActionsList = new TypeActionsSet;

        ActionsList = ActionsList + TypeActionsEnum.pm;

        ActionsList = ActionsList + TypeActionsEnum.s;
        ActionsList = ActionsList + TypeActionsEnum.gm;

        ActionsList = ActionsList + TypeActionsEnum.dt;

        // Initialize include relations.

        IncludesSet = new TypeRelationsSet;

        CreateRelation( TypeActionsEnum.s, TypeActionsEnum.gm, IncludesSet);

        CreateRelation( TypeActionsEnum.s, TypeActionsEnum.dt, IncludesSet);

        // Initialize Exclude relations.

        ExcludesSet = new TypeRelationsSet;

        CreateRelation( TypeActionsEnum.gm, TypeActionsEnum.dt, ExcludesSet);

        CreateRelation( TypeActionsEnum.dt, TypeActionsEnum.gm, ExcludesSet);

         // Initialize condition relations.

        ConditionsSet = new TypeRelationsSet;

        CreateRelation( TypeActionsEnum.s, TypeActionsEnum.pm, ConditionsSet);

        CreateRelation( TypeActionsEnum.gm, TypeActionsEnum.s, ConditionsSet);

        CreateRelation( TypeActionsEnum.dt, TypeActionsEnum.s, ConditionsSet);


       // Initialise StrongConditionsSet
       StrongConditionsSet = new TypeRelationsSet;

       CreateRelation( TypeActionsEnum.gm, TypeActionsEnum.s, StrongConditionsSet);

       CreateRelation( TypeActionsEnum.dt, TypeActionsEnum.s, StrongConditionsSet);



       // // Initialize Responses relations.

        ResponsesSet = new TypeRelationsSet;

        CreateRelation( TypeActionsEnum.pm, TypeActionsEnum.gm, ResponsesSet);

        CreateRelation( TypeActionsEnum.pm, TypeActionsEnum.s, ResponsesSet);

        CreateRelation( TypeActionsEnum.dt, TypeActionsEnum.s, ResponsesSet);




        // update included set with actions which are initially included.
        //DCRSMain.IncludedActionsList = ActionsList;

        DCRSMain.IncludedActionsList = DCRSMain.CloneActionSets(ActionsList, DCRSMain.IncludedActionsList);


    }


    static void CreateRelation(TypeActionsEnum dom, TypeActionsEnum ran, TypeRelationsSet relationSet)
    {
        Relation rel;

        rel = new Relation;

        rel.Initialise( dom, ran);

        relationSet = relationSet + rel;

    }



};

class Relation
{
    TypeActionsEnum Child;

    TypeActionsEnum Parent;

    void Initialise(TypeActionsEnum dom, TypeActionsEnum ran)
    {
        Parent = dom;

