Matteo Zavatteri

Temporal and Resource Controllability of Workflows Under Uncertainty

Ph.D. Thesis

May 16, 2018

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Luca Viganò
Dipartimento di Informatica
Università degli Studi di Verona
Strada le Grazie 15, 37134 Verona
Italia
and
Department of Informatics
King's College London
Bush House, 30 Aldwych, London WC2B 4BG
UK

Co-advisor:
Prof. Carlo Combi
Dipartimento di Informatica
Università degli Studi di Verona
Strada le Grazie 15, 37134 Verona
Italia

Series N°: **TD-10-18**

Università degli Studi di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italia

to Luisa

Abstract

Workflow technology has long been employed for the modeling, validation and execution of business processes. A workflow is a formal description of a business process in which single atomic work units (tasks), organized in a partial order, are assigned to processing entities (agents) in order to achieve some business goal(s). Workflows can also employ workflow paths (projections with respect to a total truth value assignment to the Boolean variables associated to the conditional split connectors) in order (not) to execute a subset of tasks. A workflow management system coordinates the execution of tasks that are part of workflow instances such that all relevant constraints are eventually satisfied. *Temporal workflows* specify business processes subject to temporal constraints such as controllable or uncontrollable durations, delays and deadlines. The choice of a workflow path may be controllable or not, considered either in isolation or in combination with uncontrollable durations. *Access controlled workflows* specify workflows in which users are authorized for task executions and authorization constraints say which users remain authorized to execute which tasks depending on who did what. Access controlled workflows may consider workflow paths too other than the uncertain availability of resources (users, throughout this thesis). When either a task duration or the choice of the workflow path to take or the availability of a user is out of control, we need to verify that the workflow can be executed by verifying all constraints for any possible combination of behaviors arising from the uncontrollable parts. Indeed, users might be absent before starting the execution (static resiliency), they can also become so during execution (decremental resiliency) or they can come and go throughout the execution (dynamic resiliency). *Temporal access controlled workflows* merge the two previous formalisms by considering several kinds of uncontrollable parts simultaneously. Authorization constraints may be extended to support conditional and temporal features. A few years ago some proposals addressed the temporal controllability of workflows by encoding them into temporal networks to exploit "off-the-shelf" controllability checking algorithms available for them. However, those proposals fail to address temporal controllability where the controllable and uncontrollable choices of workflow paths may mutually influence one another. Furthermore, to the best of my knowledge, controllability of access controlled workflows subject to uncontrollable workflow paths and algorithms to validate and execute dynamically resilient workflows remain unexplored. To overcome these limitations, this thesis goes for exact algorithms by addressing temporal and resource controllability of workflows under uncertainty. I provide several new classes of (temporal) constraint networks and corresponding algorithms to check their controllability. After that, I encode workflows into these new formalisms. I also provide an encoding into instantaneous timed games to model static, decremental and dynamic resiliency and synthesize memoryless execution strategies. I developed a few tools with which I carried out some initial experimental evaluations.

To me they were nothing but constraints.

Table of Contents

1 **Overview** 1
- 1.1 Context and Motivations 1
- 1.2 Contributions 4
- 1.3 Scientific Publications 5
- 1.4 Organization 6

2 **Background** 9
- 2.1 Temporal Networks 9
  - 2.1.1 Simple Temporal Networks 9
  - 2.1.2 Simple Temporal Networks with Uncertainty 10
  - 2.1.3 Conditional Simple Temporal Networks 13
  - 2.1.4 Conditional Simple Temporal Networks with Uncertainty . . . 17
- 2.2 Timed Game Automata 19
  - 2.2.1 Controller Synthesis with UPPAAL-TIGA 21
  - 2.2.2 Dynamic Controllability of CSTNUs via TGAs 23
- 2.3 Constraint Networks and Directional Consistency 27
- 2.4 Periodic Time 30
  - 2.4.1 Gap-Order and Periodicity constraints 32
  - 2.4.2 From symbolic expressions to constraints 33
- 2.5 Temporal role-based access control models 33
- 2.6 Workflow Satisfiability and Resiliency 36

3 **Related work** 41
- 3.1 Planning and scheduling 41
- 3.2 Controllability of Workflows 46
- 3.3 Role based access control models, workflow satisfiability and resiliency 48

Part I Temporal Controllability

4 Simple Temporal Networks with Decisions . . . 55
4.1 Syntax . . . 55
4.2 STND-HSCC1 . . . 58
4.3 STND-HSCC2 . . . 59
4.4 Correctness . . . 61
4.5 Kappa: A Tool for STNDs . . . 62
4.6 Equivalence with DTP . . . 66
4.6.1 Encoding DTNs into STNDs . . . 68
4.6.2 Encoding STNDs into DTNs . . . 69
4.7 Conclusions . . . 71

5 Conditional Simple Temporal Networks with Uncertainty and Decisions . . . 73
5.1 Syntax . . . 76
5.2 Semantics . . . 76
5.3 Dynamic Controllability of CSTNUDs via TGAs . . . 80
5.3.1 Extension . . . 80
5.3.2 Optimizations . . . 81
5.3.3 An optimized encoding for checking DC of CSTNUDs . . . 83
5.4 Correctness and Complexity of the Encoding . . . 83
5.5 Expressiveness of CSTNUDs . . . 86
5.5.1 Encoding STNs into CSTNUDs . . . 87
5.5.2 Encoding STNUs into CSTNUDs . . . 87
5.5.3 Encoding CSTNs into CSTNUDs . . . 87
5.5.4 Encoding STNDs into CSTNUDs . . . 88
5.5.5 Encoding STNUDs into CSTNUDs . . . 88
5.5.6 Encoding CSTNDs into CSTNUDs . . . 88
5.5.7 Encoding CSTNUs into CSTNUDs . . . 89
5.6 Esse: A Tool for CSTNUDs . . . 89
5.7 Modeling Temporal Workflows Under Uncertainty . . . 97
5.7.1 Motivating Example . . . 98
5.7.2 Process Modeling Language . . . 98
5.7.3 Controllability of temporal workflows under uncertainty . . . 101
5.7.4 Encoding TWFs into CSTNUDs . . . 104
5.8 Conclusions . . . 109

Part II Resource Controllability

6 Constraint Networks Under Conditional Uncertainty . . . 117
6.1 Syntax . . . 117
6.2 Semantics . . . 120
6.3 Controllability Checking Algorithms . . . 122
6.3.1 Weak controllability checking . . . 123
6.3.2 Strong controllability checking . . . 124
6.3.3 Dynamic controllability checking . . . 125
6.4 Correctness of the Algorithms . . . 128

6.5 Zeta: A tool for CNCUs 129
6.6 Modeling access controlled workflows under uncertainty 138
6.6.1 Motivating Example 138
6.6.2 Process Modeling Language 140
6.6.3 Controllability of ACWFs under conditional uncertainty 141
6.6.4 Encoding ACWF into CNCUs 143
6.7 Conclusions 150

7 Workflow Resiliency 153
7.1 Motivating Example 153
7.2 Workflow Resiliency via Controller Synthesis 155
7.2.1 Static Resiliency 156
7.2.2 Decremental and Dynamic Resiliency 158
7.3 Correctness and Complexity of the Encodings 160
7.3.1 Static resiliency 160
7.3.2 Decremental resiliency 164
7.3.3 Dynamic resiliency 165
7.4 Erre: A Tool for Workflow Resiliency 166
7.5 Conclusions 174

Part III Temporal And Resource Controllability Together

8 Access Controlled Temporal Networks 179
8.1 Motivating Example 179
8.2 Syntax 180
8.3 Semantics 182
8.4 Encoding ACTNs into TGAs 186
8.4.1 Internal state 188
8.4.2 Predecessors and transition range limitations 188
8.4.3 Time Point Transitions 189
8.4.4 Game Interplay, Failing Transitions, and Winning path considering disjunctive constraints 190
8.5 Correctness and complexity of the encoding 191
8.6 Conclusions 195

9 CSTNUs with Resources 197
9.1 Motivating Example 197
9.2 Syntax 199
9.3 Semantics 202
9.4 Encoding CSTNURs into TGAs 205
9.4.1 Encoding Committable Resources into Dedicated Clocks 205
9.4.2 Encoding Resource Commitments into Circular Paths 206
9.4.3 Encoding Contingent Time Points into Contingent Circular Paths 213
9.5 Correctness and Complexity of the Encoding 213
9.6 Encoding CSTNURs into CDTNUs 219
9.7 A possible implementation with Uppaal-TIGA 223

9.8 TRBAC on top of CSTNURs (and ACTNs) 225
9.8.1 From granules to real time instants 225
9.8.2 From Periodic Expressions to STNs 226
9.8.3 Case study 229
9.9 Conclusions 230

10 Conclusions and Future Work 235

References 241

List of Tables

5.1 Encoding temporal workflows into CSTNUDs. 105

6.1 Labeled relational constraints of the CNCU in Figure 6.1. 119
6.2 Encoding access-controlled workflows into CNCUs. 144
6.3 Relations of the example in Figure 6.6. 144

9.1 Examples of periodic expressions. 226
9.2 Displacement, Periodicity and Granularity of the periodic expressions given in Table 9.1. 226

List of Figures

2.1 STN distance graph. 10
2.2 STNU distance graph. 11
2.3 CSTN labeled distance graph. 14
2.4 Streamlined CSTN. 16
2.5 CSTNU labeled distance graph. 17
2.6 Examples of TA and TGA. 21
2.7 Examples of (extended) TGAs with UPPAAL-TIGA. 22
2.8 TGA encoding the CSTNU in Figure 2.5. 25
2.9 Constraint Network and Directional Consistency. 29
2.10 Graphical representation of periodic time with calendars. 31
2.11 Role states in RBAC and TRBAC. 34
2.12 An example of Role Enabling Base. 36

4.1 STND and STN-projections. 57
4.2 Experimental evaluation with KAPPA. 66
4.3 Representing and encoding DTNs into STNDs. 68
4.4 Encoding STNDs into DTNs. 70

5.1 Example of uncontrollable CSTNU. 74
5.2 Three examples of CSTNUD. 75
5.3 TGA encoding the CSTNUD in Figure 5.2b. 82
5.4 A hierarchy of simple temporal networks. 86
5.5 Time and space analysis with ESSE. 95
5.6 Experimental evaluation with ESSE. 97
5.7 Example of temporal workflow in BPMN. 99
5.8 A fragment of a structured (temporal) BPMN. 100
5.9 CSTNUD corresponding to the workflow in Figure 5.9. 107

6.1 Example of binary CNCU. 118
6.2 Experimental evaluation with ZETA (time). 136
6.3 Experimental evaluation with ZETA (space). 137
6.4 Example of access controlled workflow in BPMN. 139
6.5 A fragment of a structured (access controlled) BPMN. 140

6.6 CNCU encoding the ACWF in Figure 6.4. 145

7.1 A simplification of a loan origination process. 154
7.2 Relational constraints of the ACWF in Figure 7.1. 154
7.3 Skeletons of the encodings for static, decremental and dynamic resiliency. 156
7.4 Static resiliency in UPPAAL-TIGA. 159
7.5 Decremental resiliency in UPPAAL-TIGA. 161
7.6 Dynamic resiliency with UPPAAL-TIGA. 162
7.7 Experimental evaluation with ERRE (time). 171
7.8 Experimental evaluation with ERRE (space). 173

8.1 Official STEMI guidelines. 180
8.2 Access controlled temporal network. 183
8.3 TGA equivalent to the ACTN in Figure 8.2. 187

9.1 Temporal workflow modeling a round-trip flight. 198
9.2 An augmented CSTNU. 200
9.3 CSTNUR modeling the temporal plan in Figure 9.1. 203
9.4 Modeling resource commitments. 208
9.5 Circular paths modeling the authorized execution of the non-contingent time points of the CSTNUR in Figure 9.3. 211
9.6 Encoding RRCs specifying TEs of Type 2. 212
9.7 Modeling the executions of contingent time points. 214
9.8 Fragment of CSTNUR. 220
9.9 Modeling and validation of Figure 9.3 with UPPAAL-TIGA. 224
9.10 Graphical representations of periodic expressions. 227
9.11 Modeling role temporalities of a TRBAC by means of an STN. 228
9.12 Access-controlled workflow. 229
9.13 The Role Enabling Base of the case study. 230
9.14 CSTNUR connected to TRBAC. 231
9.15 ACTN connected to TRBAC. 232
9.16 Deciding dynamic controllability of CSTNURs via TGAs. 233

Listings

2.1 Early execution strategy for the STNU in Figure 2.2. 12
2.2 Early execution strategy for the CSTN in Figure 2.3. 16
2.3 Early execution strategy for the CSTNU in Figure 2.5. 19
2.4 Output of the analysis for Figure 2.7a. 22
2.5 Output of the analysis for Figure 2.7b. 23
4.1 KAPPA's help screen. 62
4.2 The specification of Figure 4.1a in the input language of KAPPA. . . . 63
4.3 Analyzing and verifying the STND in Figure 4.1a with KAPPA. Z is a special time point that must occur before any other. This is to avoid getting schedules with negative numbers. 64
5.1 ESSE's help screen. 89
5.2 Specification of Figure 5.2b. 90
5.3 Example of DC-checking for Figure 5.2a, Figure 5.2b and Figure 5.2c. 92
5.4 Three random executions for Figure 5.2b 92
5.5 Specification of Figure 5.9. 106
5.6 Random executions of the CSTNUD in Figure 5.9. 109
6.1 ZETA's help screen. 129
6.2 Specification of Figure 6.1 in ZETA's input language. 131
6.3 WC, SC and DC-checking of Figure 6.1 with ZETA. 132
6.4 Execution simulations for Figure 6.1 (weak controllability). 132
6.5 Execution simulations for Figure 6.1 (dynamic controllability). 133
6.6 Specification of Figure 6.6. 146
6.7 Random executions for Figure 6.6 (weak controllability) 147
6.8 Random executions for Figure 6.6 (dynamic controllability) 149
7.1 ERRE's help screen. 167
7.2 Specification of Figure 7.1. 168
7.3 Execution simulations for static, decremental and dynamic resiliency. 169

1

Overview

In this chapter I first discuss the context and motivation of this work and the contributions of my thesis. Then, I summarize the scientific publications that this thesis has led to and the organization of the contributions.

1.1 Context and Motivations

Workflow technology has emerged as one of the leading technologies for modeling, (re)designing and executing business processes in several different application domains. For instance, workflows have been used to model processes for industrial research & development, manufacturing, energy distribution, banking processes, critical infrastructures and healthcare [34, 89]. The conceptual modeling of workflows underlying business processes has been receiving increasing attention over the last years and many technical aspects have been discussed, including flexibility, structured vs. unstructured modeling, change management, authorization models, and temporal features and constraints (see, e.g., [34, 109, 123]).

A *workflow schema* (or simply *workflow*) is a formal description of a business process in which single atomic work units (*tasks*), organized in a partial order, are assigned to processing entities (*agents*) in order to achieve some business *goal(s)* [68]. A *workflow management system* must coordinate the execution of tasks to obtain workflow instances.

Recently, attention has been devoted in particular to the issue of expressing *temporal features* of workflows, such as task-duration constraints, temporal constraints between non-consecutive tasks, delays, deadlines and so on [34]. Moreover, properties of such temporal workflow models have been defined and analyzed. One of the most interesting properties is that of *dynamic controllability*, which ensures that a workflow can be executed satisfying all the given temporal constraints without the workflow management system restricting and/or controlling task durations but only assuming that each duration is within a specified range (*temporal uncertainty*) [34].

The authors of [34] also tackled dynamic controllability under another kind of uncertainty, *conditional uncertainty*, represented by the fact that some subsets of tasks have to be executed if and only if some conditions (abstracted as

Boolean propositions) are true. Similarly to what happens for uncontrollable task durations, the truth-value assignment to such propositions is *out of control*. For instance, when a patient enters the emergency room, the severity of his condition is not known a priori but it is established by a physician, while the workflow is being executed. Since such a condition discriminates which tasks have, or have not, to be executed (i.e., the choice of the workflow path), the workflow management system must be able to get to the end of the workflow satisfying all relevant temporal constraints regardless of which tasks have to be executed and which task durations have to be satisfied. In [34], the authors do not consider workflow instances specifying both controllable and uncontrollable workflow paths such that the choice of a controllable workflow path may exclude the choice of an uncontrollable one and vice versa. That is, they do not address *fallback temporal plans*, i.e., plans in which making a decision may exclude some uncontrollable part whose behavior risks violating some constraint (see, for example, [126]).

Workflows also deal with the management of associated resources in order to complete business processes. This thesis works, from a security point of view, with the most trivial of resources: *users*. When we talk about security in the business process context, we must also talk about access control models, security policies and authorization constraints (which act as primitives for security policies).

Therefore, an *access-controlled workflow* extends a classical workflow by adding users and authorization constraints. Users are authorized for tasks whereas authorization constraints say which users remain authorized for which tasks depending on who did what. *Role-based access control models (RBAC, [113])* put another layer of security on top of access controlled workflows injecting the concept of *role*, which acts as an interface between users and tasks saying, intuitively, "who can do what". For example, in a financial context, a *clerk* is authorized to process a loan request, but he is not authorized to sign the contract at the end of the process as only *managers* can do so. RBAC models can specify several kinds of constraints involving roles (e.g., mutual exclusivity, hierarchy, etc.), but they all fail to model constraints at user level such as, for example, the well-known *separation of duties* (*SoD*) and *binding of duties* (*BoD*) [31, 111]. A SoD (resp., BoD) between two tasks says that the users executing such tasks must be different (resp., equal).

Some proposals attempted at extending RBAC models to address such constraints, e.g., [14, 15], leading to a natural question: *Does there exist an assignment of tasks to users satisfying all constraints?*, or more formally, *Is the workflow satisfiable?* If it is, then it means that at least a *static plan*, precomputed before starting, to execute the workflow exists. If it is not, then either we decide not to execute the workflow or we accept that any possible execution will violate at least one constraint (and then we could, for example, look for a "least bad" plan [47] maximizing the number of satisfied constraints). Thus, the *workflow satisfiability problem* (*WSP*, [122, 123]) is a *constraint satisfaction problem* (*CSP*) where variables model tasks and domains model authorized users. Although some techniques have been provided to solve the WSP efficiently (e.g., [49]), a CSP remains in general NP-hard [52]. This is due to the non-monotonicity of the relations employed (e.g., $\neq$ [52]).

When an access controlled workflow does not specify any uncontrollable part workflow satisfiability is enough to synthesize a valid plan. Instead, when some part

is out of control (e.g., the choice of the workflow path or the absence of users), we need, as for temporal workflows discussed above, a controllability approach to decide in real time which users to commit to which tasks. For example, consider an access controlled workflow under conditional uncertainty. In this workflow, the choices of the workflow paths to take are out of control (or by abusing language we say that these workflow paths are uncontrollable). Differently from the WSP, the assignment of a user to a task might not be precomputed before starting as the workflow may in general specify different authorization constraints for different workflow paths involving the same users for some common tasks which must be considered in any execution. In that case, we must make this assignment while executing, typically after having full information on which workflow path we will go through (online planning).

Another controllability problem in the context of workflows with resources is the *workflow resiliency problem (WRP)*, i.e., a *dynamic* WSP coping with the absence of users. If a workflow is resilient, it is of course satisfiable, but the vice versa does not hold. A few years ago, Wang and Li defined three levels of resiliency: *static* (level 1), *decremental* (level 2) and *dynamic* (level 3) [122, 123]. In static resiliency, up to k users might be absent before the execution starts and never become available for that execution. In decremental resiliency, up to k users might be absent before or during execution and, again, they never become available for that execution. In dynamic resiliency, up to k (possibly different) users might be absent at any time and they may in general turn absent and available continuously, before or during the execution. Much work has been carried out to tackle static resiliency, little for decremental and, to the best of my knowledge, nothing for dynamic.

Finally, we can of course consider variants of the previously discussed aspects. For instance, a workflow employing users and temporal constraints could specify a *temporal separation (or binding) of duties*. A temporal SoD says that a user is allowed to carry out two tasks provided a further temporal constraint is satisfied. For example, a surgeon, who has just carried out a 4-hour intervention, is allowed to do another one only after resting from 2 to 4 hours. Likewise, an aircraft pilot must rest for at least 10 hours after a transatlantic flight. These temporal constraints must be considered *in conjunction* with access control as there is a mutual influence. However, when everything is under control, these constraints boil down to normal disjunctions for which a satisfiability approach is enough. The interesting part is when some part is, again, out of control. For example, consider again the transatlantic flight and suppose that once the aircraft lands in America, it will take off again after 12 hours after the expected landing time (so potentially the same pilot is fine for the return flight). However, the exact duration of the outbound flight is uncontrollable. Suppose that it takes normally 10 hours. Once boarding is complete, the take off could be delayed for extreme weather conditions and related safety procedures such as, for example, deicing. Deicing is the process of removing snow and ice from the plane surfaces (especially wings) by "power washing" the aircraft with chemicals which also remain on the surfaces in order to prevent the reformation of the ice. If the deicing process takes 3 hours[1], the

[1] Actually, deicing an aircraft does not take 3 hours, but since all leaving aircrafts have to do so following the departure scheduling, each plane queues for its turn.

flight will land after 13 hours since boarding. As a result, the next take off will be scheduled after 9 (and no longer 12 hours), so the same pilot is not going to be fine.

To the best of my knowledge security policies involving temporal, conditional and resource uncertainty still need to be explored in depth.

When facing uncontrollable parts we can in general act in three main ways:

1. We assume that we know in advance how the uncontrollable part will behave and make sure that a (possibly different) strategy to operate on the controllable part exists.
2. We assume that we have a fixed strategy operating on the controllable part always the same way no matter how the uncontrollable one will behave.
3. We assume that we have a strategy operating (possibly differently) on the same controllable part making decisions in real time depending on how the uncontrollable part is behaving.

These are the intuitions behind the three main kinds of controllability: *weak* (for presumptuous), *strong* (for anxious) and *dynamic* (for grandmasters).

1.2 Contributions

Towards the unexplored directions mentioned in Section 1.1 my contributions in this thesis are many-fold. Here, I give only a high level overview of them, postponing a more detailed description to the specific parts of the thesis.

1. I address temporal controllability of workflows specifying controllable and uncontrollable workflow paths and uncontrollable task durations. This part relies on temporal constraint networks. I provide *conditional simple temporal networks with uncertainty and decisions (CSTNUDs)* as a new temporal network formalism and then an encoding from temporal workflows into CSTNUDs. I also address *simple temporal networks with decisions (STNDs)*, a subclass of CSTNUDs equivalent to (but more compact than) DTP. I provide Esse and Kappa, two tools for CSTNUDs and STNDs, respectively, with which I carry out a few experimental evaluations.
2. I address resource controllability of workflows specifying uncontrollable workflow paths. This part relies on constraint networks. I provide *constraint networks under conditional uncertainty (CNCUs)* as a new formalism of constraint networks able to model conditional uncertainty. Then, I provide an encoding from access controlled workflows into CNCUs. After that, I also address workflow resiliency via real-time controller synthesis for timed game automata. I provide Zeta and Erre, two tools for CNCUs and workflow resiliency, respectively, with which I carry out a few experimental evaluations.
3. I address temporal and resource controllability together. This part relies on further new extensions of temporal networks whose dynamic controllability is checked via controller synthesis for the corresponding timed game automata. I provide *access controlled temporal networks (ACTNs)* and *conditional simple temporal networks with uncertainty and resources (CSTNURs)* in order to model temporal security policies. I also show how the temporal constraints of

a temporal role based access control model (TRBAC) can be represented as a simple temporal network to be connected to the temporal network modeling the workflow in order to understand if the access controlled workflow can be executed.

These contributions fall in the areas of *constraint satisfaction*, *uncertainty in AI*, *planning and scheduling*, *algorithms*, *business process management* and *security*.

1.3 Scientific Publications

The work of this thesis has led to the following published papers and the results of my work will lead to a few more to be submitted soon. In what follows I summarize titles, contents and related bib entries.

1. Carlo Combi, Luca Viganò, and Matteo Zavatteri. Security constraints in temporal role-based access-controlled workflows. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY 2016. ACM, 2016 [41]

   Summary. *In this paper we provide a mapping between workflow models and simple temporal networks with uncertainty (STNUs) to properly manage temporal constraints of workflows. We also provide a mapping between role temporalities and simple temporal networks (STNs). We discuss how to connect the two resulting networks to make explicit who can do what, when and we define Security Constraints (SCs) along with Security Constraint Propagation Rules (SCPRs) to prevent users from doing unauthorized actions. These rules allow the system to propagate security constraints at runtime depending on what is going on. We provide an algorithm to check whether a set of SCPRs is safe, and extend an existing execution algorithm for it to take into account these new security aspects.*

2. Carlo Combi, Roberto Posenato, Luca Viganò, and Matteo Zavatteri. Access controlled temporal networks. In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART 2017)*, pages 118–131. INSTICC, ScitePress, 2017 [40]

   Summary. *In this paper, we define Access-Controlled Temporal Networks (ACTNs) as an extension of Conditional Simple Temporal Networks with Uncertainty (CSTNUs) by adding users and authorization constraints that must be considered together with temporal constraints to model a temporal access control. We show that the dynamic controllability checking can be done via Timed Game Automata and we provide experimental results using* UPPAAL-TIGA *on a concrete real-world case study.*

3. Matteo Zavatteri, Carlo Combi, Roberto Posenato, and Luca Viganò. Weak, strong and dynamic controllability of access-controlled workflows under conditional uncertainty. In *Business Process Management (BPM 2017)*, 2017 [127]

   Summary. *In this paper, we address controllability analysis for access controlled workflows under conditional uncertainty. We define weak, strong and dynamic controllability of ACWFs under conditional uncertainty, we present*

algorithmic approaches to address each of these types of controllability, and we synthesize execution strategies that specify which user has been (or will be) assigned to which task.

4. Massimo Cairo, Carlo Combi, Carlo Comin, Luke Hunsberger, Roberto Posenato, Romeo Rizzi, and Matteo Zavatteri. Incorporating decision nodes into conditional simple temporal networks. In S. Schewe, T. Schneider, and J. Wijsen, editors, *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, LIPIcs, pages 9:1–9:17, 2017 [19]

   Summary. *In this paper, we incorporate decision time points into conditional simple temporal networks. A decision time-point is like an observation time-point with the difference that the truth value assignment to the associated proposition is under control. The resulting network is called a CSTN with Decisions (CSTND). We prove that the problem of determining whether any given CSTND is dynamically consistent is PSPACE-complete. We present algorithms that address two sub-classes of CSTNDs: (1) those that contain only decision time-points; and (2) those in which all decisions are made before execution begins.*

5. Matteo Zavatteri. Conditional simple temporal networks with uncertainty and decisions. In *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, volume 90 of *LIPIcs*, 2017 [126]

   Summary. *In this paper, I propose conditional simple temporal networks with uncertainty and decisions (CSTNUDs) which introduce decision time points into the specification in order to operate on a conditional part under control. I model the dynamic controllability checking (DC-checking) of a CSTNUD as a two-player game in which each player makes his moves in his turn at a specific time instant. I give an encoding into timed game automata for a sound and complete DC-checking. I also synthesize memoryless execution strategies for CSTNUDs proved to be DC.*

6. Matteo Zavatteri and Luca Viganò. Constraint networks under conditional uncertainty. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART 2018)*, pages 41–52. INSTICC, SciTePress, 2018 [128]

   Summary. *In this paper, we propose constraint networks under conditional uncertainty (CNCUs), and we define weak, strong and dynamic controllability of a CNCU. We provide algorithms to check each of these types of controllability and discuss how to synthesize (dynamic) execution strategies that drive the execution of a CNCU saying which value to assign to which variable depending on how the uncontrollable part behaves. We provide* ZETA, *a tool that we developed for CNCUs.*

1.4 Organization

I divided this thesis in three main parts according to the three main contributions highlighted in Section 1.2. I discuss essential background in Chapter 2 and related work in Chapter 3 and then,

1. I address *temporal controllability* in Part I, where Chapter 4 summarizes part of the work in [19] and also provides new contributions, whereas Chapter 5 summarizes the work in [126] with Section 5.7 as a new contribution.
2. I address *resource controllability* in Part II, where Chapter 6 summarizes the work in [128] with Section 6.6 summarizing part of the work in [127] and providing new contributions along with Chapter 7.
3. I address *temporal and resource controllability together* in Part III, where Chapter 8 summarizes the work in [40], Chapter 9 provides new contributions starting from the work in [41] and also adapting it to Chapter 8 as a minor contribution.

Chapter 10 draws conclusions and discusses future work.

2

Background

In this chapter I provide essential background on temporal networks, timed game automata, periodic time, temporal role-based access controlled models and on the workflow satisfiability and resiliency problems.

2.1 Temporal Networks

In this section I provide essential background on the extensions of simple temporal networks I use in most of this thesis.

2.1.1 Simple Temporal Networks

A simple temporal network (STN, [53]) is a formalism able to model temporal plans in which all components are under control. For STNs, consistency analysis is enough to validate the temporal plan.

Definition 2.1 (STN). *A* simple temporal network (STN) *is a pair* $\langle \mathcal{T}, \mathcal{C} \rangle$*, where:*

- $\mathcal{T} = \{X, \dots\}$ *is a finite set of time points (continuous variables).*
- $\mathcal{C} = \{(Y - X \leq k), \dots\}$ *is a finite set of constraints, where* $X, Y \in \mathcal{T}$*,* $k \in \mathbb{R} \cup \pm\infty$*. If* $(Y - X \leq k) \notin \mathcal{C}$*, then* $k = \infty$*.*

An STN is consistent if there exists a consistent schedule (see Definition 6.5).

Definition 2.2 (Schedule). *A schedule is a mapping* $S \colon \mathcal{T} \to \mathbb{R}$ *assigning real values to time points such that if* X *is executed before* Y*, then* $S(X) \leq S(Y)$ *(time points are not scheduled in the past). A schedule is consistent if the assignments it makes satisfy all constraints (i.e., for each* $(Y - X \leq k) \in \mathcal{C}$*,* $S(Y) - S(X) \leq k$ *holds).*

Before proceeding, I point out an assumption regarding constraints sharing the same time points (in the same order).

Definition 2.3 (Tightest constraint, STN). *Given a set of constraints*

$$\mathcal{C} = \{(Y - X \leq k_1), (Y - X \leq k_2), \dots, (Y - X \leq k_n)\}$$

the tightest constraint is $(Y - X \leq k)$ *where* $k = \min\{k_i \mid (Y - X \leq k_i) \in \mathcal{C}\}$*.*

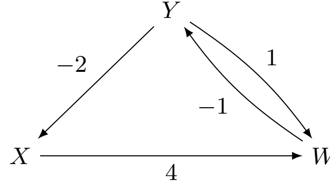


Fig. 2.1: STN distance graph.

For example, if

$$\mathcal{C} = \{(Y - X \leq 4), (Y - X \leq -1), (Y - X \leq -2)\}$$

then $(Y - X \leq -2)$ is the tightest constraint. When more constraints involve the same two time points (in the same position[1]) I always keep the tightest one.

I represent an STN as a directed weighted graph (called *distance graph*) where an edge $X \rightarrow Y$ labeled by k models $Y - X \leq k$. If $Y \rightarrow X$ has a negative weight $-k$, then it means that Y must be executed after minimum k since X (delay). If $X \rightarrow Y$ has a positive weight k, then it means that Y must be executed within k since X (deadline)[2].

For example Figure 2.1 shows the distance graph of $\mathcal{Z} = \langle \mathcal{T}, \mathcal{C} \rangle$, where:

- $\mathcal{T} = \{X, Y, W\}$
- $\mathcal{C} = \{\underbrace{(X - Y \leq -2)}_{1}, \underbrace{(W - Y \leq 1), (Y - W \leq -1)}_{2}, \underbrace{(W - X \leq 4)}_{3}\}$

In other words, $\mathcal{Z}$ specifies three time points X, Y and W requiring that:

1. Y must be executed after minimum 2 since X.
2. W must be executed after minimum 1 and within 1 since Y.
3. W must be executed after maximum 4 since X.

Consistency of STNs is in PTIME and can be tested by hunting down negative cycles in the corresponding directed weighted graph representation [53]. An early execution of an STN consists of finding a schedule executing the time points as soon as possible (e.g., by using Floyd-Warshall [53]).

The STN in Figure 2.1 is consistent. A possible consistent schedule is:

$$S(X) = 0, S(Y) = 2, S(W) = 3.$$

2.1.2 Simple Temporal Networks with Uncertainty

A simple temporal network with uncertainty (STNU) [104] augments an STN with a set of contingent links in order to model uncontrollable durations. A contingent link is a pair of distinct time points specifying a range of allowed values between their distance. One of these time points is called *activation time point* and it

[1] There is no tightest constraints for $\mathcal{C} = \{(Y - X \leq k_1), (X - Y \leq k_2)\}$.

[2] However, nothing prevents Y from being executed before X. Should this be the case, the constraints will trivially hold.

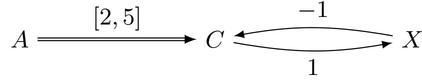


Fig. 2.2: STNU distance graph.

is under control, whereas the other one is called *contingent time point* and it is not. The real value assignment to the contingent one depends on the behavior of unpredictable external events which are only observed to occur while executing the network.

Definition 2.4 (STNU). *A* simple temporal network with uncertainty (STNU) *is a tuple* $\mathcal{Z} = \langle \mathcal{T}, \mathcal{L}, \mathcal{C} \rangle$*, where:*

- $\mathcal{T}$ *and* $\mathcal{C}$ *are the same of those given for STNs (Definition 2.1).*
- $\mathcal{L} = \{(A, x, y, C), \dots\}$ *is a finite set of contingent links, where* $A, C \in \mathcal{T}$ *(A is the activation time point, C the contingent one),* $A \neq C$ *and* $0 < x < y < \infty$ *(*$x, y \in \mathbb{R}$*).*

An STNU is well-defined iff for any pair $(A_1, x_1, y_1, C_1), (A_2, x_2, y_2, C_2) \in \mathcal{L}$ *such that* $A_1 \neq A_2$*, we have that* $C_1 \neq C_2$*.*

Thus, an STNU models an (infinite) family of STNs each obtained by fixing a duration for each contingent link.

The graphical representation of an STNU extends that of an STN by dividing the set of edges in *single edges* and *double edges*. A single edge $X \rightarrow Y$ labeled by k still represents a constraints $(Y - X \leq k) \in \mathcal{C}$, whereas a double edge $A \Rightarrow C$ labeled by $[x, y]$ represents the contingent link (A, x, y, C). Figure 2.2 shows the STNU distance graph of $\mathcal{Z} = \langle \mathcal{T}, \mathcal{L}, \mathcal{C} \rangle$, where:

- $\mathcal{T} = \{X, A, C\}$
- $\mathcal{L} = \{(A, 2, 5, C)\}$
- $\mathcal{C} = \{\underbrace{(C - X \leq -1)}_{1}, \underbrace{(X - C \leq 1)}_{2}\}$

In other words, $\mathcal{Z}$ specifies three time points X, A, C and a contingent link $(A, 2, 5, C)$ requiring that:

1. X must be executed minimum one 1 since C
2. X must be executed within 1 since C

Let S be a schedule for $\mathcal{Z}$. Once we execute A, we can only observe the execution of C which is guaranteed to occur such that $S(C) - S(A) \in [2, 5]$.

A schedule for an STNU still involves all time points, but with the difference that we cannot specify assignments having the form $S(C)$ where C is a contingent time point. We can only specify assignments $S(X)$, where X is a non-contingent time point.

Differently from an STN (where consistency is enough), in what follows I give the definitions for weak, strong and dynamic controllability of an STNU which help address the satisfaction of constraints under uncertainty. Controllability analysis is necessary for STNUs and any another kind of network specifying uncontrollable parts.

Definition 2.5 (Weak controllability, STNU). *An STNU is weakly controllable if whenever a schedule is defined for contingent time points only, we can complete such a schedule by assigning real values to all non-contingent time points such that the resulting schedule is consistent.*

The STNU in Figure 2.2 is weakly controllable. Suppose that $S(C) \in [2, 5]$ is known in advance. We can build a consistent and complete schedule by setting $S(A) = 0$ and $S(X) = S(C) + 1$.

Dealing with weak controllability is quite complex as it always requires one to predict how all uncontrollable parts will behave before starting the execution. This leads us to consider the opposite case in which we want to synthesize a schedule defined for non-contingent time points only which will be always consistent no matter how the assignments to contingent time points will complete this schedule. Thus, the second kind of controllability is strong controllability.

Definition 2.6 (Strong controllability, STNU). *An STNU is strongly controllable if there exists a schedule defined for non-contingent time points only such that the schedule is always consistent regardless of the assignments to the contingent time points that will complete it.*

The STNU in Figure 2.2 is not strongly controllable. Indeed, there is no way to precompute $S(X)$ such that S will be consistent for any $S(C)$. The problem lies in the constraints involving X and C. In a nutshell, the scheduling of X must occur exactly 1 since C, but C can occur any time in the interval $[2, 5]$ which makes impossible to set $S(X)$ to a fixed $k \in \mathbb{R}$ once we have set $S(A)$.

Strong controllability is, however, "too strong". If an STNU is not strongly controllable, it could be still executable by refining the schedule in real time depending on what durations the contingent links take. To achieve this purpose, I provide the definition of dynamic controllability.

Definition 2.7 (Dynamic controllability, STNU). *An STNU is dynamically controllable if a consistent schedule is generated in real time by assigning (possibly different) real values to non-contingent time points depending on what assignments to the contingent ones are observed in real time.*

The STNU in Figure 2.2 is dynamically controllable. The execution *strategy* shown in Listing 2.1 generates a schedule S such that X will be scheduled depending on when C occurs.

Listing 2.1: Early execution strategy for the STNU in Figure 2.2.

```
S(A) = 0
wait for C to execute (i.e., for the environment to set S(C)
S(X) = S(C) + 1
```

As pointed out in [104], it is easy to see that

$$\textit{Strong controllability} \Rightarrow \textit{Dynamic controllability} \Rightarrow \textit{Weak controllability}$$

2.1.3 Conditional Simple Temporal Networks

A Conditional Simple Temporal Network (CSTN) [75] (formerly CTP [118]) extends an STN (but not an STNU) by adding *observation time points* to model conditional temporal plans. Each observation time point is associated to a Boolean proposition. Time points and constraints may be labeled by *labels* (i.e., conjunctions of literals) saying for which *scenarios* (truth value assignments to the propositions) they must appear in the solution. Thus, a CSTN models a family of STNs each obtained as a *projection* of the initial CSTN onto a *scenario*.

Given a set $\mathcal{P}$ of Boolean propositions, a *label* $\ell = \lambda_1 \dots \lambda_n$ is any finite conjunction of literals λ_i, where a literal is either a proposition p (positive literal) or its negation $\neg p$ (negative literal). The *empty label* is denoted by $\boxdot$. The *label universe of* $\mathcal{P}$, denoted by $\mathcal{P}^*$, is the set of all possible (consistent) labels drawn from $\mathcal{P}$; e.g., if $\mathcal{P} = \{p, q\}$, then $\mathcal{P}^* = \{\boxdot, p, q, \neg p, \neg q, pq, p\neg q, \neg pq, \neg p\neg q\}$. To ease reading I will often omit the $\wedge$ connector when the context allows and write $pq\neg r\dots$ instead of $p \wedge q \wedge \neg r \wedge \dots$. Two labels $\ell_1, \ell_2 \in \mathcal{P}^*$ are *consistent* if and only if their conjunction $\ell_1 \wedge \ell_2$ is satisfiable. A label ℓ_1 *entails* a label ℓ_2 (written $\ell_1 \Rightarrow \ell_2$) if and only if all literals in ℓ_2 appear in ℓ_1 too (i.e., if ℓ_1 is more *specific* than ℓ_2). A label ℓ_1 *falsifies* a label ℓ_2 iff $\ell_1 \wedge \ell_2$ is inconsistent. For instance, if $\ell_1 = p\neg q$ and $\ell_2 = p$, then ℓ_1 and ℓ_2 are consistent since $p\neg qp$ is satisfiable, and ℓ_1 entails ℓ_2 since $p\neg q \Rightarrow p$.

Definition 2.8 (CSTN). *A* conditional simple temporal network (CSTN) *is a tuple* $\mathcal{Z} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ *where*

- $\mathcal{T}$ *is the same of that given for STNs and STNUs (Definition 2.1, Definition 2.4).*
- $\mathcal{OT} \subseteq \mathcal{T} = \{P?, Q?, \dots\}$ *is a finite set of observation time points*
- $\mathcal{P} = \{p, q, \dots\}$ *is a finite set of Boolean propositions*
- $O\colon \mathcal{OT} \to \mathcal{P}$ *is a bijection assigning a unique proposition to each observation time point*
- $L\colon \mathcal{T} \to \mathcal{P}^*$ *is a mapping assigning a label to each time point*
- $\mathcal{C} = \{(Y - X \le k, \ell), \dots\}$ *is a finite set of labeled constraints, where* $X, Y \in \mathcal{T}$, $k \in \mathbb{R} \cup \pm\infty$ *and* $\ell \in \mathcal{P}^*$. *If* $(Y - X \le k, \ell) \notin \mathcal{C}$ *for some* $\ell \in \mathcal{P}^*$, *then* $k = \infty$ *(for* that *label)*

Definition 2.9 (Scenario, CSTN). *A* scenario *for a CSTN* $\mathcal{Z} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ *is a mapping* $s\colon \mathcal{P} \to \{\top, \bot\}$ *assigning a truth value to each proposition in* $\mathcal{P}$. *A scenario* s satisfies *a label* ℓ *(written* $s \models \ell$*) if* ℓ *valuates true under the following interpretation given by* s*:*

1. $s \models \lambda$ *iff* $(\lambda = p \wedge s(p) = \top)$ *or* $(\lambda = \neg p \wedge s(p) = \bot)$,
2. $s \models \ell$ *iff* $s \models \lambda_1$ *and* $\dots$ *and* $s \models \lambda_n$ *for* $\ell = \lambda_1 \dots \lambda_n$.

The graphical representation of a CSTN extends that of an STN (but not that of an STNU) by labeling nodes and constraints with labels. An edge $X \to Y$ labeled by $\langle k, \ell \rangle$ represents the labeled constraint $(Y - X \le k, \ell) \in \mathcal{C}$. I show the labels of nodes below them.

Figure 2.3 shows the labeled distance graph of $\mathcal{Z} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{C} \rangle$, where:

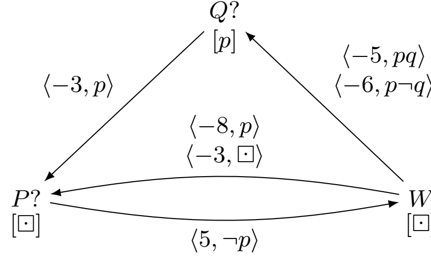


Fig. 2.3: CSTN labeled distance graph.

- $\mathcal{T} = \{P?, Q?, W\}$
- $\mathcal{OT} = \{P?, Q?\}$
- $\mathcal{P} = \{p, q\}$
- $O(p) = P?$ and $O(q) = Q?$
- $L(P?) = L(W) = \boxdot$ and $L(Q?) = p$.
- $\mathcal{C} = \{\underbrace{(P? - Q? \leq -3, p)}_{1}, \underbrace{(Q? - W \leq -5, pq)}_{2}, \underbrace{(Q? - W \leq -6, p\neg q)}_{3},$
$\underbrace{(P? - W \leq -3, \boxdot)}_{4}, \underbrace{(P? - W \leq -8, p)}_{5}, \underbrace{(W - P? \leq 5, \neg p)}_{6}\}$

In other words, $\mathcal{Z}$ specifies three time points $P?, Q?$ and W requiring that:

1. $Q?$ must be executed if $s(p) = \top$ and after minimum 3 since $P?$.
2. W must be executed after minimum 5 since $Q?$ (if $s(p) = s(q) = \top$).
3. W must be executed after minimum 6 since $Q?$ (if $s(p) = \top$ and $s(q) = \bot$).
4. W must be executed after minimum 3 since $P?$ (always)
5. W must be executed after minimum 8 since $P?$ (if $s(p) = \top$).
6. W must be executed within 5 since $P?$ (if $s(p) = \bot$).

Many $\langle k, \ell \rangle$ can be specified for the same $X \to Y$ provided their ℓ are different (e.g., $W \to P?$ in Figure 2.3). Again, if two labels are equal, I keep the smallest k.

A label ℓ labeling a time point or a constraint is *honest* if for each literal p or $\neg p$ in ℓ we have that $\ell \Rightarrow L(P?)$, where $P? = O(p)$ is the observation time point associated to p; ℓ is dishonest otherwise. For example, consider $W \to Q?$ labeled by $\langle -1, pq \rangle$ in Figure 2.3. The constraint applies only if $s(p) = s(q) = \top$. However, the truth value of q is set (by the environment) upon the execution of $Q?$, which in turn is relevant iff p was *previously* assigned true (as $L(Q?) = p$). Thus, an honest ℓ containing q or $\neg q$ should also contain p. A label on a constraint is *coherent* if it entails the labels of all variables in the scope of the constraint (e.g, p labeling the constraint $Q? \to P?$). Label honesty and coherence say when CSTNs are well-defined [75, 126]. Formally:

Definition 2.10 (Well-definedness). *A CSTN is well-defined if*

- *for each $X \in \mathcal{T}$ and any $\{p, \neg p\} \in L(X)$, we have that $L(X) \Rightarrow L(O(p))$ and $(O(p) - X \leq \epsilon) \in \mathcal{C}$ (time point label honesty), and*
- *$\ell \Rightarrow L(Y) \wedge L(X)$ for each $(Y - X \leq k, \ell) \in \mathcal{C}$ (constraint label coherence), and $\ell \Rightarrow L(O(p))$ for each literal $\{p, \neg p\} \in \ell$ (constraint label honesty)*

The CSTN in Figure 2.3 is well-defined. Since CSTNs express uncontrollable parts (truth value assignments to the propositions) we still need a controllability approach. In what follows, I provide the definitions of weak, strong and dynamic controllability as I did for STNUs but with respect to scenarios.

A schedule for a CSTN still involves all time points, but with the difference that we do not have to define $S(X)$ when the scenario containing the observations we made so far does not satisfy $L(X)$ (if we do so, the verification of constraints will ignore that assignment).

Definition 2.11 (Consistent scenario, CSTN). *A scenario s is consistent for a CSTN $\mathcal{Z} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ if there exists a schedule S whose domain consists of all time points $X \in \mathcal{T}$ such that $s \models L(X)$ and such that for each $(Y - X \leq k, \ell)$, where $s \models \ell$[3], $S(Y) - S(X) \leq k$ holds.*

Definition 2.12 (Weak controllability, CSTN). *A CSTN is weakly controllable if whenever a scenario s is known a priori, we can find a consistent schedule whose domain consists of all time points $X \in \mathcal{T}$ such that $s \models L(X)$.*

The CSTN in Figure 2.3 is weakly controllable. To prove that I show that each scenario is consistent.

- If $s(p) = \bot, s(q) = \bot$, then $S(P?) = 0, S(W) = 3$
- If $s(p) = \bot, s(q) = \top$, then $S(P?) = 0, S(W) = 3$ (impossible scenario)
- If $s(p) = \top, s(q) = \bot$, then $S(P?) = 0, S(Q) = 3, S(W) = 9$
- If $s(p) = \top, s(q) = \top$, then $S(P?) = 0, S(Q) = 3, S(W) = 8$

Note that well-defined CSTNs may lead to impossible scenarios (i.e., scenarios that never happen). In this example, $s(p) = \bot, s(q) = \top$ is impossible as $s \not\models L(Q?)$, thus $Q?$ should not be executed. However, the associated schedule, whose domain consists of $P?$ and W only, satisfies all constraints between those time points (so Definition 2.11 is correct).

Like STNUs I proceed by giving the definition for strong controllability.

Definition 2.13 (Strong controllability, CSTN). *A CSTN is strongly controllable if there exists a unique schedule[4] making consistent any scenario.*

The CSTN in Figure 2.3 is not strongly controllable. The problem lies in the pair of constraints $P? \to W$ labeled by $\langle 5, \neg p \rangle$ and $W \to P?$ labeled by $\langle -8, p \rangle$ which say that if $s(p) = \top$, then $S(W)$ cannot be assigned a value less than 8 and if $s(p) = \bot$, then $S(p)$ must be assigned a value not greater than 5. Therefore, there is no way of precomputing a static $S(W)$ without having any information on what truth value p will be assigned. Note that the CSTN is weakly controllable because weak controllability assumes to be able to predict the future.

However, the CSTN could be still executable by making scheduling decisions in real time.

[3] Note that $s \models L(X)$ and $s \models L(Y)$ implicitly (constraint label coherence, Definition 2.10).

[4] The domain of this schedule is $\mathcal{T}$. If for any scenario s there exists $X \in \mathcal{T}$ such that $s \not\models L(X)$, then the assignment $S(X)$ is ignored and does not participate in the solution.

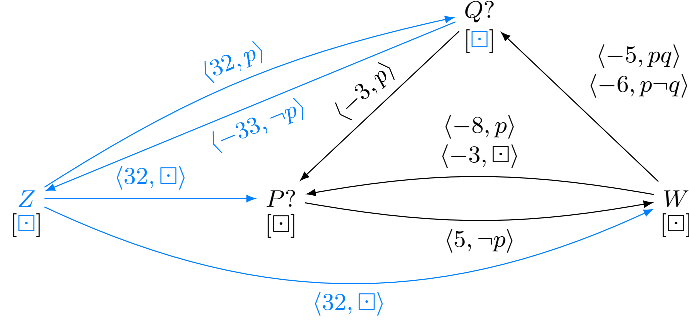


Fig. 2.4: Streamlining the CSTN in Figure 2.3 (all modifications appear in blue).

Definition 2.14 (Dynamic controllability, CSTN). *A CSTN is dynamically controllable if a consistent schedule is generated in real time by assigning (possibly different) real values to the time points according to what scenario is being generated.*

The CSTN in Figure 2.3 is dynamically controllable. Listing 2.2 shows a dynamic execution strategy to always build a consistent schedule.

Listing 2.2: Early execution strategy for the CSTN in Figure 2.3.

```
S(P?) = 0
if s(p) = ⊥, then S(W) = 3
else {
  S(Q) = 3
  if s(q) = ⊤, then S(W) = 8
  else S(W) = 9
}
```

In Figure 2.3 I gave an example of well-defined CSTN with three time points: $P?, W$ (always executed) and $Q?$ (executed if and only if $p = \top$). Since labeling time points complicates proofs and reductions towards other models by adding further conditions such as "if the time point is executed" (i.e., if it is relevant), a *streamlined model* was proposed in [21] to convert a CSTN having labels on nodes into a CSTN having labels on constraints only. Figure 2.4 shows the streamlined representation of Figure 2.3. The main idea is the following. We add a time point Z whose execution is fixed at 0 and we remove the labels from all time points (Figure 2.4). We compute an *horizon* value $h = M \times N$, where M is the maximum absolute value of any negative edge in the network and N the number of time points. Considering Figure 2.4 as the streamlined CSTN of Figure 2.3, we have that $M = 8$ and $N = 4$ (once we have added Z), therefore $h = 32$. We constrain every time point to occur within h if relevant in the original CSTN (i.e., if its label is true), and after h otherwise (modeled by as many delay constraints as the number of disjuncts arising from the negation of the label). For Figure 2.3 "after h" is modeled by a delay constraint of weight $-h-1 = -33$ (but any value "greater than" h is fine).

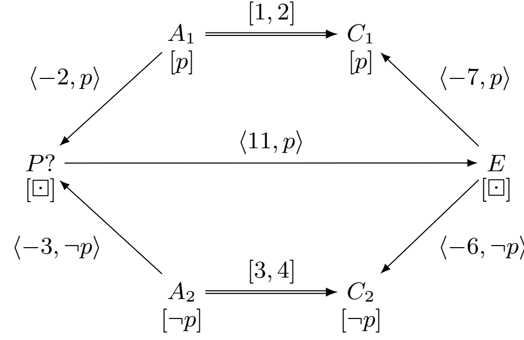


Fig. 2.5: CSTNU labeled distance graph.

A streamlined CSTN is dynamically controllable if and only if the original CSTN is so [21]. Therefore, "streamlining" a CSTN (but in general any temporal network subject to conditionals) preserves dynamic controllability (or uncontrollability) of the network getting another network in which all time points are always executed. Note that in this case well-defined properties such as label honesty, constraint honesty and label coherence on constraints become superfluous as they trivially hold [21].