        Child = ran;
    }

};
```

Bibliography

[Aalst & Weske 2001] Wil M. P. van der Aalst and Mathias Weske. *The P2P Approach to Interorganizational Workflows*. In Proceedings of the 13th International Conference on Advanced Information Systems Engineering, CAiSE '01, pages 140–156, 2001. (Cited on page 123.)

[Aalst *et al.* 2011] W M P Van Der Aalst, K M Van Hee, A H M Hofstede and N Sidorova. *Soundness of Workflow Nets : Classification , Decidability , and Analysis*. Technology, vol. 23, no. 3, pages 1–48, 2011. (Cited on page 55.)

[Aalst 2001] W M P Van Der Aalst. *Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change*. Information Systems Frontiers, vol. 3, no. 3, pages 297–317, 2001. (Cited on page 55.)

[Aalst 2004] Wil M P Van Der Aalst. *Business Process Management Demystified : A Tutorial on Models , Systems and Standards for Workflow Management*. Lectures on Concurrency and Petri Nets, vol. 3098, no. 3098, pages 1–65, 2004. (Cited on page 4.)

[Adams *et al.* 2006] Michael Adams, Arthur H M Hofstede, David Edmond and Wil M P Van Der Aalst. *Worklets : A Service-Oriented Implementation of Dynamic Flexibility in Workflows*. On the Move to Meaningful Internet Systems 2006 CoopIS DOA GADA and ODBASE, vol. 4275, no. 19, pages 291–308, 2006. (Cited on page 56.)

[Adams 2007] Michael James Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows by*. PhD thesis, Queensland University of Technology Brisbane, Australia, 2007. (Cited on page 56.)

[Andrews *et al.* 2004] Tony Andrews, Shaz Qadeer, Sriram Rajamani, Jakob Rehof and Yichen Xie. *Zing: Exploiting Program Structure for Model Checking Concurrent Software*. In Philippa Gardner and Nobuko Yoshida, editors, CONCUR 2004 - Concurrency Theory, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2004. (Cited on pages 159 and 185.)

[Antonik *et al.* 2008] Adam Antonik, Michael Huth, Kim Larsen, Ulrik Nyman and Andrzej Wasowski. *20 Years of Mixed and Modal Specifications*. Bulletin of the European Association for Theoretical Computer Science, May 2008. (Cited on page 208.)

[Ash *et al.* 2004] J.S. Ash, M. Berg and E. Coiera. *Some Unintended Consequences of Information Technology in Health Care: The Nature of Patient Care Information System-related Errors*. J Sm Med Inform Assoc., vol. 11, pages 104–112, 2004. (Cited on page 37.)

[Augusto *et al.* 2003] Juan C. Augusto, Michael Butler, Carla Ferreira and Stephen Craig. *Using SPIN and STeP to verify business processes specifications*. In In Proceedings of Eeshov Memorial Conference, pages 207–213. Springer, 2003. (Cited on page 161.)

[Bardram & Bossen 2005] J.E. Bardram and C. Bossen. *Mobility Work: The Spatial Dimension of Collaboration at a Hospital*. Computer Supported Cooperative Work (CSCW), vol. 14, no. 2, pages 131–160, April 2005. (Cited on page 29.)

[Bates *et al.* 2001] D.W. Bates, M. Cohen, L.L. Leape, J.M. Overhage, M.M. Shabot and T. Sheridan. *Reducing the frequency of errors in medicine using information technology*. J Am Med Inform Assoc, vol. 8, pages 299–308, 2001. White Paper. (Cited on page 29.)

[Ben-Ari 2008] Mordechai Ben-Ari. Principles of the spin model checker. Springer, 2008. (Cited on pages 159, 160, 167 and 168.)

[Berg & Toussaint 2003] M. Berg and P. Toussaint. *The mantra of modeling and the forgotten powers of paper: A sociotechnical view on the development of process-oriented ICT in health care*. Int J Med Inform , vol. 69, pages 223–234, 2003. (Cited on page 37.)

[Berg *et al.* 2000] M. Berg, Klasien Horstman, Saskia Plass and Michelle van Heusden. *Guidelines, professionals and the production of objectivity: standardisation and the professionalism of insurance medicine*. Sociology of Health & Illness, vol. 22, pages 765–791(27), November 2000. (Cited on page 38.)

[Bhattacharya *et al.* 2007a] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam and F. Y. Wu. *Artifact-centered operational modeling: lessons from customer engagements*. IBM Syst. J., vol. 46, pages 703–721, October 2007. (Cited on pages 56 and 161.)

[Bhattacharya *et al.* 2007b] Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu and Jianwen Su. *Towards formal analysis of artifactcentric business process models*. In In preparation, pages 288–304, 2007. (Cited on pages 4 and 161.)

[Bjørner *et al.* 2000] Nikolaj S. Bjørner, Anca Browne, Michael A. Colon, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma and Tomas E. Uribe. *Verifying temporal properties of reactive systems: A STeP tutorial*. In FORMAL METHODS IN SYSTEM DESIGN, page 2000, 2000. (Cited on page 161.)

[Bødker & Christiansen 2004] S. Bødker and E. Christiansen. *Designing for ephemerality and prototypicality*. In Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques., Cambridge, MA, USA, 2004. ACM Press. (Cited on page 29.)

[Bowen & Stavridou 1993] J. Bowen and V. Stavridou. *Safety-critical systems, formal methods and standards*. Software Engineering Journal, vol. 8, no. 4, pages 189 –209, jul 1993. (Cited on page 5.)

[Brauer *et al.* 1987] Wilfried Brauer, Wolfgang Reisig and Grzegorz Rozenberg, editors. Petri nets: Central models and their properties, advances in petri nets 1986, part ii, proceedings of an advanced course, bad honnef, 8.-19. september 1986, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987. (Cited on pages 160 and 257.)

[Bravetti & Zavattaro 2007] Mario Bravetti and Gianluigi Zavattaro. *Contract Based Multi-party Service Composition*. In International Symposium on Fundamentals of Software Engineering (FSEN), volume 4767, pages 207–222. Springer, 2007. (Cited on pages 123 and 124.)

[Bravetti & Zavattaro 2009] Mario Bravetti and Gianluigi Zavattaro. *A theory of contracts for strong service compliance*. Mathematical. Structures in Comp. Sci., vol. 19, pages 601–638, June 2009. (Cited on pages 123 and 124.)

[BURNS 1977] J. C. BURNS. *The evolution of office information systems*. Datamation, vol. vol. 23,no. 4, pages 60–64, April 1977. (Cited on page 2.)

[Bussler & Jablonski 1994] Christoph Bussler and Stefan Jablonski. *Implementing agent coordination for workflow management systems using active database systems*. In Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on, pages 53–59, Feb 1994. (Cited on pages 7 and 52.)

[Cabana *et al.* 1999] M.D. Cabana, C.S. Rand, N.R. Powe, A.W. Wu, M.H. Wilson, P.A. Abboud and H.R. Rubin. *Why don't physicians follow clinical practice guidelines? A framework for improvement.* JAMA, vol. 282, no. 15, pages 1458–1465, OCTOBER 1999. (Cited on page 30.)

[Carbone *et al.* 2007] Marco Carbone, Kohei Honda and Nobuko Yoshida. *Structured Communication-Centred Programming for Web Services*. In 16th European Symposium on Programming (ESOP'07), LNCS, pages 2–17. Springer, 2007. (Cited on pages 124, 150 and 202.)

[Carbone *et al.* 2012] Marco Carbone, Thomas Hildebrandt, Hugo A. Lopez, Gian Perrone and Andrzej Wasowski. *Refinement for Transition Systems with Responses*. In Accepted for International Workshop on Foundations of Interface Technologies, 2012. (Cited on page 208.)

[Castellani *et al.* 1999] Ilaria Castellani, Madhavan Mukund and P. Thiagarajan. *Synthesizing Distributed Transition Systems from Global Specifications*. In Foundations of Software Technology and Theoretical Computer Science, volume 1738, pages 219–231. Springer Berlin / Heidelberg, 1999. (Cited on page 124.)

[Cheng 1995] Allan Cheng. *Petri Nets, Traces, and Local Model Checking*. In Proceedings of AMAST, pages 322–337, 1995. (Cited on pages 57 and 58.)

[Cicekli & Cicekli 2006] Nihan Kesim Cicekli and Ilyas Cicekli. *Formalizing the specification and execution of workflows using the event calculus*. Information Sciences, vol. 176, no. 15, pages 2227 – 2267, 2006. (Cited on pages 55 and 90.)

[Cicekli & Yildirim 2000] Nihan K. Cicekli and Yakup Yildirim. *Formalizing Workflows Using the Event Calculus*. In Proceedings of the 11th International Conference on Database and Expert Systems Applications, DEXA '00, pages 222–231. Springer-Verlag, 2000. (Cited on page 55.)

[Cimatti *et al.* 2000] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. *NUSMV: a new symbolic model checker*. International Journal on Software Tools for Technology Transfer, vol. 2, page 2000, 2000. (Cited on page 159.)

[Cohn & Hull 2009] David Cohn and Richard Hull. *Business Artifacts : A Data-centric Approach to Modeling Business Operations and Processes*. Management, vol. 32, no. 3, pages 1–7, 2009. (Cited on page 4.)

[Coiera 2006] E. Coiera. *Communication systems in healthcare*. Clin Biochem Rev , vol. 27, pages 89–98, 2006. (Cited on page 38.)

[Crafa *et al.* 2007] Silvia Crafa, Daniele Varacca and Nobuko Yoshida. *Compositional Event Structure Semantics for the Internal pi -Calculus*. In Luís Caires and Vasco Thudichum Vasconcelos, editors, CONCUR, volume 4703 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007. (Cited on page 48.)

[D. Eastlake 2002] D. Solo D. Eastlake J. Reagle. *RFC 3275: XML-Signature Syntax and Processing*, 2002. `http://www.ietf.org/rfc/rfc3275.txt`. (Cited on page 20.)

[Damaggio *et al.* 2011] Elio Damaggio, Richard Hull and Roman Vaculín. *On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard-Stage-Milestone Lifecycles*. In Stefanie Rinderle-Ma, Farouk Toumani and Karsten Wolf, editors, BPM, volume 6896 of *Lecture Notes in Computer Science*, pages 396–412. Springer, 2011. (Cited on pages 56 and 204.)

[Dams *et al.* 1997] Dennis Dams, Rob Gerth and Orna Grumberg. *Abstract interpretation of reactive systems*. ACM Trans. Program. Lang. Syst., vol. 19, pages 253–291, March 1997. (Cited on page 208.)

[Das *et al.* 1996] S. Das, K. Kochut, J. Miller, A. Sheth and D. Worah. *ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR2*. Technical report, The University of Georgia, 1996. (Cited on page 125.)

[Davenport 1993] T.H. Davenport. Process innovation: reengineering work through information technology. Harvard Business School Press, 1993. (Cited on page 2.)

[Davis & Taylor-Vaisey 1997] D.A. Davis and A. Taylor-Vaisey. *Translating guidelines into practice. A systematic review of theoretic concepts, practical experience and research evidence in the adoption of clinical practice guidelines.* CMAJ, vol. 157, no. 4, pages 408–416, August 1997. (Cited on page 29.)

[Davulcu *et al.* 1998] Hasam Davulcu, Michael Kifer, C. R. Ramakrishnan and I.V. Ramakrishnan. *Logic Based Modeling and Analysis of Workflows.* In Proceedings of ACM SIGACT-SIGMOD-SIGART, pages 1–3. ACM Press, 1998. (Cited on pages 7, 52, 55 and 90.)

[de Jong 1991] Gjalt G. de Jong. *An Automata Theoretic Approach to Temporal Logic.* In PROCEEDINGS OF 3 RD WORKSHOP ON COMPUTER AIDED VERIFICATION (CAV91), VOLUME 575 OF LECTURE NOTES IN COMPUTER SCIENCE, pages 477–487. Springer-Verlag, 1991. (Cited on page 39.)

[Deutsch *et al.* 2009] Alin Deutsch, Richard Hull, Fabio Patrizi and Victor Vianu. *Automatic verification of data-centric business processes.* In Proceedings of the 12th International Conference on Database Theory, ICDT '09, pages 252–267, New York, NY, USA, 2009. ACM. (Cited on pages 161 and 203.)

[Diaz *et al.* 2005] Gregorio Diaz, Juan-José Pardo, María-Emilia Cambronero, Valentín Valero and Fernando Cuartero. *Automatic Translation of WS-CDL Choreographies to Timed Automata.* Formal Techniques for Computer Systems and Business Processes, pages 230–242, 2005. (Cited on page 160.)

[Dijkman *et al.* 2008] Remco M. Dijkman, Marlon Dumas and Chun Ouyang. *Semantics and analysis of business process models in BPMN.* Information and Software Technology, vol. 50, no. 12, pages 1281 – 1294, 2008. (Cited on page 160.)

[Dong *et al.* 2000] Guozhu Dong, Richard Hull, Bharat Kumar, Jianwen Su and Gang Zhou. *A Framework for Optimizing Distributed Workflow Executions.* In Revised Papers from the 7th International Workshop on Database Programming Languages: Research Issues in Structured and Semistructured Database Programming, DBPL '99, pages 152–167, London, UK, 2000. Springer-Verlag. (Cited on page 124.)

[Dong *et al.* 2006] Jin Dong, Yang Liu, Jun Sun and Xian Zhang. *Verification of Computation Orchestration Via Timed Automata.* In Zhiming Liu and Jifeng He, editors, Formal Methods and Software Engineering, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer Berlin / Heidelberg, 2006. (Cited on page 160.)

[Drucker 1993] P.F. Drucker. *The New Realities.* Harper & Row, 1993. (Cited on page 37.)

[Dun *et al.* 2008] Haiqiang Dun, Haiying Xu and Lifu Wang. *Transformation of BPEL Processes to Petri Nets.* In Theoretical Aspects of Software Engineering, 2008.

TASE '08. 2nd IFIP/IEEE International Symposium on, pages 166 –173, june 2008. (Cited on page 160.)

[Dwyer *et al.* 1998] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. *Property specification patterns for finite-state verification*. In Proceedings of the second workshop on Formal methods in software practice, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM. (Cited on page 24.)

[Ellis & Nutt 1980] Clarence A. Ellis and Gary J. Nutt. *Office Information Systems and Computer Science*. ACM Comput. Surv., vol. 12, pages 27–60, March 1980. (Cited on page 2.)

[Ellis & Nutt 1996] Clarence A Ellis and Gary J Nutt. *Workflow: The Process Spectrum*. In AmitEditor Sheth, editor, Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, pages 140–145, 1996. (Cited on page 2.)

[Ellis *et al.* 1995] Clarence Ellis, Karim Keddara and Grzegorz Rozenberg. Dynamic change within workflow systems, pages 10–21. ACM Press, 1995. (Cited on page 55.)

[Ellis 1979] Clarence A. Ellis. *Information Control Nets: A Mathematical Model of Office Information Flow*. Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems, ACM SIGMETRICS Performance Evaluation Review, vol. 8, no. 3, pages 225–240, 1979. (Cited on page 2.)

[Eshuis & Wieringa 2004] R. Eshuis and R. Wieringa. *Tool support for verifying UML activity diagrams*. Software Engineering, IEEE Transactions on, vol. 30, no. 7, pages 437 – 447, july 2004. (Cited on page 159.)

[Eshuis 2002] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, Univ. of Twente, November 2002. CTIT Ph.D.-thesis series No. 02-44. (Cited on pages 5 and 159.)

[Exformatics 2009] Exformatics, 2009. `http://exformatics.dk/`. (Cited on page 199.)

[Fahland 2007] Dirk Fahland. *Towards Analyzing Declarative Workflows*. In Autonomous and Adaptive Web Services, 2007. (Cited on page 125.)

[Fdhila & Godart 2009] Walid Fdhila and Claude Godart. *Toward synchronization between decentralized orchestrations of composite web services*. In CollaborateCom'09, pages 1–10, 2009. (Cited on page 124.)