2.1.4 Conditional Simple Temporal Networks with Uncertainty

Conditional simple temporal networks with uncertainty [74] merge, as the name suggests, STNUs and CSTNs to address conditional constraints and uncontrollable durations simultaneously.

Definition 2.15 (CSTNU). *A* Conditional Simple Temporal Network with Uncertainty *(*CSTNU*) is a tuple $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$, where:*

- *$\mathcal{T}$, $\mathcal{OT}$, $\mathcal{P}$, O, L and $\mathcal{C}$ are the same of those given for CSTNs (Definition 2.8).*
- *$\mathcal{L}$ is the same of that given for STNUs (Definition 2.4).*

A CSTNU is well-defined *if (i) $L(A) = L(C)$ for any $(A, x, y, C) \in \mathcal{L}$, and (ii) the well-defined properties for STNUs (Definition 2.4) and CSTNs (Definition 2.10) hold.*

Therefore, any contingent link has a unique implicit label $L(A) = L(C)$.

The graphical representation of a CSTNU extends that of a CSTN and also that of an STNUs. An edge $X \rightarrow Y$ labeled by $\langle k, \ell \rangle$ represents the labeled constraint $(Y - X \leq k, \ell) \in \mathcal{C}$, whereas a double edge $A \Rightarrow C$ labeled by $[x, y]$ represents a contingent link $(A, x, y, C) \in \mathcal{L}$. Like CSTNs, I show the labels of nodes below them.

Figure 2.5 shows the labeled distance graph of $\mathcal{Z} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$, where:

- $\mathcal{T} = \{P?, A_1, C_1, A_2, C_2, E\}$
- $\mathcal{OT} = \{P?\}$

- $\mathcal{P} = \{p\}$
- $O(p) = P?$
- $L(P?) = L(E) = \boxdot$, $L(A_1) = L(C_1) = p$ and $L(A_2) = L(C_2) = \neg p$.
- $\mathcal{L} = \{(A_1, 1, 2, C_1), (A_2, 3, 4, C_2)\}$
- $\mathcal{C} = \{\underbrace{(P? - A_1 \leq -2, p)}_{1}, \underbrace{(P? - A_2 \leq -3, \neg p)}_{2}, \underbrace{(C_1 - E \leq -7, p)}_{3},$
  $\underbrace{(C_2 - E \leq -6, \neg p)}_{4}, \underbrace{(E - P? \leq 11, p)}_{5}\}$

In other words, $\mathcal{Z}$ specifies six time points $P?, A_1, C_1, A_2, C_2, E$ (of which one is observation), two contingent links $(A_1, 1, 2, C_1)$, $(A_2, 3, 4, C_2)$ and requires that:

1. $(A_1, 1, 2, C_1)$ must be executed if $s(p) = \top$ with A_1 scheduled after minimum 2 since $P?$.
2. $(A_2, 3, 4, C_2)$ must be executed if $s(p) = \bot$ with A_2 scheduled after minimum 3 since $P?$.
3. E must be executed after minimum 7 since C_1 (if $s(p) = \top$).
4. E must be executed after minimum 6 since C_2 (if $s(p) = \bot$).
5. E must be executed within 11 since $P?$ (if $s(p) = \top$).

The CSTNU in Figure 2.5 is well-defined. Again, since CSTNUs embed both CSTNs and STNUs, a controllability approach is needed. A schedule for a CSTNU still involves all time points, but as for STNUs and CSTNs we can specify assignments only for non-contingent time points and we do not have to define $S(X)$ when the scenario containing the observations we made so far does not satisfy $L(X)$.

Definition 2.16 (Weak controllability, CSTNU). *A CSTNU is weakly controllable if whenever a schedule is defined for contingent time points only and a scenario s is known in advance, we can complete such a schedule by assigning real values to all non-contingent time points X such that $s \models L(X)$ in a way that the resulting schedule is consistent.*

The CSTNU in Figure 2.5 is weakly controllable. To prove that I show that there exists a consistent schedule for each combination scenario/duration.

- If $s(p) = \bot$, then $S(P?) = 0, S(A_1) = 2$, $S(E) = S(C_1) + 7$ for any $S(C_1)$ such that $S(C_1) - S(A_1) \in [1, 2]$
- If $s(p) = \bot$, then $S(P?) = 0, S(A_2) = 3$, $S(E) = S(C_2) + 6$ for any $S(C_2)$ such that $S(C_2) - S(A_2) \in [3, 4]$

I proceed by giving the definition for strong controllability.

Definition 2.17 (Strong controllability, CSTNU). *A CSTNU is strongly controllable if there exists a schedule defined for non-contingent time points only such that the schedule is always consistent regardless of the real-value assignments to the contingent time points that will complete it and the generated scenario.*

The CSTNU in Figure 2.5 is not strongly controllable. The problem lies in the constraint $P? \to E$ labeled by $\langle 11, p \rangle$ which says that if $s(p) = \top$, then $S(E)$ cannot be assigned a value greater than $S(P?) + 11$. This does not create any problem if $s(p) = \top$. Assume that $S(P?) = 0$, then $S(E) = 11$. Indeed, for any

duration of $A_1 \Rightarrow C_1$ that constraint will hold if we execute the non contingent time points as soon as possible. However, if $s(p) = \bot$, and $A_2 \Rightarrow C_2$ takes any duration in $[3 + \epsilon, 4]$, $S(E) > 11$. Therefore, there is no way of precomputing a static $S(E)$ without having any information on what truth value p will be assigned. Again, the CSTNU is weakly controllable because weak controllability assumes to be able to predict the future.

However, the CSTNU is dynamically controllable.

Definition 2.18 (Dynamic controllability, CSTNU). *A CSTNU is dynamically controllable if a consistent schedule is generated in real time by assigning (possibly different) real values to the relevant non contingent time points according to what scenario is being generated and what durations of contingent links are being observed.*

The CSTNU in Figure 2.5 is dynamically controllable. Listing 2.3 shows a dynamic execution strategy to always build a consistent schedule.

Listing 2.3: Early execution strategy for the CSTNU in Figure 2.5.

```
S(P?) = 0
if s(p) = ⊥, then {
  S(A1) = 2
  wait for C1 to execute
  S(E) = S(C1) + 7
}
else {
  S(A2) = 3
  wait for C2 to execute
  S(E) = S(C2) + 6
}
```

I now discuss how we can check dynamic controllability of STNUs, CSTNs, and CSTNUs. The approach, which is sound and complete, is based on timed game automata (TGAs).

2.2 Timed Game Automata

A *finite automaton* is a tuple $\langle S, \rightarrow \rangle$, where S is a finite set of states and $\rightarrow$ is a finite set of labeled transitions. S always contains both a *starting state* and a subset of *final states*. Each transition specifies a legal move from one state to another [69].

A *timed automaton* [3] refines a finite automaton by adding real-valued *clocks* and *clock constraints* . All clocks increase at the uniform rate keeping track of the time with respect to a fixed global time frame. Clocks model the timing properties of a system. Formally,

Definition 2.19 (Timed Automaton). *A* Timed Automaton (TA) *is a tuple* $\langle Loc, Act, \mathcal{X}, \rightarrow, Inv \rangle$*, where*

- *Loc is a finite set of* locations*. One is initial.*

- *Act is a finite set of* actions *used as transition labels (they can be viewed as input symbols).*
- $\mathcal{X}$ *is a finite set of* real-valued clocks*.*
- $\rightarrow \subseteq Loc \times \mathcal{H}(\mathcal{X}) \times Act \times 2^{\mathcal{X}} \times Loc$ *is the transition relation. An edge* (L_i, G, A, R, L_j) *represents a transition from location* L_i *to location* L_j *realizing action* A*.* $G \in \mathcal{H}(\mathcal{X})$ *is a* guard *consisting of a conjunction of clock constraints having the form* $c_1 \sim k$ *or* $c_1 - c_2 \sim k$ *with* $c_1, c_2 \in \mathcal{X}$*,* $k \in \mathbb{Z}$ *and* $\sim \in \{<, \leq, =, >, \geq\}$*.* $R \subseteq 2^{\mathcal{X}}$ *specifies the set of clocks to reset (i.e., set to* 0*).*
- $Inv : Loc \rightarrow \mathcal{H}(\mathcal{X})$ *is a function assigning an* invariant *(i.e., a conjunction of clock constraints) to each location. An invariant is a condition under which the automaton may stay in that location.*

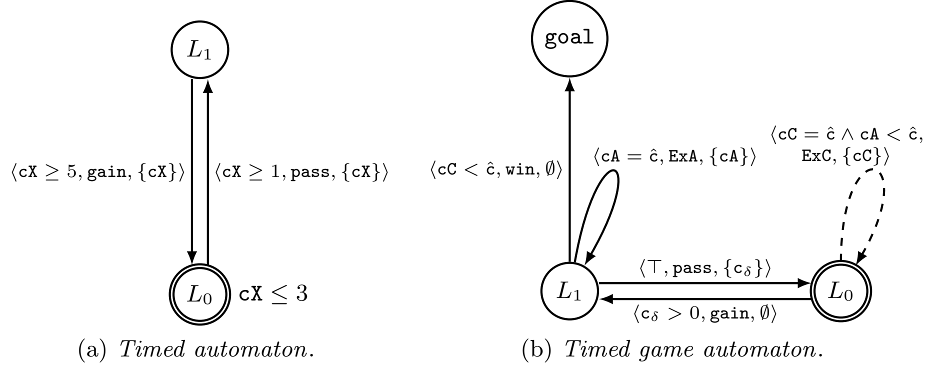
Figure 2.6a shows an example of a TA. L_0 is the initial location, drawn as double circle. The TA has one clock `cX` which is set to 0 upon the start of any execution. The TA is allowed to remain in L_0 until `cX` ≤ 3, then it has to leave the location. The `pass` transition can be taken whenever `cX` ≥ 1 and along with $Inv(L_0)$ ensures that the TA enters L_1 at any instant such that $1 \leq$ `cX` ≤ 3. When we take `pass`, `cX` resets to 0. After that, we can take the `gain` transition as soon as `cX` ≥ 5. If we do so, the TA gets back to L_0 and `cX` resets to 0. If we don't, the TA may remain in L_1 forever. However, in order to model uncontrollable parts timed automata must be extended into timed game automata [98].

Definition 2.20 (Timed Game Automaton). *A* Timed Game Automaton (TGA) *extends a TA by partitioning the set of transitions into* controllable *and* uncontrollable *ones. Uncontrollable transitions have priority over controllable ones.*

In other words, if during an execution there are a set of controllable and a set of uncontrollable transitions we want to take at the same time, then the uncontrollable ones go first and might prevent the controllable ones from being taken.

A TGA models a two-player timed game between a controller (`ctrl`) and the environment (`env`). The controller is assigned controllable transitions, whereas the environment is assigned uncontrollable ones. Moreover, in a TGA a location can be labeled as *urgent* in order to express that time freezes in that location.

Figure 2.6b shows an example of a TGA having four clocks ($\hat{\mathtt{c}}$, $\mathtt{c}_\delta$, `cA` and `cC`). Solid edges represent controllable transitions, whereas dashed ones uncontrollable transitions. The initial location is L_0, and `goal` is the location that `ctrl` must reach in order to win the game. Consider the following possible run. When all clocks are equal to 5, we take `gain` and the current location changes to L_1. At the same time, 5, `ctrl` takes `ExA` resulting in the reset of `cA`. After that, `ctrl` takes `pass` always at time 5. Therefore, the current location becomes L_0 and $\mathtt{c}_\delta$ is reset to 0. At time 6, both `ExC` and `gain` are enabled and `ctrl` decides to take `gain`. At the same time, `env` decides to take `ExC`. Since `ExC` has priority, it goes first and, therefore, `cC` resets to 0. Then, `ctrl` can take `gain` at the same time 6 or later. After `gain`, `ctrl` can take `win` to enter `goal`. For this TGA there does not exist a winning control strategy as `env` controlling the uncontrollable transitions can always refuse to take the uncontrollable transition. That is, if `env` decides to do so, `cC` will never be reset, preventing `ctrl` to take `win`.



(a) *Timed automaton.* (b) *Timed game automaton.*

Fig. 2.6: Examples of TA and TGA. Empty guards are always true ($\top$).

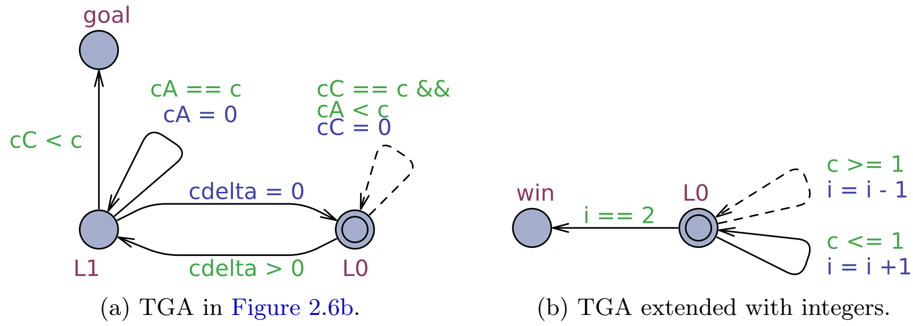
2.2.1 Controller Synthesis with UPPAAL-TIGA

Controller synthesis for both discrete and timed systems has been addressed in the past (e.g., [4, 98]). Controller synthesis allows, as the name suggests, for the synthesis of a controller able to react to the uncontrollable parts of a (timed) transition system so that a desired behavior is (always) met. *Game automata (GAs)* [4] model untimed games, whereas *timed game automata (TGAs)* extend GAs by modeling time by means of continuous variables called *clocks* [4, 98].

UPPAAL-TIGA [9] is a software tool allowing for the specification of (networks of) TGAs that involve clocks to model the temporal part, synchronization channels to model the communication between separate systems and also array data structures and user defined functions as well as integer and Boolean variables to model a discrete part. UPPAAL-TIGA relies on *timed computation tree logic* (TCTL) to express and check a variety of properties such as, for example, safety and liveness properties. Also, the query language permits to specify that we are interested in finding a *control strategy* satisfying the property we specified. Therefore, this query language provides path formulae over both discrete and continuous variables (clocks). Expressions over continuous variables do not involve logic operators such as $\neq$ or $\vee$ (Definition 9.3), whereas expressions over discrete variables are not affected by this restriction (see [10] for another definition of automata over clocks and variables).

Path formulae starting with `A<>` mean *always eventually*, whereas those starting with `A[]` mean *always globally*. If the keyword `control:` appears before a path formula φ, it means that we want to synthesize a controller able to always satisfy φ. In this thesis, I only use `control: A<>` and `control: A[]` path formulae to synthesize controllers for those games that are won if and only if `ctrl` always eventually enters a location of interest (e.g., "must reach/avoid `win`" depending on how the game is modeled). During any run, a controller follows a (memoryless) *strategy* in order to meet the desired behavior defined by φ.

A *memoryless strategy* is a mapping $\sigma\colon \mathcal{Z} \to \mathit{Act} \cup \{\texttt{wait}\}$, where $\mathcal{Z}$ models the entire state of the TGA (i.e., discrete and continuous variables) and Act is a



(a) TGA in Figure 2.6b.

(b) TGA extended with integers.

Fig. 2.7: Examples of (extended) TGAs with UPPAAL-TIGA. Nodes model locations (double-circled means initial). Solid and dashed edges model controllable and uncontrollable transitions, respectively. Guards (above, green) and updates (below, blue) are shown as labels near transitions.

set of actions augmented with `wait` which means "no action". Therefore, a strategy tells `ctrl` what to do depending on what the state looks like.

Figure 2.7a shows how Figure 2.6b looks like in UPPAAL-TIGA. As I have already said that TGA is uncontrollable. To prove that we can try to synthesize a controller by querying the system with `control: A<> goal` which tells the model checker to find a *strategy* to *always eventually* get to `goal`.

Listing 2.4 shows the result of the analysis for Figure 2.7a which is the "encoding" in UPPAAL-TIGA of Figure 2.6b. The text in output shows the strategy of `env`. That is, it shows that `env` can always beat `ctrl` by waiting forever either in L_0 (if `ctrl` does not take `pass`) or in L_1 (it `ctrl` takes `pass`). The statement `-Property is NOT satisfied` (line 4) tells us that the TGA is uncontrollable.

Listing 2.4: Output of the analysis for Figure 2.7a.

```
$ verifytga -s -q -w0 TGA.xml TGA.q

Verifying property 1 at line 6
 -- Property is NOT satisfied.
$v_gameInfoCounterPlayInitial state:
( tga.L0 )
(c==cdelta && cdelta==cA && cA==cC && cC==0)

Counter strategy to prevent from winning:

State: ( tga.L0 )
While you are in (c==cC && cdelta<=cA && cA<=c), wait.

State: ( tga.L1 )
While you are in (0<c && c==cC && cA<=c), wait.
```

Instead, Figure 2.7b shows an example of TGA extended with integer variables consisting of two locations: L_0 (initial) and `win`. The continuous part consists of a single clock `c` which starts at 0 in any run. The discrete part consists of an integer

variable `i` (initially set to 0). The TGA specifies two controllable transitions (solid edges) and an uncontrollable one (dashed edge). `ctrl` can take (i) the controllable self loop transition at L_0 if and only if the continuous value of the clock `c` is less than or equal to 1, and (ii) the transition going from L_0 to `win` if and only if the value of the discrete integer variable `i` is equal to 2. `env` can take the uncontrollable self loop transition at L_0 if and only if the continuous value of `c` is greater than or equal to 1. If `ctrl` takes his self loop, then `i` is incremented by 1, whereas if `env` takes his, then `i` is decremented by 1. If both players take their self loop transitions at $\texttt{c} = 1$, then `env` plays first. `ctrl` wins if he enters `win`.

This TGA is controllable. To prove that we try to synthesize a controller using the same query we used for Figure 2.7a.

Listing 2.5: Output of the analysis for Figure 2.7b.

```
$ verifytga -s -q -w0 ExtendedTGA.xml ExtendedTGA.q

Verifying property 1 at line 6
 -- Property is satisfied.
$v_gameInfoPlayInitial state:
( tga.L0 ) i=0
(c==0)

Strategy to win:

State: ( tga.L0 ) i=1
When you are in (c<1), take transition tga.L0->tga.L0 { c <= 1, tau, i := i
     + 1 }

State: ( tga.L0 ) i=0
When you are in (c<1), take transition tga.L0->tga.L0 { c <= 1, tau, i := i
     + 1 }

State: ( tga.L0 ) i=2
When you are in true, take transition tga.L0->tga.win { i == 2, tau, 1 }
```

Listing 2.5 shows the strategy for `ctrl` and says that `ctrl` must first take his self loop twice and then the transition going to `win` before the value of the clock `c` reaches 1, otherwise `env` will always be able to decrement `i` as many times as he wants, preventing `ctrl` from leaving L_0. The statement `- Property is satisfied` says that the TGA is controllable.

2.2.2 Dynamic Controllability of CSTNUs via TGAs

I summarize here the encoding from CSTNUs into TGAs. Since CSTNUs embed both STNUs and CSTNs this encoding implicitly works for them too.

The DC-checking problem is the problem of deciding if a CSTNU is DC. We can answer the DC-checking problem by using *sound* and *complete* TGA reachability algorithms [25, 26]. The DC-checking is modeled as a two-player game between a controller (`ctrl`) and the environment (`env`). The aim of `ctrl` is to reach a specific location as soon as all relevant time points have been executed and all

constraints are satisfied, whereas env's goal is to prevent ctrl from doing that. If ctrl wins, the network is DC, otherwise it is not. An important aspect of this encoding is that ctrl is assigned *uncontrollable* transitions, whereas env is assigned *controllable* ones. This is necessary to allow env's instantaneous reactions as in the TGA semantics, uncontrollable transitions go first [25–27]. Considering the CSTNU in Figure 2.5 the encoding into a TGA (shown in Figure 2.8) is as follows.

Clocks. $\mathcal{X}$ contains a clock $\mathtt{cX}$ for each time point $X \in \mathcal{T}$ and a clock $\mathtt{bP}$ for each proposition $p \in \mathcal{P}$. $\mathcal{X}$ also contains two special clocks $\hat{\mathtt{c}}$ (modeling the global time) and $\mathtt{c}_\delta$ (regulating the interplay of the game). $\mathtt{cX} = \hat{\mathtt{c}}$, means that X has not been executed, whereas $\mathtt{cX} < \hat{\mathtt{c}}$ means that X was executed at time $\hat{\mathtt{c}} - \mathtt{cX}$ (when this difference is > 0). Likewise, $\mathtt{bP} = \hat{\mathtt{c}}$ means that $s(p) = \top$, whereas $\mathtt{bP} < \hat{\mathtt{c}}$ means that $s(p) = \bot$ (each when $\mathtt{cP} < \hat{\mathtt{c}}$). Each $\mathtt{cX}$ and $\mathtt{bP}$ may be reset at most once. For the example I am discussing, $\mathcal{X} = \{\hat{\mathtt{c}}, \mathtt{c}_\delta, \mathtt{cP}, \mathtt{cA}_1, \mathtt{cC}_1, \mathtt{cA}_2, \mathtt{cC}_2, \mathtt{cE}, \mathtt{bP}\}$.

Locations. *Loc* contains three core locations L_0 (initial), L_1 (urgent) and goal (urgent), and $n-1$ urgent locations $L_{\ell_1}, \ldots, L_{\ell_{n-1}}$ where n is the number of distinct labels in the CSTNU. That is,

$$n = |\{L(X) \mid X \in \mathcal{T}\} \cup \{\ell \mid (Y - X \leq k, \ell) \in \mathcal{C}\}|$$

For this example, $Loc = \{L_0, L_1, L_{\boxdot}, L_p, \mathtt{goal}(= L_{\neg p})\}$ as the distinct labels are $\{\boxdot, p, \neg p\}$.

Transitions. $\rightarrow$ contains controllable and uncontrollable transitions to model the following:

- *Game interplay.* pass and gain are uncontrollable transitions regulating the game interplay. In particular gain can be taken only when $\mathtt{c}_\delta > 0$ modeling the *reaction time* needed to observe how the uncontrollable part behaves.
- *Non-contingent time point executions.* For each non-contingent time point X there is an uncontrollable self-loop transition

$$\langle L_1, \mathtt{cX} = \hat{\mathtt{c}}, \mathtt{ExX}, \{\mathtt{cX}\}, L_1\rangle$$

  modeling the execution of X. The guard says that X has not been executed yet, while the reset fixes the execution time of X to $\hat{\mathtt{c}} - \mathtt{cX}$ by resetting $\mathtt{cX}$.
- *Contingent time point executions.* For each contingent link $(A, x, y, C) \in \mathcal{L}$ there is a controllable self-loop transition

$$\langle L_0, \mathtt{cA} < \hat{\mathtt{c}} \wedge \mathtt{cC} = \hat{\mathtt{c}} \wedge \mathtt{cA} \geq x \wedge \mathtt{cA} \leq y, \mathtt{ExC}, \{\mathtt{cC}, \mathtt{c}_\delta\}, L_0\rangle$$

  to allow env to execute the contingent time point C such that $C - A \in [x, y]$, and a fail transition

$$\langle L_0, \mathtt{cA} < \hat{\mathtt{c}} \wedge \mathtt{cC} = \hat{\mathtt{c}} \wedge \mathtt{cA} > y, \mathtt{failC}, \{\emptyset\}, \mathtt{goal}\rangle$$

  to allow ctrl to move to goal if env fails or refuses to take the transition.

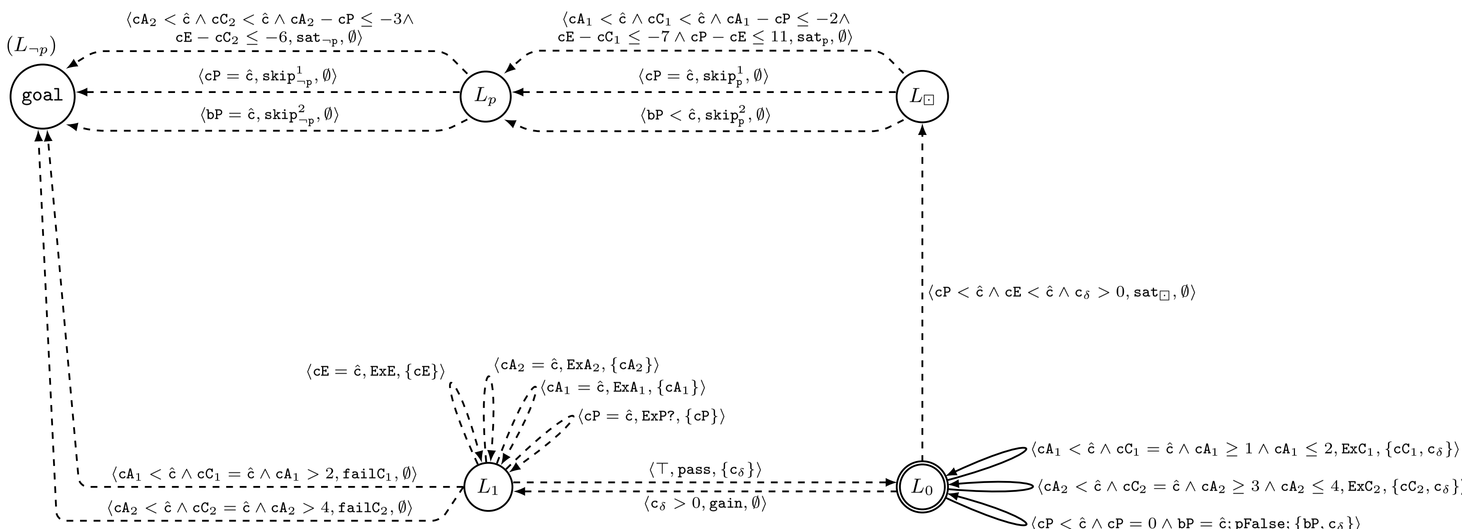


Fig. 2.8: TGA encoding the CSTNU in Figure 2.5. L_0 is the initial location, L_1, $L_\boxdot$, L_p, goal are urgent. Solid (resp., dashed) edges model controllable (resp., uncontrollable) transitions.

- *Truth value assignments.* For each proposition $p \in \mathcal{P}$ there is a controllable self-loop transition

$$\langle L_0, \texttt{cP} < \hat{\texttt{c}} \wedge \texttt{cP} = 0 \wedge \texttt{bP} = \hat{\texttt{c}}, \texttt{pFalse}, \{\texttt{bP}, \texttt{c}_\delta\}, L_0\rangle$$

  to allow `env` to assign $\bot$ to p, if it decides so. If it does not, the truth value of p will remain forever $\top$.
- *Winning conditions.* To check that all relevant time points have been executed and all constraints are satisfied we connect each pair of locations $(L_{\ell_{i-1}}, L_{\ell_i})$ in the winning path $L_0 \to L_{\boxdot} \to \cdots \to L_{\ell_{n-1}} \to \texttt{goal}$ by means of a set of uncontrollable transitions. Each set of transitions going from $L_{\ell_{i-1}}$ to L_{ℓ_i} verifies that if the current (partial) scenario $s \models \ell_i$, then all time points labeled by ℓ_i must have been executed and all constraints labeled by ℓ_i are satisfied. If $s \not\models \ell_i$, a `skip` transition allows us to ignore this check. In this way, the problem is decomposed with respect to the specific labels avoiding the combinatorial explosion of all arising cases. For example, the set of transitions going from $L_{\boxdot}$ to L_p is generated as follows. In the scenario where $P?$ has been executed and p assigned $\top$ (i.e., $s(p) = \top$), then A_1 and C_1 must have been executed, and $P? - A_1 \leq -2, C_1 - E \leq -7$ and $E - P? \leq 11$ are satisfied. In other words, the meta conditional constraint

$$(\texttt{cP} < \hat{\texttt{c}} \wedge \texttt{bP} = \hat{\texttt{c}}) \implies (\texttt{cA}_1 < \hat{\texttt{c}} \wedge \texttt{cC}_1 < \hat{\texttt{c}} \wedge \texttt{cA}_1 - \texttt{cP} \leq -2 \wedge \\ \texttt{cE} - \texttt{cC}_1 \leq -7 \wedge \texttt{cP} - \texttt{cE} \leq 11)$$

  refines to

$$\neg(\texttt{cP} < \hat{\texttt{c}} \wedge \texttt{bP} = \hat{\texttt{c}}) \vee (\texttt{cA}_1 < \hat{\texttt{c}} \wedge \texttt{cC}_1 < \hat{\texttt{c}} \wedge \texttt{cA}_1 - \texttt{cP} \leq -2 \wedge \\ \texttt{cE} - \texttt{cC}_1 \leq -7 \wedge \texttt{cP} - \texttt{cE} \leq 11)$$

  simplifying to

$$(\texttt{cP} = \hat{\texttt{c}}) \vee (\texttt{bP} < \hat{\texttt{c}}) \vee (\texttt{cA}_1 < \hat{\texttt{c}} \wedge \texttt{cC}_1 < \hat{\texttt{c}} \wedge \texttt{cA}_1 - \texttt{cP} \leq -2 \wedge \\ \texttt{cE} - \texttt{cC}_1 \leq -7 \wedge \texttt{cP} - \texttt{cE} \leq 11)$$

  since TGAs do not allow negations nor disjunctions of clock constraints in the guards. Finally, we generate a transition[5] for each disjunct ($\texttt{sat}_\texttt{p}$, $\texttt{skip}^1_\texttt{p}$,$\texttt{skip}^2_\texttt{p}$).

DC-checking is done by looking for a control strategy for `env` to always prevent `ctrl` from getting to `goal`[6]. If such a strategy exists, the initial CSTNU is not DC, otherwise it is (as `ctrl` has a counter-strategy to react to any combination of `env`'s moves). The correctness of the encoding is given in [25–27].

[5] I model $Y - X \leq k$ as $(\hat{\texttt{c}} - \texttt{cY}) - (\hat{\texttt{c}} - \texttt{cX}) \leq k$ simplifying to $\texttt{cX} - \texttt{cY} \leq k$. I might write $\texttt{cX} - \texttt{cY} \geq k$ as a short for $X - Y \leq -k$ and $\texttt{cX} - \texttt{cY} = k$ as a short for the pair $Y - X \leq k$ and $X - Y \leq -k$.

[6] In Uppaal-TIGA the query is `control: A[] not tga.goal`, where `tga` is the name of the model.

2.3 Constraint Networks and Directional Consistency

In this section, I briefly review CNs and the *adaptive consistency algorithm* for the related consistency checking [52]. I renamed some symbols for coherence with the rest of the contributions.

Definition 2.21. *A* Constraint Network (CN) *is a triple* $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$*, where*

- $\mathcal{V} = \{V_1, \ldots, V_n\}$ *is a finite set of variables*
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ *is a set of discrete domains* $D_i = \{v_1, \ldots, v_j\}$ *(one for each variable)*
- $\mathcal{C} = \{R_{S_1}, \ldots, R_{S_n}\}$ *is a finite set of constraints each one represented as a relation* R_S *defined over a* scope *of variables* $S \subseteq \mathcal{V}$ *such that if* $S = \{V_i, \ldots, V_r\}$*, then* $R \subseteq D_i \times \cdots \times D_r$*.*

A CN is consistent *if each variable* $V_i \in \mathcal{V}$ *can be assigned a value* $v_i \in D_i$ *such that* all *constraints are satisfied.*

The *constraint satisfaction problem* (*CSP*) is NP-hard [52]. A CN is k-ary if all constraints have scope cardinality $\leq k$ and therefore *binary* when $k = 2$ [52, 101].

Let R_{ij} be a shortcut to represent a binary relation having scope $S = \{V_i, V_j\}$. A binary CN is *minimal* if any tuple $(v_i, v_j) \in R_{ij} \in \mathcal{C}$ belongs to at least one global solution for the underlying CSP [101]. Thus, a minimal CN models an n-ary relation whose scope is $\mathcal{V}$ and whose tuples represent the set of all solutions. Besides for a few restricted classes of CNs, the general process of computing a minimal network is NP-hard [101]. Furthermore, even considering a binary minimal network, the problem of generating an arbitrary solution is NP-hard if there is no total order on the variables [66].

Therefore, a first crude technique is that of searching for a solution by exhaustively enumerating (and testing) all possible solutions and stopping as soon as one satisfies all constraints in $\mathcal{C}$. To speed up the search, we can combine techniques such as backtracking with pruning techniques such as *node*, *arc* and *path consistency* [97].

A variable V_i is *node-consistent* if $v \in R_i$ for each $v \in D_i$. A CN is node-consistent if each variable is node-consistent. A variable V_i is *arc-consistent* with respect to a second variable V_j if for each $v \in D_i$, there exists $u \in D_j$ such that $(v, u) \in R_{ij}$. A CN is arc-consistent if every variable is arc-consistent with respect to any other second variable.

A pair of variables (V_i, V_j) is *path-consistent* with respect to a third variable V_k if for any assignment $V_i = v, V_j = u$, where $v \in D_i$ and $u \in D_j$, there exists $k \in D_k$ such that $(v, k) \in R_{ik}$ and $(k, u) \in R_{kj}$. A CN is path-consistent if any pair of variables is path-consistent with respect to any other third variable. Path consistency is not enough for a backtrack free search [52].

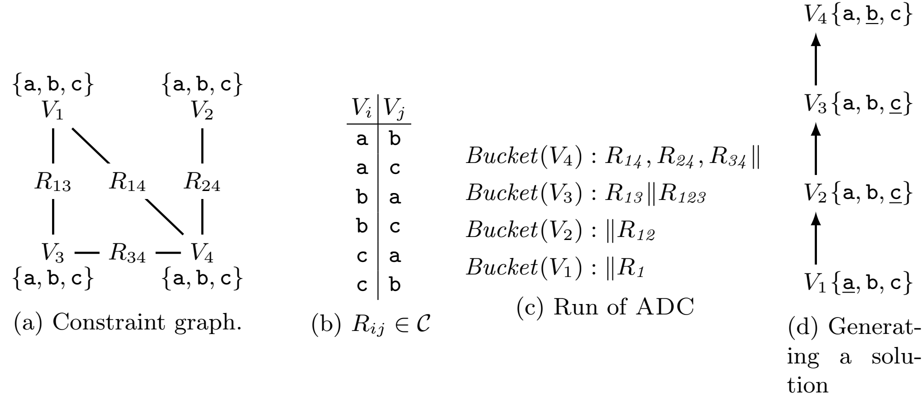
k-consistency guarantees that any (locally consistent) assignment to any subset of $(k-1)$-variables can be extended to a kth (still unassigned) variable such that all constraints between these k-variables are satisfied. *Strong k-consistency* is k-consistency for each j such that $1 \leq j \leq k$ [65]. As a result, 1, 2 and 3-consistency are node, arc and path consistency.

Algorithm 1: ADC($\mathcal{Z}, d$)

Input: A CN $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ and an ordering $d = V_1 \prec \cdots \prec V_n$
Output: A set *Buckets* of buckets (one for each variable) if $\mathcal{Z}$ is consistent, inconsistent otherwise.

```
1  for i ← n downto 1 do                         ▷ Partition the constraints as follows:
2      Put in Bucket(V_i) all unplaced constraints mentioning V_i
3  for p ← n downto 1 do
4      Let j ← |Bucket(V_p)| and S_i be the scope of R_{S_i} ∈ Bucket(V_p)
5      S' ← ⋃_{i=1}^{j} S_i \ {V_p}
6      R_{S'} ← π_{S'}(⋈_{i=1}^{j} R_{S_i})
7      if R_{S'} ≠ ∅ then
8          Bucket(V') ← Bucket(V') ∪ {R_{S'}}, where V' ∈ S' is the "latest" variable
           in d.
9      else
10         return inconsistent
11 Buckets = {{Bucket(V)} | V ∈ 𝒱}
12 return Buckets
```

Directional consistency has been introduced to speed up the process of synthesizing a solution for a constraint network limiting backtracking [54]. In a nutshell, given a total order on the variables of a CN, the network is *directional-consistent* if it is consistent with respect to the given order that dictates the assignment order of variables. In [54], an *adaptive-consistency (*ADC*) algorithm* was provided as a directional consistency algorithm adapting the level of k-consistency needed to guarantee a backtrack-free search once the algorithm terminates, if the network admits a solution (see Algorithm 1). The input of ADC is a CN $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ along with an order d for $\mathcal{V}$. At each step the algorithm *adapts* the level of consistency to guarantee that if the network passes the test, any solution satisfying all constraints can be generated without backtracking. If the network is inconsistent, the algorithm detects it before the solution generation process starts. ADC initializes a *Bucket*(V) for each variable $V \in \mathcal{V}$ and first processes all the variables top-down (i.e., from last to first following the ordering d) by filling each bucket with all (still unplaced) constraints $R_S \in \mathcal{C}$ such that $V \in S$. Then, it processes again the variables top-down and, for each variable V, it computes a new scope S' consisting of the union of all scopes of the relations in *Bucket*(V) neglecting V itself. After that, it computes a new relation $R_{S'}$ by joining all $R_S \in$ *Bucket*(V) and projecting with respect to S' ($\bowtie$ and π are the join and projection operators of relational algebra). In this way, it enforces the appropriate level of consistency. If the resulting relation is empty, then $\mathcal{Z}$ is inconsistent; otherwise, the algorithm adds $R_{S'}$ to the bucket of the *latest* variable in S' (with respect to the ordering d), and goes on with the next variable. Finally, it returns the set of *Buckets* (I slightly modified the return statement of ADC). Note that ADC takes as input a k-ary CN $\mathcal{Z}$ and returns a k'-ary CN $\mathcal{Z}'$, where $k' \geq k$ (an example of a binary CN turned into a ternary one can be found in [52, chapter 4]).



(a) Constraint graph. (b) $R_{ij} \in \mathcal{C}$ (c) Run of ADC (d) Generating a solution

Fig. 2.9: Graphical representation of a binary CN (a). Relational constraint (b). Run of ADC (c). Solution generation without backtracking (d).

Time and space complexity of ADC are $\mathcal{O}(n(2z)^{w^*+1})$ and $\mathcal{O}(nz^{w^*})$, respectively, where $n = |\mathcal{V}|$, $z = \max_{i=1,\dots,n} |D_i|$ is the maximal cardinality of variables domains, and w^* is the induced width of the graph along the order of processing [52, chapter 4]. Informally, w^* represents the maximum number of variables that can be affected by the value assumed by another variable; i.e., a characterization of the topology of the CN.

Any binary CN can be represented as a *constraint graph* where the set of nodes coincides with $\mathcal{V}$ and the set of edges represents the constraints in $\mathcal{C}$. Furthermore, nodes are labeled by their domains. Each (undirected) edge between two variables V_1 and V_2 is labeled by the corresponding $R_{12} \in \mathcal{C}$. As an example, consider the *constraint graph* in Figure 2.9a representing $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where:

- $\mathcal{V} = \{V_1, V_2, V_3, V_4\}$
- $\mathcal{D} = \{D_1, D_2, D_3, D_4\}$, with $D_1 = D_2 = D_3 = D_4 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\mathcal{C} = \{R_{13}, R_{14}, R_{24}, R_{34}\}$.

All $R_{ij} \in \mathcal{C}$ contain the same tuples; actually, they all specify the $\neq$ constraint between the pair of variables they connect. That is,

$$R_{13} = R_{14} = R_{24} = R_{34} = \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{a}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{a}), (\mathtt{c}, \mathtt{b})\}$$

(Figure 2.9b).

The CN in Figure 2.9a is consistent. To prove that I choose, without loss of generality (recall that *any* order is fine for this algorithm [52]), the order $d = V_1 \prec V_2 \prec V_3 \prec V_4$ and run ADC$(\mathcal{Z}, d)$ on the CN. I show the output of the algorithm in Figure 2.9c.

ADC first processes V_4 by filling $Bucket(V_4)$ with R_{14}, R_{24} and R_{34} (as they all mention V_4 in their scope and are still unplaced). Then, it processes V_3 by filling $Bucket(V_3)$ with R_{13} (but not R_{34}). Finally, it leaves $Bucket(V_2)$ and $Bucket(V_1)$ empty as all relations mentioning V_2 and V_1 in their scope have already been put in some other bucket. Therefore, the initialization phase fills the buckets in

Figure 2.9c with all relations on the left of $\|$ (the newly generated ones will appear on the right).

In the second phase, the algorithm computes $R_{123} = \pi_{123}(R_{14} \bowtie R_{24} \bowtie R_{34}) = \{(\mathtt{a},\mathtt{a},\mathtt{a}), (\mathtt{a},\mathtt{a},\mathtt{b}), (\mathtt{a},\mathtt{a},\mathtt{c}), (\mathtt{a},\mathtt{b},\mathtt{a}), (\mathtt{a},\mathtt{b},\mathtt{b}), (\mathtt{a},\mathtt{c},\mathtt{a}), (\mathtt{a},\mathtt{c},\mathtt{c}), (\mathtt{b},\mathtt{a},\mathtt{a}), (\mathtt{b},\mathtt{a},\mathtt{b}), (\mathtt{b},\mathtt{b},\mathtt{a}), (\mathtt{b},\mathtt{b},\mathtt{b}), (\mathtt{b},\mathtt{b},\mathtt{c}), (\mathtt{b},\mathtt{c},\mathtt{b}), (\mathtt{b},\mathtt{c},\mathtt{c}), (\mathtt{c},\mathtt{a},\mathtt{a}), (\mathtt{c},\mathtt{a},\mathtt{c}), (\mathtt{c},\mathtt{b},\mathtt{b}), (\mathtt{c},\mathtt{b},\mathtt{c}), (\mathtt{c},\mathtt{c},\mathtt{a}), (\mathtt{c},\mathtt{c},\mathtt{b}), (\mathtt{c},\mathtt{c},\mathtt{c})\}$ and adds it to $Bucket(V_3)$ (the latest variable in the scope $\{V_1, V_2, V_3\}$). Then, it goes ahead by processing $Bucket(V_3)$ generating in a similar way R_{12} and adding it to $Bucket(V_2)$. Finally, it processes $Bucket(V_2)$ by computing R_1 and adding it to $Bucket(V_1)$. Since the joins yielded no empty relation, it follows that $\mathcal{Z}$ is consistent[7].

A solution is generated by assigning the variables following the order d. For each $V \in d$ we just look for a value v in its domain such that the current solution augmented with $V = v$ satisfies all constraints in $Bucket(V)$. If the network is consistent, at least one value is guaranteed to be there. In this way, each solution can be generated efficiently without backtracking and by assigning one variable at a time. For instance any (combination of) values for V_1 and V_2 is fine (recall that $Bucket(V_1)$ and $Bucket(V_1)$ only contain universal relations). Assume that $V_1 = \mathtt{a}$ and $V_2 = \mathtt{c}$. Now we can only choose either $\mathtt{a}$ or $\mathtt{c}$ for V_3 (as $(\mathtt{a},\mathtt{c},\underline{\mathtt{b}}) \notin R_{123}$). Assume that $V_3 = \mathtt{a}$. Now the only possible value satisfying R_{14}, R_{24} and R_{34} in $Bucket(V_4)$ is $\mathtt{b}$. Therefore, a possible solution is $V_1 = \mathtt{a}$, $V_2 = \mathtt{c}$, $V_3 = \mathtt{c}$ and $V_4 = \mathtt{b}$ (Figure 2.9d).

2.4 Periodic Time

Periodic time was first addressed by Niezette and Stevenne [105] and then refined by Bertino et al. [12] and it is one of the base building blocks of temporal RBAC models that I discuss in Section 2.5.

Periodic time is represented as $\langle$`[begin,end]`,`P`$\rangle$ where `[begin,end]` is a time interval and `P` a *periodic expression*. The formalism relies on the notion of *calendars*.

Definition 2.22 (Calendar). *A* calendar C *is a countable set of contiguous intervals, numbered by integers called* indexes *of the intervals.*

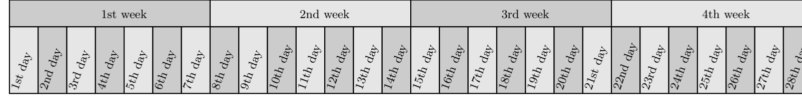
New calendars can be dynamically generated from the existing ones, by means of the function $generate(sp; C_0; (x_1, \ldots, x_n))$, where sp is a synchronization point and C_0 a reference calendar. The first tick of the new calendar corresponds to the union of the fist x_1 ticks of the reference one, the second to the union of the next x_2 ticks, and so forth.

For example, assume *Hours* (Figure 2.10a) is the base calendar (i.e. the minimum granularity). The following calendars can be defined:

$$Days = generate(1; Hours; (24));$$

[7] Note that R_{12} and R_1 should not be recored in $Bucket(V_2)$ and $Bucket(V_1)$ as they represent the universal relations $R_{12} = D_1 \times D_2$ and $R_1 = D_1$. However, doing so is superfluous but not wrong.

(a) Graphical representation of the first 24 *Hours*



(b) Graphical representation of $Days \sqsubseteq Weeks$.

Fig. 2.10: Graphical representation of periodic time with calendars.

$$\begin{aligned} Months = generate(1; Days; (&31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,\\ &31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,\\ &31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,\\ &31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31); \end{aligned}$$

$$Weeks = generate(1; Days; (7));$$

$$Years = generate(1; Days; (365, 365, 366, 365));$$

The synchronization point 1 stands for the first hour of the 1st of January 2015.

Definition 2.23 (Subcalendar). *A calendar C_1 is a* subcalendar *of a calendar C_2 (written $C_1 \sqsubseteq C_2$), if each interval of C_2 is exactly covered by a finite number of intervals of C_1.*

Let me take *Weeks* and *Days* as an example. Since 1 week consists of 7 days, $Days \sqsubseteq Weeks$ (Figure 2.10b).

Calendars can (and should) be combined to increase their expressiveness with expressions such as: *the third hour of the first day of each month*. Such a combination of calendars is called *periodic expression*.

Definition 2.24 (Periodic expression). *Given calendars $C_d, C_1, \ldots, C_n$, a* periodic expression $\mathtt{P}$ *is defined as follows:*

$$\mathtt{P} = \sum_{i=1}^{n} O_i \cdot C_i \triangleright r \cdot C_d$$

where $O_1 = all$, $O_i \in 2^{\mathbb{N}} \cup \{all\}$, $C_i \sqsubseteq C_{i-1}$ for $i = 2, \ldots, n$, $C_d \sqsubseteq C_n$ and $r \in \mathbb{N}$.

The symbol $\triangleright$ separates the first part (the starting granules of the intervals) from the duration of each interval (C_d). For example, the expression

$$all \cdot Years + \{3, 7\} \cdot Months \triangleright 2 \cdot Months$$

represents the set of intervals, whose starting granules are the third and the seventh month of every year (all), which last 2 months.

In the periodic time formalism, two important functions are defined:

1. the function Π, which models the infinite set of time intervals that belong to a periodic expression
2. the function *Sol*, which models the set of granules of a periodic expression P within the [begin,end] bounds which are formalized as date expressions.

Definition 2.25 (Date Expression). *A* date expression *has the form*[8] $\texttt{dd/mm/yy : hh}$ *(month, day, year and hour) where:*

- $\texttt{dd} \in \{01, \ldots, 31\}$
- $\texttt{mm} \in \{01, \ldots, 12\}$
- $\texttt{yy} \in \{00, \ldots, 99\}$
- $\texttt{hh} \in \{01, \ldots, 24\}$

I point out that at the time of writing the current year is 2018, so when I write yy, I will always consider the corresponding year 20yy, whereas when I omit $\texttt{hh}_1$ and $\texttt{hh}_2$ in two date expressions composing an interval of the form $[\texttt{dd}_1/\texttt{mm}_1/\texttt{yy}_1,\texttt{dd}_2/\texttt{mm}_2/\texttt{yy}_2]$, I will always consider that $\texttt{hh}_1 = 01$ and $\texttt{hh}_2 = 24$.

Definition 2.26 (Function Π). *Let* $\texttt{P} = \sum_{i=1}^{n} O_i \cdot C_i \triangleright r \cdot C_d$ *be a periodic expression; then* $\Pi(\texttt{P})$ *is a set of time intervals whose common duration is* $r \cdot C_d$ *and whose set* S *of starting granules is computed as follows:*

- *If* $n = 1$*,* S *contains all the starting granules of the intervals of* C_1
- *If* $n > 1$*, and* $O_n = \{n_1, \ldots, n_k\}$*, then* S *contains the starting granules of the* $n_1^{th}, \ldots, n_k^{th}$ *intervals of* C_n *included in each interval of* $\Pi(\sum_{i=1}^{n-1} O_i \cdot C_i \triangleright 1 \cdot C_{n-1})$

Definition 2.27 (Function *Sol*). *Let* g *be a granule,* P *a periodic expression, and* begin *and* end *two date expressions (represented as* $\texttt{dd/mm/yy : hh}$*).* $g \in Sol([\texttt{begin,end}], \texttt{P})$ *iff there exists* $\tau \in \Pi(\texttt{P})$ *such that* $g \in \tau$ *and* $g_b \leq g \leq g_e$ *where* g_b, g_e *are the granules denoting* begin *and* end*.*

2.4.1 Gap-Order and Periodicity constraints

Periodic expressions, formalized as $\sum_{i=1}^{n} O_i \cdot C_i \triangleright r \cdot C_d$ are of course convenient to be used by designers but are unsuitable to be manipulated in a deductive process. For this reason intervals [begin,end] consisting of date expressions are translated in *gap-order* constraints [110], whereas calendar expressions into *periodicity constraints* constraints [116].

Definition 2.28. *Let* u, l *be integers,* c *be a non-negative integer, and* g, g' *variables ranging over integers. A* gap-order constraint *is a formula of the form:*

- $l < g$,
- $g < u$,
- $g = g'$,
- $g + c < g'$.

If begin and end are denoted by g_b and g_e, respectively, the corresponding gap-order constraints are $c_1 < g$ and $g < c_2$, where $c_1 = g_b - 1$, and $c_2 = g_e + 1$.

[8] Please note that the original version specifies the date in the American format $\texttt{dd/mm/yy : hh}$.

Definition 2.29. *Let K be a finite set of natural numbers, g an integer variable, k an element of K, and $c \in \{0, \ldots, k-1\}$. A* simple periodicity constraint *is a formula of the form $g \equiv_k c$.*

A periodicity constraint of the form $g \equiv_k c$ denotes the set of integers of the form $c + nk$, with n ranging from $-\infty$ to $+\infty$ in $\mathbb{Z}$. In what follows I write $g \equiv_k (y + c), \forall y = 0, \ldots, u$ as a compact notation to represent the disjunction of simple periodicity constraints:

$$(g \equiv_k c) \vee (g \equiv_k c + 1) \vee \cdots \vee (g \equiv_k c + u)$$

2.4.2 From symbolic expressions to constraints

Any symbolic periodic expression can be translated into an equivalent set of simple periodicity constraints [12]. The translation of calendars in linear repeating intervals provided in [105] can be easily extended to periodic expressions.

A linear repeating interval is a mathematical expression of the form $kn+[g_b, g_e]$ denoting the sets of intervals including the interval $[g_b, g_e]$ and all intervals obtained by shifting $[g_b, g_e]$ by multiples of k. Any granule g in the intervals defined by $kn + [g_b, g_e]$ satisfies one of the constraints $g \equiv_k [g_b + y] \forall y = 0, \ldots, g_e - g_b$ and vice versa [12].

It follows that for each periodic expression P, there exists a disjunction of simple periodicity constraints such that the set of its solutions is the set of granules contained in the intervals of $\Pi(\texttt{P})$.

When the intervals in $\Pi(\texttt{P})$ have the same length, the simple periodicity constraints corresponding to P can be represented in the following compact way:

$$g \equiv_{Periodicity(\texttt{P})} (y + z - 1)$$

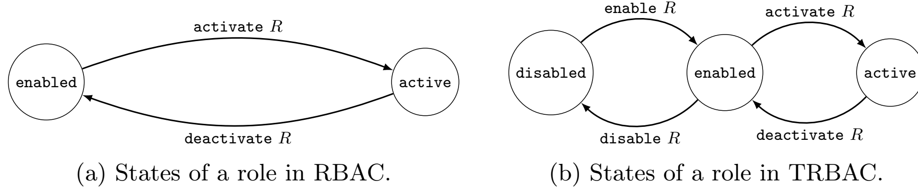
for all $y \in \{1, \ldots, Granularity(\texttt{P})\}$ and all $z \in Displacement(\texttt{P})$. The values depend on the calendar used to express the constraints, and on the definition of the calendars appearing in P.