[Fdhila *et al.* 2009] Walid Fdhila, Ustun Yildiz and Claude Godart. *A flexible approach for automatic process decentralization using dependency tables*. International Conference on Web Services, 2009. (Cited on page 124.)

[Feder *et al.* 1999] G. Feder, M. Eccles, R. Grol, C. Griffiths and J. Grimshaw. *Clinical guidelines: using clinical guidelines.* BMJ, vol. 318, pages 728–730, 1999. (Cited on page 37.)

[Fernandes *et al.* 1997] Alvaro A. A. Fernandes, M. Howard Williams and Norman W. Paton. *A logic-based integration of active and deductive databases*. New Gen. Comput., vol. 15, no. 2, pages 205–244, 1997. (Cited on pages 6 and 51.)

[Ferrara 2004] Andrea Ferrara. *Web Services: A Process Algebra Approach*. Proceedings of the 2nd international conference on Service oriented computing, pages 242–251, 2004. (Cited on page 160.)

[Field & Lohr 1992] M. J. Field and K. N. Lohr. *Guidelines for Clinical Practice: From Development to Use*, 1992. (Cited on page 30.)

[Fournet *et al.* 2004] Cedric Fournet, Tony Hoare, Sriram Rajamani and Jakob Rehof. *Stuck-Free Conformance*. In Rajeev Alur and Doron Peled, editors, Computer Aided Verification, volume 3114 of *Lecture Notes in Computer Science*, pages 314–317. Springer Berlin / Heidelberg, 2004. (Cited on pages 159 and 185.)

[Fu *et al.* 2004a] Xiang Fu, Tevfik Bultan and Jianwen Su. *Analysis of interacting BPEL web services*. In Proceedings of the 13th international conference on World Wide Web, WWW '04, pages 621–630, New York, NY, USA, 2004. ACM. (Cited on page 160.)

[Fu *et al.* 2004b] Xiang Fu, Tevfik Bultan and Jianwen Su. *Realizability of Conversation Protocols With Message Contents*. In Proceedings of the IEEE International Conference on Web Services, ICWS '04, pages 96–, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on pages 123 and 124.)

[Gabbay 1987] Dov M. Gabbay. *The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems*. In Temporal Logic in Specification, pages 409–448, London, UK, 1987. Springer-Verlag. (Cited on pages 25 and 27.)

[Georgakopoulos *et al.* 1995] Diimitrios Georgakopoulos, Mark Hornick and Amit Sheth. *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. In DISTRIBUTED AND PARALLEL DATABASES, pages 119–153, 1995. (Cited on page 3.)

[Gerede & Su 2007] Cagdas E. Gerede and Jianwen Su. *Specification and Verification of Artifact Behaviors in Business Process Models*. In Proceedings of the 5th international conference on Service-Oriented Computing, ICSOC '07, pages 181–192, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on pages 4 and 161.)

[Gerede *et al.* 2007] C.E. Gerede, K. Bhattacharya and Jianwen Su. *Static Analysis of Business Artifact-centric Operational Models*. In Service-Oriented Computing and Applications, 2007. SOCA '07. IEEE International Conference on, pages 133 –140, june 2007. (Cited on pages 4 and 161.)

[Gerth *et al.* 1995] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi and Pierre Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic.* In In Protocol Specification Testing and Verification, pages 3–18. Chapman & Hall, 1995. (Cited on page 182.)

[Gerth *et al.* 1996] Rob Gerth, Doron Peled, Moshe Y. Vardi and Pierre Wolper. *Simple on-the-fly automatic verification of linear temporal logic.* In Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd. (Cited on pages 39 and 41.)

[Grimshaw *et al.* 2004] J.M. Grimshaw, R.E. Thomas, G. MacLennan, C. Fraser, C.R. Ramsay, L. Vale, P. Whitty, M.P. Eccles, L. Matowe, L. Shirran, M. Wensing, R. Dijkstra and C. Donaldson. *Effectiveness and efficiency of guideline dissemination and implementation strategies.* Health Technol Assess 8, vol. iii-iv, pages 1–72, 2004. (Cited on page 30.)

[Grol & Grimshaw 2003] R. Grol and J. Grimshaw. *From best evidence to best practice: effective implementation of change in patients' care.* The Lancet, vol. 362, pages 1225–1230, 2003. (Cited on pages 29 and 37.)

[Guelfi & Mammar 2005] Nicolas Guelfi and Amel Mammar. *A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation.* In APSEC'2005: Asia Pacific Software Engineering Conference. IEEE Computer Society Press, 2005. (Cited on pages 159 and 161.)

[Guelfi *et al.* 2004] Nicolas Guelfi, Amel Mammar and Benoît Ries. *A Formal Approach for the Specification and the Verification of UML Structural Properties: Application to E-Business Domain.* In International Workshop on Software Verification and Validation (SVV 2004), workshop of ICFEM'04. IEEE Computer Society, 2004. (Cited on pages 159 and 161.)

[Hammer & Champy 1993] Michael Hammer and James Champy. Reengineering the corporation: A manifesto for business revolution, volume 5. Harper Business, 1993. (Cited on page 2.)

[Hammer 1990] Michael Hammer. *Reengineering Work: Don't Automate, Obliterate.* Harvard Business Review, vol. 68, pages 104–112, 1990. (Cited on page 2.)

[Harmon 2007] P. Harmon. Business process change: a guide for business managers and bpm and six sigma professionals. The MK/OMG Press. Elsevier/Morgan Kaufmann Publishers, 2007. (Cited on pages 2 and 3.)

[Havelund *et al.* 1998] Klaus Havelund, Mike Lowry and John Penix. *Formal Analysis of a Space Craft Controller using SPIN.* In In Proceedings of the 4th SPIN workshop, 1998. (Cited on page 161.)

[Hee *et al.* 2004] Kees Van Hee, Natalia Sidorova and Marc Voorhoeve. *Generalised Soundness of Workflow Nets Is Decidable.* Applications and Theory of Petri Nets 2004, vol. 3099, page 197Ű215, 2004. (Cited on page 55.)

[Heinl *et al.* 1999] Petra Heinl, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein and Michael Teschke. *A Comprehensive Approach to Flexibility in Workflow Management Systems.* In Proceedings of WACC '99, pages 79–88. ACM Press, 1999. (Cited on page 202.)

[Heljanko & Stefanescu 2005] Keijo Heljanko and Alin Stefanescu. *Complexity Results for Checking Distributed Implementability.* In Proceedings of the Fifth International Conference on Application of Concurrency to System Design, pages 78–87, 2005. (Cited on pages 124 and 125.)

[Hildebrandt & Mukkamala 2010] Thomas T. Hildebrandt and Raghava Rao Mukkamala. *Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs.* In Kohei Honda and Alan Mycroft, editors, PLACES, volume 69 of *EPTCS*, pages 59–73, 2010. (Cited on pages 51 and 191.)

[Hildebrandt & Mukkamala 2011] Thomas Hildebrandt and Raghava Rao Mukkamala. *Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs.* In Post proceedings of International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10), 2011. (Cited on page 51.)

[Hildebrandt & Sassone 1996] Thomas Hildebrandt and Vladimiro Sassone. *Comparing transition systems with independence and asynchronous transition systems.* In Ugo Montanari and Vladimiro Sassone, editors, CONCUR '96: Concurrency Theory, volume 1119 of *Lecture Notes in Computer Science*, pages 84–97. Springer Berlin / Heidelberg, 1996. (Cited on page 202.)

[Hildebrandt *et al.* 2011a] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. *Declarative Modelling and Safe Distribution of Healthcare Workflows.* In International Symposium on Foundations of Health Information Engineering and Systems, Johannesburg, South Africa, August 2011. (Cited on page 201.)

[Hildebrandt *et al.* 2011b] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. *Designing a Cross-organizational Case Management System using Dynamic Condition Response Graphs.* In Proceedings of IEEE International EDOC Conference, 2011. (Cited on page 60.)

[Hildebrandt *et al.* 2011c] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. *Nested Dynamic Condition Response Graphs.* In Proceedings of Fundamentals of Software Engineering (FSEN), April 2011. (Cited on pages 60, 93 and 152.)

[Hildebrandt *et al.* 2011d] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. *Safe Distribution of Declarative Processes.* In 9th International Conference on Software Engineering and Formal Methods (SEFM) 2011, 2011. (Cited on pages 147 and 154.)

[Hildebrandt 2008] Thomas Hildebrandt. *Trustworthy Pervasive Healthcare Processes (TrustCare) Research Project.* Webpage, 2008. http://www.trustcare.dk/. (Cited on pages 10 and 17.)

[Hildebrandt 2010] Thomas Hildebrandt. *Interest Group for Processes and IT.* Webpage, 2010. http://www.infinit.dk/dk/interessegrupper/processer_og_it/. (Cited on page 13.)

[Hill *et al.* 2006] Janelle B Hill, Jim Sinur, David Flint and Michael James Melenovsky. *GartnerŠs Position on Business Process Management , 2006.* ReVision, no. February, 2006. (Cited on page 1.)

[Hinz *et al.* 2005] Sebastian Hinz, Karsten Schmidt and Christian Stahl. *Transforming BPEL to Petri Nets.* In Proceedings of the International Conference on Business Process Management (BPM2005), volume 3649 of Lecture Notes in Computer Science, pages 220–235. Springer-Verlag, 2005. (Cited on page 160.)

[Hofstede *et al.* 2010] Arthur H M Hofstede, Wil M P Aalst, Michael Adams and NickEditors Russell, editors. Modern business process automation. Springer-Verlag, 2010. (Cited on page 5.)

[Holzmann 1997] Gerard J. Holzmann. *The Model Checker SPIN.* IEEE Trans. Softw. Eng., vol. 23, pages 279–295, May 1997. (Cited on pages 159, 160, 167, 168 and 182.)

[Holzmann 2004] Gerard J. Holzmann. Spin model checker, the: Primer and reference manual. Addison-Wesley Professional, 2004. (Cited on pages 159, 160, 167, 168 and 203.)

[Hull *et al.* 2011a] Richard Hull, Elio Damaggio, Riccardo De Masellis, Fabiana Fournier, Manmohan Gupta, Fenno Terry Heath III, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piwadee Noi Sukaviriya and Roman Vaculin. *Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events.* In Proceedings of the 5th ACM international conference on Distributed event-based system, DEBS '11, pages 51–62, New York, NY, USA, 2011. ACM. (Cited on pages 56, 113 and 204.)

[Hull *et al.* 2011b] Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, Fenno Heath, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya and Roman Vaculin. *Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles*. In Mario Bravetti and Tevfik Bultan, editors, Web Services and Formal Methods, volume 6551 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin / Heidelberg, 2011. (Cited on pages xv, 56, 204 and 205.)

[Janssen *et al.* 1998] Wil Janssen, Radu Mateescu, Sjouke Mauw and Jan Springintveld. *Verifying Business Processes using SPIN*. In Proceedings of the 4th International SPIN Workshop, pages 21–36, 1998. (Cited on page 161.)

[Karamanolis *et al.* 2000] C. Karamanolis, D. Giannakopoulou, J. Magee and S.M. Wheater. *Model checking of workflow schemas*. In Enterprise Distributed Object Computing Conference, 2000. EDOC 2000. Proceedings. Fourth International, pages 170 –179, 2000. (Cited on page 160.)

[Kesten *et al.* 1996] Yonit Kesten, Zohar Manna and Amir Pnueli. *Verification of Clocked and Hybrid Systems*. In European Educational Forum: School on Embedded Systems'96, pages 4–73. Springer-Verlag, 1996. (Cited on page 159.)

[Khalaf & Leymann 2006] R. Khalaf and F. Leymann. *Role-based Decomposition of Business Processes using BPEL*. In Web Services, 2006. ICWS '06. International Conference on, pages 770 –780, sept. 2006. (Cited on page 124.)

[Kilov 2002] Haim Kilov. Business models: A guide for business and it. Prentice Hall;, 2002. (Cited on page 1.)

[Kindler *et al.* 2000] Ekkart Kindler, Axel Martens and Wolfgang Reisig. *Interoperability of Workflow Applications: Local Criteria for Global Soundness*. In Business Process Management, Models, Techniques, and Empirical Studies, pages 235–253, London, UK, 2000. Springer-Verlag. (Cited on page 123.)

[Knapp *et al.* 2002] Alexander Knapp, Stephan Merz and Christopher Rauh. *Model Checking - Timed UML State Machines and Collaborations*. In Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2, FTRTFT '02, pages 395–416, London, UK, UK, 2002. Springer-Verlag. (Cited on page 160.)

[Kohn *et al.* 2000] L.T. Kohn, J.M. Corrigan and M.S Donaldson. To err is human. building a safer health system. National Academic Press, Washington DC, 2000. (Cited on pages 29 and 37.)

[Kowalski 1992] Robert Kowalski. *Database updates in the event calculus*. J. Log. Program., vol. 12, no. 1-2, pages 121–146, 1992. (Cited on page 55.)

[Krukow *et al.* 2008] Karl Krukow, Mogens Nielsen and Vladimiro Sassone. *A logical framework for history-based access control and reputation systems*. J. Comput. Secur., vol. 16, pages 63–101, January 2008. (Cited on page 48.)

[Kumaran *et al.* 2003] S. Kumaran, P. Nandi, T. Heath, K. Bhaskaran and R. Das. *ADoc-oriented programming.* In Applications and the Internet, 2003. Proceedings. 2003 Symposium on, pages 334 – 341, jan. 2003. (Cited on page 4.)

[Laroussinie *et al.* 2002] F. Laroussinie, N. Markey and Ph. Schnoebelen. *Temporal logic with forgettable past.* In Proceedings of 17th IEEE Symp. Logic in Computer Science (LICS'2002), pages 383–392, Copenhagen, Denmark, July 2002 2002. IEEE Computer Society Press. (Cited on page 25.)

[Larsen & Thomsen 1988] K.G. Larsen and B. Thomsen. *A modal process logic.* In Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on, pages 203 –210, jul 1988. (Cited on page 208.)

[Larsen *et al.* 1997] Kim G. Larsen, Paul Pettersson and Wang Yi. *UPPAAL in a Nutshell*, 1997. (Cited on page 160.)

[Latella & Massink 2001] D. Latella and M. Massink. *A formal testing framework for UML statechart diagrams behaviours: from theory to automatic verification.* In High Assurance Systems Engineering, 2001. Sixth IEEE International Symposium on, pages 11 –22, 2001. (Cited on page 159.)

[Latella *et al.* 1999] Diego Latella, Istvan Majzik and Mieke Massink. *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker.* Formal Aspects of Computing, vol. 11, no. 6, pages 637–664, December 1999. (Cited on page 159.)

[Lenz & Reichert 2007] Richard Lenz and Manfred Reichert. *IT support for healthcare processes - premises, challenges, perspectives.* Data Knowl. Eng., vol. 61, no. 1, pages 39–58, 2007. (Cited on pages 29 and 37.)

[Levitt 1960] Theodore Levitt. *Marketing myopia. 1960.* Harvard Business Review, vol. 82, no. 7-8, pages 138–49, 1960. (Cited on page 2.)

[Lilius & Paltor 1999] J. Lilius and I.P. Paltor. *vUML: a tool for verifying UML models.* In Automated Software Engineering, 1999. 14th IEEE International Conference on., pages 255 –258, oct 1999. (Cited on page 160.)

[Liu *et al.* 2007] Rong Liu, Kamal Bhattacharya and Frederick Wu. *Modeling Business Contexture and Behavior Using Business Artifacts.* In John Krogstie, Andreas Opdahl and Guttorm Sindre, editors, Advanced Information Systems Engineering, volume 4495 of *Lecture Notes in Computer Science*, pages 324–339. Springer Berlin / Heidelberg, 2007. (Cited on pages 56 and 161.)