In general, given a symbolic expression P and the basic calendar, the values for *Periodicity*(P), *Displacement*(P), and *Granularity*(P) can be derived as follows:

- *Periodicity*(P) is the number n of units of the basic calendar identifying the periodicity with which the time intervals in $\Pi(\texttt{P})$ repeat themselves.
- *Displacement*(P) is a set of numbers, each one representing the position within a period where a segment of the span of time defined by P begins.
- *Granularity*(P) is the length of each segment of time within the period defined by P. The granularity is expressed using the basic calendar (easily derived from the part on the right of $\triangleright$).

2.5 Temporal role-based access control models

A temporal role-based access control model (TRBAC, [13]) extends a classic role-based access control model (RBAC, [113]) by introducing temporal constraints



(a) States of a role in RBAC. (b) States of a role in TRBAC.

Fig. 2.11: A comparison of the possible states a role can assume in RBAC and TRBAC.

on role enabling and disabling. As usual in RBAC models *users* are assigned to *roles* and roles are assigned *permissions*. A role is thus an interface between users and permissions. If a user changes his/her role(s) the system administrator simply reassigns him/her to the new role(s). This flexibility is one of the main reason why these models have a so high industrial impact.

Definition 2.30 (Classical RBAC). *A* role based access control model (RBAC) *consists of the following components:*

- *Sets* `Users`*,* `Roles`*,* `Perm` *and* `Sess`*, representing the set of users, roles, permissions and sessions, respectively.*
- *UA* $\subseteq$ `Users` $\times$ `Roles`*, a many-to-many roles to permissions assignment relation.*
- *PA* $\subseteq$ `Roles` $\times$ `Perm`*, a many-to-many users to roles assignment relation.*
- *user* : `Sess` $\rightarrow$ `Users`*, a function that assigns each session to a single user.*
- *role* : `Sess` $\rightarrow 2^{\texttt{Roles}}$*, a function that assigns each session to a set of roles.*
- *RH* $\subseteq$ `Roles` $\times$ `Roles`*, a partially ordered role hierarchy (written* $\geq$*).*

TRBAC was proposed as a temporal extension of RBAC to address temporal constraints on role enabling and disabling [13]. If a user wants to activate a role, the role itself has to be first enabled by means of an event expression, a trigger or a runtime request expression. The status of role differs from that of the traditional RBAC in which a role can only be `active` or `not active`. Instead, TRBAC introduces the concept of enabling/disabling of a role, thereby extending the state of the role itself. The classical RBAC made an implicit assumption: a role *is always* `enabled` and can become `active` if some user decides to activate it. Instead, in TRBAC the status of a role is `enabled` at certain moments and `disabled` at others. To become `active` the status of a role has to be first `enabled`. So $\texttt{active}(R) \implies \texttt{enabled}(R)$. Figure 2.11 shows a comparison between the states of a role in RBAC and in TRBAC. I proceed by discussing the basic TRBAC components.

Let $(\texttt{Prios}, \preceq)$ be a totally ordered set of priorities where there are two distinct members $\top, \bot \in \texttt{Prios}$ such that for all $x \in \texttt{Prios}, \bot \preceq x \preceq \top$. I write $x \prec y$ meaning that $x \prec y \wedge x \neq y$.

Event Expressions and *Role Status Expressions* model the enabling/disabling of a role and its status, respectively.

Definition 2.31 (Event Expressions). Event Expressions *serve to change the status of a role from* `enabled` *to* `disabled` *or vice versa. They divide in simple and prioritized as follows:*

- Simple Event Expressions *have the form* `enable` R *or* `disable` R*, where* $R \in$ `Roles`*,*
- Prioritized Event Expressions *have the form* $p\colon E$*, where* $p \in$ `Prios` *and* E *is an event expression.*

Of course, simple event expressions can be viewed as a subset of the prioritized ones when the priority is "omitted" (i.e. when the system sets it by default to $p = \bot$).

Definition 2.32 (Role Status Expressions). Role status expressions *have the form* `enabled` R *or* $\neg$`enabled` R*, where* $R \in$ `Roles`*.*

I now continue by providing the definition of conflicting events which play a crucial role in defining the semantics of TRBAC.

Definition 2.33 (Conflicting Events). *Let* $E_1 =$ `enable` R *and* $E_2 =$ `disable` R *be two event expressions.* E_1 *is in conflict with* E_2 *if* $R = R'$*. Formally, either:*

$$\texttt{conf(enable R)} \stackrel{\text{def}}{=} \texttt{disable R}, \textit{ or}$$

$$\texttt{conf(disable R)} \stackrel{\text{def}}{=} \texttt{enable R}$$

Event expressions and role status expressions are the basic building blocks of the Role Enabling Base (REB), which contains temporal constraints on the enabling of roles.

Definition 2.34 (Role Enabling Base, Periodic Events, Role Triggers). *A* Role Enabling Base (REB) $\mathcal{R}$ *is a set of elements of the following kinds.*

1. Periodic events *of the form* $(I, P, p\colon E)$*, where*
   - I *is a time interval,*
   - P *is a periodic expression,*
   - $p\colon E$ *is a prioritized event expression with* $p \prec \top$*.*
2. Role triggers *of the form* $E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow p\colon E$ [`after` Δt]*, where:*
   - E_i *are simple event expressions* $(1 \leq i \leq n)$*, and*
   - C_i *are role status expressions* $(1 \leq i \leq k)$*, and*
   - $p\colon E$ *is a prioritized event expression with* $p \prec \top$*.*

Priorities (p) and delay expressions (`after` Δt) can be omitted. In that case, by default, $p = \bot$ and $\Delta t = 0$. Also, I will use the expression $B \rightarrow p\colon E$ to refer to a trigger, where B is also known as the *body* of the trigger.

$$\underbrace{E_1, \ldots, E_n, C_1, \ldots, C_k}_{B} \rightarrow p\colon E$$

An example of REB describing a medical domain is depicted in Figure 2.12, where assuming that `NightTime` goes from 7PM to 7AM and `DayTime` from 7AM

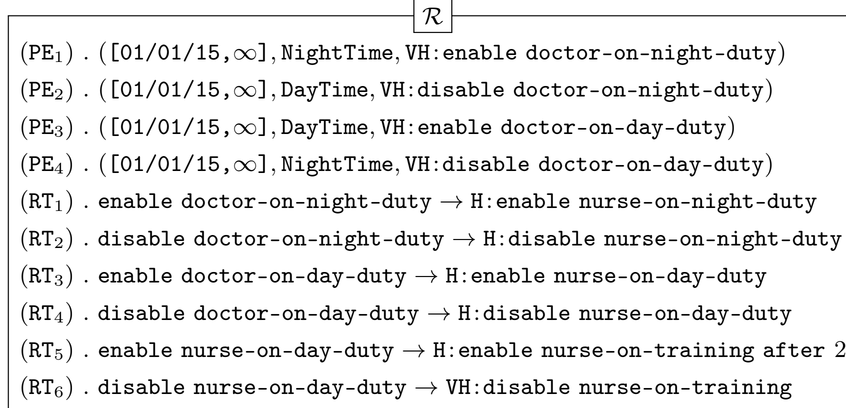


Fig. 2.12: An example of Role Enabling Base.

to 7PM, the REB says that every night a `doctor-on-night-duty` is enabled (PE_1) and disabled at the end of it (PE_2). Similarly, a `doctor-on-day-duty` is enabled every day (PE_3) and disabled at the end of it (PE_4). Enabling `doctor-on-night-duty` implies to enable the corresponding nurse (RT_1) and disabling it makes the opposite effect (RT_2). The same happens for `doctor-on-day-duty` with (RT_3) and (RT_4) but with respect to `nurse-on-day-duty` that implies in turn to enable `nurse-on-training` after 2 hours (RT_5) and disable it at the end of the day (RT_6).

In Section 9.8, I consider the fragment of TRBAC consisting of non-conflicting complementary periodic events (e.g., PE_1, PE_2). I do not consider non-complementary periodic events because the disabling/enabling of one periodic event could be fired before the interval spanned by the periodic expression of its complementary ends. I do not consider conflicting events because I do not deal with conflicts for the moment. I do not consider runtime request expressions (i.e., requests allowing an administrator to enable/disable roles dynamically if needed) and thus individual exceptions because they allow an administrator to override any execution. I also do not consider role triggers because they may lead to the previous problems. Thus, under these assumptions, priorities will not influence the behavior of the system.

2.6 Workflow Satisfiability and Resiliency

The standard definition published by the Workflow Management Coalition in [32] defines a workflow 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. The automation of a business process is defined within a *process definition*, which identifies the various process activities, procedural rules and associated control data used to manage the workflow during process execution.

For workflow satisfiability and resiliency, in this thesis I consider workflows specifying tasks assigned to users who are the only resources to commit for their

execution[9]. More specifically, the workflows I consider have a core mathematical specification defining a set of tasks $\texttt{Tasks}$ and a partial-order relation $\prec \subseteq \texttt{Tasks} \times \texttt{Tasks}$ saying in which order such tasks have to be executed. An access-controlled workflow (ACWF) extends classic workflows by adding a set of users $\texttt{Users}$, an authorization relation $UA \subseteq \texttt{Users} \times \texttt{Tasks}$ and a set of constraints $\mathcal{C}$ where each constraint is either a *counting* or an *entailment* constraint [49].

A *counting constraint* is a triple $(x, y, \texttt{Tasks}')$, where $1 \leq x \leq y \leq |\texttt{Tasks}|$ and $\texttt{Tasks}' \subseteq \texttt{Tasks}$.

An *entailment constraint* is a triple $(\rho, \texttt{Tasks}', \texttt{Tasks}'')$, where $\rho \subseteq \texttt{Users} \times \texttt{Users}$ and $\texttt{Tasks}', \texttt{Tasks}'' \subseteq \texttt{Tasks}$. An entailment constraint is:

- *Type 1* if $|\texttt{Tasks}'| = |\texttt{Tasks}''| = 1$, (e.g., $(\rho, \{T_1\}, \{T_2\})$)
- *Type 2* if $|\texttt{Tasks}'| = 1$ and $|\texttt{Tasks}''| \neq 1$, (e.g., $(\rho, \{T_1\}, \{T_2, T_3, \dots\})$)
- *Type 3* if $\texttt{Tasks}'$ and $\texttt{Tasks}''$ are arbitrary sets (e.g., $(\rho, \{T_1, \dots\}, \{T_j, \dots\})$)

as defined in [49, 122].

A *plan* is a mapping $\pi \colon \texttt{Tasks}' \rightarrow \texttt{Users}$ assigning users to tasks. A plan is *initial* if $\texttt{Tasks}' = \emptyset$, *partial* if $\texttt{Tasks}' \subset \texttt{Tasks}$, and *total* if $\texttt{Tasks}' = \texttt{Tasks}$. A plan is *consistent* if it satisfies all constraints involving tasks in its domain. When the domain of the plan does not contain all tasks but satisfies the constraints for the tasks it contains, we say that the plan is *locally consistent*. A plan satisfies a counting constraint $(x, y, \texttt{Tasks}')$ if a user performs either no task in $\texttt{Tasks}'$ or between x and y. A plan π satisfies an entailment constraint $(\rho, \texttt{Tasks}', \texttt{Tasks}'')$ iff there exist $T' \in \texttt{Tasks}'$ and $T'' \in \texttt{Tasks}''$ such that $(\pi(T'), \pi(T'')) \in \rho$. I abuse notation and write $\pi \cup \{\pi(t) = u\}$ to shorten that the domain of π is extended by adding t such that $\pi(T) = u$. In this thesis, I only consider Type 1 entailment constraints and ACWFs having the following core specification.

Definition 2.35. *An ACWF is a tuple* $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$*, where:*

- $\texttt{Tasks} = \{T_1, \dots, T_n\}$ *is a finite set of tasks.*
- $\texttt{Users} = \{u_1, \dots, u_m\}$ *is a finite set of users.*
- $UA \subseteq \texttt{Users} \times \texttt{Tasks}$ *is the authorization relation. I shorten the set of users authorized for* T *as* $\mathcal{A}(T) = \{u \mid (u, T) \in UA\}$.
- $\prec \subseteq \texttt{Tasks} \times \texttt{Tasks}$ *is a partial-order relation. I write* $(T_1, T_2) \in \prec$ *(or simply* $T_1 \prec T_2$*) meaning that* T_1 *executes* before T_2.
- $\mathcal{C}$ *is a set of constraints, each one represented as a binary relation* R_S*, where* $S \in \texttt{Tasks} \times \texttt{Tasks}$ *is the* scope*, such that if* $S = \{T_i, T_j\}$ *for* $T_i, T_j \in \texttt{Tasks}$*, then* $R_S \subseteq \mathcal{A}(T_i) \times \mathcal{A}(T_j)$.

An ACWF is satisfiable if there exists a consistent plan for it.

However, when the availability of users is uncertain, workflow satisfiability is not enough to guarantee that a consistent plan will always work. Indeed, workflow resiliency calls for the synthesis of *dynamic plans* whose assignments of users to tasks are in general different depending on if users are absent or not by the time tasks need to be executed. In [122, 123], Wang and Li defined three levels of resiliency:

[9] Instead, I consider temporal workflows in Section 5.7.

Algorithm 2: StaticResiliency

Input: $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$ and $0 \leq k < |\texttt{Users}|$
Output: Resilient if W is statically resilient up to k absent users. Breakable otherwise.

1 `Player2` chooses $Absent \subset \texttt{Users}$ s.t. $|Absent| \leq k$
2 $\texttt{Users} \leftarrow \texttt{Users} \setminus Absent$
3 **if** *there exists a consistent plan* π *for* W **then**
4   **return** *Resilient* ▷ `Player1` wins
5 **return** *Breakable* ▷ `Player2` wins

Algorithm 3: DecrementalResiliency

Input: $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$ and $0 \leq k < |\texttt{Users}|$
Output: Resilient if W is decrementally resilient up to k absent users. Breakable otherwise.

1 $Absent \leftarrow \emptyset$
2 $\pi \leftarrow$ initial plan
3 $Executed \leftarrow \emptyset$
4 **while** $Executed \neq \texttt{Tasks}$ **do**
5   `Player2` chooses $\texttt{Users}' \subset \texttt{Users}$ s.t. $|\texttt{Users}' \cup Absent| \leq k$
6     $Absent \leftarrow Absent \cup \texttt{Users}'$
7     $\texttt{Users} \leftarrow \texttt{Users} \setminus Absent$
8   `Player1` looks for $T \in \texttt{Tasks} \setminus Executed$ and $u \in U$ such that $(u, T) \in UA$ and $\pi \cup \{\pi(T) = u\}$ is locally consistent
9   **if** *no pair* (u, T) *exists* **then**
10   **return** *Breakable* ▷ `Player2` wins
11     $\pi \leftarrow \pi \cup \{\pi(T) = u\}$
12     $Executed \leftarrow Executed \cup \{T\}$
13 **return** *Resilient* ▷ `Player1` wins

- *static* (level 1)
- *decremental* (level 2)
- *dynamic* (level 3)

In *static resiliency*, up to k users might be absent *before* an execution of an ACWF starts and these users will never become available again for that execution. In *decremental resiliency* up to k users might be absent *before* execution or become so during it. As in the static case, absent users will never become available again for that execution. In *dynamic resiliency*, up to k (possibly different) users might be absent at *any time*. These users may become absent and become available again (possibly) many times. It is easy to see that *dynamic resiliency* $\Rightarrow$ *decremental resiliency* $\Rightarrow$ *static resiliency* [122, 123].

Each of these levels of resiliency can be seen as a two-player game where a controller (`Player1`) plays against an environment (`Player2`). Algorithm 2, Algorithm 3 and Algorithm 4 summarize formalizations proposed by Wang and Li in [122] for static, decremental and dynamic resiliency, respectively.

Algorithm 4: DynamicResiliency

Input: $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$ and $0 \leq k < |\texttt{Users}|$

Output: Resilient if W is dynamically resilient up to k absent users. Breakable otherwise.

```
1  π ← initial plan
2  Executed ← ∅
3  while Executed ≠ Tasks do
4      Player2 chooses Absent ⊂ Users s.t. |Absent| ≤ k
5      Users′ ← Users \ Absent
6      Player1 looks for Tasks ∈ Tasks \ Executed and u ∈ Users′ such that
        (u, T) ∈ UA and π ∪ {π(T) = u} is locally consistent if no pair (u, T) exists
        then
7          return Breakable                                        ▷ Player2 wins
8      π ← π ∪ {π(T) = u}
9      Executed ← Executed ∪ {T}
10 return Resilient                                                ▷ Player1 wins
```

3

Related work

3.1 Planning and scheduling

A Simple Temporal Network (STN) [53] models a temporal plan in which all time points and durations are under control. Consistency analysis determines whether it is possible to schedule events such that all the given temporal constraints are satisfied. The consistency checking is done by running the all pairs shortest paths algorithm [46, 53]. Therefore, the decision problem of consistency for STNs is in PTIME [53].

Drake [45] is an executive for temporal plans with choices modeled as Labeled STNs that extend STNs by labeling constraints with environments (set of instantiated choice variables). There are no decision points, therefore decisions involving discrete variable can be made anytime. After compilation, Drake uses an Assumption-based Truth Maintenance System to maintain a minimal representation of the constraints needed to execute the network. During execution, choices are discriminated by generating conflicts according to the time Drake decides to schedule some event.

A Disjunctive Temporal Problem (DTP) [53, 114, 115, 117] specifies a set of disjunctive temporal constraints. The work in [53] allows for the specification of several intervals between the *same pair* of time points, whereas that in [115] (originally appeared in [114]) does not have that restriction. DTP in [115] and Labeled STNs are equivalent [45]. In [45], it is proved that each disjunction $(l_1 \leq Y - X \leq u_1) \vee (l_2 \leq Y - X \leq u_2) \vee (l_3 \leq Y - X \leq u_3)$ (in DTP) can be mapped into a set of constraints $(\{x = 1\}\colon l_1 \leq Y - X \leq u_1)$, $(\{x = 2\}\colon l_2 \leq Y - X \leq u_2)$, $(\{x = 3\}\colon l_3 \leq Y - X \leq u_3)$ for a choice variable x having domain $\{1, 2, 3\}$. The vice versa is obtained by writing "implication constraints" (as the same choice could be related to many intervals) and then computing the CNF of the resulting formula.

Temporal plan networks (TPNs) [79] extend STNs by adding decision nodes and symbolic constraints to model temporal plans with controllable choices modeled as outgoing edges from a decision node. Taking one of these outgoing edges means making a particular decision. Time points are not labeled and activities are modeled as pair of non-decision nodes (start,end). A symbolic constraint is either Ask(c) (is c true?) and Tell(c) (c is true!) where c a literal. Symbolic constraints

may exclude activities from being executed. A plan is consistent if it satisfies both temporal and symbolic constraints. TPNs do not specify more than one temporal constraint on the same edge. Consistency is checked by means of a backtracking algorithm which maintains a set of nodes and iterates until this set becomes empty. Initially the set contains the start node. At any iteration a node is removed from the set and all nodes reachable from its outgoing edges are added (if the removed node is not a decision) or just one node is added (if the node is a decision). In the latter case, the algorithm marks the picked outgoing edge. When the STN arising from the path going from start to end is inconsistent, the algorithm backtracks to the last decision node that still has unmarked outgoing edges (i.e., it will try other decisions).

In [125], a Controllable Conditional Temporal Problem is provided to address temporal plans with choices. Differently, from LabeledSTNs and TPNs this work deals with over constrained problems and provides a Best-first Conflict-Directed Relaxation (BCDR) algorithm to enumerate the best continuous relaxation for an over-constrained conditional temporal problem with controllable choices. The work in [125] is an optimization problem.

Other related work involves the management of several kinds of uncertainty in temporal networks.

Pike [90] is an executive for temporal plans with both controllable and uncontrollable choices that achieves plan recognition and adaptation concurrently. Pike employs temporal plan networks with uncertainty (TPNUs) which extend TPNs to address both controllable and uncontrollable choices. Pike adapts controllable choices to uncontrollable ones made by the human. As stated in [90], Pike takes as input (1) a TPNU, (2) the initial and goal states (sets of PDDL predicates, [64]), (3) a stream of state estimates (sets of predicates), and (4) a stream of time assignments and outcomes to the uncontrollable choices in the TPNU. Pike outputs (1) a stream of choice assignments to the TPNU's controllable variables, and (2) a dispatch of the TPNU's events, such that there is at least one complete and consistent candidate subplan for the choice made.

In [87], Léauté and Williams provide a continuous model-based executive for systems with state variables and continuous dynamics that generates near-optimal control sequences during execution. Their approach is based on encoding subparts of the main temporal problem into disjunctive linear programs (DLPs, [5]) reformulating them as Mixed-Integer Linear Programs, and on constraint pruning policies for both state plan constraints and temporal constraints. Their experimental evaluation shows that the adopted pruning policies result in the executive performing significantly better.

In [124], a Controllable Conditional Temporal Problem (CCTPU) is defined to address temporal plans with controllable choices and uncontrollable durations. A Conflict-Directed Relaxation with Uncertainty algorithm (CDRU) is provided to deal with over constrained temporal problems. When a plan is uncontrollable some lower or upper bound of some duration (either controllable or not) might be modified to restore controllability.

In [76], TPNUs are extended to also support uncontrollable durations. The work deals with strong controllability only. When the plan is infeasible a technique

is implemented to negotiate with a human some relaxation to restore controllability.

Simple Temporal Networks with Uncertainty (STNUs) [104] specify contingent durations as the unique uncontrollable part. The execution of non-contingent time points cannot influence any contingent duration. Instead, contingent durations do influence the real-value assignments to the non-contingent time points. However, such durations never prevent any non-contingent time point from being executed. Controllability analysis aims to synthesize a strategy making (possibly different) assignments to non-contingent time points such that all constraints will be eventually satisfied whatever the contingent durations. The decision problem of dynamic controllability for STNUs is in PTIME [103].

Conditional Simple Temporal Networks (CSTNs) [75] (formerly CTP, [118]) specify conditional constraints as the unique uncontrollable part. The execution of time points cannot prevent any truth value assignment from happening. Instead, depending on what truth value a propositional variable is assigned some time point might be excluded, runtime, from the execution of the network. The decision problem of dynamic controllability for CSTNs is PSPACE-complete [22]. Controllability analysis aims to synthesize a strategy making (possibly different) assignments to the time points such that all constraints will be eventually satisfied whatever the truth value assignments to the uncontrollable propositions.

Conditional Simple Temporal Networks with Uncertainty [74] address both uncertain durations and conditional constraints simultaneously. Any combination of these two parts influences when (and if) to execute non-contingent time points. In [37] a sound but not complete algorithm is provided. The complexity of the decision problem of dynamic controllability is currently unknown.

Timed Game Automata (TGAs) [98] offer sound and complete algorithms to check the dynamic controllability of temporal networks via controller synthesis. In [27] a first proposal to synthesize memoryless execution strategies for STNUs is provided. The approach is based on the synthesis of a controller for a timed game automaton. After, [25] extended [27] to support temporal networks with observation time points and disjunctive constraints. Finally, [26] revised [27] and [25]. All these works also show the theoretical relationships between various kinds of temporal networks and timed game automata. All encodings from a temporal network into the corresponding timed game automaton run in polynomial time.

In [19], CSTNs are extended with decision nodes regulating the truth value assignment for some propositions under control. That work focuses on the complexity analysis of the DC-checking problem proving that it is PSPACE-complete and provides algorithms for two special cases in which (i) the network specifies only decisions and no observations and (ii) all decisions are made *before* any observation. Uncontrollable durations are not considered.

In [23], a new system of constraint propagation rules for STNUs, which is sound-and-complete for DC checking, is presented. That system comprises just three rules which, differently from the ones proposed in all previous works, only generate unconditional constraints. That work also proves the existence of late execution strategies for STNUs, and the existence of a faster execution algorithm for STNUs which runs in $\mathcal{O}(nk)$ for an STNUs having a $k \geq 1$ contingent links and $n \geq k$ time points.

Simple Temporal Network with Partially Shrinkable Uncertainty (STNPSU) [82] extend STNUs allowing for the representation of a contingent range in a way that can be shrunk during execution as long as the shrinking does not go beyond a given threshold.

In [21], a streamlined model for CSTNs is provided. This model mainly shows how it is possible to remove labels from time points by rewriting the network in an equivalent one having labels on constraints only. During execution, previously labeled nodes that turn irrelevant, will be executed anyway after an horizon, a point in the future after which the temporal plan can be considered complete. Therefore, time points which are assigned a value greater than the horizon can be considered as unexecuted.

Hyper Temporal Networks (HyTNs) are a strict generalization of STNs, introduced to partially overcome the limitation of allowing only conjunctions of constraint [42]. Compared to STN distance graphs HyTNs allow for a greater flexibility in the definition of the temporal constraints meanwhile offering a pseudo-polynomial tractability in the consistency checking of the instances.

In [43], Comin and Rizzi solved the CTP by converting it into a Mean Payoff Game (MPG). They also introduced a variant of dynamic consistency, called ε-DC, where $\varepsilon > 0$ represents the minimum reaction time of the executive in response to observations. They presented (1) a sharp lower-bound analysis on the critical value of the reaction time where the CSTN changes from being DC to non-DC, (2) a proof that the CTP is coNP-hard, and (3) the first singly-exponential-time algorithm for solving the CTP.

In [72], Hunsberger and Posenato showed how their DC-checking algorithm from earlier work [75] can be extended to check the ε-DC property without incurring any performance degradation. They also introduced four benchmarks for testing DC-checking algorithms.

In [73], Hunsberger and Posenato presented another optimization of the approach presented by Cimatti et al. in which the CTP is viewed as a two-player game. Its solution is determined by exploring an abstract game tree to find a "winning" strategy, using Monte Carlo Tree Search and Limited Discrepancy Search to guide its search. An empirical evaluation shows that the new algorithm is competitive with the propagation-based algorithm.

In [20], Cairo et al. improved the analysis of the ε-DC property. They showed that if $\varepsilon = 0$ (i.e., if the system can react instantaneously), it is necessary to impose a further condition to avoid a form of instantaneous circularity. In particular, they (1) proposed a new extension of dynamic consistency, called π-DC, suitable for systems that can react instantaneously, (2) showed that π-DC is not equivalent to 0-DC, and (3) proposed a sound-and-complete algorithm for checking the π-DC property having a (pseudo) singly-exponential time complexity in the number of propositional letters.

In [50], Cui and Haslum extend STNUs by conditioning temporal constraints on the assignment of controllable discrete variables (decisions) that can be done at any time. They define dynamic controllability of such networks as the existence of a strategy that decides on both the values of discrete choice variables and the scheduling of controllable time points dynamically.

A Temporal Qualitative Constraint Network (TQCN) specifies a set of variables and a set of constraints where each constraint is a set of base relations representing the possible relative positions between two temporal events (i.e., between the variables modeling them). Consistency analysis consists of deciding if a TQCN is consistent. That is, if there exists an assignment of values to the variables satisfying all constraints. In [70], an efficient technique to check the consistency of a TQCN in polynomial time is given, provided the treewidth of the network is bounded. Optimization problems have also been studied for TQCNs. For example, in [44], Condotta et al. study the minimal consistency problem (`MinCons`): given an integer k, `MinCons` is the problem of establishing whether or not a TQCN admits a solution using at most k distinct points on the line. The decision problem of minimal consistency is NP-complete for both point algebra (PA, [121]) and interval algebra (IA [1]) (reduction from the 3 coloring problem).

Satisfiability Modulo Theory (SMT, [8]) can describe (simple) temporal networks by using a fragment of linear real arithmetic (LRA) called difference logic (RDL). However, SMT-solvers do not guarantee to find early schedules. SMT can address weak and strong controllability of temporal networks via quantifier elimination techniques (e.g., [29, 30]). Some SMT-solvers support quantifiers in their input language (e.g., Z3, [51]), some others don't (e.g., MathSAT, [24]). Currently, it is unclear if SMT can address dynamic controllability.

In [106], Uppaal-TIGA is used to synthesize a controller for timeline-based plans which consider multivalued state variables and networks of TGAs. Apart from time points, in this thesis variables are real, Boolean or integer and the encodings I provide all involve one TGA only.

A constraint network (CN) specifies a finite set of variables, a set of finite discrete domains (one for each variable) and a set of relational constraints. The constraint satisfaction problem (CSP) is the problem of finding an assignment of values to the variables such that all constraints are satisfied. A CN is k-ary if all constraints have scope cardinality $\leq k$ and therefore *binary* when $k = 2$ [52, 101]. CSP is NP-hard [52]. Moreover, given a minimal network (i.e., a network where the inference of new constraints from the existing ones is no longer possible) generating an arbitrary solution is NP-hard as well [66]. This is due to the non-monotonicity of the relations employed (e.g., $\neq$ [52, 101]). If the relations are monotone (e.g., $=$) the checking is done in n^3 where n is the number of variables [101]. The same holds for non-monotone relations if each variable has no more that 2 elements in its domain [101]. Consistency checking can be done by exhaustively enumerating (and testing) all possible solutions and stopping as soon as one satisfies all constraints. To speed up the search, we can combine techniques such as backtracking with pruning techniques such as *node*, *arc* and *path consistency* [97]. However, these pruning techniques are incomplete inference rules [52]. *k-consistency* guarantees that any (locally consistent) assignment to any subset of $(k-1)$-variables can be extended to a kth (still unassigned) variable such that all constraints between these k-variables are satisfied. *Strong k-consistency* is k-consistency for each j such that $1 \leq j \leq k$ [65]. As a result, 1, 2 and 3-consistency are node, arc and path consistency. *Directional consistency* has been introduced to speed up the process of synthesizing a solution for a constraint network limiting backtracking [54]. In a nutshell, given a total order on the variables of a CN, the network is *directional-*

consistent if it is consistent with respect to the given order that dictates the assignment order of variables. In [54], an *adaptive-consistency algorithm (*ADC*)* was provided as a directional consistency algorithm adapting the level of k-consistency needed to guarantee a backtrack-free search once the algorithm terminates, if the network admits a solution. Classic CNs in [52] do not address any uncontrollable parts.

A Mixed CSP partitions the set of variables in controllable and uncontrollable and considers both full and no observability (NO). Full observability is when we get to known the uncontrollable part *before* making our decisions. In [63], Fargier et al. provide a consistency algorithm assuming full observability of the uncontrollable part.

Dynamic constraint satisfaction problems (DCSPs) [100] introduce activity constraints saying when variables are relevant or not depending on the values assigned to some other variables. No uncontrollable parts are specified, therefore the approach in [100] deals with satisfiability only.

Some probabilistic approaches (e.g., [62]) attempted to find the most probable working solution to a CSP under probabilistic uncertainty.

In a Prioritized Fuzzy Constraint Satisfaction Problem (PFCSP) (e.g., [94]) a solution threshold states the overall satisfaction degree.

3.2 Controllability of Workflows

In [84] and its extension in [85], Lanz et al, define a set of time patterns for process-aware information systems (PAIS). Such patterns are defined in four categories. Category I involves *Durations and Time Lags* allowing for the specification of time lags between activities (TP1), durations (TP2) and time lags between events (TP3). Category II involves *Restriction of process execution points* allowing for the specification of fixed date elements (TP4), schedule restricted elements (TP5), time based restrictions (TP6) and validity period (TP7). Category III involves *variability* allowing for the specification of time dependent variability (TP8). Category IV involves *Recurrent Process Elements* allowing for the specification of cyclic elements (TP9) and periodicity (TP10). A formal semantics of time patterns can be found in [83].

In [99], Marjanovic et al., define a conceptual model for temporal constraints of a process schema considering TP1, TP2 and TP4. A validation algorithm to compute the shortest and longest instances of a process is presented. However, no runtime part is supported.

In temporal workflow management, the difference between controllable and uncontrollable XOR splits is introduced in [59] and a technique based on PERT-nets computes internal activity deadlines in order to meet the global ones. Some missed deadlines require human interaction for recovery.

In [60], Eder at al., address time constraints in workflow systems and give an approach aimed at computing activity deadlines so that both the overall process deadline and all external time constraints are satisfied. Activity durations are deterministic and therefore the model does not specify starting times that can be obtained from ending times of activities minus their durations.

In [58], Eder et al., propose a Timed Workflow Graph to model the temporal properties of tasks considering TP1, TP2, TP4 and TP5. Nodes model tasks and edges regulate the control flow. Activity durations are the same no matter the process instance.

In [61], Eder et al. also give an overview of the early time management approaches for workflows. They focus on the modeling of temporal aspects, the analysis of temporal properties, synthesis of schedules, minimization of exceptions arising from the violation of temporal constraints, monitoring of workflow aspects and modeling and calculation of temporal properties for distributed workflows.

In [108] Pichler et al., propose temporal conditions in the formulation of XOR-splits and loops to give process designers explicit control over the temporal behavior of the processes they model. Task durations can be out of control. Some split connectors have an associated condition $elapsed \leq constant$ discriminating which branch to take depending on how the condition evaluates. For example, if a diamond is labeled by $elapsed \leq 100$ and the condition is true by the time the connector is executed, then the $\top$ branch is chosen, else the $\bot$ one. Loops are specified as activities labeled by *while elapsed* $\leq$ *constant* which are continuously repeated until the condition is true. An example of application can be the submission of a paper to a conference. If *while elapsed* ≤ 100 (i.e., it the submission deadline has not passed yet) continue to (re)submit the (updated version of the) paper.

The approach of Bettini in [16] can be considered one of the first relevant attempts of using temporal networks for the modeling and validation of workflows. Nodes model the start and end of activities whereas edges represent the temporal distance between them. Bettini also defines the concepts of *free-schedule* and *restricted due-time free-schedule*. In the former an agent can take any (allowed) amount of time to complete activities. In the latter an agent must complete them within the maximum upper bounds that are tighter than the maximum task durations.

In [35], Combi et al., propose a conceptual model for specifying time-aware process schema supporting TP1, TP2, TP4, TP5 and TP10. That work also discusses how consistency can be checked at design time, leaving the management of the runtime phase as future work.

In [38], Combi et al. move from consistency to controllability of time-aware process schemas and address temporal controllability of workflows specifying tasks having uncontrollable durations. The approach maps workflow paths to STNs and STNUs in order to boil down consistency and controllability analysis to those of STNs/STNUs and propose a technique to deal with conditional paths. Later, in [39] the same authors extend the algorithm in [103] to support runtime execution. They show that the controllability of each single workflow path is not enough to guarantee the controllability of the whole workflow.

In [81], CSTNUs are employed for the modeling and validation of time-aware business processes in order to verify their controllability at design time and handle their execution at run time. The authors provide a set of basic elements describing a process-aware information system supporting TP1, TP2, TP4 and TP9. Then, they provide an encoding from these basic components into a corresponding CSTNU. Finally, they provide an execution algorithm to handle controllability of the process during runtime. At that time, the algorithm for checking controllability

of a CSTNU was only known to be sound. In [75], a sound and complete algorithm for CSTNs was provided discussing issues that affect CSTNUs too. Therefore, [81] does not guarantee a correct controllability checking of the process, neither at design nor at runtime (unless this checking is done via TGAs).

In [127], Zavatteri et al., define and address weak, strong and dynamic controllability for access-controlled workflows under conditional uncertainty. That work deals with structured workflows by unfolding workflow paths, considering binary constraints only (whose labels are the conjunction of the labels of the connected tasks) and assuming that a total order for the tasks is given in input.

In [18], Cabanillas et al. address the resource allocation for business processes. They consider an RBAC environment and they do not impose any particular order on activities. They also address loops. However, the authors clearly state that their work is unable to address *History-Based Allocation* of resources. This is because consistency analysis is not enough.

Modeling languages such as YAWL [119] and BPMN [17] allow for the composition of constructs regardless of the type and position of the connected elements leading to unstructured workflows which are well-known to be error prone [86,109]. In [78], Kiepuszewski et al. prove that one can employ a structured approach without losing expressiveness. NestFlow [33] and TNest [34] are structured Process Modeling Languages (PML) integrating a control flow that can be described by a well-defined grammar. TNest is the temporal extension of NestFlow. For TNest dynamic controllability analysis is boiled down to that of the underlying conditional simple temporal network with uncertainty (CSTNU).

3.3 Role based access control models, workflow satisfiability and resiliency

A role based access control model (RBAC [113]) specifies users, roles and permissions. Roles are assigned both users and permissions acting as an interface between them. Roles are a collection of both users and permissions and are different from groups which are only a mere collection of users [112]. RBAC models do not specify constraints at user level. TRBAC [13] extends the classic RBAC by injecting temporal constraints on role enabling and disabling. Roles may be enabled in some time intervals and disabled in some others. A user can activate a role only if the role is enabled (in classic RBAC roles are always enabled).

In [14], Bertino et al. give a language for defining authorization constraints on role and user assignment to tasks in a workflow. They also provide algorithms for consistency checking and task assignment. Their proposal assumes that the workflow enforces a total order on tasks (i.e., no parallel tasks are allowed). Furthermore, temporal constraints are not investigated.

The Temporal Authorization Base model described in [12] is able to enforce authorization constraints in heterogeneous distributed systems. It allows users to assign periodic authorizations to other users on sets of objects. This model is quite expressive. In order to use it in a workflow context, we would need to restrict access modes to `execute` and constrain objects to be `tasks`.

In [120], Vasilikos et at. use a network of timed automata (TAs) to model distributed systems and provide BTCTL (behavior timed computational tree logics), an extension of TCTL [2] to express time-dependent access control policies. Such a logic allows for expressing security policies in which temporal, data and information flow aspects must be considered together. They use synchronization channels to make the various processes (each one underlain by a different TA) communicate with each other and present a reduction of a fragment of BTCTL into TCTL$^+$ (a variation of TCTL). They use UPPAAL as a model checker to validate their systems with respect to the fragment of BTCTL that can be translated into the logics supported by UPPAAL (they implemented a translator to automate this task). No uncontrollable part is supported.

The problem of verifying workflow features related to the assignment of agents to tasks is known in literature as workflow satisfiability and resiliency. More specifically, the *workflow satisfiability problem (WSP)* is the problem of finding an assignment of users to tasks such that the execution of the workflow gets to the end satisfying all authorization constraints. The *workflow resiliency problem* is WSP under the uncertainty that a maximum number of users may become (temporally) absent before or during execution. WSP does not address conditional uncertainty and always synthesizes one execution plan satisfying all constraints before starting.

Several work to tackle workflow satisfiability has been proposed over the last 20 years. In [15], Bertino et al. proposed a seminal work for the specification and enforcement of authorization constraints in workflow management systems. That work does not tackle resiliency nor provide a controllability approach.

In [122,123], Wang and Li proposed a role-and-relation based model (R^{2}BAC) for workflow systems and studied the complexity bounds of workflow satisfiability and resiliency. Their experimental evaluation involves WSP only.

In [91], Li et al. introduced the notion of resiliency policies in the context of access control systems and defined the *resiliency checking problem (RCP)*, i.e., whether an access control state satisfies a given resiliency policy. They studied the complexity bound of the RCP and provided a SAT-based approach for RCP. RCP always means static resiliency.

In [47], Crampton et al. studied the VALUED WSP to find the "least bad" plan; i.e., a plan that maximizes the number of satisfied constraints. Workflow resiliency is not addressed. After that, Crampton et al. [48] also proposed the BI-objective WSP as a generalization of [47].

In [77], Khan and Fong defined workflow feasibility (availability in some state) as the dual of workflow resiliency (availability in every state).

In [95], Mace et al. introduced the quantitative workflow resiliency as a metric of probability on how likely a workflow terminates given a security policy and a user availability model. They do so by solving a Markov Decision Process.

In [96], Mace et al. provided WRAD, a tool for workflow resiliency analysis and design that encodes a workflow specification into the PRIMS model checker. Workflow resiliency is defined as the maximum probability of finding a complete and valid plan.

In [107], Paci et al. extended the RBAC-WS-BPEL language to support the specification of resiliency constraints and provided an algorithm to check if a sys-

tem is failure-resistant. Although they use different terms, they deal with static resiliency only.

In [92], Lowalekar et al. provided failure resilience while satisfying security policies and constraints. They considered both static and decremental resiliency, but left the investigation for dynamic resiliency as future work.

In [93], Lu et al. studied the dynamic workflow adjustment; i.e., how to minimally adjust existing user-task assignments, when a sudden change, such as the absence of users, occurs.

In [56], dos Santos et al. define a class of *Scenario Finding Problems*, which are solutions solving the WSP that also satisfy other properties (e.g., a minimal number of users must be present). After, in [57], dos Santos et al. solve static workflow resiliency by pre-computing reachability graphs by model checking the system, but they do not address decremental nor dynamic resiliency.

Further surveys of literature on workflow satisfiability and resiliency can be found in [55, 88] and [67].

Part I

Temporal Controllability

Introduction

Temporal networks are a framework to model temporal plans and check the coherence of their temporal constraints which impose a minimal and maximal temporal distance between the occurrence of the events specified in the plan. Temporal plans mainly divide in plans having everything under control and plans having something out of control. The main components of a temporal network are *time points* and *constraints*. Time points are variables having continuous domain and model the occurrence of events as soon as these variables are assigned real values (i.e., executed). Constraints regulate the minimal and maximal temporal distance between the occurrence of pairs of events and are formalized as linear inequalities.

Whenever both these two components are under control we simply deal with a *consistency* problem asking us to find an assignment of real values to *all* time points satisfying all constraints. Simple temporal networks (STNs) model exactly this case [53], whereas Labeled STNs in Drake [45] address temporal plans with choices that are, however, under control; therefore, we keep dealing with a consistency problem asking us to further find suitable values for such choices.

Instead, when some component is out of control, satisfiability is, in general, not enough. In such a case, we deal with a *controllability* problem.

Section 2.1.3, Section 2.1.2 and Section 2.1.4 pointed out that CSTNs address conditional uncertainty, STNUs address temporal uncertainty, whereas CSTNUs address both conditional and temporal uncertainty simultaneously. All these three formalisms are frameworks to model controllability problems.

The controllability of a temporal network implies the existence of a *strategy* able to schedule the time points such that all constraints are eventually satisfied.

Controllability mainly divides in *weak*, *strong* and *dynamic*. Weak controllability ensures the existence of a (possible different) strategy to operate on the controllable part whenever we are able to predict how the *entire* uncontrollable part will behave before the execution starts. Strong controllability is the opposite case ensuring the existence of a strategy always operating the same way on the controllable part no matter how the uncontrollable part will behave. Dynamic controllability ensures the existence of a strategy operating on the controllable part in real time depending on how the uncontrollable one has behaved. Weak and strong controllability allow for an offline temporal planning, whereas dynamic controllability allows, in general, for an online temporal planning.

However, none of the formalisms mentioned so far tackles temporal plans in which some conditional constraints under control may influence (or be influenced by) some uncontrollable part. An initial discussion is given in [19] where CSTNs are extended with decision nodes regulating the truth value assignments to some propositions under control.

Contributions

Towards the modeling and validation of temporal plans in which decisions may influence (or be influenced by) *both* conditional and temporal uncertainty, my specific contributions for the first part of this thesis are the following.

1. I define *simple temporal networks with decisions (STNDs)* and provide two *Hybrid SAT-based Consistency Checking (HSCC)* algorithms for them.
2. I provide Kappa, a tool I developed for STNDs along with an experimental evaluation. I also provide an algorithm to generate random STNDs.
3. I prove that STNDs and disjunctive temporal problems (DTPs, [115]) are equivalent by reducing STNDs to DTPs and vice versa in polynomial time.
4. I define *conditional simple temporal networks with uncertainty and decisions (CSTNUDs)* as a unified formalism for temporal networks expressing uncontrollable parts in order to also model fallback temporal plans.
5. I provide an encoding into timed game automata (TGAs) for a sound and complete dynamic controllability checking for any CSTNUD. TGAs also allow for the synthesis of memoryless execution strategies for executing a (controllable) CSTNUD.
6. I provide Esse, a tool I developed for CSTNUDs along with an experimental evaluation. I also provide an algorithm to generate random CSTNUDs.
7. I provide a language to specify temporal workflows under conditional and temporal uncertainty. I define weak, strong and dynamic controllability of such workflows and give an encoding from temporal workflows into CSTNUDs for a sound and complete dynamic controllability checking.

Organization

Chapter 4 defines simple temporal networks with decisions (STNDs) along with two HSCC algorithms to check consistency and synthesize single or all consistent scenarios. Then, it provides Kappa a tool for STNDs along with an experimental evaluation and an algorithm to generate random STNDs. Finally, it proves the equivalence between STNDs and DTPs provided in [115]. Chapter 5 defines conditional simple temporal networks with uncertainty and decisions (CSTNUDs) and extends the encoding from CSTNUs into TGAs given in Section 2.2.2 in order to adapt it to CSTNUDs. Then, it proceeds by discussing the correctness of the encoding and provides Esse, a tool for CSTNUDs along with an experimental evaluation and an algorithm to generate random CSTNUDs. Finally, it provides a process modeling language for temporal workflows under conditional and temporal uncertainty, defines the controllability of such workflows and gives an encoding from temporal workflows into CSTNUDs for a dynamic controllability checking.

4

Simple Temporal Networks with Decisions

In this chapter I focus on simple temporal networks with decisions (STNDs). STNDs are STNs plus decision time points [19, 126]. Since STNDs do not specify any uncontrollable part, a consistency approach is enough. STNDs are a compact framework to represent temporal workflows patterns where the choice of the workflow path to take is under control. In Section 5.7 I will discuss of how decisions time points permit to choose a specific workflow path.

4.1 Syntax

Each decision time point is associated to a Boolean proposition. Time points and constraints may be labeled by labels (i.e., conjunctions of literals) saying for which scenarios (truth value assignments to the propositions) they must appear in the solution. Thus, an STND models a family of STNs each obtained as a projection of the initial STND onto a scenario. An STND is consistent if there exists a consistent scenario (i.e., a scenario such that the corresponding STN projection is consistent).

Definition 4.1. *A* Simple Temporal Network with Decisions *(*STND*) is a tuple* $\mathcal{S} = \langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$*, where:*

- $\mathcal{T} = \{X, Y, \dots\}$ *is a set of time points.*
- $\mathcal{DT} \subseteq \mathcal{T} = \{D!, E!, \dots\}$ *is a set of decision time points.*
- $\mathcal{P} = \{d, e, \dots\}$ *is a set of Boolean propositions.*
- $O\colon \mathcal{P} \to \mathcal{DT}$ *is a bijection assigning a unique proposition to each decision time point* $D!$ *that controls the truth value assignment to* d*.*
- $L\colon \mathcal{T} \to \mathcal{P}^*$ *is a function assigning labels to time points.*
- $\mathcal{C}$ *is a set of* labeled *constraints each having the form* $(Y - X \leq k, \ell)$*, where* $X, Y \in \mathcal{T}$*,* $k \in \mathbb{R} \cup \pm\infty$ *and* $\ell \in \mathcal{P}^*$*.*

The STN-projection of an STND $\mathcal{S}$ *with respect to a scenario* s *(written* $\pi_s(\mathcal{S})$*) is an STN* $\langle \mathcal{T}_s, \mathcal{C}_s \rangle$ *built as follows:*

- $\mathcal{T}_s = \{X \mid X \in \mathcal{T} \wedge s \models L(X)\}$
- $\mathcal{C}_s = \{(Y - X \leq k) \mid (Y - X \leq k, \ell) \in \mathcal{C} \wedge s \models \ell\}$

$\mathcal{S}$ is consistent if there exists a scenario s such that $\pi_s(\mathcal{S})$ is consistent. A solution is a pair $\langle s, S\rangle$, where s is a scenario, S is a schedule with domain $\mathcal{T}_s$, and $S(Y) - S(X) \leq k$ holds for each $(Y - X \leq k) \in \mathcal{C}_s$.

I graphically represent an STND exactly as I represented a CSTN in Section 2.1.3. For instance, Figure 4.1a models the following STND $\mathcal{S} = \langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C}\rangle$:

- $\mathcal{T} = \{A!, B!, C!, D, E\}$
- $\mathcal{DT} = \{A!, B!, C!\}$
- $\mathcal{P} = \{a, b, c\}$
- $O(a) = A!, O(b) = B!, O(c) = C!$
- $L(A!) = L(D) = L(E) = \boxdot, L(B!) = a, L(C!) = ab$
- $\mathcal{C} = \{(A! - B! \leq -2, a), (A! - D \leq -5, \boxdot), (A! - C! \leq 0, ab), (E - A! \leq 10, \neg a), (E - B! \leq 6, a\neg b), (E - C! \leq 4, ab\neg c), (D - E \leq -7, \boxdot), (B! - C! \leq -1, ab), (B! - A! \leq 5, a), (C! - B! \leq 2, ab)\}$.

Figure 4.1b–Figure 4.1e model all of the STN-projections of $\mathcal{S}$.

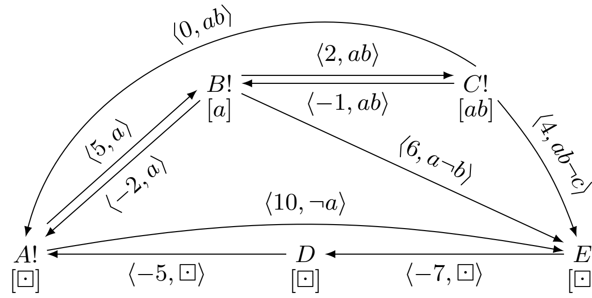
Definition 4.2 (Well-defined STND). *An STND is* well-defined *if and only if labels are honest and coherent like in CSTNs (Definition 2.10) with two differences for label honesty:*

1. *For each $X \in \mathcal{T}$, if $\lambda \in L(X)$, where $\lambda = \{d, \neg d\}$ and $d \in \mathcal{P}$, then $L(X) \Rightarrow L(O(d))$ and $(O(d) - X \leq 0, L(X)) \in \mathcal{C}$.*
2. *For each $D! \in \mathcal{DT}$, where $D! = O(d)$, we have that $d \notin L(D!)$ (no self labeling).*

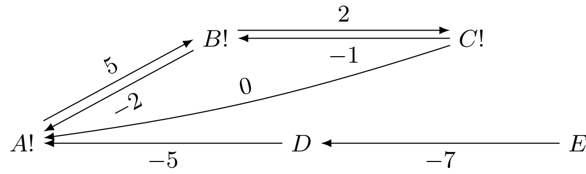
The first modification says that X can be executed at the same time of $D!$ (but instantaneously after $D!$ since time points executed at the same instant must follow an *order of execution*). This is because d is under control. The second modification serves to avoid the following circularity: If $L(D!) = d$, then $D!$ can be executed if and only if $d = \top$ (i.e., instantaneously after itself), but d can be assigned a truth value only upon the execution of $D!$. This problem does not happen with observation time points as time point label honesty would insert the implicit negative loop $(O(p) - P? \leq -\epsilon)$. However, for decision time points $(O(d) - D! \leq 0)$ trivially holds.

Figure 4.1a is an example of well-defined STND modeling a temporal plan with 3 decisions ($A!$, $B!$ and $C!$) and two (instantaneous) activities (D and E). $A!$ is the first time point to execute. We execute $B!$ if and only if we decided $\top$ for a ($L(B!) = a$) and $C!$ if we further decided $\top$ for b too ($L(C!) = ab$). Instead, $A!$, D and E are always executed ($L(A!) = L(D) = L(C) = \boxdot$). We can execute $B!$ (if we decide so) after at least 2 ($B! \to A!$ labeled by $\langle -2, a\rangle$) and within 5 time units since $A!$ was executed ($A! \to B!$ labeled by $\langle 5, a\rangle$). The same happens for $C!$ with respect to $B!$ (after 1 and within 2 time units since $B!$). Instead, we always execute D after minimum 5 time units since $A!$ ($D \to A!$ labeled by $\langle -5, \boxdot\rangle$) and E after 7 time units since D ($E \to D$ labeled by $\langle -7, \boxdot\rangle$). Furthermore,

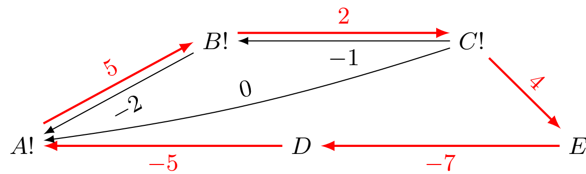
- if we decided $\perp$ for a, then we must execute E within 10 time units since $A!$ ($A! \to E$ labeled by $\langle 10, \neg a\rangle$), whereas
- if we decided $\top$ for a and $\perp$ for b, then we must execute E within 6 time units since $B!$ ($B! \to E$ labeled by $\langle 6, a\neg b\rangle$), and finally



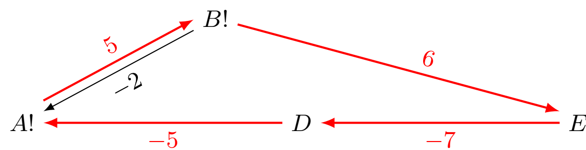
(a) *Simple Temporal Network with Decisions.*



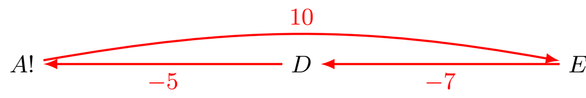
(b) *STN-projection onto* $s(a) = s(b) = s(c) = \top$.