[Lyng *et al.* 2008] Karen Marie Lyng, Thomas Hildebrandt and Raghava Rao Mukkamala. *From Paper Based Clinical Practice Guidelines to Declarative Workflow Management.* In Proceedings of 2nd International Workshop on Process-oriented information systems in healthcare (ProHealth 08), pages 36–43, Milan, Italy, 2008. BPM 2008 Workshops. (Cited on pages 7, 18, 52, 53, 73, 89, 91, 92, 150, 157 and 201.)

[Martens 2005] Axel Martens. *Analyzing Web Service Based Business Processes*. In Fundamental Approaches to Software Engineering. Springer Berlin / Heidelberg, 2005. (Cited on page 123.)

[M.Clarke *et al.* 1999] Edmund M.Clarke, Orna Grumberg and Doron A.Peled. Model checking. MIT Press, 1999. (Cited on pages 5, 24 and 41.)

[McNaughton 1966] Robert McNaughton. *Testing and generating infinite sequences by a finite automaton*. Information and Control, vol. 9, no. 5, pages 521–530, 1966. (Cited on page 89.)

[Microsoft-Research 2010] Microsoft-Research. *Zing Model Checker*. Webpage, 2010. http://research.microsoft.com/en-us/projects/zing/. (Cited on page 16.)

[Milosevic *et al.* 2006] Zoran Milosevic, Shazia Sadiq and Maria Orlowska. *Towards a Methodology for Deriving Contract-Compliant Business Processes*. In Business Process Management, volume 4102 of *Lecture Notes in Computer Science*, pages 395–400. Springer Berlin / Heidelberg, 2006. (Cited on page 124.)

[Mimnagh & Murphy. 2004] C. Mimnagh and M. Murphy. *Junior doctors working patterns: application of knowledge management theory to junior doctors training.* In Proc. of the conf. on current perspectives in healthcare computing, pages 42–47. Harrogate, 2004. (Cited on page 38.)

[Mitra *et al.* 2008] Saayan Mitra, Ratnesh Kumar and Samik Basu. *Optimum Decentralized Choreography for Web Services Composition*. In Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2, 2008. (Cited on page 124.)

[Mohan *et al.* 1995] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Guenthoer and M. Kamath. *Exotica: a project on advanced transaction management and workflow systems*. SIGOIS Bull., vol. 16, pages 45–50, August 1995. (Cited on page 125.)

[Montali 2010] Marco Montali. Specification and verification of declarative open interaction models: A logic-based approach, volume 56 of *Lecture Notes in Business Information Processing*. Springer, 2010. (Cited on page 125.)

[Morimoto 2008] Shoichi Morimoto. *A Survey of Formal Verification for Business Process Modeling*. New York, vol. 5102, pages 514–522, 2008. (Cited on page 160.)

[Mukkamala & Hildebrandt 2010] Raghava Rao Mukkamala and Thomas Hildebrandt. *From Dynamic Condition Response Structures to Büchi Automata*. In Proceedings of 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010), August 2010. (Cited on pages 51 and 191.)

[Mukkamala *et al.* 2008] Raghava Rao Mukkamala, Thomas Hildebrandt and Janus Boris Tøth. *The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL.* In Proceeding of DDBP, 2008. (Cited on pages 7, 18, 52, 53 and 89.)

[Mukund & Nielsen 1992] Madhavan Mukund and Mogens Nielsen. *CCS, locations and asynchronous transition systems.* In Rudrapatna Shyamasundar, editor, Foundations of Software Technology and Theoretical Computer Science, volume 652 of *Lecture Notes in Computer Science*, pages 328–341. Springer Berlin / Heidelberg, 1992. (Cited on page 202.)

[Mukund 2002] M. Mukund. *From Global Specifications to Distributed Implementations.* In Synthesis and Control of Discrete Event Systems. Springer, 2002. (Cited on page 124.)

[Mulyar *et al.* 2007] N. Mulyar, M. Pesic, W.M. van der Aalst and M. Peleg. *Towards the Flexibility in Clinical Guideline Modelling Languages.* BPM Center Report (Ext. rep. BPM-07-04)., vol. 8, 2007. (Cited on page 29.)

[Nanda *et al.* 2004] Mangala Gowri Nanda, Satish Chandra and Vivek Sarkar. *Decentralizing execution of composite web services.* SIGPLAN Not., vol. 39, pages 170–187, October 2004. (Cited on page 124.)

[Narayanan & McIlraith 2002] Srini Narayanan and Sheila A. McIlraith. *Simulation, verification and automated composition of web services.* In Proceedings of the 11th international conference on World Wide Web, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM. (Cited on page 160.)

[Nielsen *et al.* 1979] Mogens Nielsen, Gordon Plotkin and Glynn Winskel. *Petri nets, event structures and domains.* In Gilles Kahn, editor, Semantics of Concurrent Computation, volume 70 of *Lecture Notes in Computer Science*, pages 266–284. Springer Berlin / Heidelberg, 1979. 10.1007/BFb0022474. (Cited on pages 43 and 53.)

[Nigam & Caswell 2003] A. Nigam and N. S. Caswell. *Business artifacts: An approach to operational specification.* IBM Syst. J., vol. 42, pages 428–445, July 2003. (Cited on pages 4, 56 and 161.)

[OASIS WSBPEL Technical Committee 2007] OASIS WSBPEL Technical Committee. *Web Services Business Process Execution Language, Version 2.0*, 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`. (Cited on pages 5, 124 and 160.)

[Object Management Group BPMN Technical Committee 2011] Object Management Group BPMN Technical Committee. *Business Process Model and Notation, Version 2.0.* Webpage, january 2011. `http://www.omg.org/spec/BPMN/2.0/PDF`. (Cited on pages 5 and 160.)

[OMG 2007] OMG. *OMG Unified Modeling Language Infrastructure, Version 2.1.2*. Webpage, November 2007. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`. (Cited on page 5.)

[Orlikowski & Gash 1994] Wanda J. Orlikowski and Debra C. Gash. *Technological frames: making sense of information technology in organizations*. ACM Trans. Inf. Syst., vol. 12, no. 2, pages 174–207, April 1994. (Cited on page 38.)

[Paul *et al.* 1997] Santanu Paul, Edwin Park and Jarir Chaar. *RainMan: a workflow system for the internet*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems, 1997. (Cited on page 125.)

[Pesic *et al.* 2007] M. Pesic, H. Schonenberg and W.M.P. van der Aalst. *DECLARE: Full Support for Loosely-Structured Processes*. In Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, pages 287–. IEEE Computer Society, Washington, DC, USA, 2007. (Cited on pages 40, 41 and 53.)

[Pesic 2008] Maja Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users*. PhD thesis, Eindhoven University of Technology, Netherlands, 2008. (Cited on pages 7, 8 and 52.)

[Petri 1977] C.A. Petri. Non-sequential processes: translation of a lecture given at the immd jubilee colloquium on "parallelism in computer science", university of erlngen-nürnberg, june 1976. GMD-ISF report. GMD, Ges. für Math. und Datenverarb., 1977. (Cited on page 43.)

[Petri 1980] C. A. Petri. *Introduction to General Net Theory*. In Proceedings of the Advanced Course on General Net Theory of Processes and Systems: Net Theory and Applications, pages 1–19, London, UK, 1980. Springer-Verlag. (Cited on page 43.)

[Pnueli 1977] A. Pnueli. *The temporal logic of programs*. In Proceedings of 18th IEEE FOCS, pages 46–57, 1977. (Cited on pages 15, 17, 24, 40, 169 and 197.)

[Porres 2001] I. Porres. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, TUCS Turku Centre for Computer Science, 2001. (Cited on page 159.)