(c) *STN-projection onto* $s(a) = s(b) = \top, s(c) = \bot$.



(d) *STN-projection onto* $s(a) = \top, s(b) = \bot, s(c) \in \{\top, \bot\}$.



(e) *STN-projection onto* $s(a) = \bot, s(b), s(c) \in \{\top, \bot\}$.

Fig. 4.1: An example of STND and related STN-projections (where thick red edges highlight negative cycles).

- if we decided $\top$ for both a and b and $\bot$ for c, then we must execute E within 4 time units since $C!$ ($C! \to E$ labeled by $\langle 4, ab\neg c\rangle$)

Algorithm 5: `STND-HSCC1`($\mathcal{S}$)

Input: An STND $\mathcal{S} = \langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$.
Output: A solution $\langle s, S \rangle$ for $\mathcal{S}$ if $\mathcal{S}$ is consistent; "*inconsistent*" if $\mathcal{S}$ is inconsistent. `STN-CC` is a consistency checking algorithm for STNs.

```
1 φ ← ⋀_{p∈P}(p ∨ ¬p)                          ▷ Make every assignment possible
2 while true do
3     s ← SAT-SOLVE(φ)                          ▷ Try to find a satisfying assignment s
4     if φ is unsatisfiable then
5         return inconsistent
6     ⟨T_s, C_s⟩ ← π_s(S)                       ▷ Project S onto s
7     if STN-CC(⟨T_s, C_s⟩) then
8         return ⟨s, S⟩                         ▷ where S is a valid schedule for ⟨T_s, C_s⟩
9     φ ← φ ∧ CYCLE-CUT(S, ⟨T_s, C_s⟩)          ▷ Cut the negative cycle
```

Negative values on edges model delays whereas positive ones model deadlines.

I propose here two hybrid SAT-based consistency checking algorithms (HSCC) to check the consistency of an STND and I report on the experimental evaluation that I carried out with KAPPA, a tool for STNDs. The approach is based on shortest paths algorithms and SAT solvers.

4.2 `STND-HSCC1`

In this section I provide `STND-HSCC1` a *hybrid SAT-based consistency checking (HSCC)* algorithm to check the consistency of an STND. Algorithm 5[1] maintains a formula φ specifying CNF clauses over propositions in $\mathcal{P}$. Initially φ allows for all possible truth value assignments. In each round of the algorithm we ask the SAT solver for a truth value assignment making φ true. Such an assignment (if any) corresponds to a scenario s over which we can project the STND and check if the resulting STN is consistent ("SAT-solver influences directed weighted graph algorithm"). If so, we return this scenario and a valid schedule for the projected STN (i.e., a solution). Otherwise, we apply De Morgan's rules to the negation of the *relevant part* of the scenario containing the negative cycle (`CYCLE-CUT` in Algorithm 6) and add the resulting clause to φ and go ahead with the next round ("directed weighted graph algorithm influences the SAT-solver"). If φ has become unsatisfiable it means that all STN-projections are inconsistent and therefore the initial STND is inconsistent. This makes the approach hybrid.

More concretely, an example of round for Figure 4.1a is as follows. Suppose that

$$\texttt{SAT-SOLVE}(\varphi) = ab\neg c$$

that is, $s(a) = s(b) = \top$ and $s(c) = \bot$. Since the STN $\pi_{ab\neg c}(\mathcal{S})$ is inconsistent (Figure 4.1c admits a negative cycle), we will add to φ the clause $\neg(a \wedge b \wedge \neg c)$, which simplifies to $(\neg a \vee \neg b \vee c)$, to ask the SAT solver for a different truth value

[1] I renamed `STND-CC` [19] to `STND-HSCC1`. I also simplified the algorithm to ease reading.

Algorithm 6: `CYCLE-CUT`$(\mathcal{S}, \langle \mathcal{T}_s, \mathcal{C}_s \rangle)$

Input: An STND $\mathcal{S} = \langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ and one of its negative cycles $\langle \mathcal{T}_s, \mathcal{C}_s \rangle$
Output: A clause ψ expressing the cut of the relevant part of the negative cycle.
1 $\psi \leftarrow \top$ ▷ ψ will contain the relevant part of s
2 **foreach** *constraint* $c \in \mathcal{C}_s$ **do**
3 $\quad \psi \leftarrow \psi \wedge \ell_c$ ▷ where ℓ_c is the corresponding label of c in $\mathcal{S}$
4 **return** `DeMorgan`$(\neg(\psi))$ ▷ the clause expressing the cut of the negative cycle

assignment excluding this projection (if any). Figure 4.1a is consistent if and only if

$$s(a) = s(b) = s(c) = \top$$

A possible schedule for this projection (shown in Figure 4.1b) is

$$S(A!) = 0, S(B!) = 2, S(C!) = 3, S(D) = 5, S(E) = 12$$

Any other combination leads to a projection containing a negative cycle (Figure 4.1c-4.1e).

The complexity of `STND-HSCC1` is $\mathcal{O}(2^{2|\mathcal{P}|}(|\mathcal{T}|^2 + |\mathcal{T}||\mathcal{C}| + |\mathcal{P}|)$ as initializing φ is linear in $|\mathcal{P}|$, each SAT solver call has cost $2^{|\mathcal{P}|}$, the projection has cost $|\mathcal{T}| + |\mathcal{C}|$, `STN-CC` has cost $|\mathcal{T}||\mathcal{C}|$ using Bellman-Ford, the time to apply De Morgan's laws is linear in $|\mathcal{P}|$ and the maximum number of iterations of the while loop is $2^{|\mathcal{P}|}$.

4.3 `STND-HSCC2`

In this section, I provide `STND-HSCC2`, a faster HSCC algorithm for STNDs.

`STND-HSCC1` is correct [19], but it suffers from the limitation that the projections are tested only when the SAT solver returns a complete truth value assignment. Consider Figure 4.1a and assume that the SAT solver starts on the formula $\varphi = (a \vee \neg a) \wedge (b \vee \neg b) \wedge (c \vee \neg c)$ which makes every truth value assignment possible. Suppose that in the search tree the SAT solver decides $\bot$ for a proposition d going down to the left and $\top$ going down to the right, and assume that the default policy is going down to the left. The first truth value assignment returned is $a = \bot$, $b = \bot$ and $c = \bot$ (corresponding to the scenario $s(a) = s(b) = s(c) = \bot$). Now `STND-HSCC1` would project Figure 4.1a onto s to obtain the STN shown in Figure 4.1e and eventually detect the negative cycle. However, the negative cycle could have been detected much earlier, say, when a was assigned $\bot$. Indeed, all projections of any scenario containing $s(a) = \bot$ boil down to Figure 4.1e (no matter which Boolean values are assigned to b and c). Therefore, a clever implementation of this algorithm calls for an early detection of negative cycles. However, before proceeding with it, I must refine the concept of scenario and projection so that they support "unknown" propositions (i.e., propositions that have not been assigned a value yet).

Definition 4.3. A *scenario* is (now) a mapping $s\colon \mathcal{P} \to \{\top, \bot, -\}$ assigning either *true*, *false* or *unknown* to each proposition $d \in \mathcal{P}$. A scenario s *satisfies* a label ℓ if ℓ evaluates to true under the following interpretation given by s:

Algorithm 7: `STND-HSCC2`($\mathcal{S}$, *all*)

Input: An STND $\mathcal{S} = \langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ and a Boolean value *all* meaning *all consistent scenarios* iff $all = \top$.

Output: A single or all scenarios s along with schedule(s) S for the projection $\pi_s(\mathcal{S})$ if $\mathcal{S}$ is consistent, "*inconsistent*" if $\mathcal{S}$ is inconsistent.

```
1  φ ← ⋀_{p∈P}(p ∨ ¬p)                              ▷ Make every assignment possible
2  Sols = ∅              ▷ The set of (all) consistent scenarios (global variable)
3  Hook a listener to run of the SAT solver and make it detect negative cycles as
     early as possible (on blocks (below) define the event-driven behavior).
4  SAT-SOLVE(φ)
5  if Sols = ∅ then
6      return inconsistent
7  return Sols
8  on assume d = ⊤ or assume d = ⊥:                                  ▷ Partial model
9      Build a scenario s from the current truth value assignment extended with
         s(d) = ⊤ or s(d) = ⊥ (depending on the case)
10     ⟨T_s, C_s⟩ ← π_s(S)                                  ▷ Get the STN projection
11     if BellmanFord(⟨T_s, C_s⟩) detects a negative cycle then
12         φ ← φ ∧ CYCLE-CUT(S, ⟨T_s, C_s⟩)                   ▷ Add clause on the fly
13 on solution found:                                               ▷ Complete model
14     Build a scenario s from the current truth value assignment
15     ⟨T_s, C_s⟩ ← π_s(S)                                  ▷ Get the STN projection
16     if BellmanFord(⟨T_s, C_s⟩) does not detect any negative cycle then
17         Sols ← Sols ∪ {⟨s, S⟩}                     ▷ S is a schedule for ⟨T_s, C_s⟩
18         if all = ⊤ then
19             φ ← φ ∪ (¬s)                       ▷ Exclude this consistent scenario
20     else
21         CYCLE-CUT(S, ⟨T_s, C_s⟩)                           ▷ Cut the negative cycle
```

1. $s \models \lambda$ iff $(\lambda = d \wedge s(d) = \top)$ or $(\lambda = \neg d \wedge s(d) = \bot)$,
2. $s \models \ell$ iff $s \models \lambda_1$ and ... and $s \models \lambda_n$ for $\ell = \lambda_1 \ldots \lambda_n$.

s never satisfies any literal whose embedded proposition d is unknown (i.e., $s \not\models \lambda$, iff $\lambda \in \{d, \neg d\}$ and $s(d) = -$).

The definition of STN projection remains the same as that given at the end of Definition 4.1 but extended to the new definition of scenario. As a result, Figure 4.1e now becomes a representative also for any scenario s such that $s(a) = \bot$ and $s(b), s(c) \in \{\top, \bot, -\}$. Another example is Figure 4.1d, extending $s(c) \in \{\top, \bot\}$ to $s(c) \in \{\top, \bot, -\}$. Now I have everything I need to hunt down inconsistent scenarios as early as possible.

`STND-HSCC2` (Algorithm 7) is a faster algorithm to check the consistency of STNDs. It allows for the synthesis of a single or all scenarios admitting a consistent schedule for the corresponding STN-projection. `STND-HSCC2` still initializes a CNF formula φ making all truth value assignments possible. Then, it starts the SAT-solver and hooks a listener to the corresponding run. Such a listener is able to

operate on φ by adding CNF clauses on the fly if needed and is triggered by two main events: *assume* and *solution found*.

- An *assume* ($d = \top$ or $d = \bot$) event (Algorithm 7, lines 8-12) triggers an action of the listener to "look ahead" if the STN-projection obtained by projecting the STND onto the scenario built from the current truth value assignment and *extended* with this assumption contains a negative cycle. If so, we extend φ by adding (on the fly) a CNF clause modeling the negation of the part of s containing a negative cycle in order to avoid getting the same scenario again. If the projection is consistent, `STND-HSCC2` does nothing and lets the run go.
- A *solution found* event (Algorithm 7, lines 13-21) extends the behavior of the listener described for *assume* as follows. When triggered, the listener builds a scenario from the current truth value assignment (which does not need to be extended with anything else this time). Then, it checks if the corresponding STN-projection contains a negative cycle. If it does not, then it computes an (early) schedule S for the STN projected onto s and adds the pair $\langle s, S\rangle$ to the set of solutions. Then, if all consistent scenarios are sought, a clause negating the current consistent scenario is added to φ. Instead, if the projection contains a negative cycle, then it acts as for *assume* events.

Eventually, when the run of the SAT solver ends, either $Sols = \emptyset$ (and thus the starting STND is inconsistent), or $Sols$ contains at least 1 solution (scenario-schedule).

The complexity of `STND-HSCC2` is $\mathcal{O}(|\mathcal{P}|2^{2|\mathcal{P}|}(|\mathcal{T}|^2 + |\mathcal{T}||\mathcal{C}| + |\mathcal{P}|)$ as each call of the SAT solver now also tests projections whenever the truth value of a new literal is assumed. Although the complexity looks a little bit worse, the experimental evaluation will show that `STND-HSCC2` is actually a better choice overall, thanks to its pruning technique aimed at detecting inconsistent scenarios early. If an STND is consistent for all $2^{|\mathcal{P}|}$ scenarios, `STND-HSCC1` will perform better.

4.4 Correctness

I proceed by discussing correctness of `STND-HSCC1` and `STND-HSCC2` and by reporting on the experimental evaluation that I carried out to compare `STND-HSCC2` with `STND-HSCC1`.

Definition 4.4. *A consistency algorithm (for a temporal network) is* sound *if whenever it says "inconsistent", the temporal network is really inconsistent, and it is* complete *if the algorithm says "inconsistent" for each inconsistent temporal network.*

`STND-HSCC1` and `STND-HSCC2` are sound and complete because they are based on a SAT-solver that allows for the iteration on all models. Whenever we add a clause, we exclude a relevant part of a scenario that we do not want to get anymore. The sooner, the better (`STND-HSCC2`).

Besides the SAT solver, all other internal sub-procedures (mostly, algorithms for directed weighted graphs) are well known to be sound and complete, and run in polynomial time. A possible bottleneck is, however, given by the size of the

initial set of constraints $\mathcal{C}$ of the STND. In the worst case, an STND may specify $((2^{|\mathcal{T}|}+1)|\mathcal{T}|^2)$ constraints. This happens when all time points are decision time points ($\mathcal{T} = \mathcal{DT}$) labeled by $\boxdot$, the STND graph is dense (admits $|\mathcal{T}|^2$ edges $X! \to Y!$) and each edge $X! \to Y!$ is labeled by $2^{|\mathcal{T}|}+1$ labels $\langle k, \ell \rangle$ (the $+1$ considers the case $\ell = \boxdot$), where each $\langle k_1, \ell_1 \rangle$ is such that there does not exist any (superfluous) $\langle k_2, \ell_2 \rangle$ such that $\ell_2 \Rightarrow \ell_1$ and $k_1 < k_2$; e.g., $\langle -3, a \rangle$ is enough to make $\langle -2, ab \rangle$ hold on the same edge.

4.5 Kappa: A Tool for STNDs

I developed Kappa, a tool for STNDs that takes in input a specification of an STND and acts both as a solver and a solution verifier. Kappa relies on SAT4J [11], a Java library compliant with the IPASIR interface that specifies how to interact with a SAT solver [6].

Kappa implements both `STND-HSCC1` and `STND-HSCC2`. I extended `STND-HSCC1` so that it allows for the synthesis of all consistent scenarios as well. In this way, I was able to carry out a more accurate experimental evaluation comparing the two algorithms when seeking single or all consistent scenarios. Listing 4.1 shows Kappa's help screen.

Listing 4.1: Kappa's help screen.

```
Usage: java -jar kappa.jar <Network.stnd> <Action> <Network.s> [--silent]
Action:
       --hscc1       seeks a single consistent scenario (STND-HSCC1)
       --hscc1-all   seeks all consistent scenarios (STND-HSCC1)
       --hscc2       seeks a single consistent scenario (STND-HSCC2)
       --hscc2-all   seeks all consistent scenarios (STND-HSCC2)
       --verify      verifies (all) synthesized solution(s)
       --silent      suppresses output (optional)
```

The input language of Kappa comprises the following three main sections. The section `Propositions`

```
Propositions {
  p q ... r
}
```

specifies the set $\mathcal{P}$, here comprising of $p, q, \dots, r$ as an example. The section `TimePoints`

```
TimePoints {
  ...
  (D! : d : p !q ...)
  (X  :   : p !q ...)
  ...
}
```

specifies the sets $\mathcal{T}$ and $\mathcal{DT}$ as well as the mappings O and L, and provides here examples of the specification of: a decision time point $D!$ with $d = O(D!)$ and

$L(D!) = p\neg q\ldots$ and a general time point $X \notin \mathcal{DT} \cup \mathcal{OT}$ with $L(X) = p\neg q$. The section `Constraints`

```
Constraints {
  ...
  (X - Y <= 10 : !d)
  ...
}
```

specifies the set $\mathcal{C}$ and provides here an example of the specification of a constraint $(X - Y \leq 10, \neg d) \in \mathcal{C}$.

Listing 4.2 shows the specification of Figure 4.1a.

Listing 4.2: The specification of Figure 4.1a in the input language of KAPPA.

```
1  Propositions {
2    a b c
3  }
4
5  TimePoints {
6    (A! : a : )
7    (B! : b : a )
8    (C! : c : a b)
9    (D : )
10   (E : )
11 }
12
13 Constraints {
14   (A - B <= -2 : a)
15   (A - D <= -5 : )
16   (A - C <= 0 : a b)
17   (E - A <= 10 : !a)
18   (E - B <= 6 : a !b)
19   (E - C <= 4 : a b !c)
20   (D - E <= -7 : )
21   (B - C <= -1 : a b)
22   (B - A <= 5 : a)
23   (C - B <= 2 : a b)
24 }
```

Given an STND specification file `Network.stnd`, we check the consistency of the network by running one of the following:

```
1 $ java -jar kappa.jar Network.stnd --hscc1 Network.sol
2 $ java -jar kappa.jar Network.stnd --hscc2 Network.sol
3 $ java -jar kappa.jar Network.stnd --hscc1-all Network.sol
4 $ java -jar kappa.jar Network.stnd --hscc1-all Network.sol
```

where (1) seeks a single consistent scenario using `STND-HSCC1`, (2) does the same using `STND-HSCC2`, whereas (3) and (4) seek all consistent scenarios by using `STND-HSCC1` and `STND-HSCC2`, respectively. If the STND is consistent, KAPPA saves to file (`Network.sol`) the solution(s). We verify the synthesized solution(s) by running

```
$ java -jar kappa.jar Network.stnd --verify Network.sol
```

I ran Kappa on the STND in Figure 4.1a. I used a FreeBSD virtual machine run on top of a VMWare ESXi Hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM. The VM was assigned 16GB of RAM and full CPU power. Kappa proved in about 274 milliseconds for `--hscc1` and in about 264 milliseconds for `--hscc2` that the STND in Figure 4.1a is consistent. Kappa verified that the unique consistent scenario is $s(a) = s(b) = s(c) = \top$ whose associated schedule is that given at the end of the second section (see Listing 4.3 as an example of use). This example is available at http://regis.di.univr.it/ExampleSTND.tar.bz2.

Listing 4.3: Analyzing and verifying the STND in Figure 4.1a with Kappa. Z is a special time point that must occur before any other. This is to avoid getting schedules with negative numbers.

```
$ java -jar kappa.jar Example.stnd --hscc2-all Example.hscc2.all.sol
hscc2
all = true
Consistent
Saving schedules for 1 scenario(s) to
Example.hscc2.all.sol

$ java -jar kappa.jar Example.stnd --verify Example.hscc2.all.sol
Scenario: a b c
Schedule: {Z = 0, A = 0, B = 2, C = 3, D = 5, E = 12}
SAT!
```

I also implemented Kappa in order to carry out an automated experimental evaluation to compare the performances of `STND-HSCC1` and `STND-HSCC2` when seeking single or all consistent scenarios.

I generated 2200 STNDs partitioned in 11 sets of benchmarks each one containing 100 consistent STNDs and 100 inconsistent STNDs. Regardless of the set, each STND has exactly 100 time points. The first set (`100TimePoints/10Decisions`) specifies 10 decision time points, the second set (`100TimePoints/11Decisions`) specifies 11 decision time points and so on, up to the eleventh one (`100TimePoints/20Decisions`) that specifies 20 decision time points. Each STND has a maximum number of constraints of $|\mathcal{T}| \times |\mathcal{DT}|$. Time points and constraints are randomly labeled so that the resulting STND is well defined. The weights on labeled edges range from -100 to 100. These sets of benchmarks are available at http://regis.di.univr.it/EE_STND2018.tar.bz2.

Algorithm 8 shows the pseudo-code of the algorithm I developed to to generate the networks.

I ran Kappa on these sets of benchmarks without imposing any timeout. I ran it four times in order to collect data (time and space) for both `STND-HSCC1` and `STND-HSCC2` when seeking a single or all consistent scenarios.

I show the graphical data in Figure 4.2, where x-axes always represent the number (#) of decision time points (i.e., the set of benchmarks under analysis)

Algorithm 8: `STND-Generator`(n, k, m, l, u)

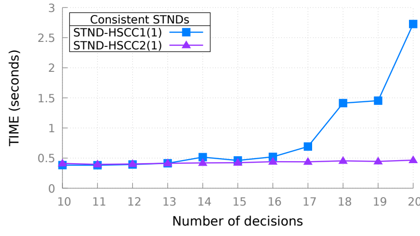
Input: An exact number of time points n, an exact number of decision time points $k \leq n$, a maximal number of constraints m, a lower and upper bound for the weights $l \leq u$.

Output: A well-defined STND where each proposition will label some component.

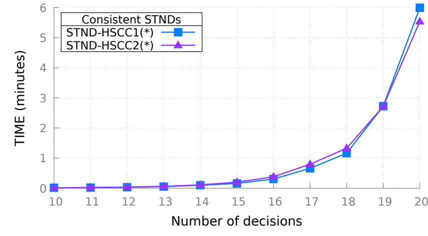
```
1  Let 𝒵 ← ⟨𝒯, 𝒟𝒯, 𝒫, O, L, 𝒞⟩                                ▷ Empty STND
   ▷ Generate time points and related propositions
2  𝒟𝒯 ← {D_i! | 1 ≤ i ≤ k}
3  𝒫 ← {d_i | 1 ≤ i ≤ k}
4  O(d_i) = D_i! for 1 ≤ i ≤ k
   ▷ Generate the rest of time points
5  𝒯 ← 𝒟𝒯 ∪ {X_i | 1 ≤ i ≤ (n − k)}
6  L(D_1!) = ⊡                                    ▷ One decision must be unlabeled
   ▷ Label time points as follows
7  for X ∈ 𝒯 where X ≠ D_1! do
8      L(X) = ⊡
9      maxLength ← Random(0, |𝒫|)                    ▷ Random length of the label
10     for i ← 0 to maxLength do
11         p ← Random(𝒫)                                   ▷ Random proposition
12         if L(X) ∧ p ∧ L(O(p)) is consistent then
13             L(X) ← L(X) ∧ p ∧ L(O(p))
   ▷ Generate constraints
14 for i ← 1 to m do
15     X, Y ← two Random time points in 𝒯
16     if L(X) ∧ L(Y) is consistent then
17         maxLength ← Random(0, |𝒫|)             ▷ Random extension of the label
18         for j ← 0 to maxLength do
19             p ← Random(𝒫)                               ▷ Random proposition
20             if L(X) ∧ L(Y) ∧ p ∧ L(O(p)) is consistent then
21                 ℓ ← L(X) ∧ L(Y) ∧ p ∧ L(O(p))
22         k ← Random(l, u)                                    ▷ Random weight
23         Add (Y − X ≤ k, ℓ) to 𝒞
   ▷ Final check
24 if Some proposition never appears in any label then
25     Throw away the network
26 return 𝒵
```

and y-axes represent either the average time elapsed or space consumed when analyzing the instances in that set.

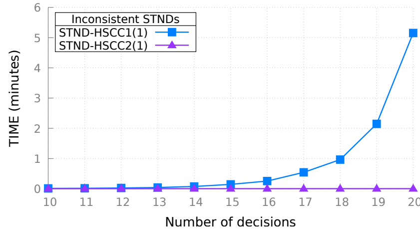
Figure 4.2a shows the results of the analysis run on the sets of benchmarks containing consistent STNDs when seeking a single consistent scenario. The graph shows that `STND-HSCC2` is significantly faster than `STND-HSCC1` for STNDs specifying more than 16 decision time points. Figure 4.2b shows the results of the same analysis when seeking all consistent scenarios: despite a normal general worsening of performances (all consistent scenarios are sought and not just one) `STND-HSCC2`



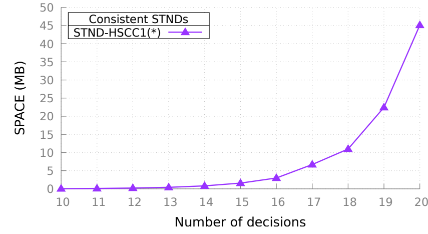
(a) *Consistency checking time on consistent STNDs when seeking a single consistent scenario.*



(b) *Consistency checking time on consistent STNDs when seeking all consistent scenarios.*



(c) *Consistency checking time on inconsistent STNDs.*



(d) *Space consumed when synthesizing all consistent scenarios.*

Fig. 4.2: Experimental evaluation with Kappa.

starts begin faster than `STND-HSCC1` for STNDs specifying more that 20 decisions. Figure 4.2c shows the results of the analysis on the sets of benchmarks containing inconsistent STNDs. `STND-HSCC2` has no competitors here, whereas `STND-HSCC1` starts having a serious exponential blow up for STNDs specifying more than 14 decisions. Figure 4.2d shows the average space consumed when synthesizing all consistent scenarios. The curve grows exponentially according to the number of decision time points (recall that `STND-HSCC1` and `STND-HSCC2` return the same set of consistent scenarios in such an analysis).

Finally, I verified all synthesized solutions. No constraint was violated.

4.6 Equivalence with DTP

Disjunctive temporal problems (DTPs) allow for disjunctions of temporal constraints (i.e., alternatives) in a temporal problem. For example, we might want that once an event modeled by a time point X happened, another event modeled by a time point Y happens *either* after 10 (seconds, minutes, hours, ...), *or* within 5. Such a constraint would look like

$$(X - Y \leq -10) \vee (Y - X \leq 5)$$

Any assignment of real values to X and Y satisfies the constraint if it satisfies (at least) one disjunct.

Differently from the first proposals of disjunctive temporal networks where disjunctions of intervals were allowed on the *same pairs of time points* only [53], the work I consider here is the one in [115] (originally proposed in [114]) not having such a restriction. I refer to the formalism in that work as the *disjunctive temporal network (DTN)*.

Definition 4.5. *A disjunctive temporal network (DTN) is a pair* $\langle \mathcal{T}, \mathcal{C} \rangle$*, where*

- $\mathcal{T}$ *is the usual finite set of time points, and*
- $\mathcal{C}$ *is a finite set of temporal constraints each one having the form*

$$\underbrace{(Y_1 - X_1 \leq k_1) \vee \cdots \vee (Y_n - X_n \leq k_n)}_{n \text{ disjuncts (atoms)}}$$

where $X_i, Y_i \in \mathcal{T}$ *and* $k_i \in \mathbb{R}$*. A temporal constraint is* non-disjunctive *if it contains one and only one disjunct,* disjunctive *otherwise.*
A DTN is consistent if there exists an assignment of real values to the time points (i) always satisfying all non-disjunctive constraints, and (ii) satisfying at least one disjunct for each disjunctive constraint.

We write $D(i)$ to shorten the ith disjunctive constraint and more specifically $D(i, j)$ to refer to the jth disjunct of the ith disjunctive constraint [115].

Since DTNs allow for the specification of disjunctions over different pairs of time points, as a minor contribution, I graphically represent a DTN through a *colored multi graph*, where *black edges* model non-disjunctive constraints (i.e., those constraints that must always hold), whereas *colored edges* (different from black) model disjunctive constraints (i.e., those $D(i)$s for which only a non-empty subset of disjuncts must hold). Each disjunctive constraint is assigned to a different color.

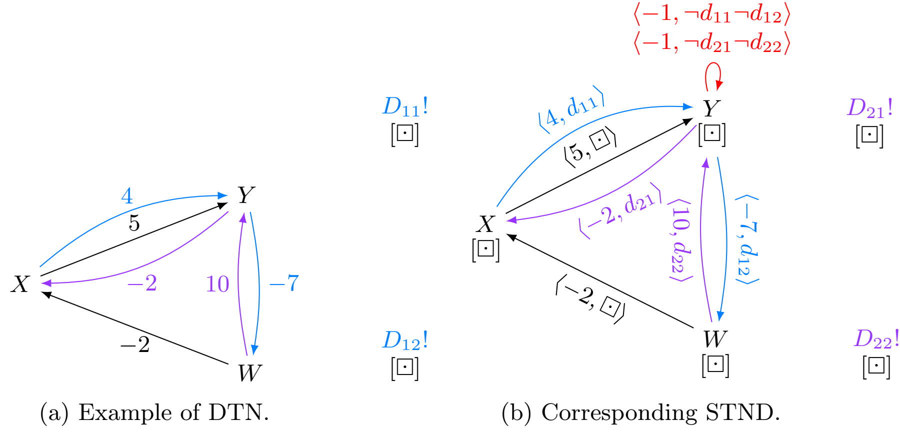
To give an example, consider the following DTN whose corresponding colored multi graph is shown in Figure 4.3a.

- $\mathcal{T} = \{X, Y, W\}$
- $\mathcal{C} = \{\overbrace{(Y - X \leq 5), (X - W \leq -2)}^{\text{must always hold}}, \overbrace{\overbrace{(Y - X \leq 4)}^{D(1,1)}, \overbrace{(W - Y \leq -7)}^{D(1,2)}}^{D(1)},$
$\underbrace{\underbrace{(X - Y \leq -2)}_{D(2,1)}, \underbrace{(Y - W \leq 10)}_{D(2,2)}}_{D(2)}\}$

The DTN in Figure 4.3a is consistent, the assignment $X = 0$, $Y = 3$ and $W = 5$ satisfies:

- all black constraints
- $D(1, 1)$ (but not $D(1, 2)$) for $D(1)$
- $D(2, 1)$ (and also $D(2, 2)$) for $D(2)$

I now proceed by proving that STNDs and DTNs are equivalent. I first give a strongly polynomial time encoding from DTNs to STNDs and then the vice versa.



(a) Example of DTN. (b) Corresponding STND.

Fig. 4.3: Representing and encoding DTNs into STNDs.

4.6.1 Encoding DTNs into STNDs

I encode the DTN in Figure 4.3a into the corresponding STND in Figure 4.3b as follows.

I generate a "core" STND containing all time points and all non-disjunctive constraints of the starting DTN and labeling them by ⊡ since all time points must always assigned a value and all non-disjunctive constraints must always be satisfied (black time points and constraints in Figure 4.3b).

For each disjunctive constraint $D(i)$ in the DTN, I add to the STND as many decision time points $D_{ij}!$ as the number of disjuncts $D(i, j)$. These decision time points are not constrained to any other time point in the STND (i.e., free to take any value). Any disjunct $D(i, j)$ in the DTN appears as a constraint in the STND labeled by d_{ij} (the proposition associated to $D_{ij}!$) so that when $d_{ij} = \top$, the disjunct of the DTN (labeled constraint in the STND) must hold and when $d_{ij} = \top$ we are not obliged to satisfy it. Furthermore, I must impose that at least one disjunct $D(i, j)$ for any disjunctive constraint $D(i)$ must hold (otherwise, it would be possible to disable them all by setting all d_{ij} to $\perp$). I enforce this condition by adding a negative self loop labeled by $\neg d_{ij_1} \ldots \neg d_{ij_n}$ on any time point of the STND.

In Figure 4.3b I added four decision time points $D_{11}!$, $D_{12}!$, $D_{21}!$ and $D_{21}!$ and the labeled constraints $X \to Y$ labeled by $\langle 4, d_{11}\rangle$, $Y \to W$ labeled by $\langle -7, d_{12}\rangle$, $Y \to X$ labeled by $\langle -2, d_{21}\rangle$ and $W \to Y$ labeled by $\langle 10, d_{22}\rangle$ to switch on and off $D(1, 1)$, $D(1, 2)$, $D(2, 1)$ and $D(2, 2)$ through the truth value assignments to $d_{11}!$, $d_{12}!$, $d_{21}!$ and $d_{22}!$. Finally, I added two negative self loops $Y \to Y$ labeled by $\langle -1, \neg d_{11} \neg d_{12}\rangle$ and $\langle -1, \neg d_{21} \neg d_{22}\rangle$ to prevent a disjunctive constraint $D(i)$ from being excluded. Note that the "-1" is meaningless: *any negative number* (e.g., -3, -159 or $-\epsilon$) is fine for this purpose. Likewise, the time point Y is meaningless too. Any time point would be fine for this purpose (e.g., $X \to X$ labeled by the same constraints). Negative self loops are the more intuitive way to enforce these

conditions. However, nothing would have prevented me from creating cycles of negative sum with respect to these labels involving many time points.

To ease reading, I colored the STND in Figure 4.3b with the same colors of the DTN in Figure 4.3a and showed the added negative cycles in red.

This encoding is strongly polynomial. The number of time points in the STND is the number of time points in the DTN plus as many decision time points as the number of disjuncts in the DTN (finite number), whereas the number of constraints in the STND is the the number of non-disjunctive constraints and disjuncts $D(i, j)$ in the DTN plus the number of disjunctive constraints $D(i)$ to model negative loops (finite number).

Any consistent scenario in the STND says which disjuncts (at least one for each disjunctive constraint) are satisfied for the synthesized solution. If the STND is inconsistent, so it the DTN.

4.6.2 Encoding STNDs into DTNs

I encode the STND in Figure 4.4a into the corresponding STND in Figure 4.4b as follows.

First of all, if the STND is not streamlined, I streamline it to get an STND having labels on constraints only (the process for streamlining CSTNs given at the end of Section 2.1.3 is applicable to STNDs too). Then, I generate a "core" DTN having the same set of time points of the STND and all constraints labeled by $\boxdot$ in the STND as non-disjunctive constraints in the DTN.

For each proposition d (associated to a decision time point $D!$) in the STND, I add to the DTN a time point d and the disjunctive constraint

$$\underbrace{(d - D! \leq 0)}_{\text{means } d = \bot} \vee \underbrace{(D! - d \leq -1)}_{\text{means } d = \top}$$

where the former says that d "occurs within" $D!$, whereas the latter says that d occurs after (minimum 1) since $D!$ (same way of reasoning of streamlined networks and horizons).

Now, every constraint $X \to Y$ labeled by $\langle k, d\neg ef \ldots\rangle$ (in the STND) implies the following "meta constraint" in the DTN:

$$(\underbrace{D! - d \leq -1}_{d} \wedge \underbrace{e - E! \leq 0}_{\neg e} \wedge \underbrace{F! - f \leq -1}_{f} \ldots) \Rightarrow Y - X \leq k$$

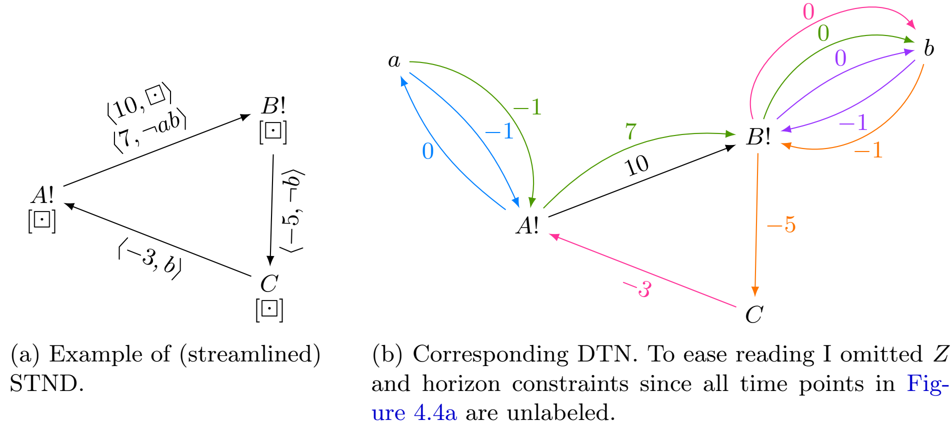
which can be rewritten as

$$\neg(D! - d \leq -1 \wedge e - E! \leq 0 \wedge F! - f \leq -1 \ldots) \vee Y - X \leq k$$

and finally simplified to

$$\underbrace{(d - D! \leq 0)}_{\neg d} \vee \underbrace{(E! - e \leq -1)}_{e} \vee \underbrace{(f - F! \leq 0)}_{\neg f} \cdots \vee (Y - X \leq k)$$

Note that for any $D!$ and d I define $\neg(D! - d \leq -1) \equiv d - D! \leq 0$ and $\neg(d - D! \leq 0) \equiv D! - d \leq -1$ (as a consequence of the first disjunctive constraint I



(a) Example of (streamlined) STND.

(b) Corresponding DTN. To ease reading I omitted Z and horizon constraints since all time points in Figure 4.4a are unlabeled.

Fig. 4.4: Encoding STNDs into DTNs.

added). Therefore, for any labeled constraint in the STND I add such a disjunctive constraint to the DTN.

In Figure 4.4b I add two time points a and b and the following constraints:

- $B! - A! \leq 10$
- $D(1)\colon (a - A! \leq 0) \vee (A! - a \leq -1)$
- $D(2)\colon (b - B! \leq 0) \vee (B! - b \leq -1)$
- $D(3)\colon (A! - a \leq -1) \vee (b - B! \leq 0) \vee (B! - A! \leq 7)$
- $D(4)\colon (B! - b \leq -1) \vee (C - B! \leq -5)$
- $D(5)\colon (b - B! \leq 0) \vee (A! - C \leq -3)$

I show the "colored" DTN graph in Figure 4.4b. Now, the DTN is consistent if and only if the STND is so. A solution of the DTN corresponds to a consistent scenario in the STND. The truth value assignment to the propositions in the STND depends on the real value assignments to the time points modeling those propositions in the DTN. For any proposition d in the STND, d is false iff in the DTN the time point d has a value not greater than $D!$ and d is true in the STND iff in the DTN the value of time point d is greater than $D!$ (the assignment to the other time points defines a schedule consistent for the scenario).

This encoding is strongly polynomial. The number of time points in the DTN is the same of that in the STND plus as many time points as the number of propositions in the STND, whereas the number of constraints in the DTN is given by the number of unlabeled constraints in the STND (non-disjunctive constraints in the DTN), plus as many disjunctive constraints as the number of labeled constraints in the STND (whose labels are different from ⊡). Also, for any disjunctive constraint $D(i)$ in the DTN, the number of disjuncts of $D(i)$ is $n+1$ where n is the number of literals contained in the label of the corresponding constraint in the STND and the "+1" refers to the inequality (numeric value of the constraint).

4.7 Conclusions

I defined simple temporal networks with decisions (STNDs) and provided `STND-HSCC1` and `STND-HSCC2` to synthesize a single or all consistent scenarios. These novel algorithms rely on a SAT solver and well-known shortest paths algorithms for directed weighted graphs. `STND-HSCC1` tests STN-projections for negative cycles by iterating on complete models returned by the SAT solver, whereas `STND-HSCC2` on partial ones. If the projected STN is inconsistent, I add a clause to the SAT solver to exclude the relevant part of that scenario, else I let the solver go. I provided Kappa, a tool for STNDs. I also discussed how to generate random temporal networks and carried out an experimental evaluation. In general `STND-HSCC2` is a better choice than `STND-HSCC1`.

STNDs are equivalent to DTNs which are a possible formalism to represent a DTP [115]. However, STNDs offer a more compact language to model any DTP. Also, STNDs are more compact than Labeled STNs (employed in Drake [45]) because Labeled STNs do not label nodes. Drake can make decisions anytime, whereas STNDs can make them only upon the execution of a decision time point. Also, STNDs employ a more structured approach and never change any constraints during execution. STNDs differ from TPNs [79] because decisions are not modeled as the choice of the outgoing edge from a decision node. STNDs differ from [125] as they model a decision problem and not an optimization problem. STNDs differ from all formalisms specifying uncontrollable parts (e.g., STNUs [104], CSTNs [75], CSTNUs [36, 74], TPNUs [90]) as all components are under control.

5

Conditional Simple Temporal Networks with Uncertainty and Decisions

In this chapter I address *fallback temporal plans*. I start by discussing a current limitation of CSTNUs that I solve in this chapter.

Figure 5.1 shows an example of CSTNU having two observation time points $P?, D?$ and four contingent links $(A_1, 1, 6, C_1)$, $(A_2, 8, 12, C_2)$, $(A_3, 3, 5, C_3)$ and $(A_4, 6, 10, C_4)$. $P?$ is the first time point to execute, whereas E is the last. If $P?$ assigns $\top$ to p (i.e., $s(p) = \top$), then A_2 and C_2 along with the constraints labeled by $\neg p$ turn irrelevant as $s \not\models \neg p$. If $P?$ assigns $\bot$ to p (i.e., $s(p) = \bot$), we will ignore A_1, C_1 and all constraints labeled by p. Likewise, if $D?$ assigns $\top$ (resp., $\bot$) to d, we will ignore A_4 and C_4 (resp., A_3 and C_3) and all constraints labeled by $\neg d$ (resp., d). The CSTNU in Figure 5.1 is uncontrollable. For example, assume that each contingent time point (if relevant) takes its maximal duration. If $s(p) = \top$ and $s(d) = \bot$, then the execution sequence is

$$P? = 0, A_1 = 1, C_1 = 7, D? = 8, A_4 = 9, C_4 = 19, E = 20$$

which violates $E \to P?$ labeled by $\langle -21, \neg d \rangle$ requiring that E must be executed after 21 since $P?$. If $s(p) = \bot$ and $s(d) = \top$, then the execution sequence is

$$P? = 0, A_2 = 1, C_2 = 13, D? = 14, A_3 = 15, C_3 = 20, E = 21$$

which violates $P? \to E$ labeled by $\langle 20, d \rangle$ requiring that E must be executed within 20 since $P?$. So,

What if we could decide d?

To achieve this purpose, I extend CSTNUs by injecting *decision time points*. A decision time point $D!$ dualizes an observation one $P?$ as the truth value assignment to the associated proposition is *under control* (recall STNDs, Chapter 4). As a result, the controllable and uncontrollable part may now mutually influence one another. That is, deciding some truth value may restrict (or even exclude) some uncontrollable part and vice versa. Several interesting cases may arise depending on if a few truth values are decided before or after having full information on how the uncontrollable part will or have behaved. I go ahead with this discussion by taking Figure 5.2 as an example. There, I took the initial CSTNU in Figure 5.1 and substituted decision time points for observation ones in all possible combinations.

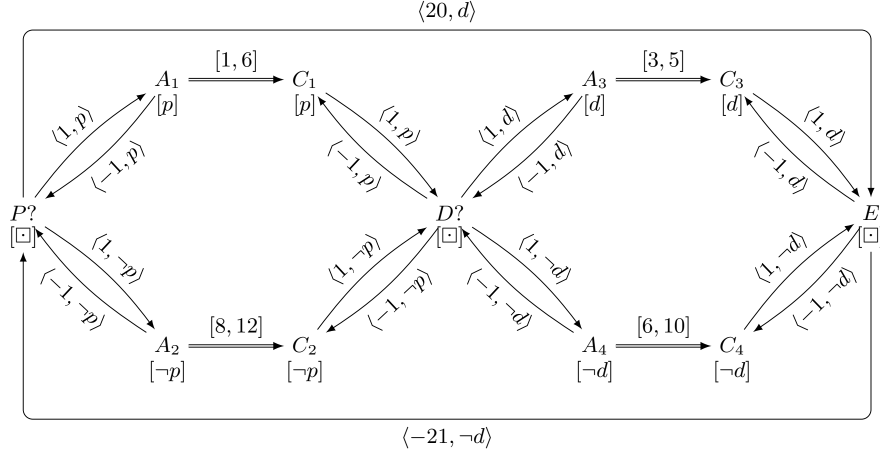
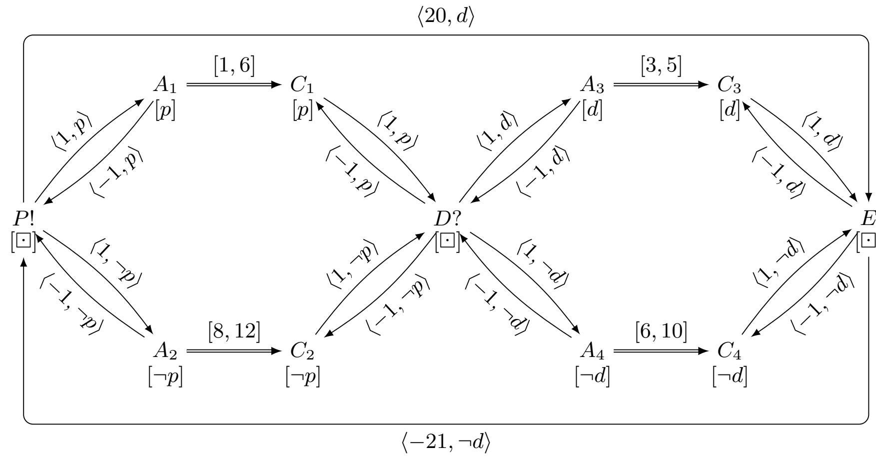


Fig. 5.1: Example of uncontrollable CSTNU.

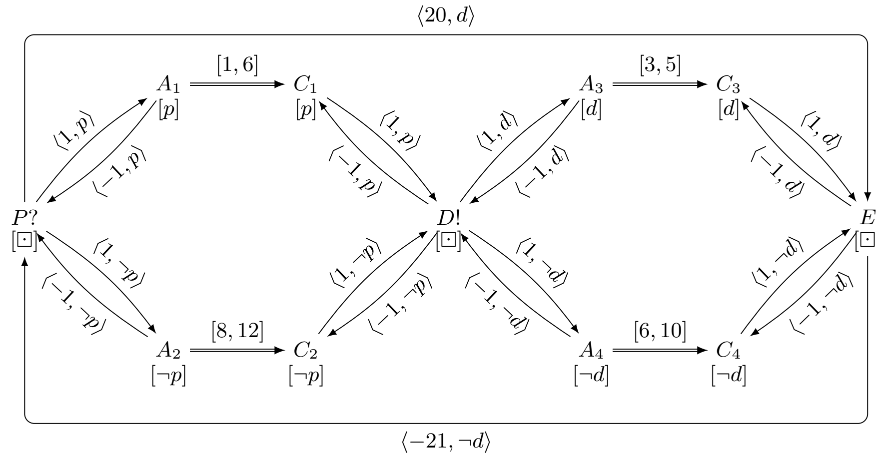
I discuss these examples by focusing on the combinations of minimal and maximal durations of contingent links only. If it works for them, then it must work for any other combination of durations.

1. In Figure 5.2a $P!$ is a decision time point. The resulting CSTNUD is uncontrollable. If we *decide* p (i.e., assign $\top$ to p), then *observe* $\neg d$ (i.e., $D?$ assigns $\bot$ to d) and C_1, C_4 take their maximal durations, then we will have to execute E at 20 violating $(P? - E \leq -21, \neg d)$ as $P?$ is executed at 0. Conversely, if we decide $\neg p$, then observe d and C_2 and C_3 take their maximal durations, then we will have to execute E at 21 (violating $E - P? \leq 20, d$).
2. In Figure 5.2b $D!$ is a decision time point. The resulting CSTNUD is DC. Assume that we observe p. Regardless on what duration C_1 takes, we can only decide d. Indeed, if we decided $\neg d$, regardless of the duration of C_4 we would have to execute E before time 21 violating $(P? - E \leq -21, \neg d)$. Assume now that we observe $\neg p$. If C_2 takes its minimal duration, d is the only good decision. If we decided $\neg d$ and then C_4 took its minimal duration, we would execute E at 18 violating $(P? - E \leq -21, \neg d)$. On the contrary, if C_2 takes its maximal duration then we can only decide $\neg d$. If we decided d and C_3 took its maximal duration, we would have to execute E at 21 violating $(E - P? \leq 20, d)$.
3. In Figure 5.2c $P!$ and $D!$ are both decision time points. The resulting STNUD (i.e., a CSTNUD without the "C" meaning no observation time points) is of course[1] dynamically controllable. If we decide p, then deciding d is always going to be fine. If we decide $\neg p$, then we will decide either d or $\neg d$ depending on how long C_2 lasts. If C_2 takes its minimal duration, then we will decide d (but not $\neg d$ since C_4 could then take its minimal duration). If C_2 takes its

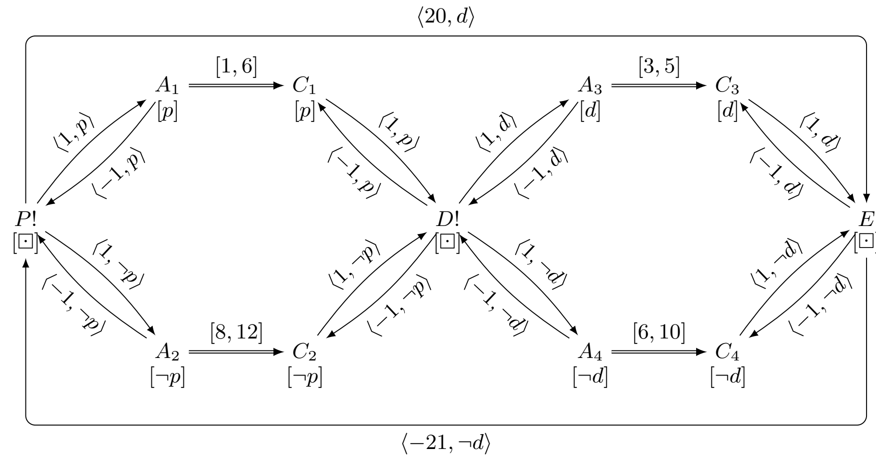
[1] If a network is DC (e.g., Figure 5.2b), then turning controllable some uncontrollable part (e.g., Figure 5.2c) *cannot worsen* the situation turning the network uncontrollable (it is like turning a $\forall$ into an $\exists$). The vice versa does not hold in general.



(a) *A decision before any uncontrollable part.*



(b) *A decision after all observations and some contingent.*



(c) *A decision after another decision and a contingent.*

Fig. 5.2: Possible cases of the CSTNU in Figure 5.1 when substituting decision time points for observation ones.

maximal duration, then we will decide $\neg d$ (but not d since if C_3 could then take its maximal duration).

Hence, *decisions are dynamic.*

5.1 Syntax

Definition 5.1 (CSTNUD). *A* Conditional Simple Temporal Network with Uncertainty and Decisions (CSTNUD) *is a tuple* $\langle \mathcal{T}, \mathcal{OT}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$*, where:*

1. $\mathcal{T}, \mathcal{OT}, \mathcal{P}, L, \mathcal{L}, \mathcal{C}$ *are exactly the same of those given for CSTNUs (Definition 2.15). Furthermore, I denote the set of contingent time points as* $Contingent = \{C \mid (A, x, y, C) \in \mathcal{L}\}$*.*
2. $\mathcal{DT} \subseteq \mathcal{T} = \{D!, \dots\}$ *is a set of* decision time points *such that* $\mathcal{OT} \cap \mathcal{DT} = \emptyset$*.*
3. $O \colon \mathcal{P} \to \mathcal{DT} \cup \mathcal{OT}$ *is a bijection associating a unique observation or decision time point to each proposition. If* $O(p) \in \mathcal{OT}$*, then* p *is called* observable*, whereas if* $O(d) \in \mathcal{DT}$*, then* d *is called* decidable*.* $\mathcal{OP} \subseteq \mathcal{P} = \{p \mid O(p) \in \mathcal{OT}\}$ *and* $\mathcal{DP} \subseteq \mathcal{P} = \{d \mid O(d) \in \mathcal{DT}\}$ *shorten the sets of all observable and decidable propositions, where* $\mathcal{OP} \cap \mathcal{DP} = \emptyset$ *(any proposition is either under control or out of control).*

A CSTNUD is well-defined *if and only if well-definedness properties for the "underlying" CSTNU part and STND part hold.*

5.2 Semantics

I model the execution semantics of a CSTNUD as a two-player game in which `Player1` models the controller and `Player2` models the environment. I employ *execution sequences* [102] to model the state of the game and define players' strategies as mappings from execution sequences considered at specific time instants to *moves*.