[Quaglini *et al.* 2001] S. Quaglini, M. Stefanelli, G. Lanzola, V. Caporusso and S. Panzarasa. *Flexible guideline-based patient careflow systems*. Artif Intell Med ., vol. 22, pages 65–80, 2001. (Cited on page 29.)

[Ranno & Shrivastava 1999] F. Ranno and S. K. Shrivastava. *A Review of Distributed Workflow Management Systems*. In Proceedings of the international joint conference on Work activities coordination and collaboration, 1999. (Cited on page 125.)

[Reddy *et al.* 2001] Madhu C. Reddy, Paul Dourish and A. Pratt. *Coordinating Heterogeneous Work: Information and Representation in Medical Care*. In In Prinz et al, pages 239–258. Kluwer Academic Publishers, 2001. (Cited on page 37.)

[Reichert & Bauer 2007] Manfred Reichert and Thomas Bauer. *Supporting Ad-Hoc Changes in Distributed Workflow Management Systems*. In Robert Meersman and Zahir Tari, editors, On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, volume 4803 of *Lecture Notes in Computer Science*, pages 150–168. Springer Berlin / Heidelberg, 2007. (Cited on page 125.)

[Reichert & Dadam 1997] Manfred Reichert and Peter Dadam. *A Framework for Dynamic Changes in Workflow Management Systems*. In in 'Proceedings, 8th Int'l Conference on Database and Expert Systems Applications (DEXA-97, pages 42–48. IEEE Computer Society Press, 1997. (Cited on page 6.)

[Reichert & Dadam 1998] M Reichert and P Dadam. *ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control*. Journal of Intelligent Information Systems, vol. 10, pages 93–129, 1998. (Cited on page 55.)

[Reichert *et al.* 2003] Manfred Reichert, Stefanie Rinderle and Peter Dadam. *ADEPT Workflow Management System Flexible Support for Enterprise-Wide Business Processes - Tool Presentation -*. In Proc. 1st Int'l Conf. on Business Process Management (BPM '03), number 2678 of LNCS, pages 371–379. Springer, June 2003. (Cited on page 55.)

[Reichert *et al.* 2009] M. U. Reichert, T. Bauer and P. Dadam. *Flexibility for Distributed Workflows*. In Handbook of Research on Complex Dynamic Process Management: Techniques for Adaptability in Turbulent Environments, pages 137–171. IGI Global, Hershey, PA, 2009. (Cited on page 125.)

[Reisig 1991] Wolfgang Reisig. *Petri nets and algebraic specifications*. Theor. Comput. Sci., vol. 80, pages 1–34, March 1991. (Cited on page 160.)

[Resultmaker 2008] Resultmaker, 2008. `http://www.resultmaker.com/`. (Cited on page 197.)

[Rinderle *et al.* 2003] Stefanie Rinderle, Manfred Reichert and Peter Dadam. *Evaluation of Correctness Criteria for Dynamic Workflow Changes*. German Research, pages 41–57, 2003. (Cited on page 55.)

[Rinderle *et al.* 2004] Stefanie Rinderle, Manfred Reichert and Peter Dadam. *Correctness criteria for dynamic changes in workflow systemsŰŰa survey*. Data & Knowledge Engineering, vol. 50, no. 1, pages 9–34, 2004. (Cited on page 55.)

[Rinderle *et al.* 2006] Stefanie Rinderle, Andreas Wombacher and Manfred Reichert. *Evolution of Process Choreographies in DYCHOR*. In On the Move to Meaningful

Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, volume 4275 of *LNCS*, pages 273–290. Springer, 2006. (Cited on pages 123 and 124.)

[Rummler & Brache 1990] G A Rummler and A P Brache. *How to Manage the White Space on the Organization Chart*. JosseyBass Inc California USA, 1990. (Cited on page 2.)

[Rummler & Brache 1995] Geary A Rummler and Alan P Brache. Improving performance: How to manage the white space on the organization chart. Jossey-Bass Publishers, 1995. (Cited on page 2.)

[Russel & Ter Hofstede 2009] Nick Russel and Arthur H M Ter Hofstede. *new YAWL: Towards Workflow 2.0*. Transactions on Petri Nets and Other Models of Concurrency II, vol. 5460/2009, pages 79–97, 2009. (Cited on page 3.)

[Sadiq *et al.* 2001] Shazia W Sadiq, Wasim Sadiq and Maria E Orlowska. *Pockets of flexibility in workflow specification*. Science, vol. 2224, pages 513–526, 2001. (Cited on page 56.)

[Sadiq *et al.* 2006] W. Sadiq, S. Sadiq and K. Schulz. *Model Driven Distribution of Collaborative Business Processes*. In Services Computing, 2006. SCC '06. IEEE International Conference on, pages 281 –284, sept. 2006. (Cited on page 124.)

[Salaun *et al.* 2004] G Salaun, L Bordeaux and M Schaerf. *Describing and reasoning on web services using process algebra*. Proceedings IEEE International Conference on Web Services 2004, vol. 1, no. 2, pages 43–50, 2004. (Cited on page 160.)

[Sara & Aguilar-Saven 2004] Ruth Sara and Aguilar-Saven. *Business process modelling: Review and framework*. International Journal of Production Economics, vol. 90, no. 2, pages 129 – 149, 2004. (Cited on pages 1 and 2.)

[Saunders-Evans & Winskel 2007] Lucy Saunders-Evans and Glynn Winskel. *Event Structure Spans for Nondeterministic Dataflow*. Electron. Notes Theor. Comput. Sci., vol. 175, pages 109–129, June 2007. (Cited on page 48.)

[Scheer 1998] August-Wilhelm W. Scheer. Aris-business process frameworks. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd édition, 1998. (Cited on page 5.)

[Scott 1970] D Scott. *Outline of a mathematical theory of computation*. Fourth Annual Princeton Conference on Information Sciences and Systems, pages 169–176, 1970. (Cited on page 43.)

[Scott 1976] D Scott. *Data Types as Lattices*. SIAM Journal on Computing, vol. 5, no. 3, pages 522–587, 1976. (Cited on page 43.)

[Scott 1982] Dana S Scott. *Domains for Denotational Semantics*. Automata languages and programming, no. 140, pages 577–610, 1982. (Cited on page 43.)

[Senkul *et al.* 2002] Pinar Senkul, Michael Kifer and Ismail H. Toroslu. *A Logical Framework for Scheduling Workflows Under Resource Allocation Constraints*. In In VLDB, pages 694–705, 2002. (Cited on pages 7, 52 and 55.)

[Shojania *et al.* 2007] K. G. Shojania, M. Sampson, M. T. Ansari, J. Ji, S. Doucette and D. Moher. *How quickly do systematic reviews go out of date? A survival analysis*. Ann Intern Med, vol. 147, no. 4, pages 224–233, August 2007. (Cited on page 29.)

[Shrivastava *et al.* 1998] Wheater Shrivastava, S. M. Wheater, S. K. Shrivastava and F. Ranno. *A CORBA Compliant Transactional Workflow System for Internet Applications*. In Proc. Of IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 98, pages 1–85233. Springer-Verlag, 1998. (Cited on page 125.)

[Sim *et al.* 2001] I. Sim, P. Gorman, R. A. Greenes, R. B. Haynes, B. Kaplan, H. Lehmann and P. C. Tang. *Clinical decision support systems for the practice of evidence-based medicine*. J Am Med Inform Assoc, vol. 8, no. 6, pages 527–534, 2001. (Cited on page 37.)

[Singh *et al.* 1995] Munindar P. Singh, Greg Meredith, Christine Tomlinson and Paul C. Attie. *An Event Algebra for Specifying and Scheduling Workflows*. In Proceedings of DASFAA, pages 53–60. World Scientific Press, 1995. (Cited on pages 7 and 52.)

[Sistla *et al.* 1983] A.P. Sistla, M. Vardi and P. Wolper. *Reasoning about infinite computation paths*. In Proceedings of 24th IEEE FOCS, pages 185–194, 1983. (Cited on pages 24 and 25.)

[Slaats 2009] Tijs Slaats. Workflow and business process execution based on temporal logic models. Master's thesis, IT University of Copenhagen, Denmark, 2009. (Cited on page 39.)

[Spin 2007] Spin. *Basic Spin Manual*. http://spinroot.com/spin/Man/Manual.html, 2007. (Cited on pages 16, 159, 168 and 172.)

[Spin 2008] Spin. *ON-THE-FLY, LTL MODEL CHECKING with SPIN*. Webpage, 2008. http://spinroot.com/spin/whatispin.html. (Cited on pages 16, 159, 160, 167 and 168.)