A *sequence* $\langle x_1, x_2, \dots, x_n \rangle$ is a totally ordered collection of elements such that for any pair of elements x_i, x_j, if $i < j$ (resp., $i > j$), then it means that x_i is before (resp., after) x_j. I abuse notation and write $\langle x_1, x_2, \dots, x_n \rangle \cup \langle x_p \rangle$ to mean the appending operation resulting in $\langle x_1, x_2, \dots, x_n, x_p \rangle$ where $n < p$. I write $x_i \in \langle x_1, x_2, \dots, x_n \rangle$ iff there exists $j \in \mathbb{N}$, $1 \leq j \leq n$ such that $x_i = x_j$ (membership), and $|\langle x_1, x_2, \dots, x_n \rangle| = n$ (cardinality). A *partial schedule* for a subset of time points $\mathcal{T}' \subseteq \mathcal{T}$ is a mapping $S_{\mathcal{T}'} \colon \mathcal{T}' \to \mathbb{R}$ assigning a real value to each $X \in \mathcal{T}'$. A *partial schedule* for a subset of Boolean propositions $\mathcal{P}' \subseteq \mathcal{P}$ is a mapping $S_{\mathcal{P}'} \colon \mathcal{P}' \to \{\top, \bot\}$ assigning either $\top$ or $\bot$ to each $p \in \mathcal{P}'$. I write $\mathtt{b}$ for a generic Boolean value (i.e., $\mathtt{b} \in \{\top, \bot\}$). I write $S_{\mathcal{T}'} \cup \{S_{\mathcal{T}'}(Y) = k\}$ to shorten that the domain of $S_{\mathcal{T}'}$ extends by adding time point Y such that $S_{\mathcal{T}'}(Y) = k$. Similarly, I write $S_{\mathcal{P}'} \cup \{S_{\mathcal{P}'}(p) = \mathtt{b}\}$ for Boolean propositions.

Definition 5.2 (Instantiation sequence). *An* instantiation sequence *is a quadruple* $\langle E, K, S_E, S_K \rangle$*, where* E *is a finite sequence of distinct time points in* $\mathcal{T}$*,* K *is a finite sequence of distinct propositions in* $\mathcal{P}$*, and* S_E*,* S_K *are partial schedules whose domains are* E *and* K*, respectively.*

Definition 5.3 (Execution sequence). *An* execution sequence $Z = \langle E, K, S_E, S_K \rangle$ *is an instantiation sequence satisfying the following properties:*

- S_E ***Monotonicity:*** *For any pair* $X_i, X_j \in E$ *if* $i < j$*, then* $S_E(X_i) \leq S_E(X_j)$*.*
- ***Time Point Label Honesty:*** *For each* $X \in E$ *and each literal* $\lambda \in L(X)$ *where* $\lambda \in \{p, \neg p\}$*, then* $O(p) \in E$ *and* $O(p)$ *is before* X*,* $p \in K$*,* $S_K(p) = \top$ *(if* $\lambda = p$*) and* $S_K(p) = \bot$ *(if* $\lambda = \neg p$*). Also,* $S_E(O(p)) < S_E(X)$ *(if* $p \in \mathcal{OP}$*) and* $S_E(O(p)) \leq S_E(X)$ *(if* $p \in \mathcal{DP}$*).*

Z^* *represents the set of all execution sequences.* $t_{last}(Z) = \max\{S_E(X) \mid X \in E\}$ *represents the last time instant in which a time point was executed in* Z*.* $last(Z) = \{X \mid X \in E \wedge S_E(X) = t_{last}\}$ *represents the set of the last executed time point(s).*

Therefore, an execution sequence models the ordered sequence of executed time points and assigned propositions according to the well-definedness of a CSTNUD. As an example, consider again Figure 5.2b. Assume that we execute $P?$ at 0 and observe $\neg p$. Assume then that we execute A_2 at 1 and observe C_2 to occur at 13 (i.e., at its maximal duration). The execution sequence is:

$$Z = \langle \langle P?, A_2, C_2 \rangle, \langle p \rangle, \{S_E(P?) = 0, S_E(A_2) = 1, S_E(C_2) = 13\}, \{S_K(p) = \bot\} \rangle$$

I can now compute the current partial scenario as the conjunction of all positive and negative literals arising from all propositions in K according to S_K and define local consistency.

Definition 5.4 (Current partial scenario). *Given any* $Z = \langle E, K, S_E, S_K \rangle$*, the* current partial scenario *is given by a label* $\ell_{cps} = \lambda_1 \wedge \cdots \wedge \lambda_k$*, where for each* $p_i \in K$*,* $\lambda_i = p_i$ *(if* $S_K(p_i) = \top$*) and* $\lambda_i = \neg p_i$ *(if* $S_K(p_i) = \bot$*).*

For Z, $\ell_{cps} = \neg p$ since $p \in K$ and $S_K(p) = \bot$.

Definition 5.5 (Local consistency). *An execution sequence* $Z = \langle E, K, S_E, S_K \rangle$*, is* locally consistent *if and only if for each* $(Y - X \leq k, \ell) \in \mathcal{C}$ *where* $X, Y \in E$ *and* $\ell_{cps} \Rightarrow \ell$*,* $S_E(Y) - S_E(X) \leq k$ *holds.*

Z is locally consistent since the schedule S_E satisfies $(A_2 - P? \leq 1, \neg p)$ and $(P? - A_2 \leq -1, \neg p)$. An execution sequence evolves over time according to the evolution of the game that `Player1` (the controller) plays against `Player2` (the environment). Each player follows a strategy saying what moves to make and when. Moreover, many moves can be made at the same time instant (provided that they respect an order) and sometimes moves are mandatory.

Definition 5.6 (Move). *A* move m *is either* X meaning "execute time point X" *or* $(p, \mathtt{b})$ meaning "assign $\mathtt{b} \in \{\top, \bot\}$ to proposition p". *A move for* `Player1` *requires that* X *is a* non-contingent *time point and* p *is a* decidable *proposition. A move for* `Player2` *requires that* X *is a* contingent *time point and* p *is an* observable *proposition.* M_1^* *and* M_2^* *represent the sets of all moves for* `Player1` *and* `Player2`*, respectively.*

A *move-based strategy* is a mapping from execution sequences considered at particular time instants to moves augmented with a `wait` condition modeling the absence of move. A strategy tells a player to make a move at a particular time instant only if the move is applicable at that particular time. Therefore, a strategy specifies *applicability conditions* saying when a move *can* be made, *obligations* saying when a move *has to* be made and *postconditions* saying how the execution sequence evolves accordingly.

Definition 5.7 (Move-based strategy). *A* move-based strategy *for* `Player1` *is a mapping* $\sigma_1 \colon Z^* \times \mathbb{R} \to M_1^* \cup \{\texttt{wait}\}$ *such that its* applicability conditions are*:*

1. *For any execution sequence* Z *and any time instant* t*,* $\sigma_1(Z,t)$ *is* applicable *iff* $t \sim t_{last}(Z)$*, where* $\sim$ *is* $>$ *if* $last(Z)$ *contains a contingent time point* C *or an observation time point* $P?$ *such that* K *contains its related proposition* p *(reaction time enforcement),* $\geq$ *otherwise.*
2. *For any execution sequence* Z *and any time instant* t*,* $\sigma_1(Z,t) = X$ *is applicable if (1) holds and* X *is an unexecuted non-contingent time point such that the current partial scenario entails* $L(X)$ *(i.e.,* $X \notin E \wedge X \notin \mathit{Contingent} \wedge \ell_{cps} \Rightarrow L(X)$*).*
3. *For any execution sequence* Z *and any time instant* t*,* $\sigma_1(Z,t) = \texttt{wait}$ *is applicable if (1) holds and there is no obligation at time* t*.*

The unique obligation *involves decidable propositions requiring that whenever a decision time point* $D!$ *has been executed and its related proposition* d *has not been assigned yet, then the strategy must issue a move to assign* d *a truth value instantaneously. That is,*

$$D! \in E \wedge d \notin K \implies \sigma_1(Z, S_E(D!)) = (d, \mathtt{b})$$

for any decision time point $D! \in \mathcal{DT}$*.*

A move-based strategy *for* `Player2` *is a mapping* $\sigma_2 \colon Z^* \times \mathbb{R} \to M_2^* \cup \{\texttt{wait}\}$ *such that its* applicability conditions *are:*

1. *For any execution sequence* Z *and any time instant* t*,* $\sigma_2(Z,t)$ *is* applicable *iff* $t \geq t_{last}(Z)$ *(no reaction time enforcement needed).*
2. *For any execution sequence* Z*, any time instant* t *and any contingent link* $(A,x,y,C) \in \mathcal{L}$*,* $\sigma_2(Z,t) = C$ *is applicable if (1) holds,* A *has already been executed,* C *has not, and executing* C *at this time satisfies* $C - A \in [x,y]$ *(i.e.,* $A \in E \wedge C \in \mathit{Contingent} \wedge C \notin E \wedge t - S_E(A) \in [x,y]$*).*
3. *For any execution sequence* Z *and any time instant* t*,* $\sigma_2(Z,t) = \texttt{wait}$ *is applicable if (1) holds and there is no obligation at time* t*.*

Obligations *are of two kinds. The first obligation involves observable propositions requiring that whenever an observation time point* $P?$ *has been executed and its related proposition* p *has not been assigned yet, then the strategy must issue a move to assign* p *a truth value instantaneously. That is,*

$$(P? \in E \wedge p \notin K) \implies \sigma_2(Z, S_E(P?)) = (p, \mathtt{b})$$

for any observation time point $P? \in \mathcal{OT}$*.*

The second obligation involves contingent links (A, x, y, C) requiring that if A has already been executed, C has not and the current time t is the last instant at which C can be executed, then the strategy must issue a move to execute C at t. That is,

$$(A \in E \wedge C \notin E \wedge t - S_E(A) = y) \implies \sigma_2(Z, t) = C$$

for any $(A, x, y, C) \in \mathcal{L}$ and any real time instant $t \in \mathbb{R}$.

Postconditions *for both σ_1 and σ_2 are the same. If the strategy tells the player to execute a time point X at time t, then Z updates by appending X to E and extending S_E such that $S_E(X) = t$. If the strategy tells the player to assign the truth value $\mathtt{b}$ to the proposition p, then Z updates by appending p to K and extending S_K such that $S_K(p) = \mathtt{b}$. In symbols:*

- *If $\sigma_i(Z, t) = X$, then $Post(Z, \sigma_i, t) = \langle E \cup \langle X \rangle, K, S_E \cup \{S_E(X) = t\}, S_K \rangle$*
- *If $\sigma_i(Z, t) = (p, \mathtt{b})$, then $Post(Z, \sigma_i, t) = \langle E, K \cup \langle p \rangle, S_E, S_K \cup \{S_K(p) = \mathtt{b}\} \rangle$*

Getting back to our example we have that $t_{last}(Z) = 13$ and $last(Z) = \{C_2\}$. Suppose that current time is $t = 14$. $\sigma_1(Z, 14) = D!$ is applicable since $t > t_{last}$ and $D!$ has not been executed yet, whereas $\sigma_1(Z, 14) = (d, \top)$ is not since $D! \notin E$. If $\sigma_1(Z, 14) = D!$ is taken into consideration (i.e., $Z' = Post(Z, \sigma_1, t)$), then $\sigma_1(Z', 14) = (d, \top)$ instantaneously after.

I model `Player2` as the *most powerful player possible*. If `Player1` can beat this (worst-case of) environment, then `Player1` must be able to beat any other less powerful environment playing the same game. To achieve this purpose I model the game in *turns*. That is, at any time instant t, there exist two turns: $T_1(t)$ (occurring first) and $T_2(t)$ (occurring last). `Player1` makes his moves in $T_1(t)$, whereas `Player2` makes his in $T_2(t)$. If player i does not make any move in $T_i(t)$, then he loses forever the possibility to play at time t. As a result, `Player2`, making his moves in $T_2(t)$, is guaranteed to always have full information on what `Player1` has done in $T_1(t)$ (worst-case scenario). In what remains of this section I define the concept of *snapshot* modeling an execution sequence at a particular time instant t (after the players are done in $T_1(t)$ and $T_2(t)$), *continuous game evolution* modeling how the execution sequence evolves and *winning conditions* for each player.

Definition 5.8 (Snapshot). *Let $Z = \langle E, K, S_E, S_K \rangle$ be any execution sequence. $Z(t) = \langle E', K', S'_E, S'_K \rangle$ models the* snapshot *of Z at time t, where*

- *$E' = \langle X \mid X \in E \wedge S_E(X) \leq t \rangle$*[2]
- *$K' = \langle p \mid p \in K \wedge O(p) \in E' \rangle$*
- *$\forall X \in E', S'_E(X) = S_E(X)$*
- *$\forall p \in K', S'_K(p) = S_K(p)$*

To give an example, let me get back to the execution sequence I have discussed before. At $t = 11$,

$$Z(11) = \langle \langle P?, A_2 \rangle, \langle p \rangle, \{S_E(P?) = 0, S_E(A_2) = 1\}, \{S_K(p) = \bot\} \rangle$$

[2] That is, E' is the subsequence of E up to time point X_i where $S_E(X_i) \leq t'$ (sequence comprehension).

Definition 5.9 (Continuous game evolution). *Let $t \in \mathbb{R}^{\geq 0}$ be the global time. The continuous game evolution is modeled by an infinite sequence of snapshots $Z(t)$ defined as:*

$$Z(t) = \begin{cases} T_2(T_1(\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle, t), t) & \text{if } t = 0 \\ T_2(T_1(Z(t-\epsilon), t), t) & \text{if } t > 0 \end{cases}$$

$$T_i(Z, t) = \begin{cases} Z & \text{if } \sigma_i(Z, t) = \texttt{wait} \\ T_i(Post(Z, \sigma_i, t), t) & \text{otherwise} \end{cases}$$

where $T_i(t)$ models the evolution of Z during turn i at time t according to σ_i, whereas an (arbitrary small) $\epsilon > 0$ models the reaction time.

Definition 5.10 (Winning conditions). `Player1` *wins the game if and only if the game evolution leads to a snapshot $Z(t)$ such that for each unexecuted time point X, ℓ_{cps} falsifies $L(X)$ and for each constraints $(Y - X \leq k, \ell)$ where $X, Y \in E$ and $\ell_{cps} \Rightarrow \ell$, $S_E(Y) - S_E(X) \leq k$ holds.* `Player2` *wins otherwise. σ_i is a* winning strategy *if player i wins the game by following σ_i.*

Definition 5.11 (Dynamic controllability). *A CSTNUD is dynamically controllable if* `Player1` *has a winning strategy such that for any $t > 0$ and any pair of execution sequences Z_1, Z_2,*

$$\text{If } \sigma_2(Z_1, t') = \sigma_2(Z_2, t') \text{ for } 0 \leq t' < t, \text{ then } \sigma_1(Z_1, t) = \sigma_1(Z_2, t)$$

In other words, whenever `Player2` has made the same sequence of moves up to time $t - \epsilon$, then `Player1` will make the same move(s) at time t.

5.3 Dynamic Controllability of CSTNUDs via TGAs

In this section I extend and optimize the encoding from CSTNUs into TGAs given in Section 2.2.2.

5.3.1 Extension

Consider, as an example, Figure 5.3 depicting the encoding of the CSTNUD in Figure 5.2b. Once again, there are three core locations but this time I borrow a few names from Section 2.2.2 and rename them to $\mathtt{T}_1$ (ex L_1), $\mathtt{T}_2$ (ex L_0) and `win` (ex `goal`). $\mathtt{T}_1$ and $\mathtt{T}_2$ model the two turns $T_1(t)$ and $T_2(t)$ when global time is > 0. $\mathtt{T}_2$ is the initial location. The winning path is computed in the same way, only renaming each L_{ℓ_i} to $\mathtt{w}_{\ell_i}$. `gain` and `pass` regulate the turns at any time instant t. I still have a clock `cX` for each $X \in \mathcal{T}$ (considering decision time points too) and a clock `bP` for each $p \in \mathcal{P}$ (considering decidable propositions too).

Let me now discuss how to model the truth value assignment to the decidable propositions. Dually to observable propositions, for each decidable proposition $d \in \mathcal{DP}$, I generate an uncontrollable self-loop transition $\langle \mathtt{T}_1, \mathtt{cD} < \hat{\mathtt{c}} \wedge \mathtt{cD} = 0 \wedge \mathtt{bD} = \hat{\mathtt{c}}, \mathtt{dFalse}, \{\mathtt{bD}\}, \mathtt{T}_1 \rangle$ at $\mathtt{T}_1$. If we take this transition, it means that we decide $\neg d$. If we don't take it, it means that we decide (actually confirm) d. In

the former case, such a transition has to be taken at the same instant at which $D!$ was executed but instantaneously after $\mathtt{ExD}$ was taken. In this way, I model "how" to decide the truth values of the propositions in $\mathcal{DP}$. All other transitions remain the same as those given for CSTNUs.

5.3.2 Optimizations

I have discussed the main extension of the encoding that concerns how to make decisions in our temporal plan. That is, how to assign truth values to the decidable propositions upon the execution of decision time points. Let me now optimize this whole encoding. I refine the guard of each uncontrollable self-loop at $\mathtt{T}_1$ by exploiting what I know of the CSTNUD. That is, I extend the guards so that they enforce time point label honesty as well as the partial order among the time points when not ambiguous. I first discussed in this optimization in [40] but there I dealt with disjunctive constraints and exploited internal data structures provided by the UPPAAL-TIGA software. Here, I propose a more formal definition avoiding such data structures. Moreover, in [40] I did not address decisions.

As an example of this optimization, consider time points A_1 and A_4 of the CSTNUD in Figure 5.2b. $L(A_1) = p$ and $L(A_4) = \neg d$. Recall that the encoding models p and d as two dedicated clocks $\mathtt{bP}$ and $\mathtt{bD}$ such that if one of these clocks is equal to (resp., less than) $\hat{\mathtt{c}}$, once its related observation or decision time point has been executed, then the related proposition is $\top$ (resp., $\bot$). Moreover, time point label honesty also requires that $P? - A_1 \leq -\epsilon$ (observations) and $D! - A_4 \leq 0$ (decisions).

Therefore, considering the time point label honesty property for CSTNUDs, it is possible to extend the guards of $\mathtt{ExA}_1$ and $\mathtt{ExA}_4$ by appending $\mathtt{bP} = \hat{\mathtt{c}} \wedge \mathtt{cP} < \hat{\mathtt{c}} \wedge \mathtt{cP} > 0$ and $\mathtt{bD} < \hat{\mathtt{c}} \wedge \mathtt{cD} < \hat{\mathtt{c}} \wedge \mathtt{cD} \geq 0$, respectively. The former models the fact that A_1 must be executed if only if $p = \top$ (i.e., $\mathtt{bP} = \hat{\mathtt{c}}$), which also implies that A_1 must be executed after $P?$ (i.e., $P?$ has been executed ($\mathtt{cP} < \hat{\mathtt{c}}$)) and a positive amount of time ϵ has elapsed ($\mathtt{cP} > 0$). The latter models the fact that A_4 must be executed if only if $d = \bot$ (i.e., $\mathtt{bD} < \hat{\mathtt{c}}$), which also implies that A_4 must be executed after $D!$ (i.e., $D!$ has been executed ($\mathtt{cD} < \hat{\mathtt{c}}$)) either instantaneously or after a positive amount of time has elapsed ($\mathtt{cD} \geq 0$).

Definition 5.12 (Encoding time point label honesty). *A* label encoder *is a mapping $L_{enc} \colon \mathcal{T} \to \mathcal{H}(\mathcal{X})$ translating the label of a time point into the equivalent clock constraint $L_{enc}(X) = L^{\mathcal{OP}}_{enc}(X) \wedge L^{\mathcal{DP}}_{enc}(X)$, where $L^{\mathcal{OP}}_{enc}(X)$ and $L^{\mathcal{DP}}_{enc}(X)$ encode all literals containing observable and decidable propositions, respectively:*

- $L^{\mathcal{OP}}_{enc}(X)$: $\bigwedge_{p \in L(X)} (\mathtt{bP} = \hat{\mathtt{c}} \wedge \mathtt{cP} < \hat{\mathtt{c}} \wedge \mathtt{cP} > 0) \bigwedge_{\neg q \in L(X)} (\mathtt{bQ} < \hat{\mathtt{c}} \wedge \mathtt{cQ} < \hat{\mathtt{c}} \wedge \mathtt{cQ} > 0)$.
- $L^{\mathcal{DP}}_{enc}(X)$: $\bigwedge_{d \in L(X)} (\mathtt{bD} = \hat{\mathtt{c}} \wedge \mathtt{cD} < \hat{\mathtt{c}} \wedge \mathtt{cD} \geq 0) \bigwedge_{\neg f \in L(X)} (\mathtt{bF} < \hat{\mathtt{c}} \wedge \mathtt{cF} < \hat{\mathtt{c}} \wedge \mathtt{cF} \geq 0)$. □

I now focus on constraints. Consider the pair of constraints $P? \to A_1$, $A_1 \to P?$ labeled by $\langle 1, p\rangle$ and $\langle -1, p\rangle$, respectively in the CSTNUD that I am discussing. Such constraints say that A_1 must be executed after 1 and within 1 since $P?$ (thus, exactly after 1 since $P?$). These constraints also have an important characteristic: $L(A_1)$ coincides with the label of the constraints. Therefore, whenever A_1 is

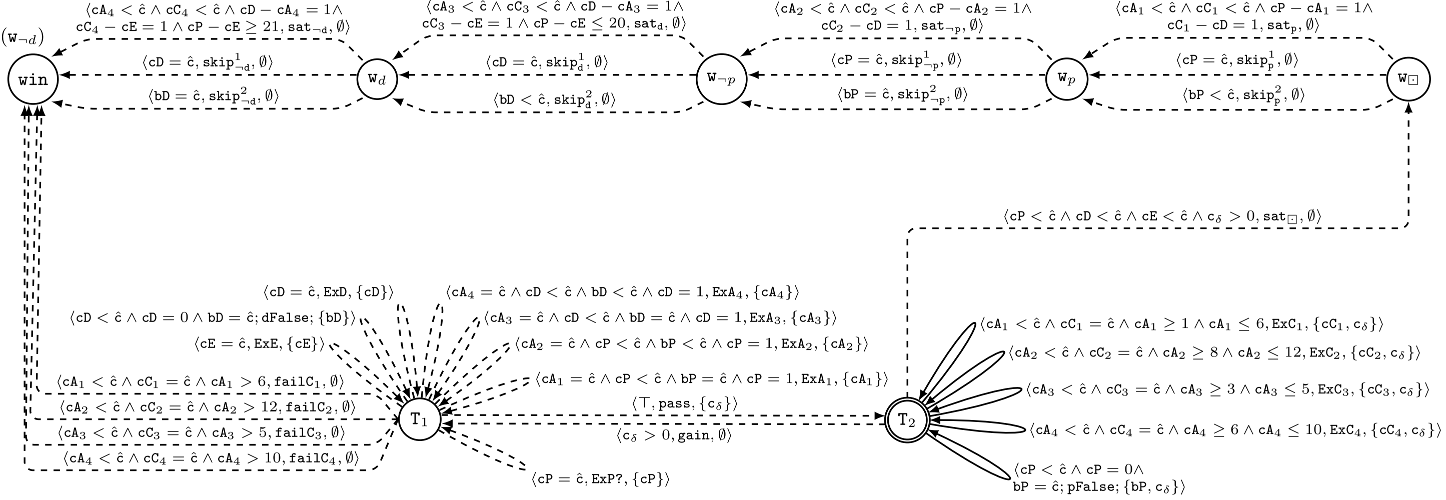


Fig. 5.3: TGA encoding the CSTNUD in Figure 5.2b. $\mathtt{T}_2$ (ex L_0) is the initial location (modeling $T_2(t)$ for $t > 0$). $\mathtt{T}_1$ (ex L_1) models $T_1(t)$ for $t > 0$). $\mathtt{w}_{\boxdot}$, $\mathtt{w}_p$, $\mathtt{w}_{\neg p}$, $\mathtt{w}_d$, $\mathtt{win}$ model the winning path. Some guards have been written in a compact notation. For example, we shorten $\mathtt{cD} \leq 1 \wedge \mathtt{cD} \geq 1$ as $\mathtt{cD} = 1$, $\mathtt{cD} \geq 0 \wedge \mathtt{cD} = 1$ as $\mathtt{cD} = 1$, and so on.

executed, the constraints must hold. Thus, I extend the original guard of $\texttt{ExA}_1$ (formerly $\texttt{cA}_1 = \hat{\texttt{c}}$) to $\texttt{cA}_1 = \hat{\texttt{c}} \wedge \texttt{cP} < \hat{\texttt{c}} \wedge \texttt{cP} = 1$, where the new conjuncts say that $P?$ has already been executed ($\texttt{cP} < \hat{\texttt{c}}$) and $A_1 - P? \in [1, 1]$ ($\texttt{cP} = 1$). More formally:

Definition 5.13 (Encoding predecessors). *Given a CSTNUD, a* predecessor *of a time point $Y \in \mathcal{T}$ is a time point $X \in \mathcal{T}$ such that there exists a constraint $(X - Y \leq -x, L(Y)) \in \mathcal{C}$ where $x > 0$. $\Pi : \mathcal{T} \rightarrow 2^{\mathcal{T}}$ returns the predecessors of a given time point and it is formalized as $\Pi(Y) = \{X \mid (X - Y \leq -x, \ell) \in \mathcal{C} \wedge x > 0 \wedge \ell = L(Y)\}$. A* predecessor encoder *is a mapping $\Pi_{enc} \colon \mathcal{T} \rightarrow \mathcal{H}(\mathcal{X})$ translating each $X \in \Pi(Y)$ (along with its temporal bounds) into an equivalent clock constraint as follows: $\Pi_{enc}(Y) = \bigwedge_{X \in \Pi(Y)} \texttt{cX} < \hat{\texttt{c}} \wedge \texttt{cX} \geq x \wedge \texttt{cX} \leq y$, where $\texttt{cX} \geq x$ models $(X - Y \leq -x, L(Y))$ and $\texttt{cX} \leq y$ models $(Y - X \leq y, L(Y))$ (if any).* □

Therefore, for each non-contingent time point X, the guard of $\texttt{ExX}$ becomes $\texttt{cX} = \hat{\texttt{c}} \wedge L_{enc}(X) \wedge \Pi_{enc}(X)$.

5.3.3 An optimized encoding for checking DC of CSTNUDs

Let me now join my extension and optimizations to define the full encoding.

Definition 5.14. *Encoding CSTNUDs into TGAs is achieved by extending and optimizing the encoding for CSTNUs discussed in Section 2.2.2 to:*

1. *model transitions to operate on decidable propositions (Section 5.3.1),*
2. *enforce time point label honesty in the transition guards (Definition 5.12 in Section 5.3.2), and*
3. *enforce predecessors in the transition guards (Definition 5.13 in Section 5.3.2).*

I point out that (1) serves to adapt the encoding to CSTNUDs, whereas (2) and (3) are optimizations to boost the model checking phase (therefore, they are also applicable to the old encoding for CSTNUs).

5.4 Correctness and Complexity of the Encoding

In this section, I prove the correctness and discuss the complexity of the encoding provided in Definition 5.14 in Section 5.3.

A controllability algorithm for a temporal network is *sound* if whenever the algorithm says "uncontrollable", the temporal network is really uncontrollable and it is *complete* if the algorithm says "uncontrollable" for each uncontrollable temporal network (e.g., see [75]). A controllability algorithm is *correct* if it is sound and complete.

Theorem 5.1. *Encoding CSTNUDs into TGAs according to Definition 5.14 is correct.*

Proof. To prove that, I start by showing that the encoding in Section 5.3 correctly models the move-based semantics given in Section 5.2. A state of the TGA is given by a pair $(L, \vec{c})$, where L is a locations and $\vec{c}$ represents the values of all clocks. The state of a CSTNUD during execution is given by its execution sequence Z. I show that the game interplay correctly models the continuous game evolution given in Definition 5.9 for all $t > 0$. I exclude the case for $t = 0$, so `Player1` does not play in $T_1(0)$ and `Player2` does not play in $T_2(0)$ either.

(Invariant) At any instant $t > 0$ the snapshot $Z(t) = \langle E, K, S_E, S_K \rangle$ corresponds to a state of the TGA $(L, \vec{c})$ in which $L = \mathtt{T}_2$ and $\vec{c}$ is as follows: $\hat{\mathtt{c}} = t$, $\mathtt{c}_\delta = 0$, for each $X \in \mathcal{T}$, $\mathtt{cX} < \hat{\mathtt{c}}$ and $\hat{\mathtt{c}} - \mathtt{cX} = k$ (if $X \in E \wedge S_E(X) = k$), $\mathtt{cX} = \hat{\mathtt{c}}$ otherwise. For each $p \in \mathcal{P}$, $\mathtt{cP} < \hat{\mathtt{c}} \wedge \mathtt{bP} = \hat{\mathtt{c}}$ (if $p \in K$ and $S_K(p) = \top$) and $\mathtt{cP} < \hat{\mathtt{c}} \wedge \mathtt{bP} < \hat{\mathtt{c}}$ (if $p \in K$ and $S_K(p) = \top$), $\mathtt{cP} = \mathtt{bP} = \hat{\mathtt{c}}$ otherwise. Finally, `Player2` has finished taking controllable transitions at t.

When $t = 0$ (i.e., $\hat{\mathtt{c}} = 0$) `Player2` cannot play in $\mathtt{T}_2$ as no controllable transition is enabled. `Player1` cannot play either because the current location is not $\mathtt{T}_1$ and he can only get there after a positive amount of time has elapsed. Therefore, at $t = 0$, $Z(0) = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$.

When $t > 0$ (i.e., $\hat{\mathtt{c}} > 0$) both `Player1` and `Player2` can play in their respective turns $T_1(t)$ and $T_2(t)$. `Player1` can take `gain` to enter $\mathtt{T}_1$ at time t. `Player2` cannot prevent him from doing so because `gain`, being urgent, has priority over any other controllable transition that `Player2` could take at that time. So, `Player1` plays first. Once entered $\mathtt{T}_1$, `Player1` can take (in general) a non-empty sequence of transitions to execute a few non contingent time points and decide the truth values of some decidable propositions (if he has executed some decision time points instantaneously before). Such a sequence is finite since there is a finite number of time points to execute and a finite number of decidable propositions to assign (one for each decision time point). Furthermore, each time point (resp., proposition) can be executed (resp., assigned a value) only once. When this sequence of transitions is over, `Player1` ends his turn by taking `pass` to lead the run back to $\mathtt{T}_2$. Since $\mathtt{T}_1$ is urgent, time has not elapsed. Therefore, the sequence of transitions taken at $\mathtt{T}_1$ corresponds to the sequence of moves made by `Player1` in $T_1(t)$. Instead, if `Player1` wants to wait at time t, he can either take `gain` and `pass` immediately after or just avoid taking `gain`. Now, at $\mathtt{T}_2$, `Player2` does the same for contingent time points and observable propositions if some observation time points have been executed by `Player1` in $T_1(t)$. When `Player2` is done, the sequence of transitions taken, models the sequence of moves made in $T_2(t)$. Since `Player2` does not make any other move in $T_2(t)$, $Z(t)$ no longer changes.

`Player1` and `Player2` are driven by their strategies σ_1 and σ_2 which say what moves to make (i.e., transitions to take) in $T_1(t)$ and $T_2(t)$ (i.e., locations $\mathtt{T}_1$ and $\mathtt{T}_2$) at any time t depending on the current Z. The purpose of σ_1 is to keep $Z(t)$ locally consistent, whereas that of σ_2 is the opposite.

The strategies also satisfy their applicability conditions as `Player1` can make his moves in $T_1(t)$ according to σ_1 iff `Player2` has not played yet in $T_2(t)$, whereas `Player2` can make his moves in $T_2(t)$ according to σ_2 if and only if either `Player1` has not played in $T_1(t)$ or `Player2` is not done in $T_2(t)$. I have already proved that for any $t > 0$, `Player1` plays first.

The strategies satisfy their obligations as each time `Player1` executes a decision time point $D!$, he also assigns the associated decidable proposition d a truth value as well. This occurs at the same time but instantaneously after the execution of $D!$. `Player1` assigns $\top$ to d by not taking `dFalse` and assigns $\bot$ to d by taking `pFalse`. If `Player1` takes the transition, then he will never be able to take it again in the same or following turns (as the guard of `pFalse` invalidates). If he does not, then he will never be able to take `dFalse` in any $T_1(t')$ where $t' > t$. Likewise, σ_2 satisfies its similar obligation for observable propositions. Furthermore, σ_2 also satisfies the obligation regarding contingent time points as the encoding generates a `failC` transition for each contingent time point C (belonging to a $(A, x, y, C) \in \mathcal{L}$) allowing `Player1` to move to `win` if `Player2` does not take `ExC` within its maximum upper bound y. Since `Player2` wants to prevent `Player1` from getting to `win`, σ_2 is obliged to schedule C such that $C - A \in [x, y]$.

Both σ_1 and σ_2 satisfy their postconditions: the reset of `cX` clocks says when the time points were executed, whereas the values of `bP` clocks say what truth values the propositions have been assigned. Finally, winning conditions are modeled differently with respect to the player. For `Player1` they are abstracted as a winning path checking that all time points and constraints whose labels are not falsified by ℓ_{cps} have been executed and satisfied, respectively. For `Player2` winning conditions correspond to schedule a contingent time point at a particular time or decide a truth value for an observable propositions (or any combination of these moves) such that `Player1` is unable to satisfy at least one constraint and ends up blocked somewhere while going through the winning path before entering `win`.

Theorem 5.2. *Any CSTNUD can be encoded into a TGA in polynomial time (with respect to the size of the network).*

Proof. Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{OT}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ be any CSTNUD, let $\mathit{Labels} = \{L(X) \mid X \in \mathcal{T}\} \cup \{\ell \mid (Y - X \leq k, \ell) \in \mathcal{C}\}$ be the set of different labels in the CSTNUD and let $\mathit{length}\colon \mathcal{P}^* \to \mathbb{N}$ be the mapping returning the length of a label (i.e., the number of literals), where

$$\mathit{length}(\ell) = \begin{cases} 0 & \text{if } \ell = \boxdot \\ n & \text{if } \ell = \lambda_1 \wedge \cdots \wedge \lambda_n) \end{cases}$$

The encoding generates $3 + |\mathit{Labels}| - 1$ locations: $\mathtt{T}_1$, $\mathtt{T}_2$, `win` and $(|\mathit{Labels}| - 1)$ $\mathtt{w}_{\ell_i}$. Thus

$$\mathit{Locations}(\mathcal{S}) = \mathcal{O}(|\mathit{Labels}|)$$

The encoding also generates a polynomial number of transitions: 2 transitions for the game interplay (`gain` and `pass`), 2 transitions for each observation time point $P?$ (`ExP?` and `pFalse`), 2 transitions for each decision time point $D!$ (`ExD` and `dFalse`), 2 transitions for each contingent time point C (`ExC` and `failC`), 1 transition for each remaining non-contingent time point X (`ExX`) and w transitions going from $\mathtt{T}_2$ to `win` (winning path). Since the number of different labels is fixed in the CSTNUD in input, I am left to prove that each set of transitions connecting $\mathtt{w}_{\ell_{i-1}}$ to $\mathtt{w}_{\ell_i}$ is polynomial in the label $\ell_i = \lambda_1 \wedge \ldots \lambda_n$. The encoding generates a set of $1 + 2 \times \mathit{length}(\ell_i)$ transitions: 1 `sat` verifying that all time points labeled

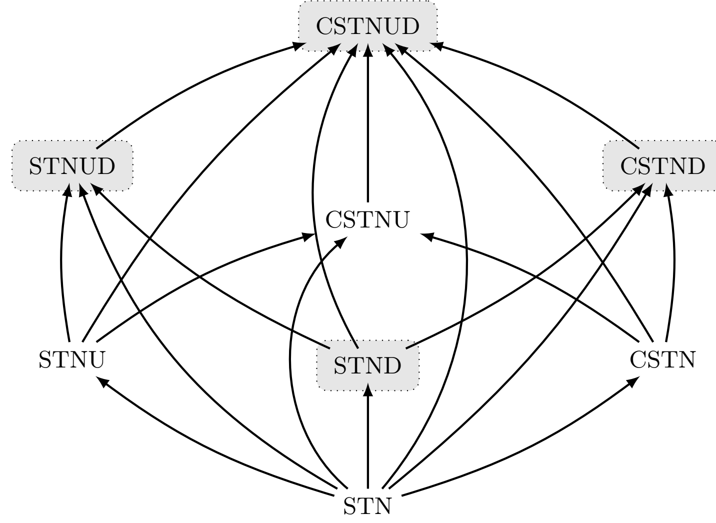


Fig. 5.4: A hierarchy of simple temporal networks. An arrow $A \to B$ means that formalism B embeds formalism A. I highlight new (sub)formalisms as grayed boxes.

by ℓ_i have been executed and all constraints labeled by ℓ_i are satisfied, and $2 \times \mathit{length}(\ell_i)$ `skip` transitions (for each $\lambda \in \ell_i$ there is a `skip` transition testing that the related observation or decision time point has not been executed and another `skip` transition testing if the literal does not hold (i.e., if $\neg\lambda_i$ holds). Therefore, $w = 1 + \sum_{\ell \in Labels \setminus \{\boxdot\}} (1 + 2 \times \mathit{length}(\ell))$, and consequently the total number of transitions is

$$\mathit{Transitions}(\mathcal{S}) = 2 + |\mathcal{P}| + |\mathcal{T}| + |\mathit{Contingent}| + w = \mathcal{O}(|\mathcal{P}| + |\mathcal{T}| + |\mathit{Labels}|)$$

Furthermore, the encoders to enforce time point label honesty and the partial order of time points run in polynomial time too. Indeed, for any non-contingent time point X, $L_{enc}(X)$ scans all literals in $L(X)$ once, and $\Pi_{enc}(X)$ scans all constraints in $\mathcal{C}$ once. Therefore, even though a CSTNUD can express an exponential number of constraints (worst case $\mathcal{O}(2^n)$ labeled constraints where $n = |\mathcal{P}|$), the function $\mathit{Locations}(\mathcal{S}) + \mathit{Transitions}(\mathcal{S})$ does not employ any component of the CSTNUD as an exponent. A more precise statement is that this encoding is *linear* in the size of the CSTNUD given in input.

5.5 Expressiveness of CSTNUDs

In this section, I discuss the expressiveness of CSTNUDs by providing a hierarchy of simple temporal networks (three of which are new) and showing that all other subformalisms can be embedded into CSTNUDs. Figure 5.4 shows the hierarchy, where an edge $A \to B$ means that formalism B embeds formalism A (e.g., STN $\to$ CSTN means that CSTNs can embed STNs).

Defining CSTNUDs as an extension of CSTNUs implicitly results in also defining three new kinds of networks:

1. *Simple temporal networks with decisions (STNDs)*, where the bottom formalism STN is augmented with decisions (Chapter 4).
2. *Simple temporal networks with uncertainty and decisions (STNUDs)*, where STNUs are augmented with decisions.
3. *Conditional simple temporal networks with decisions (CSTNDs)*, where CSTNs are augmented with decisions.

I now discuss the straightforward encodings from STNs, STNUs, CSTNs, STNDs, STNUDs, CSTNDs and CSTNUs into CSTNUDs.

5.5.1 Encoding STNs into CSTNUDs

I encode an STN $\langle \mathcal{T}, \mathcal{C} \rangle$ (Section 2.1.1) into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{OT}' = \mathcal{DT}' = \mathcal{P}' = \mathcal{L}' = \emptyset$.
3. O is undefined.
4. $L'(X) = \boxdot$ for each $X \in \mathcal{T}$.
5. $\mathcal{C}' = \{(Y - X \leq k, \boxdot) \mid (Y - X \leq k) \in \mathcal{C}\}$.

5.5.2 Encoding STNUs into CSTNUDs

I encode an STNU $\langle \mathcal{T}, \mathcal{L}, \mathcal{C} \rangle$ (Section 2.1.2) into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{OT}' = \mathcal{DT}' = \mathcal{P}' = \emptyset$.
3. $\mathcal{L}' = \mathcal{L}$.
4. O is undefined.
5. $L'(X) = \boxdot$ for each $X \in \mathcal{T}$.
6. $\mathcal{C}' = \{(Y - X \leq k, \boxdot) \mid (Y - X \leq k) \in \mathcal{C}\}$.

5.5.3 Encoding CSTNs into CSTNUDs

I encode a CSTN $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ (Section 2.1.3) into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{OT}' = \mathcal{OT}$.
3. $\mathcal{P}' = \mathcal{P}$.
4. $\mathcal{DT}' = \mathcal{L}' = \emptyset$.
5. $O' = O$.
6. $L'(X) = L(X)$ for each $X \in \mathcal{T}$.
7. $\mathcal{C}' = \mathcal{C}$.

5.5.4 Encoding STNDs into CSTNUDs

I encode an STND $\langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ (Chapter 4) into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{DT}' = \mathcal{DT}$.
3. $\mathcal{P}' = \mathcal{P}$.
4. $\mathcal{OT}' = \mathcal{L}' = \emptyset$.
5. $O' = O$.
6. $L'(X) = L(X)$ for each $X \in \mathcal{T}$.
7. $\mathcal{C}' = \mathcal{C}$.

5.5.5 Encoding STNUDs into CSTNUDs

A *Simple Temporal Network with Uncertainty and Decisions* (*STNUD*, [126]) is a tuple $\langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$, where $\mathcal{T}$ is a set of time points, $\mathcal{DT} \subseteq \mathcal{T}$ is a set of decision time points, $\mathcal{P}$ is a set of decidable propositions, $O \colon \mathcal{P} \to \mathcal{DT}$ is a bijection assigning a unique decision time point to each proposition, $\mathcal{L}$ is a set of contingent links and $\mathcal{C}$ is a set of *labeled* constraints. I encode an STNUD $\langle \mathcal{T}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{DT}' = \mathcal{DT}$.
3. $\mathcal{P}' = \mathcal{P}$.
4. $\mathcal{OT}' = \emptyset$.
5. $\mathcal{L}' = \mathcal{L}$.
6. $O' = O$.
7. $L'(X) = L(X)$ for each $X \in \mathcal{T}$.
8. $\mathcal{C}' = \mathcal{C}$.

5.5.6 Encoding CSTNDs into CSTNUDs

A *Conditional Simple Temporal Network with Decisions* (*CSTND*, [19, 126]) is a tuple $\langle \mathcal{T}, \mathcal{OT}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$, where $\mathcal{T}$ is a set of time points, $\mathcal{OT} \subseteq \mathcal{T}$ is a set of observation time points, $\mathcal{DT} \subseteq \mathcal{T}$ is a set of decision time points, $\mathcal{P}$ is a set of decidable and observable propositions, $O \colon \mathcal{P} \to \mathcal{DT} \cup \mathcal{OT}$ is a bijection assigning a unique decision or observation time point to each proposition and $\mathcal{C}$ is a set of *labeled* constraints. I encode a CSTND $\langle \mathcal{T}, \mathcal{OT}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{C} \rangle$ into a CSTNUD $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{OT}' = \mathcal{OT}$.
3. $\mathcal{DT}' = \mathcal{DT}$.
4. $\mathcal{P}' = \mathcal{P}$.
5. $O' = O$.
6. $L'(X) = L(X)$ for each $X \in \mathcal{T}$.
7. $\mathcal{L}' = \emptyset$.
8. $\mathcal{C}' = \mathcal{C}$.

5.5.7 Encoding CSTNUs into CSTNUDs

I encode a CSTNU $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ (Section 2.1.4) into a CSTNUDs $\langle \mathcal{T}', \mathcal{OT}', \mathcal{DT}', \mathcal{P}', O', L', \mathcal{L}', \mathcal{C}' \rangle$ as follows:

1. $\mathcal{T}' = \mathcal{T}$.
2. $\mathcal{OT}' = \mathcal{OT}$.
3. $\mathcal{DT}' = \emptyset$.
4. $\mathcal{P}' = \mathcal{P}$.
5. $O' = O$.
6. $L'(X) = L(X)$ for each $X \in \mathcal{T}$.
7. $\mathcal{L}' = \mathcal{L}$.
8. $\mathcal{C}' = \mathcal{C}$.

Similar encodings apply between all other formalisms $A \to B$ in Figure 5.4 (e.g., [74] shows how to encode STNs, CSTNs and STNUs into CSTNUs).

5.6 Esse: A Tool for CSTNUDs

I developed Esse, a tool for CSTNUDs that takes in input the specification of a CSTNUD and acts both as a solver for dynamic controllability and as an executor simulator. Esse is available at `http://regis.di.univr.it/Esse.tar.bz2` along with the case studies discussed in this thesis and the set of benchmarks used in [126] containing 1000 randomly generated CSTNUDs. Esse is a software that completely supersedes the first prototype provided in [126], although, for comparison purposes, I also compiled a different version of Esse implementing the old Uppaal-TIGA encoding.[3]

Esse relies on Uppaal-TIGA for the model checking phase, but it differs from the previous prototype for two main reasons. First, it makes the interaction with Uppaal-TIGA transparent. I no longer need to run Uppaal-TIGA manually as Esse handles it internally. Second, the new Uppaal-TIGA encoding is optimized by exploiting Boolean variables (instead of clocks) to model propositions and organizing all clocks in array data structures. These two modifications result in the model checking phase performing better and Esse becoming usable also for designers with little or no knowledge on TGAs. Listing 5.1 shows Esse's help screen.

Listing 5.1: Esse's help screen.

```
Usage: java -jar esse.jar <Network.cstnud> <Action> dynamic <Network.s> [N]
     [--silent]

<Action>:
  --check    internally encodes the CTSNUD in input into an UPPAAL-TIGA
             specification ready to check dynamic controllability (saves
             the strategy to Network.s)
```

[3] Due to a few coding issues, it was not possible to use directly the old prototype in the environment that I set up for the experimental evaluation. In order to make the comparison I compiled a different version of Esse supporting the old approach.

```
  --execute  performs [N] (default 1) executions of the CSTNUD in input
             (if controllable) according to the strategy (.s) synthesized
             by UPPAAL-TIGA.

  --silent   suppresses output (optional)

Examples:
  java -jar esse.jar Network.cstnud --check dynamic Network.s
  java -jar esse.jar Network.cstnud --execute dynamic Network.s 10
```

The input language of ESSE extends that of KAPPA by also allowing for the specification of observation time points (`?` identifier instead of `!` in the `TimePoint` section) and contingent links. The section `TimePoints`

```
TimePoints {
  ...
  (D! : d : p !q ...)
  (P? : p : !q ...)
  (X  :   : p !q ...)
  ...
}
```

specifies the sets $\mathcal{T}$, $\mathcal{OT}$ and $\mathcal{DT}$ as well as the mappings O and L, and provides here examples of the specification of: a decision time point $D!$ with $d = O(D!)$ and $L(D!) = p\neg q\ldots$, an observation time point $P?$ with $p = O(P?)$ and $L(D!) = \neg q\ldots$ (note the substitution of `?` for `!`), and a general time point $X \notin \mathcal{DT} \cup \mathcal{OT}$ with $L(X) = p\neg q$. The section `ContingentLinks`

```
ContingentLinks {
  ...
  (A1,10,20,C1)
  ...
}
```

specifies the set $\mathcal{L}$ and provides here an example of the specification of a contingent link $(A_1, 10, 20, C_1) \in \mathcal{L}$. All other sections remain the same.

Listing 5.2 shows the specification of Figure 5.2b into ESSE's input language. I omit the specifications of Figure 5.2a and Figure 5.2c as they are very similar (the only things that change are the suffixes `!` and `?` for `P` and `D` depending on the case).

Listing 5.2: Specification of Figure 5.2b.

```
1 Propositions {
2   d p
3 }
4
5 TimePoints {
6   (P? : p : )
7   (A1 : p)
8   (C1 : p)
9   (A2 : !p)
```

```
10   (C2 : !p)
11   (D! : d : )
12   (A3 : d)
13   (C3 : d)
14   (A4 : !d)
15   (C4 : !d)
16   (E : )
17 }
18
19 ContingentLinks {
20   (A1,1,6,C1)
21   (A2,8,12,C2)
22   (A3,3,5,C3)
23   (A4,6,10,C4)
24 }
25
26 Constraints {
27   (A1 - P <= 1 : p)
28   (P - A1 <= -1 : p)
29   (A2 - P <= 1 : !p)
30   (P - A2 <= -1 : !p)
31   (D - C1 <= 1 : p)
32   (C1 - D <= -1 : p)
33   (D - C2 <= 1 : !p)
34   (C2 - D <= -1 : !p)
35   (A3 - D <= 1 : d)
36   (D - A3 <= -1 : d)
37   (A4 - D <= 1 : !d)
38   (D - A4 <= -1 : !d)
39   (E - C3 <= 1 : d)
40   (C3 - E <= -1 : d)
41   (E - C4 <= 1 : !d)
42   (C4 - E <= -1 : !d)
43   (E - P <= 20 : d)
44   (P - E <= -21 : !d)
45 }
```

Given a specification of a CSTNUD, we can check the CSTNUD's dynamic controllability and then carry out `N` execution simulations (if DC) by running

```
$ java -jar esse.jar Network.cstnud --check dynamic Network.s
$ java -jar esse.jar Network.cstnud --execute dynamic Network.s N
```

I ran Esse on the specifications of Figure 5.2a, Figure 5.2b and Figure 5.2c to check whether or not they were dynamically controllable. I used a Linux virtual machine run on top of a VMWare ESXi hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM for the experimental evaluation. The VM was assigned full CPU power and 16GB of RAM (which is plenty, since Uppaal-TIGA is provided for 32bit architectures). For Figure 5.2a the analysis took 3.85 seconds answering `Uncontrollable`, for Figure 5.2b the analysis took 3.23 seconds answering `Controllable` and saving a 119-action strategy for

`Player1` of 268KB, and for Figure 5.2c the analysis took 3.05 seconds answering `Controllable` and saving a 92-action strategy for `Player1` of 196KB, as shown in Listing 5.3.

Listing 5.3: Example of DC-checking for Figure 5.2a, Figure 5.2b and Figure 5.2c.

```
$ java -jar esse.jar A.cstnud --check dynamic A.s
Checking A.cstnud with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Uncontrollable (strategy for Player2 exists)

$ java -jar esse.jar B.cstnud --check dynamic B.s
Checking B.cstnud with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Saving a 119-action strategy to B.s

$ java -jar esse.jar C.cstnud --check dynamic C.s
Checking C.cstnud with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Saving a 92-action strategy to C.s
```

The same analyses would have taken 2 minutes for Figure 5.2a and about 2 minutes 21 seconds for both Figure 5.2b and Figure 5.2c using the old UPPAAL-TIGA encoding.[4] Therefore, ESSE carried out the analysis 31.16 times faster on Figure 5.2a, 43.65 times faster on Figure 5.2b and 46.53 times faster on Figure 5.2c. Finally, I executed the latter two controllable cases. The simulator correctly scheduled all non-contingent time points satisfying all constraints. Listing 5.4 shows three random executions for Figure 5.2b:

Listing 5.4: Three random executions for Figure 5.2b

```
$ java -jar esse.jar B.cstnud --execute dynamic B.s 3
P = 0.1
p = true
A1 = 1.1
C1 = 4.0
D = 5.0
d = true
A3 = 6.0
C3 = 9.1
E = 10.1
Verifying ... SAT!

P = 0.1
p = false
A2 = 1.1
C2 = 9.3
```

[4] Moreover, I noted that the performance of the model checking phase also depends on the order of the statements (clocks in particular) in the UPPAAL-TIGA specification. Two specifications defining the same clocks in different orders might perform differently. That is why the times for Figure 5.2a, Figure 5.2b and Figure 5.2c in this thesis differ from those given in [126] when using ESSE compiled for the old encoding.

```
D = 10.3
d = true
A3 = 11.3
C3 = 14.7
E = 15.7
Verifying ... SAT!

P = 0.1
p = false
A2 = 1.1
C2 = 13.1
D = 14.1
d = false
A4 = 15.1
C4 = 23.2
E = 24.2
Verifying ... SAT!
```

In the first execution `Player1` observes p and $C_1 = 4$, and therefore decides d. In the second execution `Player1` observes $\neg p$ and $C_2 = 9.3$, and therefore decides d. In the third execution `Player1` observes $\neg p$ and $C_2 = 13.1$, and therefore decides $\neg d$.