[Strong & Miller 1995] Diane M. Strong and Steven M. Miller. *Exceptions and exception handling in computerized information processes*. ACM Trans. Inf. Syst., vol. 13, pages 206–233, April 1995. (Cited on page 6.)

[ter Hofstede *et al.* 2003] Arthur ter Hofstede, Rob van Glabbeek and David Stork. *Query Nets: Interacting Workflow Modules That Ensure Global Termination*. In Business Process Management. Springer Berlin / Heidelberg, 2003. (Cited on page 123.)

[Thorsen & Makela 1999] Thorkil Thorsen and Marjukka Makela, editors. Changing professional practice., volume Vol. 99.05. Danish Institute for Health Services Research and Development, 1999. (Cited on page 37.)

[Uppaal-Group 2009] Uppaal-Group. *Uppaal Model Checker*. Webpage, 2009. http://www.uppaal.org/. (Cited on page 203.)

[van der Aalst & Pesic 2006a] Wil M.P van der Aalst and Maja Pesic. *A Declarative Approach for Flexible Business Processes Management*. In Proceedings DPM 2006, LNCS. Springer Verlag, 2006. (Cited on pages xiii, 7, 8, 9, 15, 17, 24, 25, 39, 40, 41, 49, 52, 53, 54, 55, 90, 125 and 207.)

[van der Aalst & Pesic 2006b] Wil M.P van der Aalst and Maja Pesic. *DecSerFlow: Towards a Truly Declarative Service Flow Language*. In M. Bravetti, M. Nunez and Gianluigi Zavattaro, editors, Proceedings of Web Services and Formal Methods (WS-FM 2006), volume 4184 of *LNCS*, pages 1–23. Springer Verlag, 2006. (Cited on pages 15, 17, 24, 25, 30, 33, 39, 40, 41, 49, 53, 54, 55 and 125.)

[Van Der Aalst *et al.* 1997] W M P Van Der Aalst, D Hauschildt and H M W Verbeek. Petri-net-based tool to analyze workflows, pages 78–98. University of Hamburg (FBI-HH-B-205/97), 1997. (Cited on page 55.)

[van der Aalst *et al.* 2003] W.M.P. van der Aalst, A. H. M. Ter Hofstede and M. Weske. *Business Process Management: A Survey*. In Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS, pages 1–12. Springer-Verlag, 2003. (Cited on pages xiii, 1, 2, 3, 4, 5 and 6.)

[van der Aalst *et al.* 2009] Wil M. P. van der Aalst, Maja Pesic and Helen Schonenberg. *Declarative workflows: Balancing between flexibility and support*. Computer Science - R&D, vol. 23, no. 2, pages 99–113, 2009. (Cited on pages 6, 7, 9, 40, 41, 52, 53, 54, 55, 90, 202 and 207.)

[van der Aalst *et al.* 2010a] Wil van der Aalst, Maja Pesic, Helen Schonenberg, Michael Westergaard and Fabrizio M. Maggi. *Declare*. Webpage, 2010. http://www.win.tue.nl/declare/. (Cited on pages 15, 17, 40, 49, 53 and 197.)

[van der Aalst *et al.* 2010b] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl and Karsten Wolf. *Multiparty Contracts: Agreeing and Implementing Interorganizational Processes*. The Computer Journal, vol. 53, no. 1, pages 90–106, January 2010. (Cited on page 123.)

[van der Aalst 1999a] W. M. P. van der Aalst. *Interorganizational Workflows: An Approach based on Message Sequence Charts and Petri Nets*. Systems Analysis - Modelling - Simulation, vol. 34, no. 3, pages 335–367, 1999. (Cited on page 123.)

[van der Aalst 1999b] W. M. P. van der Aalst. *Woflan: a Petri-net-based workflow analyzer*. Syst. Anal. Model. Simul., vol. 35, pages 345–357, May 1999. (Cited on page 160.)

[van der Aalst 2003] W.M.P. van der Aalst. *Inheritance of Interorganizational Workflows: How to Agree to Disagree Without Loosing Control?* Information Technology and Management, vol. 4, pages 345–389, 2003. (Cited on page 123.)

[Vanderaalst *et al.* 2005] W Vanderaalst, M Weske and D Grunbauer. *Case handling: a new paradigm for business process support.* Data & Knowledge Engineering, vol. 53, no. 2, pages 129–162, 2005. (Cited on page 55.)

[Vardi & Wolper 1986] M. Y. Vardi and P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* In Symposium on Logic in Computer Science (LICS'86), pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press. (Cited on pages 160, 167 and 168.)

[Verbeek & Aalst 2000] Eric Verbeek and Wil M P Van Der Aalst. Woflan 2.0: a petri-net-based workflow diagnosis tool, volume 1825, pages 475–484. Springer, 2000. (Cited on page 55.)

[Verbeek & van der Aalst 2000] Eric Verbeek and Wil M. P. van der Aalst. *Woflan 2.0: a Petri-net-based workflow diagnosis tool.* In Proceedings of the 21st international conference on Application and theory of petri nets, ICATPN'00, pages 475–484, Berlin, Heidelberg, 2000. Springer-Verlag. (Cited on page 160.)

[Weske 2007] M. Weske. Business process management: Concepts, languages, architectures. Springer, 2007. (Cited on page 1.)

[Westergaard 2011] Michael Westergaard. *Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL.* In Proc. of BPM, 2011. (Cited on page 42.)

[WfMC 1999] WfMC. *Workflow Management Coalition Terminology & Glossary.* Management, vol. 39, no. 3, pages 1–65, 1999. `http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf`. (Cited on page 3.)

[Winskel & Nielsen 1993] Glynn Winskel and Mogens Nielsen. *Models for Concurrency.* Technical Report DAIMI PB-463, Computer Science Department, Aarhus University, Denmark, 1993. (Cited on pages 43, 45 and 53.)

[Winskel & Nielsen 1995] Glynn Winskel and Mogens Nielsen. *Models for Concurrency.* In S. Abramsky, Dov M. Gabbay and T. S. E. Maibaum, editors, Handbook of Logic and the Foundations of Computer Science, volume 4, chapter Models for Concurrency, pages 1–148. Oxford University Press, Oxford, UK, 1995. (Cited on page 43.)

[Winskel 1982] Glynn Winskel. *Event Structure Semantics for CCS and Related Languages.* In Proceedings of the 9th Colloquium on Automata, Languages and Programming, pages 561–576, London, UK, 1982. Springer-Verlag. (Cited on page 48.)

[Winskel 1986] Glynn Winskel. *Event Structures*. In Brauer et al. [Brauer *et al.* 1987], pages 325–392. (Cited on pages 15, 17, 43, 49, 53, 89 and 197.)

[Winskel 2011] Glynn Winskel. *Events, Causality and Symmetry*. The Computer Journal, vol. 54, no. 1, pages 42–57, 2011. (Cited on page 43.)

[Wodtke & Weikum 1997] Dirk Wodtke and Gerhard Weikum. *A Formal Foundation for Distributed Workflow Execution Based on State Charts*. In Proceedings of the 6th International Conference on Database Theory, pages 230–246, London, UK, 1997. Springer-Verlag. (Cited on page 123.)

[Workflow Management Coalition 1993] Workflow Management Coalition. *Workflow Management Coalition*, 1993. `http://www.wfmc.org/`. (Cited on page 4.)

[Workflow Management Coalition 2008] Workflow Management Coalition. *Process Definition Interface - XML Process Definition Language*. Webpage, October 2008. `http://www.wfmc.org/index.php?option=com_docman&task=doc_download&Itemid=72&gid=132`. (Cited on page 4.)

[Yi & Kochut 2004a] X. Yi and K.J Kochut. *Process composition of web services with complex conversation protocols*. In Design, Analysis, and Simulation of Distributed Systems Symposium at Adavanced Simulation Technology, 2004. (Cited on pages 123 and 124.)

[Yi & Kochut 2004b] Xiaochuan Yi and Krys Kochut. *A CP-nets-based Design and Verification Framework for Web Services Composition*. In ICWS'04, pages 756–760, 2004. (Cited on page 160.)

[Zielonka 1987] W. Zielonka. *Notes on finite asynchronous automata*. Informatique Théorique et Applications, vol. 21(2), pages 99–135, 1987. (Cited on page 124.)

[Zisman 1977] M. D. Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, Wharton School, University of Pennsylvania, 1977. (Cited on page 2.)