The development of Esse allowed me to carry out again the experimental evaluation discussed in [126] and verify performance improvements. When I generated that set of benchmarks for [126] I did not enforce any particular distribution of CSTNUDs for which the analysis outputs consistent, inconsistent or timeout nor did I fix any particular number for any component (e.g., 100 time points as I discussed in Section 4.5). Each CSTNUD in that set has 5 to 20 time points, max $\lfloor \frac{|\mathcal{T}|}{5} \rfloor$ contingent links, max $\lfloor \frac{|\mathcal{T}|}{5} \rfloor$ observation time points, max $\lfloor \frac{|\mathcal{T}|}{5} \rfloor$ decision time points, a number of constraints obtained as the floor of 5 to 10% of $|\mathcal{T}|^2 - 2 \times |\mathcal{L}|$. The CSTNUDs were generated by trying to maximize these numbers which looked a good setting to avoid both under-constrained and over-constrained networks (resulting in a concrete chance of getting both controllable and uncontrollable CSTNUDs). Algorithm 9 shows the pseudo code of the generator, which was embedded in the first prototype in [126] and used to generate the set of benchmarks.

I ran the analysis on this set, always following the same order, by imposing a time out of 900 seconds on each instance. I show the graphical data in Figure 5.5, where x-axes represent the number (#) of analyzed instances, whereas y-axes represent either the overall time elapsed or space consumed. The analysis proved that 326 networks were DC and 27 non-DC. The remaining networks reached the timeout limit. I executed each CSTNUD proved dynamically controllable 1000 times by randomly assigning truth values to observable propositions and durations to contingent links. No execution crashed.

Figure 5.5a shows the DC-checking on the whole set of benchmarks comparing the performances of Esse with those of the old experimental evaluation in [126]. Figure 5.5b shows the space consumed by the strategies related to those CSTNUDs proved dynamically controllable. The strategies synthesized by Esse take more disk space than those synthesized by Uppaal-TIGA (average space of Esse's is 912.63KB, average space of Uppaal-TIGA's is 351.83KB). This is be-

Algorithm 9: `CSTNUD-Gen`$(nTPs, maxObs, maxDec, maxCL, maxConstr)$

Input: The number of time points ($nTPs$), the maximum number of observation time points ($maxObs$), the maximum number of decision time points ($maxDec$), the maximum number of contingent links ($maxCL$), and the maximum number of constraints ($maxConstr$).

Output: A well-defined CSTNUD with at least 1 observation, 1 decision, 1 contingent link and such that each proposition labels some component. That is, a *real* CSTNUD.

1. Check that $(maxObs + maxDec + (2 * maxCL)) \leq nTPs$ ▷ it must be possible
2. $\mathcal{Z} \leftarrow \langle \mathcal{T}, \mathcal{OT}, \mathcal{DT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ ▷ Empty CSTNUD

   ▷ Final cardinality of sets $\mathcal{OT}$, $\mathcal{DT}$ and $\mathcal{L}$
3. $nObs \leftarrow Random(1, maxObs)$, $nDec \leftarrow Random(1, maxDec)$, $nCL \leftarrow Random(1, maxCL)$

   ▷ Generate observation time points and related propositions
4. $\mathcal{OT} \leftarrow \{P_i? \mid 1 \leq i \leq nObs\}$, $\mathcal{P} \leftarrow \{p_i? \mid 1 \leq i \leq nObs\}$, $O(p_i) = P_i?$ for $1 \leq i \leq nObs$, $\mathcal{T} \leftarrow \mathcal{OT}$

   ▷ Generate decision time points and related propositions
5. $\mathcal{DT} \leftarrow \{D_i! \mid 1 \leq i \leq nDec\}$, $\mathcal{P} \leftarrow \mathcal{P} \cup \{d_i? \mid 1 \leq i \leq nDec\}$, $O(d_i) = D_i!$ for $1 \leq i \leq nDec$, $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{DT}$

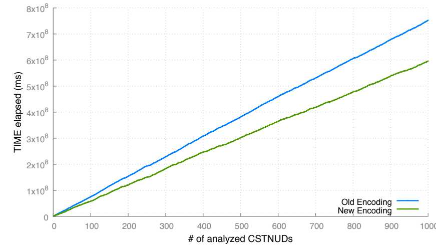
   ▷ Generate contingent links where each $x_i = Random(1, 10)$ and $y_i = Random(11, 20)$
6. $\mathcal{L} \leftarrow \{(A_i, x_i, y_i, C_i) \mid 1 \leq i \leq nCL\}$, $\mathcal{T} \leftarrow \mathcal{T} \cup \{A, C \mid (A, x, y, C) \in \mathcal{L}\}$

   ▷ Generate the rest of time points
7. $\mathcal{T} \leftarrow \mathcal{T} \cup \{X_i \mid 1 \leq i \leq (nTPs - nObs - nDec - (2 * nCL))\}$
8. $p \leftarrow Random$ element in $\mathcal{P}$
9. $P = O(p)$, $L(P) = \boxdot$ ▷ One decision/observation must be unlabeled

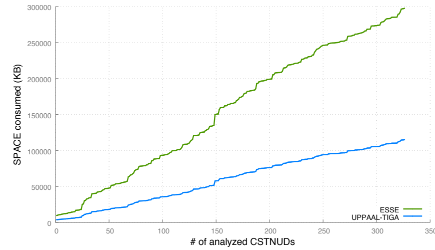
   ▷ Label time points as follows
10. $Unlabeled \leftarrow [\mathcal{OT}, \mathcal{DT}, \{\mathcal{T} \setminus \mathcal{OT} \setminus \mathcal{DT} \setminus \{C \mid (A, x, y, C) \in \mathcal{L}\}]$ ▷ List
11. Remove P from $Unlabeled$ ▷ already assigned $\boxdot$
12. $maxLength \leftarrow Random(0, |\mathcal{P}|)$ ▷ Max length of a label
13. **while** $Unlabeled \neq \emptyset$ **do**
14. $X \leftarrow$ pop first element from $Unlabeled$
15. $tmp \leftarrow$ Random sample of $\mathcal{P}$ of size $maxLength$
16. $L(X) = \boxdot$
17.   **while** $tmp \neq \emptyset$ **do**
18. $p \leftarrow$ pop a proposition from tmp and decide a sign for it
19. $P \leftarrow O(p)$
20.   **if** *L(P) has already been specified and* $P \neq X$ *and* $L(X) \wedge L(P) \wedge (\neg)p$ *is consistent* **then**
21. $L(X) \leftarrow L(X) \wedge L(P) \wedge (\neg)p$ **if** X *is an activation time point* **then**
22. $L(C) = L(X)$ where C is the associated contingent

    ▷ Generate constraints
23. **for** $i \leftarrow 1$ *to* $maxConstr$ **do**
24. $X, Y \leftarrow$ two Random time points in $\mathcal{T}$
25.   Check that $X \neq Y$ and X, Y do not specify a contingent link
26. $\ell \leftarrow L(X) \wedge L(Y)$
27.   Randomly decide to extend ℓ ▷ Probability of this choice is $\frac{1}{2}$
28.   **if** *so* **then**
29.   Get a sample of propositions from $\mathcal{P}$ not appearing in ℓ and try to extend ℓ the same way as we did with time points
30. $k \leftarrow Random(-100, 100)$
31.   Add $(Y - X \leq k, \ell)$ to $\mathcal{C}$

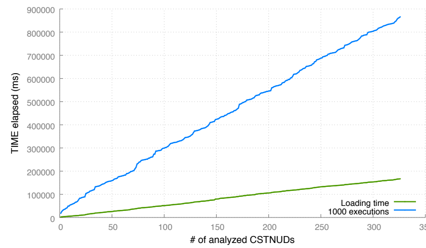
    ▷ Final check
32. **if** *Some proposition never appears in any label* **then**
33.   Throw away the network
34. **return** $\mathcal{Z}$



(a) Dynamic controllability checking on all CSTNUDs with the old encoding (blue, above) and the new one (green, below).



(b) Strategy space consumed by Uppaal-TIGA (blue, below) and Esse (green, above) for dynamically controllable CSTNUDs.



(c) Strategy loading time (green, below), then 1000 executions (blue, above).

Fig. 5.5: Time and space analysis with Esse on 1000 CSTNUDs.

cause Uppaal-TIGA returns a strategy in text format that is not directly usable, whereas Esse translates it into a dedicated data structure of objects ready for the execution. Figure 5.5c shows how long it takes to load the strategies in memory and then carry out 1000 execution simulations.

I also carried out a second experimental evaluation in which I analyzed time performances and space consumption with respect to the number of contingent links, observation time points and decision time points. I used a FreeBSD virtual machine run on top of a VMWare ESXi Hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM. The VM was assigned 16GB of RAM and full CPU power. I generated 1800 CSTNUDs partitioned in 9 sets of benchmarks each one containing 100 dynamically controllable CSTNUDs and 100 uncontrollable. These sets of benchmarks are available at `http://regis.di.univr.it/EE_CSTNUD2018.tar.bz2`.

The first (three) sets (`6tps/0c1o1d`, `6tps/1c1o1d` and `6tps/2c1o1d`) specify CSTNUDs with 1 observation time point, 1 decision time point and differ for the number of contingent links which range from 0 to 2 (note that `6tps/0c1o1d` actually specifies CSTNDs).

The second (four) sets (`6tps/1c0o1d`, `6tps/1c1o1d`, `6tps/1c2o1d` and `1c3o1d`) specify CSTNUDs with 1 contingent link, 1 decision time point and differ for the

number of observation time points which range from 0 to 3 (note that `6tps/1c1o1d` is the same of that contained in the first three sets and that `6tps/1c0o1d` actually specifies STNUDs).

The third (four) sets (`6tps/1c1o0d`, `6tps/1c1o1d`, `6tps/1c1o2d` and `6tps/1c1o3d`) specify CSTNUDs with 1 contingent link, 1 observation time point and differ for the number of decision time points which range from 0 to 3 (again, `6tps/1c1o1d` is the same of that contained in the first three sets and in the second four sets and that `6tps/1c1o0d` actually specifies CSTNUs).

Regardless of the set, each CSTNUD has exactly 6 time points and a maximum number of constraints of 35% of $|\mathcal{T}| \times (|\mathcal{DT}| + |\mathcal{OT}| + |\mathcal{L}|)$. Time points and constraints are labeled randomly. The minimum and maximum values for the ranges of contingent links are 1 and 5, whereas the weight values for labeled constraints range from -10 to 10. The generator is similar to the previously discussed one (just enforcing these new policies). I proceed to discuss the graphical data regarding time and space shown in Figure 5.6 where x-axes always represent the ranging number of a specific component (i.e., the set of benchmarks under analysis) and y-axes represent either the average time elapsed or space consumed when analyzing the instances in that set.

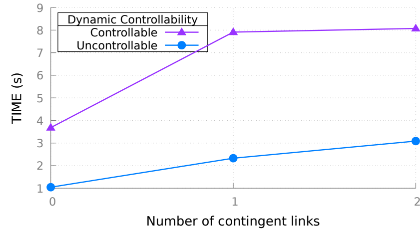
In Figure 5.6a I focused on the time performances with respect to the number of contingent links (first three sets). The results show that augmenting the number of contingent links results in augmenting the time of the analysis too. Note that uncontrollable networks suffer less from this problem.

In Figure 5.6b I focused on the time performances with respect to the number of observation time points (second four sets). The results show that augmenting the number of observation time points results in augmenting the time of the analysis too. This is mainly because the resulting TGA has a longer winning path. Both controllable and uncontrollable CSTNUDs suffer from this issue.

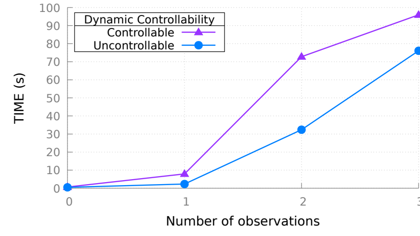
In Figure 5.6c I focused on the time performances with respect to the number of decision time points (third four sets). The results show that augmenting the number of decisions results in augmenting the time of the analysis too. Again, because we get a longer winning path. However, the general time of the analysis is worse than the corresponding analysis with respect to observation time points. This is because decisions are under control and therefore, during the validation phase, runs of the TGA detecting some constraint violation may be excluded (making the right decision) in order to continue the analysis and look for (another) strategy.

Figure 5.6d, Figure 5.6e and Figure 5.6f show the space consumption with respect to the number of contingent links, observation and decision time points, respectively. Despite a general expected growth in size when augmenting the specific uncontrollable components, the results confirm, as shown by the previous rough analysis in Figure 5.5b, that the strategies saved by ERRE consume more space than those saved by UPPAAL-TIGA.

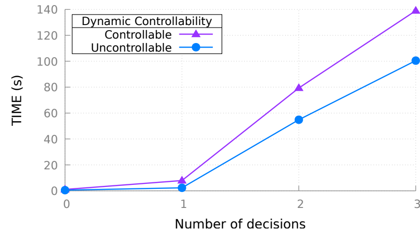
Finally, I executed all controllable CSTNUDs proved dynamically controllable 1000 times by randomly assigning truth values to observable propositions and durations to contingent links. Overall, ESSE simulated 900,000 of random executions. No one crashed.



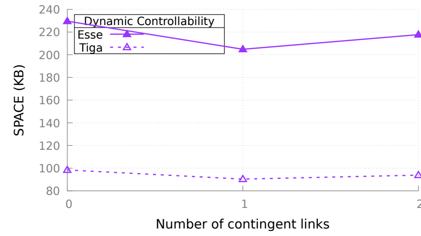
(a) *Dynamic controllability checking time on with respect to the number of contingent links.*



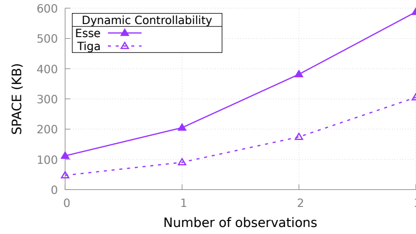
(b) *Dynamic controllability checking time with respect to the number of observation time points.*



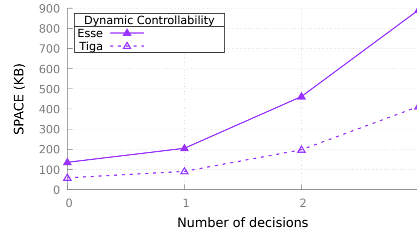
(c) *Dynamic controllability checking time with respect to the number of decision time points.*



(d) *Strategy space with respect to the number of contingent links.*



(e) *Strategy space with respect to the number of observation time points.*



(f) *Strategy space with respect to the number of decision time points.*

Fig. 5.6: Experimental evaluation with ESSE.

5.7 Modeling Temporal Workflows Under Uncertainty

In this section, I define a structured process modeling language (PML) to specify temporal workflows under conditional and temporal uncertainty (simply shortened as TWFs), then I define weak, strong and dynamic controllability of TWFs and finally provide an encoding from TWFs into CSTNUDs. I start with a motivational example that I use throughout the section.

5.7.1 Motivating Example

As a motivating example, I consider simplification of a goods delivery process that I show in Figure 5.7. The WF starts by processing orders collected online at least 1 hour ago (`ProcO`). This task lasts from 1 to 2 hours and its duration is under control. Then, after exactly 1 hour, the execution flow enters a conditional block and splits into two mutually exclusive branches depending on whether the order requires a one-day delivery or not (diamond labeled by `1dd?`). If `1dd?` $= \top$ (i.e., if `1dd?` is true), then the goods in the order must be delivered within 1 day and a fast collection process (`FastC`) starts after exactly 1 hour since the flow has split. `FastC` lasts from 1 to 3 hours and its exact duration is out of control. Instead, if `1dd?` $= \bot$ (i.e., if `1dd?` is false), then no express delivery is required and the flow of execution continues in the other branch with a normal collection process (`NormC`) starting after exactly 1 hour since the flow has split. `NormC` lasts from 10 to 20 hours and its exact duration is out of control. This conditional block lasts from 3 to 22 hours (dashed edge between the two leftmost diamonds) and concludes after exactly 1 hour since the task in the chosen branch completed.

Now that the shipment is ready, we must choose the correct delivery method in order to satisfy the temporal constraints. Therefore, after exactly 1 hour the flow of execution enters another conditional block but this time we can *decide* which delivery method to use (diamond labeled by `hurry!`). There are two different types of delivery, each one specifying an uncontrollable duration. If we decide to hurry up, we assign `hurry!` $= \top$ and go for a fast delivery (`FastD`) which takes 1 to 12 hours, whereas if we don't, we assign `hurry!` $= \bot$ and go for a normal delivery (`NormD`), which takes 24 to 48 hours. The conditional block lasts globally from 3 to 50 hours (dashed edge between the two rightmost diamonds) and completes after one hour since the task in the chosen branch completed. The whole process completes after minimum another hour. The WF lasts at most 1 day in case of one-day delivery (dashed edge above), and (more than) 1 to 3 days (dashed edge below) otherwise. The goal of this temporal plan is to always satisfy the temporal constraints no matter what, which means guaranteeing customer satisfaction.

5.7.2 Process Modeling Language

I consider a structured approach employing a fragment of BPMN decorated with temporal ranges.

I restrict the analysis on loop-free workflows and follow the ideas of the structured approach of the conceptual model TNest [34], where the specification of a workflow is given by a *workflow schema*, where nodes correspond to *activities* and arcs represent the control flow defining dependencies between the order of execution of such activities. There exist two different types of activity: *tasks* (rounded boxes) and *connectors* (diamonds). Tasks represent elementary work units that cannot be decomposed further. Connectors (or *gateways* in BPMN) represent internal activities executed by the workflow management system to achieve a correct and coordinated execution of tasks. Connectors are restricted to being of two types: *total* (+) and *conditional split* ($\times$). A connector is total when it splits the flow of the execution into $n > 1$ parallel branches or it joins $n > 1$ incoming parallel

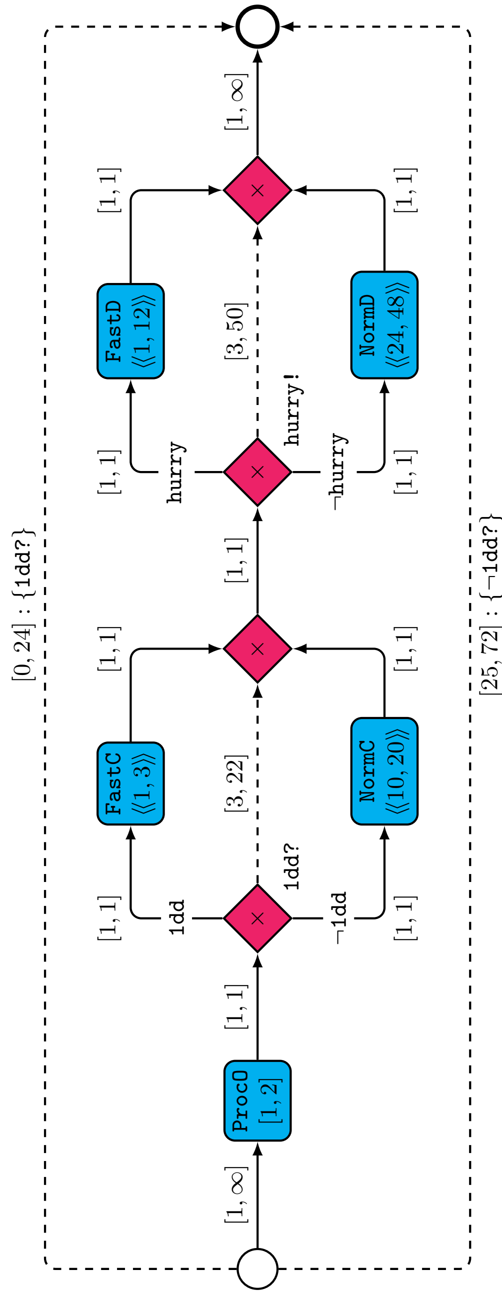


Fig. 5.7: Example of a temporal workflow in BPMN for a good delivery process.

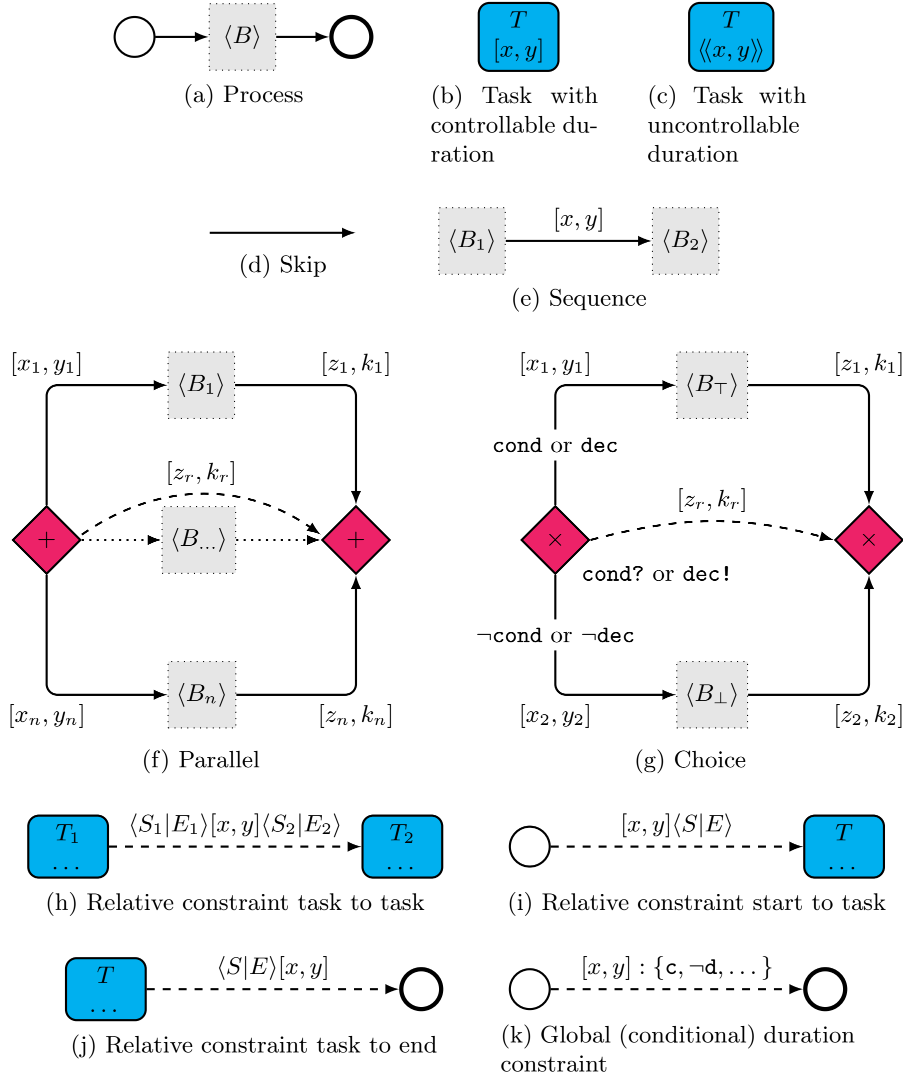


Fig. 5.8: A fragment of a structured (temporal) BPMN.

branches into a single outgoing flow. A connector is conditional when it splits a single flow of execution in exactly two mutually-exclusive branches or it joins two mutually-exclusive branches into a single one.

Differently from TNest, the encoding provided here has the following characteristics:

- BPMN components model the workflow
- Conditional blocks specifying decisions are supported
- Tasks with both controllable and uncontrollable durations are supported

- Conditional relative constraints between any combination of start or end of the process and tasks and relative constraints to restrict a global duration of a parallel or a conditional block are supported.

Figure 5.8 shows the process modeling language that I propose to model TWFs under conditional uncertainty.

Each workflow block can be thought of as a symbol in a context-free grammar. In particular, a *process* can be thought of as the starting symbols of such a grammar embedding the non-terminal symbol block $\langle B \rangle$ (Figure 5.8a). A terminal block can be a *task* whose duration is under control (Figure 5.8b) or out of control (Figure 5.8c) or a *skip* (Figure 5.8d). A non-terminal block can be a *sequence* (Figure 5.8e), a parallel (Figure 5.8f) or a conditional block where the associated condition can be under control (`dec!`) or out of control (`cond?`) (Figure 5.8g).

Several relative constraints (drawn as dashed edges) may be expressed to restrict the duration of a parallel (Figure 5.8f) or conditional block (Figure 5.8g), between the start and/or the end of two different tasks (Figure 5.8h), between the start of the process and the start or end of a task (Figure 5.8i), the vice versa but with respect to the end of the process (Figure 5.8j) or between the start and the end of the whole process (Figure 5.8k). In the latter case, I also (optionally) augment the label with a finite set of literals $\{\mathtt{c}, \neg\mathtt{d}, \dots\}$ requiring that the constraint holds only if the observations and decisions are according to the set of literals (e.g., $[0, 24] : \{\mathtt{1dd?}\}$ and $[25, 72] : \{\neg\mathtt{1dd?}\}$ labeling the relative constraints between the start and the end of the process in Figure 5.7). If a literal appears in this set, then all other literals according to the conditional nesting levels of the blocks (if any) should appear too. Also, this set should never contain pairs of inconsistent literals such as $\{\mathtt{d}, \neg\mathtt{d}\}$. These last two assumptions are, however, captured by the well-definedness properties of CSTNUDs [126], so enforcing them at design time is not mandatory. If the resulting CSTNUD is not well-defined, the controllability algorithm will find out.

Relative constraints do not breach the structuredness of the temporal workflow as they only impose further temporal constraints which have nothing to do with the information flow that is regulated by the grammar. The example in Figure 5.7 introduced in Section 5.7.1 is a structured TWF under conditional and temporal uncertainty.

5.7.3 Controllability of temporal workflows under uncertainty

In this section, I define *weak*, *strong* and *dynamic controllability* of TWFs specifiable with the language in Figure 5.8. My main goal is the synthesis of execution strategies saying when to schedule tasks and connectors, and what decisions to make so that in the arising WF path the execution satisfies all relevant temporal constraints. I recall that the uncontrollable parts I deal with are: (i) truth value assignments to `cond?` variables and (ii) uncontrollable task durations.

Definition 5.15. *Let* $\mathcal{B} = \{\mathtt{cond?}, \mathtt{dec!}, \dots\}$ *be the set of Boolean variables associated to all conditional split connectors. A* scenario $s \colon \mathcal{B} \to \{\bot, \top\}$ *is a complete truth value assignment to the Boolean variables in* $\mathcal{B}$*. A scenario has a* controllable part *consisting of the assignments to the variables* `dec!` *(decision scenario,*

ds) and an uncontrollable part *consisting of the assignments to the variables* `cond?` *(condition scenario, cs). Σ_D and Σ_C model the space of all decision and condition scenarios.*

A *workflow path (WF path)* is the projection of a workflow onto a scenario. That is, a new unconditional WF in which all components incoherent with the truth value assignment are removed. For example, if in an execution of Figure 5.7 we observe "one-day delivery" ($s(\texttt{1dd?}) = \top$) and then we decide not to hurry up ($s(\texttt{hurry!}) = \bot$), the WF path that arises is $\texttt{ProcO} \to \texttt{FastC} \to \texttt{NormD}$ as `NormC` and `FastD` are removed. We write $T_1 \to \cdots \to T_n$ instead of $\texttt{Start} \to \texttt{Split} \to T_1 \to \cdots \to \texttt{Join} \to \ldots T_n \to \texttt{End}$ to save space and ease reading.

Definition 5.16. *Let* `UTasks` *be the set of all tasks with uncontrollable durations. A* situation $sit\colon \texttt{UTasks} \to \mathbb{R}$ *is a complete real value assignment to all tasks having uncontrollable durations.*

For example, if in the previous WF path `FastC` lasts 2 hours and `NormC` 14 hours, then $sit(\texttt{FastC}) = 2$ and $sit(\texttt{NormC}) = 14$ (note that `ProcO` has a controllable duration so the domain of *sit* does not contain it).

Definition 5.17. *A* drama *is a pair $\delta = (cs, sit)$, where cs is a condition scenario and sit a situation. The set of all dramas is denoted by Δ.*

Dramas allow us to reason on all possible combinations of uncontrollable behaviors by fixing them one at a time.

Definition 5.18. *A* temporal reference *is* $r ::= \texttt{Start}(P) \mid \texttt{Start}(T) \mid \texttt{End}(T) \mid \texttt{Execute}(C) \mid \texttt{End}(P)$, *where P is the whole process, T a task and C a connector. The set of all temporal references is denoted by $\mathcal{R}$. If $T \in \texttt{UTasks}$, then $\texttt{End}(T)$ is uncontrollable, else controllable. All other temporal references are controllable.*

Definition 5.19. *A* schedule *is a mapping $\psi\colon \mathcal{R} \to \mathbb{R}$ from temporal references to real time instants saying when the events modeled by these references occur. A schedule is consistent if the assignments it makes satisfy all relevant temporal constraints involving tasks and connectors whose related events are in the domain of ψ. The set of all schedules is denoted by Ψ.*

Thus, a schedule containing $\texttt{Start}(P) = 0$ means "start the process at 0".

Definition 5.20. *An* execution strategy *is a mapping $\sigma\colon \Delta \to \Sigma_D \times \Psi$ from dramas to pairs (σ^d, σ^t), where σ^d is a decision strategy mapping dramas to decision scenarios and σ^t is a temporal strategy mapping dramas to schedules. An execution strategy is* viable *if for every drama δ there exists a decision scenario $\sigma^d(\delta)$ such that the corresponding schedule $\sigma^t(\delta)$ is consistent.*

I write $[\sigma^t(\delta)]_r$, instead of $\sigma^t(\delta)(r)$, to denote when the reference r is scheduled in $\psi = \sigma^t(\delta)$ and I write $[\sigma^t(\delta)]_{\texttt{dec!}}$, instead of $\sigma^d(\delta)(\texttt{dec!})$, to denote the truth value assignment to the controllable Boolean variable `dec!`.

Definition 5.21. *A TWF is* weakly controllable *if there exists a viable strategy.*

That is, once fixed a drama we can find a decision scenario and a schedule for which the resulting WF path is consistent. The TWF in Figure 5.7 is weakly controllable (I don't discuss the strategy).

Definition 5.22. *A TWF is* strongly controllable *if there exists a viable strategy working for all dramas.*

That is, a single decision scenario and a single schedule make consistent all WF paths. The TWF in Figure 5.7 is not strongly controllable mainly because the decision `hurry!` depends on both the `1dd?` condition and the `NormC` duration (see the discussion for dynamic controllability).

To define dynamic controllability I first define the history of a drama.

Definition 5.23. *The* history $\mathcal{H}(t, \delta, \sigma^t)$ *of the drama* $\delta = (cs, sit)$ *at the real time instant* $t > 0$ *for the temporal strategy* σ^t *is the pair* $(\mathcal{H}_{cs}(t, \delta, \sigma^t), \mathcal{H}_{sit}(t, \delta, \sigma^t))$*, where*

- $\mathcal{H}_{cs}(t, \delta, \sigma^t) = \{(\texttt{cond?}, cs(\texttt{cond?})) \mid [\sigma^t(\delta)]_{\texttt{Execute}(C)} < t\}$ *is the* condition scenario history *representing the set of uncontrollable truth value assignments observed before* t *(upon the execution of the corresponding conditional split connectors* C *having associated* `cond?` *variables) in the schedule* $\sigma^t(\delta)$*,*
- $\mathcal{H}_{sit}(t, \delta, \sigma^t) = \{(T, [\sigma^t(\delta)]_{\texttt{End}(T)} - [\sigma^t(\delta)]_{\texttt{Start}(T)}) \mid [\sigma^t(\delta)]_{\texttt{End}(T)} < t\}$ *is the* situation history *representing the set of uncontrollable task durations observed before* t *in the schedule* $\sigma^t(\delta)$*.*

Consider Figure 5.7 and assume that $\texttt{Start}(P) = 0$, $\texttt{Start}(\texttt{Proc0}) = 1$, $\texttt{End}(\texttt{Proc0}) = 2$ and $\texttt{Execute}(C) = 1$, where C is the conditional split connector having `1dd?` associated. Suppose that we observe the request for a one-day delivery. Then, $\mathcal{H}_{cs}(t, \delta, \sigma^t) = \emptyset$ at $t = 3$, whereas $\mathcal{H}_{cs}(t, \delta, \sigma^t) = \{\texttt{1dd?}\}$ at $t = 3 + \epsilon$. Similarly, suppose that $\texttt{Start}(\texttt{FastC}) = 4$ and $\texttt{End}(\texttt{FastC})$ is observed at 6. Then, $\mathcal{H}_{sit}(6, \delta, \sigma^t) = \emptyset$, whereas $\mathcal{H}_{sit}(6 + \epsilon, \delta, \sigma^t) = \{(\texttt{FastC}, 2)\}$.

Definition 5.24. *An execution strategy* $\sigma = (\sigma^t, \sigma^d)$ *is* dynamic *if* σ^t *and* σ^d *are dynamic, where:*

- *A decision strategy* σ^d *is dynamic if for any pair of dramas* $\delta_1, \delta_2 \in \Delta$ *and any time instant* $t > 0$*, whenever the drama histories look the same, then the strategy makes the same decisions. That is, for all* $\delta_1, \delta_2 \in \Delta$ *and any* `dec!`*, if* $\mathcal{H}_\delta(t, \delta_1, \sigma^t) = \mathcal{H}_\delta(t, \delta_2, \sigma^t)$*, then* $[\sigma^d(\delta_1)]_{\texttt{dec!}} = [\sigma^d(\delta_2]_{\texttt{dec!}}$*.*
- *A temporal strategy* σ^t *is dynamic if for any pair of dramas* $\delta_1, \delta_2 \in \Delta$ *and any time instant* $t > 0$*, whenever the drama histories look the same, then the strategy schedules the controllable temporal references at the same times. That is, for all* $\delta_1, \delta_2 \in \Delta$ *and any controllable* $r \in \mathcal{R}$*, if* $\mathcal{H}_\delta(t, \delta_1, \sigma^t) = \mathcal{H}_\delta(t, \delta_2, \sigma^t)$*, then* $[\sigma^t(\delta_1)]_r = [\sigma^t(\delta_2)]_r$*.*

Definition 5.25. *A TWF is* dynamically controllable *if there exists a dynamic execution strategy.*

The TWF in Figure 5.7 is dynamically controllable. Assume that `Proc0` always lasts 1 hour (otherwise the process is not controllable). If $\texttt{1dd?} = \top$, then regardless

of the duration of `FastC`, we will always go for a `FastD` as for any combination of durations the process will end within 1 day. Instead, suppose now that `1dd?` $= \bot$. We will go for `FastD` or `NormD` depending on the duration of `NormC` (I focus again on the combinations of minimal and maximal durations of `NormC`, `FastD` and `NormD`). If `NormC` takes its minimal duration, we cannot, in general, choose `FastD` because if `FastD` lasted 1 hour (its minimal duration), the process would end in 20 hours providing a one-day delivery service to the customers who did not pay for it. In this case, we will always choose `NormD` because if `NormD` lasts its minimal duration the process ends after 43 hours, whereas if `NormD` lasts its maximal duration, then the process ends after 67 hours. Either way, the process ends within 3 days but after 1 day. Instead, if `NormC` takes its maximal duration, we must always go for `FastD`. If we went for `NormD`, and `NormD` took its maximal duration, then we would end the process after 77 hours violating the guarantee that the goods will be delivered within 3 days. This combination is correct because if `FastD` lasts its minimal duration, then the process ends after 30 hours, whereas if `FastD` lasts its maximal, then the process ends after 41 hours. Either way, the process ends after 1 day and within 3 days.

Hence, the controllability of TWFs boils down to that of CSTNUDs when temporal references are modeled as time points, tasks with uncontrollable durations as contingent links, conditional split connectors as observation time points (`cond?` variables) or decision time points (`dec!` variables), tasks with controllable durations, delays, deadlines and relative constraints as requirement links.

5.7.4 Encoding TWFs into CSTNUDs

Table 5.1 provides the encodings from each workflow block discussed in Figure 5.8 into the corresponding CSTNUD fragment. I used a compact notation to specify the constraints in the CSTNUD fragments called *requirement link*. That is, $X \to Y$ labeled by $[x, y], \ell$ shortens the pair of constraints $X \to Y$ and $Y \to X$ labeled by $\langle y, \ell \rangle$ and $\langle -x, \ell \rangle$, respectively. The start and end of the process are encoded into two non-contingent time points S and E whose labels are both $\boxdot$ (row 1). All arcs regulating the control flow are encoded into requirement links labeled by the same intervals and extended with propositional labels according to the nesting level in which the WF block appears.

A task T is encoded into a pair of time points T_S and T_E modeling its start and end. The labels of these tasks are according to the nesting level of the WF block in which the task appears. If T specifies a controllable duration, then T is encoded into a requirement link, whereas if T specifies an uncontrollable duration, then T is encoded into a contingent link. Either way, the link is labeled by the same interval and propositional label (row 2). Again, the label on contingent links is the same of the activation and contingent time points (thus not shown).

A skip connector is encoded as a requirement link whose label ℓ is the conjunction of the labels of the connected time points in the resulting CSTNUD (row 3). Similarly, a sequence is encoded into a requirement link connecting the ending time point of the first block to the starting one of the second. The requirement link is again labeled by the same interval and the conjunction of the labels of the connected time points (row 4).

Table 5.1: Encoding TWFs into CSTNUDs. Ranges $[x, y]$ $(0 < x < y < \infty)$ and $\langle\langle x, y\rangle\rangle$ model controllable and uncontrollable durations $(0 \le x \le y \le \infty)$, resp.

WORKFLOW BLOCK	CSTNUD FRAGMENT
$[x_1, y_1]$ $\langle B\rangle$ $[x_2, y_2]$	S $[\boxdot]$ $[x_1, y_1], \boxdot$ $\dots$ $[x_2, y_2], \boxdot$ E $[\boxdot]$
T $[x, y]$ or T $\langle\langle x, y\rangle\rangle$	T_S $[\ell]$ $[x, y], \ell$ T_E $[\ell]$ or T_S $[\ell]$ $[x, y]$ T_E $[\ell]$
$[x, y]$	$[x, y], \ell$
$\langle B_1\rangle$ $[x, y]$ $\langle B_2\rangle$	$\dots$ $[x, y], \ell$ $\dots$
$[x_1, y_1]$ $\langle B_1\rangle$ $[z_1, k_1]$; $[z_r, k_r]$; + $\langle B_{\dots}\rangle$ +; $[x_n, y_n]$ $\langle B_n\rangle$ $[z_n, k_n]$	$[x_2, y_1], \ell$ $\dots$ $[\ell \wedge \dots]$ $[z_1, k_1], \ell$; $[z_r, k_r], \ell$; P_S $[\ell]$ $\dots$ $[\ell \wedge \dots]$ P_E $[\ell]$; $[x_n, y_n], \ell$ $\dots$ $[\ell \wedge \dots]$ $[z_n, k_n], \ell$
$[x_1, y_1]$ $\langle B_\top\rangle$ $[z_1, k_1]$; cond or dec; $[z_r, k_r]$; × cond? or dec! ×; ¬cond or ¬dec; $[x_2, y_2]$ $\langle B_\bot\rangle$ $[z_2, k_2]$	$[x_2, y_1], \ell \wedge p$ $\dots$ $[\ell \wedge p \wedge \dots]$ $[z_1, k_1], \ell \wedge p$; $[z_r, k_r], \ell$; $P?$ or $P!$ $[\ell]$ P_E $[\ell]$; $[x_2, y_2], \ell \wedge \neg p$ $\dots$ $[\ell \wedge \neg p \wedge \dots]$ $[z_2, k_2], \ell \wedge \neg p$
$[x, y]\langle S \vert E\rangle$ T $\dots$ or T $\dots$ $\langle S \vert E\rangle[x, y]$	S $[\boxdot]$ $[x, y], \ell$ $T_S \vert T_E$ $[\ell]$ or $T_S \vert T_E$ $[\ell]$ $[x, y], \ell$ E $[\boxdot]$
T_1 $\dots$ $\langle S_1 \vert E_1\rangle[x, y]\langle S_2 \vert E_2\rangle$ T_2 $\dots$	$T^1_S \vert T^1_E$ $[\ell_1]$ $[x, y], \ell_1\ell_2$ $T^2_S \vert T^2_E$ $[\ell_2]$
$[x, y] : \{\mathtt{c}, \neg\mathtt{d}, \dots\}$	S $[\boxdot]$ $[x, y], c\neg d \dots$ E $[\boxdot]$

Parallel blocks are encoded by substituting requirement links for arcs regulating the control flow. Total split and join connectors are modeled as two time points P_S ("parallel start") and P_E ("parallel end") whose labels are the same according to the nesting level. If a relative temporal constraint restricting the global duration of the entire block is specified (dashed edge labeled by $[z_r, k_r]$), then I encode it as a requirement link in the CSTNUD (row 5). Conditional blocks are encoded the same way but with two modifications. First, if the conditional split connector is

associated to a `cond?` variable, then an observation time point models it, whereas if it is associated to a `dec!` variable, then a decision time point models it. Second, the propositional labels of all time points and links modeling the sub-blocks are augmented with a positive literal p ($\top$ branch) and a negative one $\neg p$ ($\bot$ branch) (row 6).

Relative constraints (dashed edges) are all encoded as requirement links connecting the time points corresponding to the "references" of the connected components and are labeled by the same intervals and conjunction of the labels of the corresponding time points in the CSTNUD (rows 7, 8 and 9).

Figure 5.9 shows the CSTNUD encoding the TWF in Figure 5.7. There, the observation time point $O?$ models the conditional split connector and o models the associated `1dd?` (O_E models the corresponding join connector), whereas the decision time point $H!$ models the conditional split connector and h models the associated `dec!` (H_E models the corresponding join connector). The requirement link $\texttt{ProcOS} \to \texttt{ProcOE}$ labeled by $[1, 2], \boxdot$ models `ProcO`, whereas the contingent links $(\texttt{FastCS}, 1, 3, \texttt{FastCE})$, $(\texttt{NormCS}, 10, 20, \texttt{NormCE})$, $(\texttt{FastDS}, 1, 12, \texttt{FastDE})$ and $(\texttt{NormDS}, 24, 48, \texttt{NormDE})$ model `FastC`, `NormC`, `FastD` and `NormD`. Therefore, the truth value assignment to o as well as the real value assignments to the contingent time points `FastCE`, `NormCE`, `FastDE` and `NormDE` (modeling the end of the corresponding tasks) are out of control, whereas any other component is under control.

I have discussed a process modeling language for workflows under uncertainty. Then, I have provided an encoding from TWFs into CSTNUDs. Now, I check dynamic controllability of the motivating example discussed in this section by feeding ESSE with the CSTNUD in Figure 5.9 whose specification is given in Listing 5.5.

Listing 5.5: Specification of Figure 5.9.

```
1  Propositions {
2          o h
3  }
4
5  TimePoints {
6          (S : )
7          (ProcOS : )
8          (ProcOE : )
9          (O? : o : )
10         (FastCS : o)
11         (FastCE : o)
12         (NormCS : !o)
13         (NormCE : !o)
14         (OE : )
15         (H! : h : )
16         (FastDS : h)
17         (FastDE : h)
18         (NormDS : !h)
19         (NormDE : !h)
20         (HE : )
21         (E : )
```

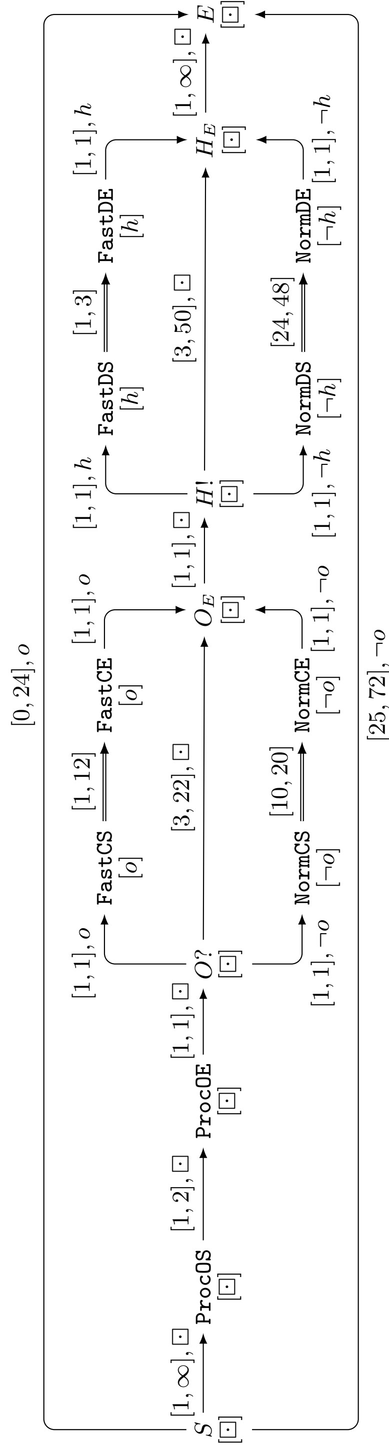


Fig. 5.9: CSTNUD corresponding to the workflow in Figure 5.9.

```
22 }
23
24 ContingentLinks {
25         (FastCS,1,3,FastCE)
26         (NormCS,10,20,NormCE)
27         (FastDS,1,12,FastDE)
28         (NormDS,24,48,NormDE)
29 }
30
31 Constraints {
32         (S - ProcOS <= -1 : )
33         (ProcOE - ProcOS <= 2 : )
34         (ProcOS - ProcOE <= -1 : )
35         (O - ProcOE <= 1 : )
36         (ProcOE - O <= -1 : )
37         (FastCS - O <= 1 : o)
38         (O - FastCS <= -1 : o)
39         (NormCS - O <= 1 : !o)
40         (O - NormCS <= -1 : !o)
41         (OE - FastCE <= 1 : o)
42         (FastCE - OE <= -1 : o)
43         (OE - NormCE <= 1 : !o)
44         (NormCE - OE <= -1 : !o)
45         (H - OE <= 1 : )
46         (OE - H <= -1 : )
47         (FastDS - H <= 1 : h)
48         (H - FastDS <= -1 : h)
49         (NormDS - H <= 1 : !h)
50         (H - NormDS <= -1 : !h)
51         (HE - FastDE <= 1 : h)
52         (FastDE - HE <= -1 : h)
53         (HE - NormDE <= 1 : !h)
54         (NormDE - HE <= -1 : !h)
55         (E - S <= 24 : o)
56         (S - E <= 0 : o)
57         (E - S <= 72 : !o)
58         (S - E <= -25 : !o)
59         (OE - O <= 22 : )
60         (O - OE <= -3 : )
61         (HE - H <= 50 : )
62         (H - HE <= -3 : )
63         (HE - E <= -1 : )
64 }
```

The workflow in Figure 5.7 dynamically controllable (because the underlying CSTNUD is so). The workflow is also weakly controllable because strong controllability $\Rightarrow$ dynamic controllability $\Rightarrow$ weak controllability. However, it is not strongly controllable mainly because the decision to hurry up is dynamic and depends on both the `1dd?` condition and the `NormC` duration.

Listing 5.6 shows a few execution simulations in which I isolated the scenarios of interest to prove that `hurry!` is made dynamically.

Listing 5.6: Random executions of the CSTNUD in Figure 5.9.

```
$ java -jar esse.jar DP.cstnud --execute dynamic DP.s 10000
Execution 1          |     Execution 2          |     Execution 3
-----------------------------------------------------------------------
S = 0.1              |     S = 0.1              |     S = 0.1
ProcOS = 1.1         |     ProcOS = 1.1         |     ProcOS = 1.1
ProcOE = 2.1         |     ProcOE = 2.1         |     ProcOE = 2.1
O = 3.1, o = true    |     O = 3.1, o = false   |     O = 3.1, o = false
FastCS = 4.1         |     NormCS = 4.1         |     NormCS = 4.1
FastCE = 6.1         |     NormCE = 23.5        |     NormCE = 18.6
OE = 7.1             |     OE = 24.5            |     OE = 19.6
H = 8.1, h = true    |     H = 25.5, h = true   |     H = 20.6, h = false
FastDS = 9.1         |     FastDS = 26.5        |     NormDS = 21.6
FastDE = 21.1        |     FastDE = 33.1        |     NormDE = 47.0
HE = 22.1            |     HE = 34.1            |     HE = 48.0
E = 23.1             |     E = 35.1             |     E = 49.0
Verifying ... SAT!   |     Verifying ... SAT!   |     Verifying ... SAT!
```

In all executions `ProcO` lasts exactly 1, starts at 1.1 and ends at 2.1 (lines 5,6). In the first execution, the handled order requires a one-day delivery (line 7) so `FastC` starts at 4.1 and is observed to end at 6.1 (lines 8,9). `FastD` is chosen, starts at 9.1 and is observed to end at 21.1 (lines 12,13). The process completes in time at 23.1 (line 15). In the second execution, the handled order does not require a one-day delivery so `NormC` starts at 4.1 and is observed to end at 23.5. Then, `FastD` is chosen (and not `NormD` as this type of delivery could lasts 48 hours making the process terminate at 75.5), starts at 26.5 and is observed to end at 33.1. The process completes in time at 35.1. In the third execution, the handled order does not require a one-day delivery either so `NormC` starts at 4.1 and is observed to end at 18.6. This time `NormD` is chosen (and not `FastD` as this type of delivery could lasts 1 hour making the process terminate at 23.6), starts at 21.6 and is observed to end at 47.0. The process completes in time at 49. This case study is available at `http://regis.di.univr.it/DP.tar.bz2`.

5.8 Conclusions

I defined *conditional simple temporal networks with uncertainty and decisions* (CSTNUDs) as a unified formalism. CSTNUDs implicitly embed all minor temporal network formalisms based on STNs (see Figure 5.4 for a hierarchy of simple temporal networks). I modeled the DC-checking via controller synthesis for timed game automata and provided an encoding from CSTNUDs into TGAs as an optimized extension of that given for CSTNUs in [25, 26]. I discussed the correctness and complexity of such an encoding and I provided Esse, a tool for CSTNUDs. I also discussed how to generate random temporal networks and carried out two experimental evaluations. I showed how temporal networks can be employed for the modeling, validation and execution of temporal workflows under conditional and temporal uncertainty.

CSTNUDs differ from STNs [53], Labeled STNs [45], TPNs [79], DTNs [117] and STNDs [19, 126] (see Chapter 4 for the latter) as none of these formalisms

specifies any uncontrollable part (consistency analysis is enough). CSTNUDs differ from TPNUs [90] and CSTNUs [36, 74] (and thus also from STNUs [104] and CSTNs [75]) because these formalism do not employ decisions. CSTNUDs differ from CCTPU [124] as a CCTPU do not employ observations and also differ from [76], which includes decisions, observations and uncontrollable durations, because that work deals with strong controllability only.

Part II

Resource Controllability

Introduction

Assume that we are given a resource-scheduling problem under uncertainty, and that we are then asked to schedule (some of the) resources in a way that meets all relevant constraints, or to prove that such a scheduling does not exist. We are also permitted to make our scheduling decisions as we like.

In this part I face two kinds of uncontrollable parts: conditional uncertainty and resource uncertainty (i.e., the uncontrollable availability of resources).

In recent years, a considerable amount of research has been carried out, especially in the temporal networks community, to investigate controllability analysis in order to deal with temporal and conditional uncertainty, either in isolation or simultaneously. A number of extensions of *simple temporal networks* have been proposed and already discussed in Section 2.1 and Part I.

Research has also been carried out in the "discrete" world of classic *constraint networks* (*CNs*) [52] in order to address different kinds of uncertainty. For example, a *mixed constraint satisfaction problem* (*Mixed CSP*, [63]) divides the set of variables in controllable and uncontrollable, whereas a *dynamic constraint satisfaction problem* (*DCSP*, [100]) introduces activity constraints saying when variables are relevant depending on what values some other variables have been assigned. Probabilistic approaches such as [62] aim instead at finding the most probable working solution.

Despite all this, a formal model to extend classic CNs [52] with conditional uncertainty adhering to the modeling ideas employed by CSTNs (Section 2.1.3) is still missing. In a CSTN, for instance, time points (variables) and linear inequalities (constraints) are labeled by conjunctions of literals where the truth value assignments to the embedded Boolean propositions are out of control. Every proposition has an associated *observation time point*, a special kind of time point that reveals the truth value assignment to the associated proposition upon its execution (i.e., as soon as it is assigned a real value). Equivalently, this truth value assignment can be thought of as being under the control of the environment.

Imagine now that the availability of users is *out of control*.

What can go wrong?

Some users might wake up feeling sick and call in to say that they are not going to come to work. Other users could be involved in a traffic jam on their way to work. And other users could simply decide to go on strike. In such a case, we must make sure that a successful execution is still possible with the only remaining users who agree not to leave until the workflow completes.

Are we safe now?

No, we are not. The situation could get worse than the previous one. Some of the users (with which the execution started) might get a call sadly announcing an emergency in the family. In such a case, we must make sure that a successful execution is still possible with, again, the only remaining users who might, this time, leave before the workflow completes.

Are we safe now?

Still, no. We could have our worst day ever. Some users might not arrive in time at work or might not arrive at all. Others could leave before the workflow ends and never come back in time to finish the work they started. Others could be absent only for a short period of time, for example to go to the doctor to collect some medical prescriptions. And so on. In such a case, we must make sure that at any time we have enough users left to keep executing the tasks until the workflow completes.

Now we are safe.

The previous three examples provide the intuitions behind the three main kinds of *workflow resiliency* defined in [122,123] by Wang and Li. *Static resiliency* is when users are absent before starting and do not come back (first scenario). *Decremental resiliency* is when they can also become absent during execution but, again, they do not come back (second scenario). *Dynamic resiliency* is when (possibly different sets of) users can become and stay absent for any (possibly big) time interval; they can also come back and become absent over and over again (third scenario).

Some works have addressed workflow resiliency probabilistically, e.g., [95, 96], whereas other works addressed it by modifying the constraints, e.g., [48, 93]. Most of these approaches consider only static resiliency (e.g., [57, 92, 107]), but one considers also decremental resiliency (e.g., [92]). However, to the best of my knowledge, dynamic resiliency remains unexplored (see also [55] for a recent survey).

For the sake of simplicity — and since these contributions are one of the first attempts to address the issues mentioned above — when used to abstract workflows, the formalisms proposed in this part will always consider a single workflow and assume to run one instance of that workflow at a time. However, I point out that this is not a limitation because multiple processes (or multiple instances of the same process) can be connected in parallel before translating this macro process into the corresponding underlying formalism. This will allow, for example, to consider resource scheduling problems, in which multiple processes share resources. The same ideas can be applied to Part I and Part III too.

Contributions

Towards the modeling, validation and execution of plans dealing with conditional uncertainty or the uncertain availability of the resources, my contributions in this part are the following.

1. I define *constraint networks under conditional uncertainty (CNCUs)* as an extension of classic constraint networks and give the semantics for weak, strong and dynamic controllability of a CNCU.
2. I provide algorithms to check each of these types of controllability and to execute a controllable CNCU.
3. I provide Zeta, a tool I developed for CNCUs along with an experimental evaluation. I also provide an algorithm to generate random CNCUs.
4. I provide a language to specify access controlled workflows under conditional uncertainty. I define the controllability of such workflows and give an encoding

from access controlled workflows to CNCUs for the weak, strong and dynamic controllability checking.

5. I focus on the uncertain resource availability and I provide encodings into timed game automata extended with variables to model static, decremental and dynamic resiliency as instantaneous games.
6. I show how to get dynamic plans for each kind of resiliency via controller synthesis by using UPPAAL-TIGA and provide Erre, a tool for resiliency analysis along with an experimental evaluation. I also provide an algorithm to generate random access-controlled workflows.

Organization

Chapter 6 introduces *constraint networks under conditional uncertainty* along with the semantics and the algorithms to check weak, strong and dynamic controllability and execute a controllable CNCU. It also discusses correctness results of the proposed algorithms and provides Zeta, a tool for CNCUs along with an experimental evaluation and an algorithm to generate random CNCUs. Finally, it provides a process modeling language for access controlled workflows under conditional uncertainty, defines the controllability of such workflows and gives an encoding from access controlled workflows to CNCUs for a dynamic controllability checking. Chapter 7 addresses static, decremental and dynamic resiliency. It provides three encodings from access controlled workflows into instantaneous timed games and provides Erre, a tool for workflow resiliency along with an experimental evaluation and an algorithm to generate random access controlled workflows.

6

Constraint Networks Under Conditional Uncertainty

In this chapter, I extend the CNs discussed in Section 2.3 to address conditional uncertainty. I call this new kind of network *Constraint Network under Conditional Uncertainty (CNCU)*. CNCUs are obtained by extending CNs with

- a set of Boolean *propositions* whose truth value assignments are out of control (or, equivalently, can be thought of as being under the control of the environment),
- *observation variables* to observe such truth value assignments, and
- *labels* to enable or disable a subset of variables and constraints, and therefore introduce an (implicit) notion of partial order among the variables.

Like temporal networks discussed in Part I, I will also talk about *execution* meaning that we *execute a variable* by assigning it a value and we *execute a CNCU* by executing all relevant variables. Variables and constraints are *relevant* if they must be considered during execution. Like observation time points, each observation variable $P?$ is associated to a (unique) Boolean proposition p. When $P?$ is still unexecuted, the truth value of p is unknown, whereas when $P?$ executes, it becomes known. Despite all this, my specification is mathematically backward-compatible with classic CNs (when there are no observation variables and no partial order).

6.1 Syntax

Labels and label operations are the same of those defined for CSTNs, CSTNUs and CSTNUDs (Section 2.1.3, Section 2.1.4 and Chapter 5). I further consider the *difference* of two labels ℓ_1 and ℓ_2 as a new label $\ell_3 = \ell_1 - \ell_2$ consisting of all literals of ℓ_1 minus those shared with ℓ_2. For instance, if $\ell_1 = p\neg q$ and $\ell_2 = p$, then $\ell_1 - \ell_2 = \neg q$ and $\ell_2 - \ell_1 = \boxdot$.

Definition 6.1. *A* constraint network under conditional uncertainty (CNCU) *is a tuple* $\langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$*, where:*

- $\mathcal{V} = \{V_1, V_2, \dots\}$ *is a finite set of variables.*
- $\mathcal{D} = \{D_1, D_2, \dots\}$ *is a set of discrete domains.*

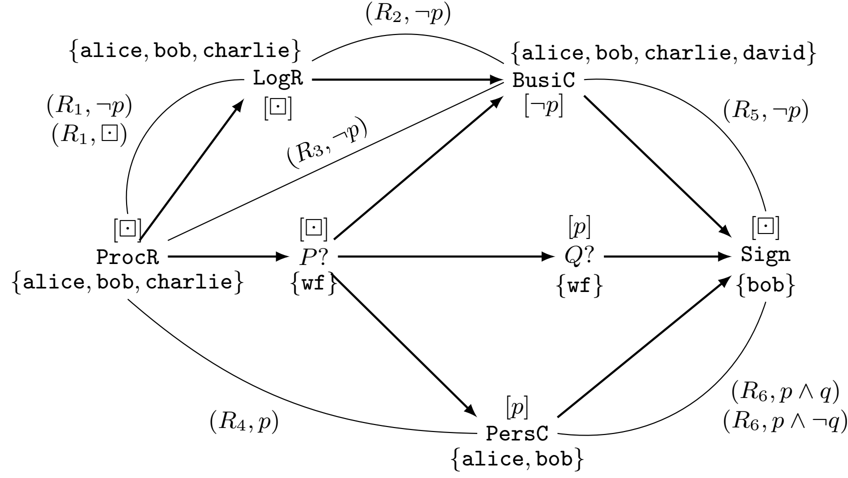


Fig. 6.1: Binary CNCU modeling the loan origination process.

- $D\colon \mathcal{V} \to \mathcal{D}$ *is a mapping assigning a domain to each variable, where more variables can share the same domain.*
- $\mathcal{OV} \subseteq \mathcal{V} = \{P?, Q?, \dots\}$ *is a set of* observation variables.
- $\mathcal{P} = \{p, q, \dots\}$ *is a set of Boolean* propositions *whose truth values are all initially unknown.*
- $O\colon \mathcal{P} \to \mathcal{OV}$ *is a bijection assigning a unique observation variable* $P?$ *to each proposition* p. *When* $P?$ *executes, the truth value of* p *becomes known and no longer changes.*
- $L\colon \mathcal{V} \to \mathcal{P}^*$ *is a mapping assigning a label* ℓ *to each variable* V *saying when* V *is relevant.*
- $\prec \subseteq \mathcal{V} \times \mathcal{V}$ *is a precedence relation on the variables. I write* $(V_1, V_2) \in \prec$ *(or* $V_1 \prec V_2$*) to express that* V_1 *is assigned* before V_2.
- $\mathcal{C}$ *is a finite set of* labeled relational constraints *of the form* (R_S, ℓ)*, where* $S \subseteq \mathcal{V}$ *and* $\ell \in \mathcal{P}^*$. *If* $S = \{V_1, \dots, V_n\}$*, then* $R_S \subseteq D(V_1) \times \cdots \times D(V_n)$.

I graphically represent a (binary) CNCU by extending the constraint graph discussed for CNs into a *labeled constraint (multi)graph*, where each variable is also labeled by its label $L(V)$, and the edges are of two kinds: *order edges* (directed unlabeled edges) and *constraint edges* (undirected labeled edges). An order edge $V_1 \to V_2$ models $V_1 \prec V_2$. A constraint edge between V_1 and V_2 models (R_{12}, ℓ). Many constraint edges may possibly be specified between the same pair of variables, as long as ℓ is different (e.g., $(R_1, \boxdot)$ and $(R_1, \neg p)$ between `ProcR` and `LogR` in Figure 6.1).

Consider now the CNCU in Figure 6.1 modeling an access controlled workflow under conditional uncertainty describing a loan origination process (LOP) for customers whose financial records have already been approved. The LOP starts by processing a request (`ProcR`) with Alice, Bob and Charlie being the only authorized users. After that, the request is logged for future accountability purposes (`LogR`) with the same users of `ProcR` authorized for this task. The flow of execution

Table 6.1: Labeled relational constraints of the CNCU in Figure 6.1.

(a) $(R_1, \boxdot)$

ProcR	LogR
alice	bob
alice	charlie
bob	alice
bob	charlie
charlie	alice
charlie	bob

(b) $(R_1, \neg p)$

ProcR	LogR
alice	alice
alice	charlie
bob	bob
bob	charlie
charlie	alice
charlie	bob
charlie	charlie

(c) $(R_2, \neg p)$

LogR	BusiC
alice	bob
alice	charlie
alice	david
bob	alice
bob	charlie
bob	david
charlie	alice
charlie	bob
charlie	david

(d) $(R_3, \neg p)$

ProcR	BusiC
alice	bob
alice	charlie
alice	david
bob	alice
bob	charlie
bob	david
charlie	alice
charlie	bob
charlie	david

(e) (R_4, p)

ProcR	PersC
charlie	alice
charlie	bob

(f) $(R_5, \neg p)$

BusiC	Sign
alice	bob
charlie	bob
david	bob

(g) $(R_6, p \wedge q)$

PersC	Sign
alice	bob

(h) $(R_6, p \wedge \neg q)$

PersC	Sign
bob	bob

then splits into two (mutually-exclusive) branches upon the execution of the observation variable P? acting as a "conditional split connector" which sets the truth value of p according to the *discovered* type of loan. A workflow engine (`wf`) is authorized to execute this conditional split connector. If p is true, it means that the workflow will handle a *personal loan* and that the flow of the execution continues by preparing a personal contract (`PersC`), with Alice and Bob the only authorized users. Moreover, when processing personal loans, different security policies hold depending on what truth value a second Boolean variable (q) is assigned (see below). The truth value of q is set upon the execution of the observation variable Q? (acting as a second "conditional split connector") whose authorized user is again `wf`. Note that no variable will be prevented from executing depending on the value of q, only the users assigned to them will. Instead, if p is false, the workflow will handle a *business loan* and the flow of execution continues by preparing a business contract (`BusiC`) with Alice, Bob, Charlie and David authorized users. Regardless of the truth values of p and q the LOP concludes with the signing of the contract (`Sign`) with Bob the only authorized user. Finally, the labeled constraints enforce the following security policies, where I recall that a *separation of duties (SoD)* (resp., *binding of duties (BoD)*) between two tasks says that the users executing such tasks must be different (resp., equal).

- $(R_1, \boxdot)$ calls for a SoD between `ProcR` and `LogR` (always, Table 6.1a), whereas $(R_1, \neg p)$ requires that the users executing `ProcR` and `LogR` are not relatives whenever p turns out to be false (Table 6.1b).
- $(R_2, \neg p)$ calls for a SoD between `LogR` and `BusiC` (implicitly when p is false, Table 6.1c).
- $(R_3, \neg p)$ calls for a SoD between `ProcR` and `BusiC` (implicitly when p is false, Table 6.1d).

- (R_4, p) calls for a SoD between `ProcR` and `PersC` *and* also requires that the users executing these two tasks must not be relatives (implicitly when p is true, Table 6.1e).
- $(R_5, \neg p)$ calls for a SoD between `BusiC` and `Sign` (implicitly when p is false, Table 6.1f).
- $(R_6, p \wedge q)$ calls for a SoD between `PersC` and `Sign` if p and q are both true (Table 6.1g), whereas $(R_6, p \wedge \neg q)$ calls for a BoD between the same variables if p is true and q is false (Table 6.1h).

In this example, Alice and Bob are married and thus the only relatives.

In the rest of this section, I say when CNCUs are well-defined. I inherit the notions of label honesty and coherence from those given for CSTNs in Section 2.1.3.

Definition 6.2. *A CNCU* $\langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$ *is* well defined *iff all labels are consistent and the following properties hold.*

- Variable Label Honesty. *$L(V)$ is honest for any $V \in \mathcal{V}$, and $O(p) \prec V$ for any p or $\neg p$ belonging to $L(V)$. That is, V only executes when the honest $L(V)$ becomes completely known and evaluates to true; e.g.,* `BusiC` after $P?$ if *$\neg p$ in Figure 6.1.*
- Constraint Label Honesty. *ℓ is honest for any $(R_S, \ell) \in \mathcal{C}$. That is, R_S only applies when the honest ℓ becomes completely known and evaluates to true; e.g., $(R_6, p \wedge q)$ in Figure 6.1 if* after *$P?$ and $Q?$, p and q* are observed *true.*
- Constraint Label Coherence. *$\ell \Rightarrow L(V)$ for any $(R_S, \ell) \in \mathcal{C}$ and any $V \in S$. That is, the label of a constraint is at least as specific as any label of the variables in its scope; e.g., $(R_6, p \wedge q)$ in Figure 6.1.*
- Precedence Relation Coherence. *For any $V_1, V_2 \in \mathcal{V}$, if $V_1 \prec V_2$ then $L(V_1) \wedge L(V_2)$ is consistent. That is, no partial order can be specified between variables not taking part together in any execution; e.g.* `PersC` *and* `BusiC` *in Figure 6.1.*

The CNCU in Figure 6.1 is well-defined.

6.2 Semantics

In this section, I give the semantics for weak, strong and dynamic controllability of CNCUs. My goal is to synthesize execution strategies saying which value to assign to which variable (and in which order) so that in the arising projection (see below) the execution satisfies both the partial order and all constraints.

Definition 6.3. *A* scenario *$s \colon \mathcal{P} \rightarrow \{\bot, \top\}$ is a complete truth value assignment to the Boolean propositions in $\mathcal{P}$. A scenario s satisfies a label ℓ (in symbols, $s \models \ell$), if ℓ is true under the interpretation given by s. Σ models the set of all scenarios.*

Consider Figure 6.1 and $s(p) = \top$ and $s(q) = \bot$. We have that $s \models L(\texttt{PersC})$ and $s \not\models L(\texttt{BusiC})$ (as $L(\texttt{BusiC}) = \neg p$ would require $s(p) = \bot$).

Definition 6.4. *Let $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$ be a CNCU and s any scenario. The* projection *of $\mathcal{Z}$ onto s is a CN $\mathcal{Z}_s = \langle \mathcal{V}_s, \mathcal{D}, \mathcal{C}_s \rangle$ such that:*

- $\mathcal{V}_s = \{V \mid V \in \mathcal{V} \wedge s \models L(V)\}$
- $\mathcal{C}_s = \{R_S \mid (R_S, \ell) \in \mathcal{C} \wedge s \models \ell\}$

For example, the projection of Figure 6.1 with respect to $s(p) = \top$ and $s(q) = \bot$ results in a CN, where $\mathcal{V}_s = \{\texttt{ProcR}, \texttt{LogR}, P?, \texttt{PersC}, \texttt{Sign}\}$ and $\mathcal{C}_s = \{R_1, R_4, R_6\}$, where R_1 is the relation of the original $(R_1, \boxdot) \in \mathcal{C}$ (Table 6.1a), R_4 is (R_4, p) (Table 6.1e), whereas R_6 is $(R_6, p \wedge \neg q)$ and not $(R_6, p \wedge q)$ or the intersection of the two (Table 6.1h).

Definition 6.5. *A* schedule *for a subset of variables $\mathcal{V}' \subseteq \mathcal{V}$ is a mapping $\psi \colon \mathcal{V}' \to \bigcup_{V \in \mathcal{V}'} D(V)$ from variables to values saying which values are assigned to which variables. A schedule is* consistent *if the assignments it makes satisfy all constraints. Ψ represents the set of all schedules.*

Consider $\mathcal{V}' = \{\texttt{ProcR}, \texttt{LogR}, P?, \texttt{Sign}\}$ containing the relevant variables when the truth values of p and q are still unknown. A consistent schedule is

$$\psi(\texttt{ProcR}) = \texttt{charlie}, \quad \psi(\texttt{LogR}) = \texttt{alice}, \quad \psi(P?) = \texttt{wf}, \quad \psi(\texttt{Sign}) = \texttt{bob}$$

satisfying the only relevant constraint $(R_1, \boxdot)$ (Table 6.1a). However, a schedule is nothing but a fixed plan for executing a bunch of variables (not even saying in which order). The interesting part is how we generate it. To do so, we need a *strategy*. Let Δ be the set of all possible orderings on the variables of a CNCU.

Definition 6.6. *An* execution strategy *is a pair $\sigma = (\sigma^v, \sigma^o)$ where $\sigma^v \colon \Sigma \to \Psi$ is a* value strategy *mapping scenarios to schedules, whereas $\sigma^o \colon \Sigma \to \Delta$ is an* order strategy *mapping scenarios to total orderings on the variables. An execution strategy is* viable *if for any $s \in \Sigma$, there exists an ordering $\sigma^o(s)$ such that the schedule $\sigma^v(s)$ is consistent.*

I write $[\sigma^v(s)]_V$ (instead of $\sigma^v(s)(V)$) to denote the value assigned to V and $[\sigma^o(s)]_V$ (instead of $\sigma^o(s)(V)$) to denote the index of V in the order $\sigma^o(s)$.

The first kind of controllability is *weak controllability* which ensures that each projection is consistent.

Definition 6.7. *A CNCU is* weakly controllable *(WC) if there exists a viable execution strategy.*

Figure 6.1 is weakly controllable. I prove that at the end of Section 6.3.1. Like temporal networks, dealing with weak controllability is quite complex as it always requires one to predict what the truth value assignments to the Boolean propositions will be before starting the execution. This leads me to consider the opposite case in which I want to synthesize a strategy working for all possible scenarios (or in other words, a solution which is not influenced by the uncontrollable part). Thus, the second kind of controllability is *strong controllability*.

Definition 6.8. *A CNCU is* strongly controllable *(SC) if there exists a viable execution strategy σ working for all scenarios.*

Figure 6.1 is not strongly controllable. I discuss why that at the end of Section 6.3.2. Strong controllability is, however, "too strong". If a CNCU is not strongly controllable, it could be still executable by refining the schedule in real time depending on the scenario being generated. To achieve this purpose, I introduce *dynamic controllability*. Since the truth values of propositions are revealed incrementally, I first introduce the formal definition of history that I use to define dynamic controllability.

Definition 6.9. *Given a strategy σ, a scenario s and a variable V, the* scenario history $\mathcal{H}(V, s, \sigma)$ *of V in s with respect to σ is the set of truth value assignments observed before V upon the execution of the corresponding observation variables $P?$ in the schedule $\sigma(s)$. Formally,*

$$\mathcal{H}(V, s, \sigma) = \{(p, s(p)) \mid [\sigma^o(s)]_{P?} < [\sigma^o(s)]_V\}$$

for any $P? \in \mathcal{OV}$.

Consider the ordering $\texttt{ProcR} \prec \texttt{LogR} \prec P? \prec Q? \prec \texttt{BusiC} \prec \texttt{PersC} \prec \texttt{Sign}$, the scenario $s(p) = s(q) = \top$, a strategy σ and the variable $\texttt{Sign}$. Then $\mathcal{H}(\texttt{Sign}, s, \sigma) = \emptyset$ before $P?$ and $Q?$ execute, $\mathcal{H}(\texttt{Sign}, s, \sigma) = \{(p, \top)\}$ after $P?$ and before $Q?$ executes, and $\mathcal{H}(\texttt{Sign}, s, \sigma) = \{(p, \top), (q, \top)\}$ after $P?$ and $Q?$ execute.

Definition 6.10. *An execution strategy $\sigma = (\sigma^v, \sigma^o)$ is* dynamic *if σ^v and σ^o are dynamic, where:*

- *A value strategy σ^v is dynamic if whenever the scenario history looks the same, then the strategy assigns the same values to the same variables. That is, for all $s_1, s_2 \in \Sigma$ and any $V \in \mathcal{V}$, if $\mathcal{H}(V, s_1, \sigma) = \mathcal{H}(V, s_2, \sigma)$, then $[\sigma^v(s_1)]_V = [\sigma^v(s_2)]_V$.*
- *An order strategy σ^o is dynamic if whenever the scenario history looks the same, then the strategy orders the variables always in the same way. That is, for all $s_1, s_2 \in \Sigma$ and any $V \in \mathcal{V}$, if $\mathcal{H}(V, s_1, \sigma) = \mathcal{H}(V, s_2, \sigma)$, then $[\sigma^o(s_1)] = [\sigma^o(s_2)]$.*

Definition 6.11. *A CNCU is* dynamically controllable *if there exists a dynamic and viable execution strategy.*

Figure 6.1 is dynamically controllable. I prove that at the end of Section 6.3.3.

Abusing grammar, I use WC, SC and DC as both nouns and adjectives (the use will be clear from the context). As for temporal networks [104], it is easy to see that SC $\Rightarrow$ DC $\Rightarrow$ WC.

6.3 Controllability Checking Algorithms

In this section, I provide the algorithms to check the three kinds of controllability introduced in Section 6.2. Since I am going to exploit directional consistency, I first need to address how to get a suitable total order for the variables meeting the restrictions specified by $\prec$. I will always classify as uncontrollable those CNCUs for which no total order exists. Although for weak and strong controllability the

problem of getting an order is important up to a certain extent (I only need to make sure that one exists), it will be absolutely necessary to get the *most conservative order* when dealing with dynamic controllability, otherwise the algorithm faces for sure incompleteness (see Section 6.3.3).

Given a CNCU, to get a possible total order coherent with $\prec$, I build a directed graph G where the set of nodes is $\mathcal{V}$ and the set of edges is such that there exists a directed edge $V_1 \rightarrow V_2$ in G for any $(V_1, V_2) \in \prec$. I refer to this graph as $G = \langle \mathcal{V}, \prec \rangle$. For example, in Figure 6.1, G is the graph that remains after removing all labels and constraint edges.

From graph theory, we know that an ordering of the vertexes of a directed acyclic graph (DAG) meeting a given restriction $\prec$ can be found in polynomial time by running the TopologicalSort algorithm on G [46, 80]. At every step, TopologicalSort chooses a vertex V without any predecessor (i.e., one without incoming edges), outputs V and removes V and all directed edges from V to any other vertex (equivalently, removes every $(V, V_2) \in \prec$). Then, TopologicalSort recursively applies to the reduced graph until the set of vertexes becomes empty. If no total order exists, TopologicalSort gets stuck in some iteration because of a cycle $V_i \rightarrow \cdots \rightarrow V_i$, which makes impossible to find a vertex without any predecessor.

6.3.1 Weak controllability checking

The idea behind the *weak controllability checking (WC-checking)* is quite simple: *every projection must have a total order and a solution.* Given a CNCU $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$, I run the classic ADC on each projection $\mathcal{Z}_s$ according to a complete scenario s. Since each $\mathcal{Z}_s$ is a classic CN, any ordering (meeting the relevant part of $\prec$ for $\mathcal{Z}_s$) will be fine. I get one by running TopologicalSort on $G_s = \langle \mathcal{V}_s, \prec_s \rangle$, where $\prec_s = \{(V_1, V_2) \mid (V_1, V_2) \in \prec \wedge V_1, V_2 \in \mathcal{V}_s\}$ (this is the relevant part of $\prec$). After that, I synthesize a strategy $\sigma(s)$ by generating a solution for the projection $\mathcal{Z}_s$ following the ordering d computed initially (Algorithm 10, line 4). Although Definition 6.7 says that one strategy is enough, my approach is able to handle all possible strategies for each scenario s as during the solution-generation process the value assignments do not depend on any uncontrollable part. WC-checking (Algorithm 10) shows the pseudo-code of the algorithm.

The CNCU in Figure 6.1 is WC. To prove that, I give an assignment of values to variables for each scenario.

- If $s(p) = \bot$, $s(q) = \{\bot, \top\}$, then $\psi(\texttt{ProcR}) = \texttt{alice}, \psi(P?) = \texttt{wf}, \psi(\texttt{LogR}) = \texttt{charlie}, \psi(\texttt{BusiC}) = \texttt{david}, \psi(\texttt{Sign}) = \texttt{bob}$ with $\texttt{ProcR} \prec P? \prec \texttt{LogR} \prec \texttt{BusiC} \prec \texttt{Sign}$.
- If $s(p) = \top, s(q) = \top$, then $\psi(\texttt{ProcR}) = \texttt{charlie}, \psi(P?) = \texttt{wf}, \psi(\texttt{PersC}) = \texttt{alice}, \psi(Q?) = \texttt{wf}, \psi(\texttt{Sign}) = \texttt{bob}, \psi(\texttt{LogR}) = \texttt{alice}$ and $\texttt{ProcR} \prec P? \prec \texttt{PersC} \prec Q? \prec \texttt{Sign} \prec \texttt{LogR}$.
- If $s(p) = \top, s(q) = \bot$, then $\psi(\texttt{ProcR}) = \texttt{charlie}, \psi(P?) = \texttt{wf}, \psi(\texttt{PersC}) = \texttt{bob}, \psi(Q?) = \texttt{wf}, \psi(\texttt{Sign}) = \texttt{bob}, \psi(\texttt{LogR}) = \texttt{bob}$ and $\texttt{ProcR} \prec P? \prec \texttt{PersC} \prec Q? \prec \texttt{Sign} \prec \texttt{LogR}$.

Algorithm 10: WC-CHECKING ($\mathcal{Z}$).

Input: A CNCU $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$
Output: A set of solutions each one having the form $\langle s, d, \mathit{Buckets} \rangle$, where s is a scenario, d an ordering for $\mathcal{V}_s$ and *Buckets* is a set of buckets (one for each variable in $\mathcal{V}_s$) if $\mathcal{Z}_s$ is WC, uncontrollable otherwise.

```
1  Solutions ← ∅
2  foreach s ∈ Σ do                                    ▷ for each scenario
3      Let Z_s be the projection of Z onto s
4      d ← TopologicalSort(G)                          ▷ where G ← ⟨V_s, ≺_s⟩
5      if no order is possible then
6          return uncontrollable
7      Buckets ← ADC(Z_s, d)
8      if Z_s is inconsistent then
9          return uncontrollable
10     Solutions ← Solutions ∪ {⟨s, d, Buckets⟩}
11 return Solutions
```

Note that in the first scenario, the value of $s(q)$ is not important because $Q?$ is not executed when $s(p) = \bot$. Therefore, the first case holds for both $s(p) = \bot$, $s(q) = \bot$ and $s(p) = \bot$, $s(q) = \top$. The relevant part of the complexity of WC-CHECKING is $2^{|\mathcal{P}|} \times \mathit{Complexity}(ADC)$ as the worst case is a CNCU specifying $2^{|\mathcal{P}|}$ complete scenarios (all other sub-algorithms run in polynomial time).

6.3.2 Strong controllability checking

The *strong controllability checking (SC-checking)* does not need to unfold all honest scenarios at all. From an algorithmic point of view it is even easier to understand: *a single ordering and a single solution must work for all projections.* To achieve this purpose, I start with a simple operation: *I wipe out all the labels in the CNCU.* Then, I run ADC on this "super-projection" by choosing an order obtained by TOPOLOGICALSORT run on the related G (Algorithm 11). Strong controllability forces solutions (if any) to also satisfy constraints that are inconsistent one another. If the buckets survive to the filling phase (i.e., no empty relation is added), ADC tries to hunt down an empty relation enforcing the adequate level of k-consistency. If this resulting network is consistent, it means that there exists (at least) a solution which is so strong that it does not depend on any uncontrollable part (i.e., a solution that just works).

The CNCU in Figure 6.1 is *not* SC. Although a total order exists once we have wiped out all the labels ($d =$ `ProcR` $\prec P? \prec$ `PersC` $\prec Q? \prec$ `LogR` $\prec$ `BusiC` $\prec$ `Sign`), there is no way to find a consistent assignment to `Sign` that always works for the initial CNCU. It is not difficult to see that the problem lies in the constraints of the original CNCU shown in Table 6.1. In the first phase, when ADC fills the buckets, each original constraint (R_S, ℓ) is deprived of its label ℓ (becoming $(R_S, \boxdot)$) and added to the bucket of the latest variable in S.

Consider the original $(R_6, p \wedge q)$ (Table 6.1g) and $(R_6, p \wedge \neg q)$ (Table 6.1h). ADC transforms them into (two) *unlabeled* constraints $(R_6, \boxdot)$ and then adds both

Algorithm 11: SC-CHECKING ($\mathcal{Z}$)

Input: A CNCU $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$

Output: A tuple $\langle d, \mathit{Buckets} \rangle$, where d is a total ordering for $\mathcal{V}$ and *Buckets* is a set of buckets (one for each variable) if $\mathcal{Z}$ is SC, uncontrollable otherwise.

1 Compute a CN $\mathcal{Z}_* \leftarrow \langle \mathcal{V}, \mathcal{D}, \mathcal{C}_* \rangle$ where $\mathcal{C}_* \leftarrow \{R_S \mid (R_S, \ell) \in \mathcal{C}\}$
2 $d \leftarrow$ TOPOLOGICALSORT(G) ▷ where $G \leftarrow \langle \mathcal{V}, \prec \rangle$
3 **if** *no order is possible* **then**
4   **return** *uncontrollable*
5 **return** ADC$(\mathcal{Z}_*, d)$

to *Bucket*(S). Since the labels of the two relations are the same, *Bucket*(`Sign`) actually contains the intersection of the two (as both must hold). However, $(\{(\mathtt{a}, \mathtt{b})\}, \boxdot) \cap (\{(\mathtt{b}, \mathtt{b})\}, \boxdot) = (\underline{\emptyset}, \boxdot)$.

In other words, in Figure 6.1 Bob always does the signing. The problem is that the user who prepares the personal contract must be different according to which truth value q will be assigned. If the system calls for a SoD (i.e., $s(q) = \top$), then Alice prepares the contract, else Bob does it. However, the intersection of the users allowed to carry out this task according to q is empty, which means that the user who prepares the contract for a personal loan *cannot* be decided before the execution starts.

The complexity of SC-CHECKING is $|\mathcal{C}| + \mathit{Complexity}(\mathrm{ADC})$. Despite computing a total ordering and wiping out the labels on variables run in polynomial time, I recall that in the worst case $\mathcal{C}$ may specify $K - 1$ relational constraints (where K is the number of all subsets of $\mathcal{V}$) and each relation may in turn appear $(2^{|\mathcal{P}|} + 1)$ times according to all possible different labels.

6.3.3 Dynamic controllability checking

The *dynamic controllability checking (DC-checking)* addresses the most appealing type of controllability. If a CNCU is not SC, it could be DC by deciding which value to assign to which variable depending on how the uncontrollable part behaves. This subsection discusses this algorithm. I start with LABELEDADC (Algorithm 13), a main sub-algorithm I make use of, which extends ADC to address the conditional part by refining the adding or tightening of constraints to the buckets and the constraint-propagation.

When I add a constraint (R_S, ℓ) to a *Bucket*(V), I *lighten* ℓ by removing all literals p or $\neg p$ in ℓ that will still be *unknown* by the time V executes. That is, those whose related observation variables are either V itself or will be assigned after V according to d.

When propagating constraints, LABELEDADC enforces the adequate level of k-consistency for all combinations of relevant (partial) scenarios arising from the conjunctions of all labels related to the constraints in the buckets. That is, for each V, it runs CCLOSURE on the set $\mathit{Closure} = \{\ell \mid (R_S, \ell) \in \mathit{Bucket}(V)\}$. After that, it generates a new constraint (R_{S_n}, ℓ_n) for each $\ell_n \in \mathit{Closure}$, where S_n is the union of the scopes of the constraints in *Bucket*(V) (whose labels are entailed

Algorithm 12: CCclosure(*Labels*)

Input: A set of labels *Labels*
Output: The closure of all possible consistent conjunctions

1 $Closure \leftarrow Labels$
2 **do**
3   Pick two labels ℓ_1 and ℓ_2 from *Closure*
4   **if** $\ell_1 \wedge \ell_2$ *is consistent and* $\ell_1 \wedge \ell_2 \notin Closure$ **then**
5     $Closure \leftarrow Closure \cup \{\ell_1 \wedge \ell_2\}$
6 **while** *Any adding is possible*
7 **return** *Closure*

Algorithm 13: LabeledADC($\mathcal{Z}, d$)

Input: A CNCU $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$ and an ordering $d = V_1 \prec \cdots \prec V_n$
Output: A set *Buckets* of buckets (one for each variable) if $\mathcal{Z}$ is consistent *along* d, inconsistent otherwise.

1 **foreach** $(R_S, \ell) \in \mathcal{C}$ **do** ▷ Partition constraints as follows
2   Let V be the latest variable in S according to d
3   Let ℓ_{Rem} be the conjunction of all literals p or $\neg p$ in ℓ such that either $V = P?$ or $V \prec P?$ in d, where $P? = O(p)$ ▷ Remove unknown literals
4   Add $(R_S, \ell - \ell_{Rem})$ to $Bucket(V)$
5 **foreach** V *in* d *taken in reverse order* **do** ▷ Process buckets
6   $Closure \leftarrow$ CCclosure($\{\ell \mid (R_S, \ell) \in Bucket(V)\}$)
7   **for** $\ell_n \in Closure$ **do** ▷ new constraint's label
8     $Entailed \leftarrow \{R_S \mid (R_S, \ell) \in Bucket(V) \wedge \ell_n \Rightarrow \ell\}$
9     $S_n \leftarrow \bigcup_{R_S \in Entailed} S \setminus \{V\}$ ▷ new constraint's scope
10   Compute $R_{tmp} \leftarrow \bowtie_{R_S \in Entailed} R_S$ ▷ enforce k-consistency
11   **if** $R_{tmp} = \emptyset$ **then**
12   **return** inconsistent
13   **if** $S_n \neq \emptyset$ **then** ▷ propagate the new constraint
14       $R_{S_n} \leftarrow \pi_{S_n}(R_{tmp})$ ▷ Project onto the new scope
15   Let V_n be the latest variable in S_n according to d
16   Compute ℓ_{Rem} as before but w.r.t. ℓ_n
17   Add $(R_{S_n}, \ell_n - \ell_{Rem})$ to $Bucket(V_n)$
18 $Buckets \leftarrow \{\{Bucket(V)\} \mid V \in \mathcal{V}\}$
19 **return** *Buckets*

by ℓ_n) deprived of V. R_{S_n} contains all tuples surviving the join of the entailed constraints projected onto S_n (as in the classic ADC). If no empty relation is computed, then the new constraint is added to the bucket of the latest variable in S_n (if any). If $S_n = \emptyset$, then it means that the algorithm computed a (implicit) unary constraint for V.

Finally, LabeledADC returns the set of buckets from which any solution can be built *according to* d (or inconsistent if no solution exists).

Algorithm 14: DC-CHECKING ($\mathcal{Z}$)

Input: A CNCU $\mathcal{Z} = \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$
Output: A tuple $\langle d, \mathit{Buckets} \rangle$, where d is an ordering for $\mathcal{V}$ and *Buckets* is a set of buckets (one for each variable) if $\mathcal{Z}$ is DC along d, uncontrollable otherwise.

```
1 G ← ⟨V, ≺⟩                                    ▷ All variables, whole partial order
2 foreach ordering d ∈ AllTopologicalSorts(G) do
3     Buckets ← LabeledADC(Z, d)
4     if Z is consistent (along d) then
5         return ⟨d, Buckets⟩
6 return uncontrollable
```

However, given an ordering d, if LABELEDADC "says no", it could be a matter of wrong ordering. Consider `PersC`, $Q?$ and `Sign` in Figure 6.1, and assume that those three variables are ordered as `PersC` $\prec Q? \prec$ `Sign`. Furthermore, consider Table 6.1 and suppose that `PersC` $=$ `alice`. When $Q?$ is executed ($Q? =$ `wf`), the truth value of q becomes known (recall that in this partial scenario $s(p) = \top$). If $s(q) = \top$, then `Sign` $=$ `bob` and $(\texttt{alice}, \texttt{bob}) \in (R_6, p \wedge q)$ (Table 6.1g), but if $s(q) = \bot$, then `Sign` $=$ `bob` and $(\texttt{alice}, \texttt{bob}) \notin (R_6, p \wedge \neg q)$ (Table 6.1h).

More simply, if Alice executes `PersC` and afterwards $s(q) = \bot$, then no valid user remains for `Sign` as our example calls for a binding of duties between the two tasks (Table 6.1h). If Bob executes `PersC` and afterwards $s(q) = \top$, then the problem is the same (so there is no user who can be assigned conservatively to `PersC` without any information on the truth value of q). Fortunately, `PersC` and $Q?$ are *unordered* (no precedence is specified between the two variables). This situation allows me to act in a more clever way: *What if I executed* $Q?$ *before executing* `PersC`*?* In such a case, I would have full information on $s(q)$ and my *strategy* would be: if $s(q) = \top$, then Alice, else Bob.

Formally, DC-CHECKING (Algorithm 14) works by looking for an ordering d coherent with $\prec$ such that LABELEDADC "says yes" when analyzing $\mathcal{Z}$ along d. If no ordering works, then the network is uncontrollable. DC-CHECKING iterates on all possible orderings by using internally the ALLTOPOLOGICALSORTS algorithm [80]. Every time a total order is found, DC-CHECKING runs LABELEDADC on the CNCU with respect to that order. If the CNCU is consistent, then DC-CHECKING stops and returns the order and the sets of buckets.

For example, the CNCU in Figure 6.1 is DC along the ordering $d_1 =$ `ProcR` $\prec$ `LogR` $\prec P? \prec Q? \prec$ `BusiC` $\prec$ `PersC` $\prec$ `Sign` and uncontrollable along $d_2 =$ `ProcR` $\prec$ `LogR` $\prec P? \prec$ `BusiC` $\prec$ `PersC` $\prec Q? \prec$ `Sign` (as `PersC` is assigned before $Q?$).

I execute a CNCU proved to be DC as follows. Let ℓ_s be the label corresponding to the current scenario. Initially $\ell_s = \boxdot$. For each variable V along the ordering d, if V is relevant for ℓ_s (i.e., if $s \models L(V)$), then I look for a value v in the domain of V satisfying all relevant constraints in $\mathit{Bucket}(V)$. If V is irrelevant (as ℓ_s falsifies $L(V)$), then I ignore V and go ahead with the next variable (if any). Moreover, if V is an observation variable, where p is the associated proposition, then ℓ_s extends

to $\ell_s \wedge p$ iff p is assigned true (i.e., $s(p) = \top$), and to $\ell_s \wedge \neg p$ otherwise. In this way, a partial scenario extends to a complete one, one observation variable at a time.

A strategy to execute the CNCU in Figure 6.1 is the following: Charlie executes `ProcR`, Alice `LogR` and the workflow engine executes the first conditional split connector (always). If $s(p) = \bot$, then David executes `BusiC`. If $s(p) = \top$, then the workflow engine executes the second split connector to have full information on q. If $s(q) = \top$, then Alice executes `PersC`, else Bob. Bob executes `Sign` (always).

The complexity of DC-checking is $|\mathcal{V}|! \times \mathit{Complexity}(\text{LabeledADC})$ as in the worst case there are $|\mathcal{V}|!$ orderings. I leave the investigation of the complexity of LabeledADC as future work.

6.4 Correctness of the Algorithms

I discuss some correctness results of the algorithms I proposed in Section 6.3. I start by defining what *soundness* and *completeness* for a controllability checking algorithm are.

Definition 6.12. *A controllability algorithm is* sound *if, whenever it classifies a CNCUs as uncontrollable, the CNCU is really uncontrollable, and it is* complete *if, whenever a CNCU is uncontrollable, the algorithm classifies it as so. A controllability algorithm is correct if it is sound and complete.*

I sketch the proofs of the soundness and completeness of my algorithms. Given a CNCU $\mathcal{Z}$ and any scenario $s \in \Sigma$, WC-checking runs TopologicalSort and subsequently ADC on $\mathcal{Z}_s$. If no total order exists for $\mathcal{Z}_s$ or $\mathcal{Z}_s$ is inconsistent, then the original CNCU is uncontrollable as there is no way to satisfy the constraints if s happens (regardless of whether we know it before starting the execution). Thus, WC-checking is sound. WC-checking is also complete because it does so for all possible scenarios guaranteeing that if some projection is inconsistent or no total order exists for that projection, WC-checking will find out. Thus, WC-checking is correct.

SC-checking first wipes out the conditional part of the original CNCU obtaining a super-projection whose set of constraints corresponds to the intersection of all sets of constraints (even inconsistent with each other) related to all possible projections. Afterwards, SC-checking runs TopologicalSort and then ADC on the resulting super-projection. Hence, SC-checking is sound and complete as it computes a total order in a correct way (provided one exists) with TopologicalSort which is known to be correct [46, 80] and tests the resulting projection along that order with ADC which is known to be sound and complete [52]. Since all variables and constraints are kept, we are sure that a solution (if any) will satisfy the "for all scenarios"-part as requested by Definition 6.8.

Note that WC-checking and SC-checking carry out the analysis on (possibly many) *unconditional* CNs. I point out that the chosen ordering according to $\prec$ given in input to ADC never breaches soundness and completeness of ADC but might only affect its complexity (see the discussion on induced width in [52]).

LabeledADC extends ADC to accommodate the propagation of labeled constraints. When it adds a constraint to the bucket of a variable V it lightens the label

of the constraint by removing all literals whose truth value will be still unknown by the time V executes. This is because the observation variables associated to the propositions embedded in those literals will be executed *after* V or coincide with V itself. For this reason, we must be conservative and consider the constraint as if it just held, since we are unable to predict "what is going to be". LABELEDADC propagates the constraints enforcing the adequate level of k-consistency for all possible combinations of honest (partial) scenarios arising from the labels of the constraints in a bucket. If LABELEDADC detects an inconsistency, it means that there exists a (partial) scenario for which the value assignments to the variables of the CNCU (along with the ordering in input) will violate some constraint. I believe that DC-CHECKING is correct as it runs LABELEDADC on all possible orderings. If LABELEDADC detects inconsistency for all orders (or no total ordering exists), then the CNCU is uncontrollable (soundness), whereas if LABELEDADC finds an order for which LABELEDADC "says yes", then the CNCU is dynamically controllable with respect to that order (completeness). I leave a formal proof as future work.

6.5 ZETA: A tool for CNCUs

I developed ZETA, a tool for CNCUs that takes in input a specification of a CNCU and acts both as a solver for weak, strong and dynamic controllability and as an execution simulator. Listing 6.1 shows ZETA's help screen.

Listing 6.1: ZETA's help screen.

```
Usage: java -jar zeta.jar <network.cncu> ACTION <network.ob> [N] [--silent]

ACTION:
    --WCchecking performs weak controllability checking.
    --SCchecking performs strong controllability checking.
    --DCchecking performs dynamic controllability checking.
    --execute    performs [N] executions of a (weakly/strongly/dynamically)
                 controllable network. If N is not specified, then the
                 default value is 1.
    --silent     [--silent] suppresses the output (optional). If --silent
                 is specified, then check the return value when doing --WC,
                 --SC and --DCchecking. 0 means controllable, 1 means
                 uncontrollable.

Examples:
---------
  java -jar zeta.jar Network.cncu --WCchecking Network.ob
  java -jar zeta.jar Network.cncu --SCchecking Network.ob
  java -jar zeta.jar Network.cncu --DCchecking Network.ob
  java -jar zeta.jar Network.cncu --execute Network.ob 1000
```

The input language comprises five main sections. The section `Domain`

```
Domains {
  ...
```

```
  (D : v1 ... vn)
  ...
}
```

specifies the set $\mathcal{D}$ and provides here an example of $D = \{v_1, \ldots, v_n\}$. The section `Proposition` is exactly the same of that given for KAPPA and ESSE. The section `Variables`

```
Variables {
  ...
  (P : p : q !r ...)
  (X : : p q !r ...)
  ...
}
```

specifies the sets $\mathcal{V}$, $\mathcal{OV}$ as well as the mappings O, D and L, and gives here an example of an observation variable $P?$ such that $O(p) = P?$ and $L(P?) = q\neg r$ and a normal variable X where $L(X) = pq\neg r$. The section `Precedence`

```
Precedence {
  ...
  (V1 < V2)
  ...
}
```

specifies the precedence relation $\prec$ and provides here an example of $V_1 \prec V_2$. The section `Constraints`

```
Constraints {
  ...
  (X Y ... : (v1 ...) ... : p !q ...)
  ...
}
```

specifies the set $\mathcal{C}$ and provides here an example of (R_S, ℓ), where $S = \{X, Y, \ldots\}$, $\ell = p\neg q \ldots$ and $R = \{(v_1, \ldots), \ldots\}$.

Given a CNCU specification file `network.cncu`, weak, strong and dynamic controllability are respectively checked by running

```
$ java -jar zeta.jar network.cncu --WCchecking network.ob
$ java -jar zeta.jar network.cncu --SCchecking network.ob
$ java -jar zeta.jar network.cncu --DCchecking network.ob
```

If the CNCU is proved controllable, ZETA saves to file the *order and buckets* needed to later generate any solution (for weak controllability, ZETA does so for any complete scenario). A controllable CNCU is executed by running

```
$ java -jar zeta.jar network.cncu --execute network.ob [N]
```

where `[N]` (default 1) is the number of simulations we want to carry out. For weak controllability, ZETA executes the CNCU with respect to each complete scenario, whereas for strong and dynamic, it executes the CNCU generating a random scenario (that is why ZETA allows for multiple simulations).

Listing 6.2 shows the specification of Figure 6.1 into ZETA's input language.

Listing 6.2: Specification of Figure 6.1 in ZETA's input language.

```
Domains {
  (D1 : alice bob charlie)
  (D2 : alice bob charlie david)
  (D3 : alice bob)
  (D4 : wf)
  (D5 : bob)
}

Propositions {
  p q
}

Variables {
  (ProcR : : D1 : )
  (LogR : : D1 : )
  (P : p : D4 : )
  (Q : q : D4 : p)
  (BusiC : : D2 : !p)
  (PersC : : D3 : p)
  (Sign : : D5 : )
}

Precedence {
  (ProcR < LogR)
  (ProcR < P)
  (P < Q)
  (P < BusiC)
  (P < PersC)
  (LogR < BusiC)
  (PersC < Sign)
  (BusiC < Sign)
  (Q < Sign)
}

Constraints {
  # (R1, )
  (ProcR LogR : (alice bob) (alice charlie) (bob alice) (bob charlie)
              (charlie alice) (charlie bob) : )
  # (R1, !p)
  (ProcR LogR : (alice alice) (alice charlie) (bob bob) (bob charlie)
              (charlie alice) (charlie bob) (charlie charlie) : !p)
  # (R2, !p)
  (LogR BusiC : (alice bob) (alice charlie) (alice david) (bob alice)
              (bob charlie) (bob david) (charlie alice) (charlie bob)
              (charlie david) : !p)
  # (R3, !p)
  (ProcR BusiC : (alice bob) (alice charlie) (alice david) (bob alice)
              (bob charlie) (bob david) (charlie alice) (charlie bob)
              (charlie david) : !p)
  # (R4, p)
```

```
51   (ProcR PersC : (charlie alice) (charlie bob) : p)
52
53   # (R5, !p)
54   (BusiC Sign : (alice bob) (charlie bob) (david bob) : !p)
55
56   # (R6, p q)
57   (PersC Sign : (alice bob) : p q)
58
59   # (R6, p !q)
60   (PersC Sign : (bob bob) : p !q)
61 }
```

I ran Zeta on the CNCU in Figure 6.1. I used a FreeBSD virtual machine run on top of a VMWare ESXi Hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM. The VM was assigned 16GB of RAM and full CPU power. Zeta proved in about 234 milliseconds that the CNCU in Figure 6.1 is weakly controllable (saving an `ob`-file of 12Kb), is not strongly controllable (in about 200 milliseconds) but is dynamically controllable in about 274 milliseconds (saving an `ob`-file of 8Kb). For weak and dynamic controllability, the CNCU was correctly executed. This example is available at http://regis.di.univr.it/LOP_LNAI2018.tar.bz2. Listing 6.3 shows the output of Zeta.

Listing 6.3: WC, SC and DC-checking of Figure 6.1 with Zeta.

```
$ java -jar zeta.jar Example.cncu --WCchecking Example.weak.ob
Weakly Controllable

$ java -jar zeta.jar Example.cncu --SCchecking Example.strong.ob
Uncontrollable

$ java -jar zeta.jar Example.cncu --DCchecking Example.dynamic.ob
Dynamically Controllable
```

Listing 6.4 shows one execution simulation for weak controllability. In the execution, Zeta provides a solution (i.e., a consistent schedule plus a total order) for each scenario. Zeta just picks up a random user for each variable among those valid for the specific projection.

Listing 6.4: Execution simulations for Figure 6.1 (weak controllability).

```
$ java -jar zeta.jar Example.cncu --execute Example.weak.ob
====================================
Scenario: !p
Order: ProcR -> P -> LogR -> BusiC -> Sign
------------------------------------
ProcR = alice
P = wf
LogR = charlie
BusiC = david
Sign = bob
------------------------------------
Verifying ... SAT!
```

```
====================================
====================================
Scenario: p q
Order: ProcR -> P -> PersC -> Q -> Sign -> LogR
------------------------------------
ProcR = charlie
P = wf
PersC = alice
Q = wf
Sign = bob
LogR = alice
------------------------------------
Verifying ... SAT!
====================================
====================================
Scenario: p !q
Order: ProcR -> P -> PersC -> Q -> Sign -> LogR
------------------------------------
ProcR = charlie
P = wf
PersC = bob
Q = wf
Sign = bob
LogR = bob
------------------------------------
Verifying ... SAT!
====================================
```

Listing 6.5 shows three (random) execution simulations for dynamic controllability. In the execution, ZETA always provides a solution no matter which scenario will arise (I isolated the executions generating all scenarios).

Listing 6.5: Execution simulations for Figure 6.1 (dynamic controllability).

```
$ java -jar zeta.jar Example.cncu --execute Example.dynamic.ob 3
====================================
Order: ProcR -> LogR -> P -> Q -> BusiC -> PersC -> Sign
------------------------------------
ProcR = charlie
LogR = alice
P = wf, p = true
Q = wf, q = false
PersC = bob
Sign = bob
------------------------------------
Verifying ... SAT!
====================================
====================================
Order: ProcR -> LogR -> P -> Q -> BusiC -> PersC -> Sign
------------------------------------
ProcR = charlie
LogR = alice
```

```
P = wf, p = true
Q = wf, q = true
PersC = alice
Sign = bob
-----------------------------------
Verifying ... SAT!
===================================
===================================
Order: ProcR -> LogR -> P -> Q -> BusiC -> PersC -> Sign
-----------------------------------
ProcR = charlie
LogR = alice
P = wf, p = false
BusiC = david
Sign = bob
-----------------------------------
Verifying ... SAT!
===================================
```

Once again, having a tool allowed me to carry out an automated experimental evaluation to compare the performances of DC-CHECKING. I summarize my findings in the following.

I generated 3000 CNCUs partitioned in 3 sets of benchmarks: `weak/`, `strong/` and `dynamic/`. Each set contains a directory `6vars/` specifying CNCUs with 6 variables and 5 sub-directories `2obs/`, `3obs/`, `4obs/`, `5obs/` and `6obs/` partitioning them by the number of observation variables, where each `Xobs` contains 2 further directories `controllable/` and `uncontrollable/`, each one containing 100 CNCUs. I generated the networks such that: (i) all weakly controllable CNCUs are neither strongly nor dynamically controllable and (ii) all dynamically controllable CNCUs are not strongly controllable. For example, `weak/6vars/3obs/controllable` contains weakly controllable CNCUs (only) with 6 variables, 3 of which are observation variables, `strong/6vars/4obs/uncontrollable` contains strongly uncontrollable CNCUs with 6 variables, 4 of which are observation variables, whereas `dynamic/6vars/6obs/controllable` contains dynamically (and not strongly) controllable CNCUs with 6 (observation) variables.

In this way, for each kind of controllability I have the same number of controllable and uncontrollable CNCUs specifying the same number of variables and varying the number of observation ones. These sets of benchmarks (along with the analysis that I am about to discuss) are available at http://regis.di.univr.it/EE_CNCU_LNAI2018.tar.bz2.

Regardless of the set, each CNCU has exactly 6 six variables, where each variable has the same 6 values in its domain. Each CNCU specifies a maximum number of relational constraints of 40% of $|\mathcal{V}| \times |\mathcal{OV}|$, where each (binary) relation (R_{ij}, ℓ) has a maximum number of tuples of 50% of $|D(V_i)| \times |D(V_j)|$ and the label ℓ is generated randomly. Furthermore, all variables are *unlabeled* and no partial order is specified. This contributes to generating "hard" instances for DC-CHECKING as it forces it to run on potentially all orders.

Algorithm 15 shows the pseudo code of the generator I developed to generate this set of benchmarks. I proceed by discussing the graphical data of the experi-

Algorithm 15: `CNCU-Gen`(n, o, m, c, k)

Input: An exact number of variables v, an exact number of observation variables o, an exact number of values m, a maximum number of constraints c, a maximum number of tuples k per constraint.

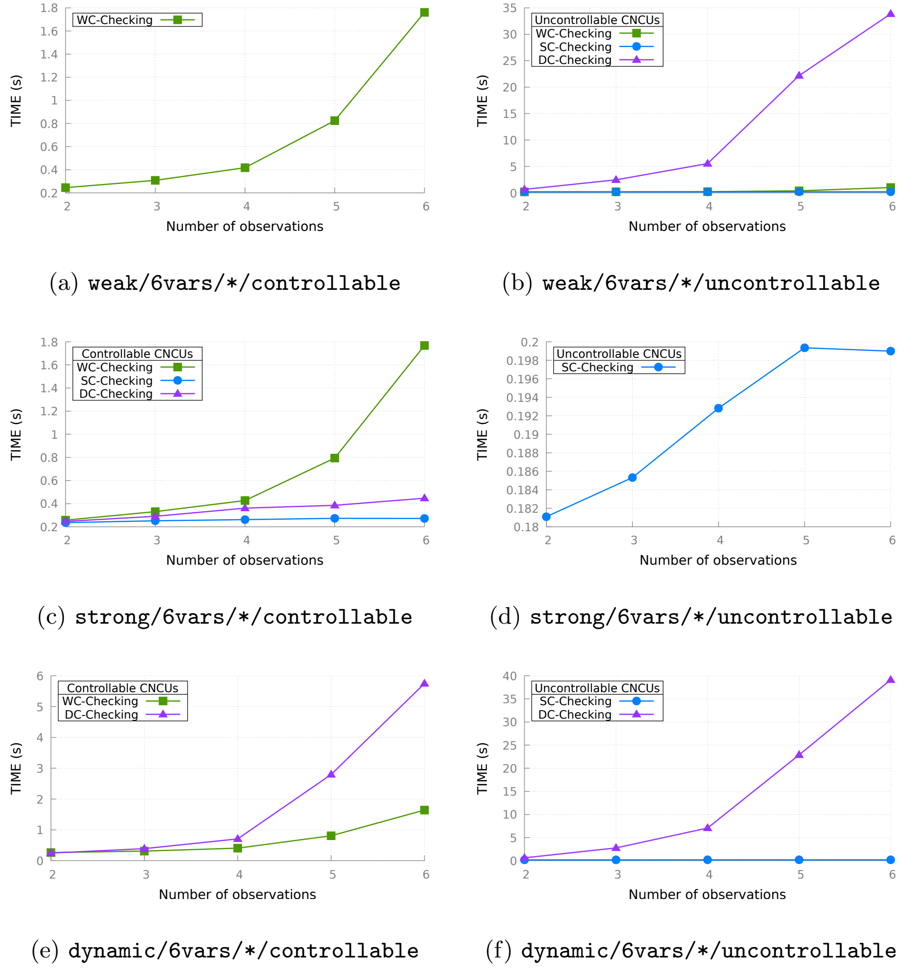
Output: A well-defined CNCU. If the output network contains observation variables then each proposition will label some component.

1. $\mathcal{Z} \leftarrow \langle \mathcal{V}, \mathcal{D}, D, \mathcal{OV}, \mathcal{P}, O, L, \prec, \mathcal{C} \rangle$ ▷ Empty CNCU
   ▷ Generate a single domain
2. $D \leftarrow \{v_1, \ldots, v_m\}$
   ▷ Generate variables and related propositions
3. $\mathcal{OV} \leftarrow \{P_i? \mid 1 \leq i \leq o\}$
4. $\mathcal{P} \leftarrow \{p_i \mid 1 \leq i \leq o\}$
5. $O(d_i) = P_i?$ for $1 \leq i \leq o$
   ▷ Generate the rest of time points
6. $\mathcal{V} \leftarrow \mathcal{OV} \cup \{V_i \mid 1 \leq i \leq (n - o)\}$
7. $L(V) = \boxdot$ for $V \in \mathcal{OV}$
   ▷ Generate constraints
8. **for** $i \leftarrow 1$ *to* c **do**
9. $V_1, V_2 \leftarrow$ two Random variables in $\mathcal{V}$
10. $R_{\{V_1, V_2\}} \leftarrow \emptyset$
11. $\ell \leftarrow \boxdot$
12. $maxLength \leftarrow Random(0, |\mathcal{P}|)$ ▷ Random extension of the label
13.   **for** $j \leftarrow 0$ *to* $maxLength$ **do**
14. $p \leftarrow Random(\mathcal{P})$ ▷ Random proposition
15.   **if** $\ell \wedge p$ *is consistent* **then**
16. $\ell \leftarrow \ell \wedge p$
17.   **for** $j = 1$ *to* k **do**
18. $t \leftarrow Random(v_1, v_2)$ ▷ For $(v_1, v_2) \in D(V_1) \times D(V_2)$
            $R_{\{V_1, V_2\}} \leftarrow R_{\{V_1, V_2\}} \cup \{t\}$
19. $\mathcal{C} \leftarrow \mathcal{C} \cup (R_{\{V_1, V_2\}}, \ell)$
   ▷ Final check
20. **if** *Some proposition never appears in any label* **then**
21.   Throw away the network
22. **return** $\mathcal{Z}$

mental evaluation in Figure 6.2 (time) and Figure 6.3 (space), where x-axes always represent the number of observation variables (i.e., the set of benchmarks under analysis) and y-axes represent either the average time elapsed or the space consumed when saving the "order and buckets" of a controllable CNCU in the specific set.

Figure 6.2a shows the time performance of WC-checking on weakly controllable CNCUs only. The results confirm that augmenting observation variables worsens the time performance of the analysis. No other comparison is possible here since this set of benchmarks contains neither strongly nor dynamically controllable CNCUs.

Figure 6.2b shows the time performance of WC-checking, SC-checking and DC-checking on weakly uncontrollable CNCUs. I recall that a weakly un-



(a) `weak/6vars/*/controllable` (b) `weak/6vars/*/uncontrollable`

(c) `strong/6vars/*/controllable` (d) `strong/6vars/*/uncontrollable`

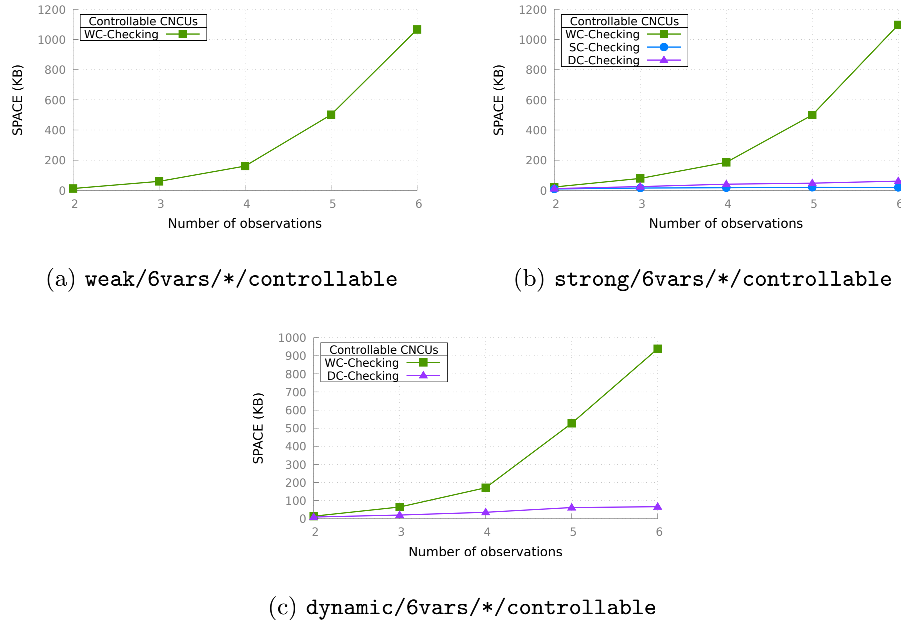
(e) `dynamic/6vars/*/controllable` (f) `dynamic/6vars/*/uncontrollable`

Fig. 6.2: Experimental evaluation with Zeta (time).

controllable CNCU cannot be strongly controllable nor dynamically controllable. Therefore, when CNCUs are uncontrollable for all three kinds of controllability, SC-Checking is faster than WC-Checking which, in turn, is faster than DC-Checking to prove uncontrollability of the CNCUs.

Figure 6.2c shows the time performance of WC-Checking, SC-Checking and DC-Checking on strongly controllable CNCUs. I recall that a strongly controllable CNCU is also weakly and dynamically controllable. Therefore, when CNCUs are controllable for all three kinds of controllability, SC-Checking is faster than DC-Checking which, in turn, is faster than WC-Checking to validate the CNCUs (note that in this case any order is fine for DC-Checking).

Figure 6.2d shows the time performance of SC-Checking on strongly uncontrollable CNCUs. Note that a strongly uncontrollable CNCU could be weakly



(a) `weak/6vars/*/controllable` (b) `strong/6vars/*/controllable`

(c) `dynamic/6vars/*/controllable`

Fig. 6.3: Experimental evaluation with Zeta (space).

and/or dynamically controllable (that's why I excluded WC-checking and DC-checking from this comparison). The graph shows little difference on the average times, which is because of the conditional part that is wiped out.

Figure 6.2e shows the time performance of WC-checking and DC-checking on dynamically controllable CNCUs. I recall that a dynamically controllable CNCU is also weakly but in general can be *not* strongly controllable (in the sets I generated these networks so that they were not SC). Therefore, when CNCUs are dynamically but not strongly controllable, WC-checking is faster than DC-checking to validate them. This is because WC does not need to brute-force all the orders.

Figure 6.2f shows the time performance of SC-checking and DC-checking on dynamically uncontrollable CNCUs. I recall that a dynamically uncontrollable CNCU is also strongly uncontrollable (but could be weakly controllable, which is why WC-checking was excluded from this comparison). Therefore when CNCUs are dynamically uncontrollable, SC-checking is faster than DC-checking to prove uncontrollability. This depends on the fact that the conditional part is wiped out but also that SC-checking does not brute-force all the orders.

Figure 6.3a shows the space consumption of WC-checking for weakly controllable CNCUs, confirming that augmenting the number of observations will augment the size of the saved strategy. I recall that all variables in any CNCU are unlabeled, therefore K observation variables imply 2^K saved solutions.

Figure 6.3b shows the space consumption of WC-checking, SC-checking and DC-checking for strongly controllable CNCUs (which are also weakly and

dynamically controllable). SC-CHECKING saves strategies that are smaller than those saved by DC-CHECKING, which, in turn, are smaller than those saved by WC-CHECKING. This graph reveals that in a strategy, unfolding all scenarios (WC-CHECKING) is worse than keeping them compact (DC-CHECKING). This is also confirmed by Figure 6.3c, which shows the difference in space consumption between WC-CHECKING and DC-CHECKING for dynamically controllable CNCUs (which are also weakly but not strongly controllable).

Finally, I executed all controllable CNCUs 1000 times. No one crashed.

6.6 Modeling access controlled workflows under uncertainty

In this section, I define a process modeling language to specify access controlled workflows under conditional uncertainty (simply shortened as ACWFs), then define weak, strong and dynamic controllability of ACWFs and finally provide an encoding from ACWFs to CNCUs.

6.6.1 Motivating Example

As a running, motivating example coming from the financial domain, I consider (an excerpt of) a simplified loan origination process that I show in Figure 6.4. The workflows has 7 tasks, 5 roles (`Clerk`, `Auditor`, `AMLOfficer`, `IRSOfficer` and `Manager`) and 6 users (`alice`, `bob`, `evie`, `kate`, `mike` and `ted`). `Clerk` contains `alice` and `bob`, `Auditor` contains `bob` and `kate`, `AMLOfficer` contains `mike` only, `IRSOfficer` contains `evie` only, `Manager` contains `kate` and `ted`.

A `Clerk` starts the workflow by processing a loan request (`ProcR`). After that, the flow of execution splits by entering an unconditional parallel block (leftmost diamond). In this block an `Auditor` checks the financial records of the customer (`CheckFR`) and at the same time another further verification takes place depending on if the amount of money requested is huge or not (diamond labeled by `hugeA?`). If `hugeA?` $= \top$, then an `AMLOfficer` carries out an anti money laundering assessment (`AntiML`). If `hugeA?` $= \bot$, an `IRSOfficer` carries out a simple tax fraud assessment. The `Auditor` who executes `CheckFR` must be different from the `Clerk` who executed `ProcR`, (as some users, e.g., `bob`, might belong to both roles) and must also not be a relative of the `AMLOfficer` who executed `AntiML` nor of the `IRSOfficer` who executed `TaxFA` (different is not necessary since the bank requires `AMLOfficer`s and `IRSOfficer`s to be external disjoint consultants). After that both the internal conditional block and the external parallel one (in this order) complete and the flow of execution enters a new conditional block to carry out the final tasks depending on if the loan has been approved or not (diamond labeled by `app?`). If `app?` $= \top$, then a `Manager` prepares the contract (`PrepC`) and another one, who must be different from the first, signs it (`Sign`). If `app?` $= \bot$, then the same `Clerk` who executed the initial `ProcR` rejects the request (`Reject`). Regardless of the chosen branch, the workflow completes by executing the rightmost split connector.

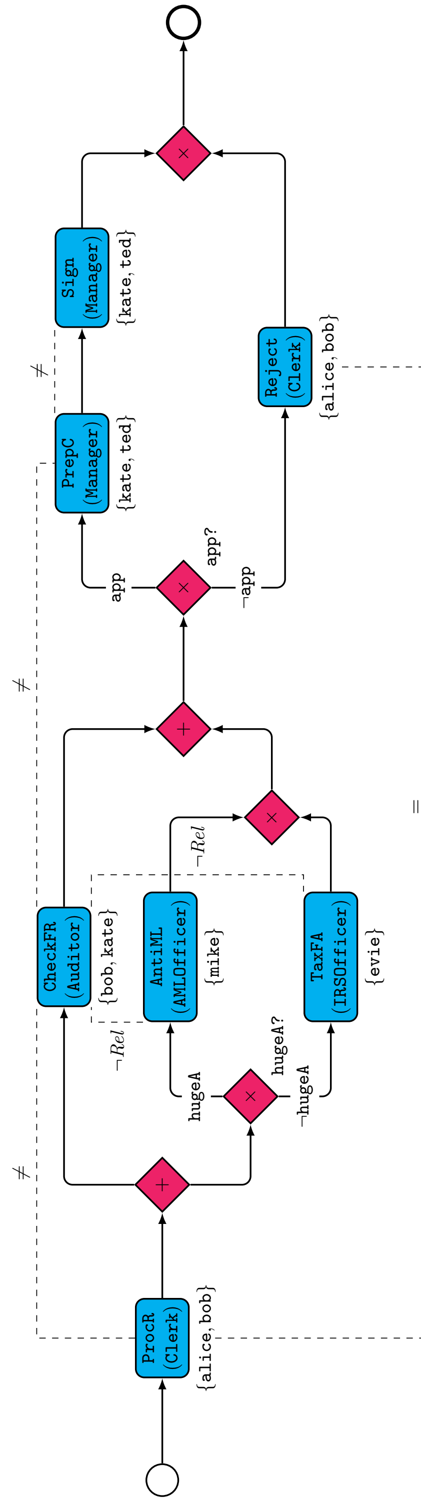
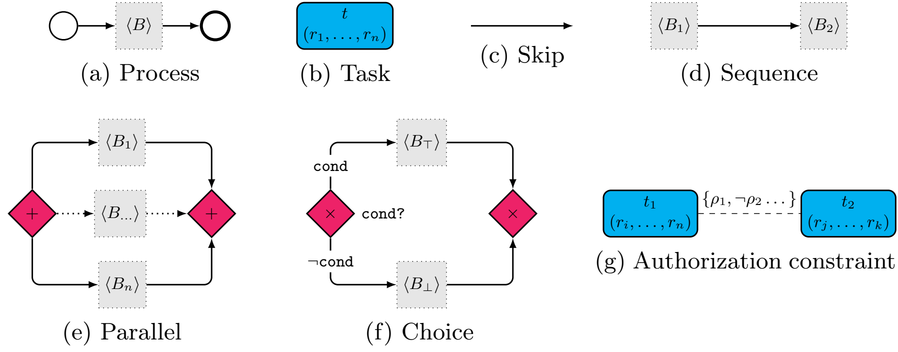


Fig. 6.4: Example of an access controlled workflow in BPMN for a loan origination process. bob and mike are brother. kate and evie are sisters.



(a) Process (b) Task (c) Skip (d) Sequence (e) Parallel (f) Choice (g) Authorization constraint

Fig. 6.5: A fragment of a structured (access controlled) BPMN.

6.6.2 Process Modeling Language

I consider again a structured approach similar to the fragment of BPMN I discussed in Section 5.7.2 but with an authorization part instead of a temporal part. I inject a role-based access-control model (RBAC, [113]) and formalize authorization constraints at user level into the process specification. Figure 6.5 shows the process modeling language I propose to model ACWFs under conditional uncertainty.

Differently from CSTNUDs, since CNCUs do not model decisions (yet), I assume that each conditional split connector is associated to a unique proposition whose truth value assignment is *uncontrollable*.

Instead, as for the structured fragment given in Figure 5.8, I assume that all tasks of the workflow are implicitly labeled by labels according to the nesting levels of the choice blocks (this is implicit since the workflow is structured).

The RBAC part of a business process consists of roles, users and tasks as the set of permissions. Roles are associated to both users and tasks, acting as an interface between them. The *UA* relation remains the same, whereas the *TA* (task-assignment) relation replaces *PA* in Definition 2.30. I write

- $\mathit{users}(r) = \{u \mid (u, r) \in \mathit{UA}\}$ for the set of users belonging to a role r
- $\mathit{roles}(t) = \{r \mid (t, r) \in \mathit{TA}\}$ for the set of roles authorized for a task t.

I abuse notation and write $\mathit{users}(t) = \{u \mid (u, t) \in \mathit{UA} \wedge r \in \mathit{roles}(t)\}$ for the set of users authorized for a task.

I label a task t by a finite set of roles $\{r_1, \ldots, r_e\} \subseteq \texttt{Roles}$ meaning that $(t, r_1), \ldots, (t, r_e) \in \mathit{TA}$ (Figure 6.5b). Assigning roles to tasks models "who does what".

I express *authorization constraints* as undirected dashed edges between pairs of tasks t_1, t_2 (i.e., tasks that do not belong to mutually exclusive workflow paths) as a finite set of binary relations $\{\rho_1, \neg\rho_2 \ldots\}$ over users ($\rho_i \subseteq \texttt{Users} \times \texttt{Users}$) where each relation may appear positive (ρ) or negative ($\neg\rho$) and such that if $u_1 \in \mathit{users}(t_1)$ and $u_2 \in \mathit{users}(t_2)$ and the pair (u_1, u_2) also satisfies all $(\neg)\rho_i$ in the set, then any execution assigning t_1 to u_1 and t_2 to u_2 *satisfies* the authorization constraint (Figure 6.5g).

Like temporal relative constraints discussed in Figure 5.8, adding authorization constraints does not breach the structuredness of the workflow either. The example in Figure 6.4 introduced in Section 6.6.1 is a structured ACWF under conditional uncertainty.

6.6.3 Controllability of ACWFs under conditional uncertainty

In this section, I define *weak*, *strong* and *dynamic controllability* of ACWFs under (conditional) uncertainty. My goal is to synthesize execution strategies identifying which users to commit for which tasks (and in which order) so that in the arising workflow path the execution satisfies both the partial order and all authorization constraints between tasks.

Let `Components` be the set of components of the WF containing tasks (t_i), connectors (c_i) and two special symbols P and E representing the start and end of the process. `Users` is the set of users. Let $\mathcal{B} = \{\texttt{cond?}, \dots\}$ be the set of Boolean variables associated to the conditional split connectors.

Definition 6.13. *A* scenario $s\colon \mathcal{B} \to \{\bot, \top\}$ *is a complete truth value assignment to the Boolean variables in* $\mathcal{B}$*.* Σ *models the space of all scenarios.*

A *WF path* is the projection of a WF onto a scenario. That is, a new unconditional WF in which all components incoherent with the truth value assignment are removed. For example, Figure 6.4 specifies 4 WF paths (I omitted the connectors to ease reading):

1. `ProcR` $\to$ `CheckFR` $\to$ `AntiML` $\to$ `PrepC` $\to$ `Sign` (if `hugeA?` and `app?`)
2. `ProcR` $\to$ `CheckFR` $\to$ `TaxFA` $\to$ `PrepC` $\to$ `Sign` (if $\neg$`hugeA?` and `app?`)
3. `ProcR` $\to$ `CheckFR` $\to$ `AntiML` $\to$ `Reject` (if `hugeA?` and $\neg$`app?`)
4. `ProcR` $\to$ `CheckFR` $\to$ `TaxFA` $\to$ `Reject` (if $\neg$`hugeA?` and $\neg$`app?`)

Definition 6.14. *A* plan *is a mapping* $\pi\colon \texttt{Components} \to \texttt{Users} \times \mathbb{N}$ *from components to users and (positive) integers saying which users execute which components and in which order. If* $\pi(X) = (u_1, i_1)$ *and* $\pi(Y) = (u_2, i_2)$ *and* $i_1 < i_2$*, then* u_1 *executes* X before u_2 *executes* Y*. A plan is consistent if the assignments it makes satisfy all authorization constraints involving tasks in its domain and the partial order.* Π *represents the set of all plans.*

However, a plan is nothing but a fixed schedule for executing a bunch of components. The interesting part is how we generate it. To do so, we need a *strategy*.

Definition 6.15. *An* execution strategy *is a mapping* $\sigma\colon \Sigma \to \Pi$ *from scenarios to plans whose domains consist of the components coherent with the scenario. An execution strategy is viable if for* $s \in \Sigma$*, there exists a consistent plan* $\sigma(s)$*.*

To avoid heaving the notation, I write $[\sigma(s)]_X$ (instead of $\sigma(s)(X)$) to denote the pair (u, i) assigned to the component X in the plan $\pi = \sigma(s)$. I am now ready to provide the three main kinds of controllability.

Definition 6.16. *An ACWF under conditional uncertainty is* weakly controllable *if there exists a viable strategy (every WF path is consistent).*

Figure 6.4 is weakly controllable. To prove that I give an assignment of users to tasks for each WF path previously discussed.

1. $\texttt{ProcR} = \texttt{bob}, \texttt{AntiML} = \texttt{mike}, \texttt{CheckFR} = \texttt{kate}, \texttt{PrepC} = \texttt{ted}, \texttt{Sign} = \texttt{kate}$
2. $\texttt{ProcR} = \texttt{alice}, \texttt{TaxFA} = \texttt{evie}, \texttt{CheckFR} = \texttt{bob}, \texttt{PrepC} = \texttt{kate}, \texttt{Sign} = \texttt{ted}$
3. $\texttt{ProcR} = \texttt{alice}, \texttt{AntiML} = \texttt{mike}, \texttt{CheckFR} = \texttt{kate}, \texttt{Reject} = \texttt{alice}$
4. $\texttt{ProcR} = \texttt{alice}, \texttt{TaxFA} = \texttt{evie}, \texttt{CheckFR} = \texttt{bob}, \texttt{Reject} = \texttt{alice}$

However, weak controllability requires one to predict how the entire uncontrollable part will behave (i.e., it requires to predict the future). This leads me to discuss the opposite kind of controllability, where a single assignment makes consistent all WF paths no matter which one we take during execution.

Definition 6.17. *An ACWF under conditional uncertainty is* strongly controllable *if there exists a viable strategy working for all scenarios.*

Figure 6.4 is not strongly controllable. The problem lies in the authorization constraints specified between `CheckFR` and `AntiML` and between `CheckFR` and `TaxFA`. Suppose that before starting we decide we will assign `bob` to `CheckFR`. If during execution we observe $\texttt{hugeA?} = \top$, thus the WF goes for `AntiML` with no user available for it since `bob` and `mike` are brothers. Then, we could change our mind and decide to assign `kate` (instead of `bob`) to `CheckFR` so that `mike` for `AntiML` would be fine as `kate` and `mike` are not relatives. But if during execution $\texttt{hugeA?} = \bot$, `evie` would not be OK for `TaxFA` as she is `kate`'s sister. Therefore, there is no way to preassign a user to `CheckFR`.

Yet, the WF is still executable as long as we decide (during execution) which user to assign to `CheckFR` *after* observing which truth value `hugeA?` has been assigned. This leads me to consider the most appealing type of controllability: *dynamic controllability*, a history-based controllability.

Definition 6.18. *The* history $\mathcal{H}(t, s, \sigma)$ *of a task t in the scenario s for the strategy σ is the set of truth value assignments observed before t upon the execution of the corresponding conditional split connectors c having associated uncontrollable Boolean variables* `cond?` *in the strategy $\sigma(s)$. Formally,*

$$\mathcal{H}(t, s, \sigma) = \{(\texttt{cond?}, s(\texttt{cond?})) \mid [\sigma(s)]_c = (u_c, i_c) \wedge [\sigma(s)]_t = (u_t, i_t) \wedge i_c < i_t\}$$

where `cond?` *is the Boolean variable associated to the conditional split connector c and $u_c, u_t \in \texttt{Users}$ are the users executing c and t, respectively.*

For example, consider `Reject` in Figure 6.4, a scenario s such that $s(\texttt{hugeA?}) = \bot$ and $s(\texttt{app?}) = \top$ and any dynamic strategy σ for s (see below). Before the conditional split connector labeled by `hugeA?` executes, $\mathcal{H}(\texttt{Reject}, s, \sigma) = \emptyset$ and $\mathcal{H}(\texttt{Reject}, s, \sigma) = \{\neg\texttt{hugeA}\}$ after. Likewise, before the conditional split connector labeled by `app?` executes we have that $\mathcal{H}(\texttt{Reject}, s, \sigma)$ is still equal to $\{\neg\texttt{hugeA}\}$ and $\mathcal{H}(\texttt{Reject}, s, \sigma) = \{\neg\texttt{hugeA}, \texttt{app}\}$ after.

Definition 6.19. *An execution strategy σ is* dynamic *if for any pair of scenarios $s_1, s_2 \in \Sigma$ and any task t, whenever the scenario history looks the same, then the strategy assigns the same users and integers to the same components of the WF. That is, for all $s_1, s_2 \in \Sigma$ and any $t \in \texttt{Tasks}$, if $\mathcal{H}(t, s_1, \sigma) = \mathcal{H}(t, s_2, \sigma)$, then $[\sigma(s_1)]_X = [\sigma(s_2)]_X$ for any $X \in \texttt{Components}$.*

Definition 6.20. *An ACWF under conditional uncertainty is* dynamically controllable *if there exists a dynamic execution strategy.*

The ACWF in Figure 6.4 is dynamically controllable. An example of dynamic execution strategy is the following. `ProcR` = `alice` (always). Then, the WfMS executes the leftmost total connector and conditional split connector labeled by `hugeA?`. If `hugeA?` = $\top$, then `CheckFR` = `kate` and `AntiML` = `mike`. After that, the WfMS executes the corresponding join connectors and also the second split connector labeled by `app?`. If `app?` = $\top$, then `PrepC` = `ted` and `Sign` = `kate`, whereas if `app?` = $\bot$, then `Reject` = `alice`. The process concludes with the WfMS executing the last join connector. Instead, beside the execution of the connectors (which is the same), if `hugeA?` = $\bot$, then `CheckFR` = `bob` and `TaxFA` = `evie`. After that, if `app?` = $\top$, then `PrepC` = `kate` and `Sign` = `ted`, whereas if `app?` = $\bot$, then `Reject` = `alice`. The important thing is to execute the conditional split connector labeled by `hugeA?` *before* executing `CheckFR` (as being in a parallel block, `CheckFR` could be executed before the connector). ACWFs like our example in Figure 6.4 are witnesses of the ordering problem.

Hence, it should not be difficult to see that weak, strong and dynamic controllability of ACWFs boil down to those of CNCUs when tasks are modeled as variables, conditional split connectors as observation variables, the partial order as order edges and authorization constraints as constraints edges.

6.6.4 Encoding ACWF into CNCUs

As I did for temporal workflows and CSTNUDs I provide here an encoding from ACWF to CNCUs. Table 6.2 shows such an encoding.

All arcs regulating the control flow are encoded as order edges in the CNCU.

The start and end of a process are encoded as two variables S and E occurring before and after all other variables, respectively. $L(S) = L(E) = \boxdot$ since the start and the end of a process always occur. `wf` is the unique authorized user for these variables. No constraint edge involves S and E (Table 6.2, row 1).

A task t having authorized roles $r_1, \ldots, r_n$ is encoded as a homonymous variable whose domain consists of the union of all users belonging to $r_1, \ldots, r_n$ authorized for t; i.e., $users(t)$, whereas its label contains the propositions modeling the Boolean variables associated to the conditional split connectors according to the nesting level of the block in which the task appears (Table 6.2, row 2).

Skip and sequence blocks are encoded as order edges connecting the last variable of the left block to the first one of the right (Table 6.2, rows 3,4).

Parallel and conditional blocks are straightforwardly encoded mirroring the partial order of the ACWFs in the CNCU. If the block is a parallel, a variable P_S models the total split connector, whereas a variable P_E models the join connector. $L(P_S) = L(P_E)$ according to the nesting level of the block in the ACWF. All labels of the variables modeling $\langle B_1 \rangle$, $\langle B_{\ldots} \rangle$ and $\langle B_n \rangle$ in the ACWF (if any) must entail $L(P_S)$. If a block is a choice then an observation variable $P?$ having associated proposition p models the conditional split connector. P_E still models the join connector. Again, $L(P?) = L(P_E)$ but this time as well as entailing $L(P?)$, all labels of the variables modeling $\langle B_\top \rangle$ and $\langle B_\bot \rangle$ in the ACWF (if any) are augmented

Table 6.2: Encoding access-controlled workflows into CNCUs.

Workflow Block	CNCU Fragment
○ → $\langle B \rangle$ → ○	$[\boxdot]$ S $\{\texttt{wf}\}$ → ··· → $[\boxdot]$ E $\{\texttt{wf}\}$
t $(r_1, \ldots, r_n)$	$[\ell]$ T $\{users(t)\}$
→	→
$\langle B_1 \rangle$ → $\langle B_2 \rangle$	··· → ···
+ → $\langle B_1 \rangle$, $\langle B_{\ldots} \rangle$, $\langle B_n \rangle$ → +	$[\ell]$ P_S $\{\texttt{wf}\}$ → $[\ell \wedge \ldots]$ ···, $[\ell \wedge \ldots]$ ···, $[\ell \wedge \ldots]$ ··· → $[\ell]$ P_E $\{\texttt{wf}\}$
× cond? → cond $\langle B_\top \rangle$, ¬cond $\langle B_\bot \rangle$ → ×	$[\ell]$ $P?$ $\{\texttt{wf}\}$ → $[\ell \wedge p \wedge \ldots]$ $\ldots\top$, $[\ell \wedge \neg p \wedge \ldots]$ $\ldots\bot$ → $[\ell]$ P_E $\{\texttt{wf}\}$
t_i $(r_m, \ldots, r_n)$ - - $\{\rho_1, \neg\rho_2, \ldots\}$ - - t_j $(r_z, \ldots, r_k)$	$[\ell_i]$ T_i $\{users(t_i)\}$ — $(\rho_1 \cap \neg\rho_2 \ldots, \ell_i \wedge \ell_j)$ — $[\ell_j]$ T_j $\{users(t_j)\}$

Table 6.3: Relations of the example in Figure 6.6.

(a) $(R_{S_1}, \boxdot)$

ProcR	CheckFR
alice	bob
alice	kate
bob	kate

(b) (R_{S_2}, h)

CheckFR	AntiML
kate	mike

(c) $(R_{S_3}, \neg h)$

CheckFR	TaxFA
bob	evie

(d) (R_{S_4}, a)

CheckFR	PrepC
bob	kate
bob	ted
kate	ted

(e) (R_{S_5}, a)

PrepC	Sign
kate	ted
ted	kate

(f) (R_{S_6}, a)

ProcR	Reject
alice	alice
bob	bob

with p or $\neg p$, respectively (Table 6.2, row 6). All variables modeling connectors are all executed by `wf`.

An authorization constraint between two tasks is encoded as a constraint edge whose relation is the intersection of all relations appearing on the authorization constraint in the WF block and the label is the conjunction of the labels of the variables modeling tasks (Table 6.2, row 7).

Figure 6.6 shows the CNCU encoding the ACWF in Figure 6.4. S and E encode the start and end of the process. P_S and P_E encode the total split and join connectors of the unconditional parallel block. $H?$ and H_E encode the conditional split and join connectors of the leftmost conditional block (h models `hugeA?`). $A?$ and A_E encode the conditional split and join connectors of the rightmost

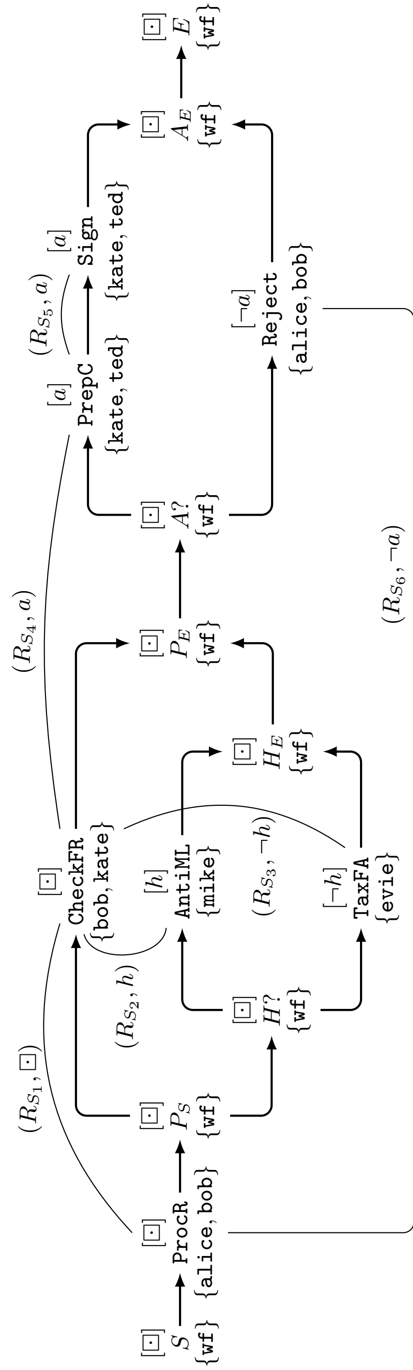


Fig. 6.6: CNCU encoding the ACWF in Figure 6.4.

conditional block (a models `app?`). `ProcR`, `CheckFR`, `AntiML`, `TaxFA`, `PrepC`, `Sign` and `Reject` model the homonymous tasks.

Table 6.3 shows the labeled relational constraints of Figure 6.6 translating the authorization constraints of the workflow in Figure 6.4. ($S_1 = \{\texttt{ProcR}, \texttt{CheckFR}\}$, $S_2 = \{\texttt{CheckFR}, \texttt{AntiML}\}$, $S_3 = \{\texttt{CheckFR}, \texttt{TaxFA}\}$, $S_4 = \{\texttt{CheckFR}, \texttt{PrepC}\}$, $S_5 = \{\texttt{PrepC}, \texttt{Sign}\}$ and $S_6 = \{\texttt{ProcR}, \texttt{Reject}\}$ shorten the scopes). Any relation has the form $R_{t_i t_j}$ where t_i and t_j are variables modeling tasks. A tuple $(u_i, u_j) \in R_{t_i t_j}$ means that $t_i = u_i$ and $t_j = u_j$ satisfies the authorization constraint between t_i and t_j. For example, $(\texttt{alice}, \texttt{bob}) \in (R_1, \boxdot)$ (Table 6.3a) means that any execution assigning `alice` to `ProcR` and `bob` to `CheckFR` will satisfy the authorization constraint between them.

I have discussed a process modeling language for ACWFs under conditional uncertainty. Then, I have provided an encoding from ACWFs into CNCUs. Now, I check weak, strong and dynamic controllability of the case study I am discussing by feeding Zeta with the CNCU in Figure 6.6 whose specification is given in Listing 6.6. The implementation of this case study is available at http://regis.di.univr.it/LOP.tar.bz2.

Listing 6.6: Specification of Figure 6.6.

```
1  Domains {
2      (System : wf)
3      (ProcRDom : alice bob)
4      (CheckFRDom : bob kate)
5      (AntiMLDom : mike)
6      (TaxFADom : evie)
7      (PrepCDom : kate ted)
8      (SignDom : kate ted)
9      (RejectDom : alice bob)
10 }
11
12 Propositions {
13     h a
14 }
15
16 Variables {
17     (S : : System : )
18     (ProcR : : ProcRDom : )
19     (PS : : System : )
20     (CheckFR : : CheckFRDom : )
21     (H : h : System : )
22     (AntiML : : AntiMLDom : h)
23     (TaxFA : : TaxFADom : !h)
24     (HE : : System : )
25     (PE : : System : )
26     (A : a : System : )
27     (PrepC : : PrepCDom : a)
28     (Sign : : SignDom : a)
29     (Reject : : RejectDom : !a)
30     (AE : : System : )
31     (E : : System : )
```

```
32 }
33
34 Precedence {
35     (S < ProcR)
36     (ProcR < PS)
37     (PS < CheckFR)
38     (PS < H)
39     (CheckFR < PE)
40     (H < AntiML)
41     (H < TaxFA)
42     (AntiML < HE)
43     (TaxFA < HE)
44     (HE < PE)
45     (PE < A)
46     (A < PrepC)
47     (A < Reject)
48     (PrepC < Sign)
49     (Sign < AE)
50     (Reject < AE)
51     (AE < E)
52 }
53
54 Constraints {
55     (ProcR CheckFR : (alice bob) (alice kate) (bob kate) : )
56     (CheckFR PrepC : (bob kate) (bob ted) (kate ted) : a)
57     (PrepC Sign : (kate ted) (ted kate) : a)
58     (CheckFR AntiML : (kate mike) : h)
59     (CheckFR TaxFA : (bob evie) : !h)
60     (ProcR Reject : (alice alice) (bob bob) : !a)
61 }
```

The workflow in Figure 6.4 is weakly, dynamically but not strongly controllable mainly because the user committed to `CheckFR` cannot be preassigned before having full information on the truth value assignment of `hugeA?`.

For weak controllability there are 4 possible assignments to the variables according to the 4 possible scenarios (Listing 6.7).

Listing 6.7: Random executions for Figure 6.6 (weak controllability)

```
1  $ java -jar zeta.jar LoanOriginationProcess.cncu --execute
       LoanOriginationProcess.weak.ob
2  ====================================
3  Scenario h a
4  Order: S -> ProcR -> PS -> H -> AntiML -> HE -> CheckFR -> PE -> A -> PrepC
        -> Sign -> AE -> E
5  ------------------------------------
6  S = wf
7  ProcR = bob
8  PS = wf
9  H = wf
10 AntiML = mike
11 HE = wf
```

```
12 CheckFR = kate
13 PE = wf
14 A = wf
15 PrepC = ted
16 Sign = kate
17 AE = wf
18 E = wf
19 -----------------------------------
20 Verifying ... SAT!
21 ===================================
22 ===================================
23 Scenario !h a
24 Order: S -> ProcR -> PS -> H -> TaxFA -> HE -> CheckFR -> PE -> A -> PrepC
       -> Sign -> AE -> E
25 -----------------------------------
26 S = wf
27 ProcR = alice
28 PS = wf
29 H = wf
30 TaxFA = evie
31 HE = wf
32 CheckFR = bob
33 PE = wf
34 A = wf
35 PrepC = kate
36 Sign = ted
37 AE = wf
38 E = wf
39 -----------------------------------
40 Verifying ... SAT!
41 ===================================
42 ===================================
43 Scenario h !a
44 Order: S -> ProcR -> PS -> H -> AntiML -> HE -> CheckFR -> PE -> A ->
       Reject -> AE -> E
45 -----------------------------------
46 S = wf
47 ProcR = alice
48 PS = wf
49 H = wf
50 AntiML = mike
51 HE = wf
52 CheckFR = kate
53 PE = wf
54 A = wf
55 Reject = alice
56 AE = wf
57 E = wf
58 -----------------------------------
59 Verifying ... SAT!
60 ===================================
```

```
61 ====================================
62 Scenario !h !a
63 Order: S -> ProcR -> PS -> H -> TaxFA -> HE -> CheckFR -> PE -> A -> Reject
        -> AE -> E
64 ------------------------------------
65 S = wf
66 ProcR = alice
67 PS = wf
68 H = wf
69 TaxFA = evie
70 HE = wf
71 CheckFR = bob
72 PE = wf
73 A = wf
74 Reject = alice
75 AE = wf
76 E = wf
77 ------------------------------------
78 Verifying ... SAT!
79 ====================================
```

Listing 6.8 shows a few dynamic executions in which I have isolated the scenarios of interest to prove that the user committed to `CheckFR` is different depending on the uncontrollable truth value assignment of `hugeA?`.

Listing 6.8: Random executions for Figure 6.6 (dynamic controllability)

```
1  $ java -jar zeta.jar LoanOriginationProcess.cncu --execute
       LoanOriginationProcess.dynamic.ob 1000
2  ...
3  ====================================
4  Order: S -> ProcR -> PS -> H -> CheckFR -> AntiML -> TaxFA -> HE -> PE -> A
        -> PrepC -> Sign -> Reject -> AE -> E
5  ------------------------------------
6  S = wf
7  ProcR = alice
8  PS = wf
9  H = wf, h = true
10 CheckFR = kate
11 AntiML = mike
12 HE = wf
13 PE = wf
14 A = wf, a = false
15 Reject = alice
16 AE = wf
17 E = wf
18 ------------------------------------
19 Verifying ... SAT!
20 ====================================
21 ====================================
22 Order: S -> ProcR -> PS -> H -> CheckFR -> AntiML -> TaxFA -> HE -> PE -> A
        -> PrepC -> Sign -> Reject -> AE -> E
```

```
23 -----------------------------------
24 S = wf
25 ProcR = alice
26 PS = wf
27 H = wf, h = false
28 CheckFR = bob
29 TaxFA = evie
30 HE = wf
31 PE = wf
32 A = wf, a = true
33 PrepC = kate
34 Sign = ted
35 AE = wf
36 E = wf
37 -----------------------------------
38 Verifying ... SAT!
39 ===================================
40 ...
```

In all executions `alice` executes `ProcR` (lines 7,25). If `bob` was committed to `ProcR`, `kate` would be only valid user to execute `CheckFR`, but then if `hugeA?` was assigned $\perp$ there would be no valid user for `TaxFA`. `alice` is fine because allows both `bob` and `kate` to be committed to `CheckFR`. All variables representing components not modeling tasks are executed by `wf`. Any execution waits to have full information on `hugeA?` before deciding which user to commit to `CheckFR` (note that in the order of variables $H?$ is executed before `CheckFR`).

In the first execution, the amount of money is huge (line 9) so we know that will go for `AntiML` where `mike` is the only user available, thus `kate` executes `CheckFR` (line 10). This execution does not approve the loan request (line 14), so `alice` executes `Reject` (line 15).

In the second execution, the amount of money is not huge (line 27) so we know that will go for just `TaxFA` where `evie` is the only user available, thus `bob` executes `CheckFR` (line 28). This execution approves the loan request (line 32), so `kate` executes `PrepC` (line 33) and `ted` executes `Sign` (line 34).

6.7 Conclusions

I defined *constraint networks under conditional uncertainty* (CNCUs) to address a kind of CSP under conditional uncertainty. I defined weak, strong and dynamic controllability of a CNCU and provided algorithms to check each type of controllability. I developed ZETA, a tool for CNCUs for an experimental evaluation. I also discussed an algorithm to generate random CNCUs. I showed how CNCUs can be employed for the modeling, validation and execution of access controlled workflows under conditional uncertainty.

CNCUs differ from classic CNs [52] and DCSPs [100] as these formalisms do not employ any uncontrollable part. CNCUs differ from Mixed CSPs [63] as CNCUs do not have the restriction of working under full observability. CNCUs differ from probabilistic approaches such as [62] as CNCUs have exact controllability

algorithms. CNCUs differ from all formalisms of temporal networks mentioned before as they do not deal with temporal constraints.

7

Workflow Resiliency

In this chapter I address workflow resiliency; i.e., a controllability analysis with respect to the uncertain availability of the employed resources, here users.

I start by modeling the games for static, decremental and dynamic resiliency defined by Wang and Li in [122, 123] as timed game automata extended with variables and use the controller synthesis technique discussed in Section 2.2.1 to first answer the problem of deciding if any access controlled workflow is resilient or breakable and then to also synthesize a memoryless execution strategy to execute a resilient access controlled workflow. The encodings I provide do not employ clocks as in classic workflow resiliency temporal constraints are not investigated. Since UPPAAL-TIGA extends TGAs with variables and user defined functions, these encodings can be realized in UPPAAL-TIGA. Like the formalization given in Chapter 5, throughout this chapter `Player1` models a controller and `Player2` models the environment.

7.1 Motivating Example

As a motivating example, I consider an adaptation of a standard workflow/business-process describing a *loan origination process* (*LOP*) for eligible customers whose financial records have already been approved. I tuned the example in order to focus on a few characteristics of interest. This example is different from that I gave in Figure 6.1.

The workflow shown in Figure 7.1 follows a structured approach, specifying 4 tasks, 4 roles, 7 users and 4 authorization constraints. The LOP starts with a pre-processing clerk (`PreClerk`), who processes a request (`ProcR`). After that, the flow of execution splits unconditionally (leftmost diamond labeled by +) and enters a parallel block, where an auditor (`Auditor`) logs the just processed request for future accountability purposes (`Log`) and a post-processing clerk (`PostClerk`) prepares the contract (`PrepC`). The order of these two tasks does not matter. Finally, the flow of execution joins and exits the parallel block (rightmost diamond labeled by +) with a manager (`Manager`) who signs the contract (`Sign`). In this workflow, `PreClerk` contains Alice, Bob and Charlie, `Auditor` contains David and Emma, `PostClerk` contains Bob, David and Emma, and `Manager` contains

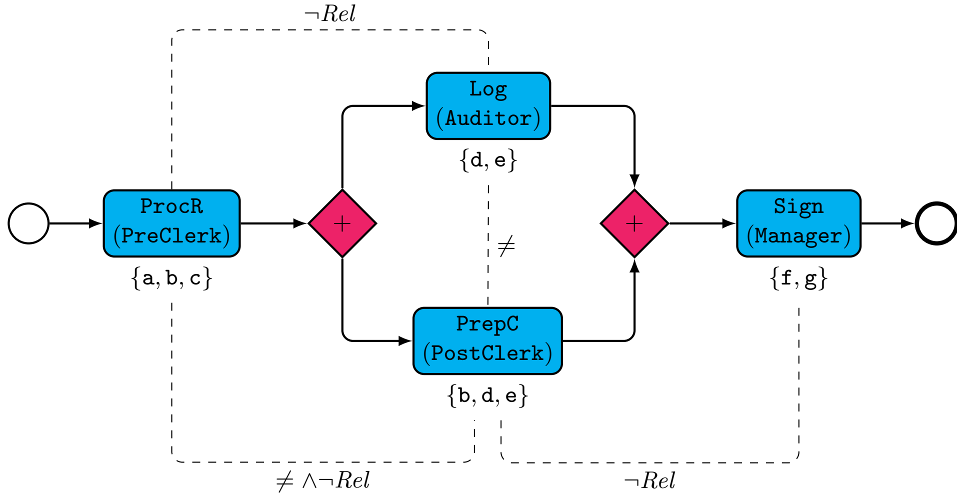


Fig. 7.1: A simplification of a loan origination process.

(a) $R_{\{\texttt{ProcR},\texttt{Log}\}}$

ProcR	Log
a	d
a	e
b	e
c	d
c	e

(b) $R_{\{\texttt{ProcR},\texttt{PrepC}\}}$

ProcR	PrepC
a	b
a	d
a	e
b	e
c	b
c	d
c	e

(c) $R_{\{\texttt{Log},\texttt{PrepC}\}}$

Log	PrepC
d	b
d	e
e	b
e	d

(d) $R_{\{\texttt{PrepC},\texttt{Sign}\}}$

PrepC	Sign
b	f
b	g
d	f
d	g
e	g

Fig. 7.2: Relational constraints of the ACWF in Figure 7.1.

Frank and Grace. In Figure 7.1 directed edges model the partial order among tasks, whereas undirected edges model security policies. Following the modeling language introduced in Figure 6.5 roles are shown inside tasks. For convenience I also show corresponding users below. I shorten Alice, Bob, Charlie, David, Emma, Frank and Grace as `a`, `b`, `c`, `d`, `e`, `f`, `g`. Bob and David are brothers, whereas Emma is Frank's daughter. Note that some user may belong to more than one role.

The access controlled workflow enforces four security policies:

1. the users who execute `ProcR` and `Log` must not be relatives
2. a separation of duties between `ProcR` and `PrepC` must hold and the users who execute them must not be relatives
3. a separation of duties between `Log` and `PrepC` must hold
4. the users executing `PrepC` and `Sign` must not be relatives

The formal specification of the ACWF in Figure 7.1 is

- $\texttt{Tasks} = \{\texttt{ProcR}, \texttt{Log}, \texttt{PrepC}, \texttt{Sign}\}$

- $\texttt{Users} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}\}$
- $UA = \{(\texttt{a}, \texttt{ProcR}), (\texttt{b}, \texttt{ProcR}), (\texttt{c}, \texttt{ProcR}), (\texttt{d}, \texttt{Log}), (\texttt{e}, \texttt{Log}), (\texttt{b}, \texttt{PrepC}), (\texttt{d}, \texttt{PrepC}), (\texttt{e}, \texttt{PrepC}), (\texttt{f}, \texttt{Sign}), (\texttt{g}, \texttt{Sign})\}$
- $\prec = \{(\texttt{ProcR}, \texttt{Log}), (\texttt{ProcR}, \texttt{PrepC}), (\texttt{Log}, \texttt{Sign}), (\texttt{PrepC}, \texttt{Sign})\}$
- $\mathcal{C} = \{R_{\{\texttt{ProcR},\texttt{Log}\}}, R_{\{\texttt{ProcR},\texttt{PrepC}\}}, R_{\{\texttt{Log},\texttt{PrepC}\}}, R_{\{\texttt{PrepC},\texttt{Sign}\}}\}$ (see Figure 7.2).

The workflow in Figure 7.1 is satisfiable. A possible plan is that in which Alice processes the request, David logs it, Bob prepares the contract and Frank signs it:

$$\pi(\texttt{ProcR}) = \texttt{a}, \pi(\texttt{Log}) = \texttt{d}, \pi(\texttt{PrepC}) = \texttt{b}, \pi(\texttt{Sign}) = \texttt{f}$$

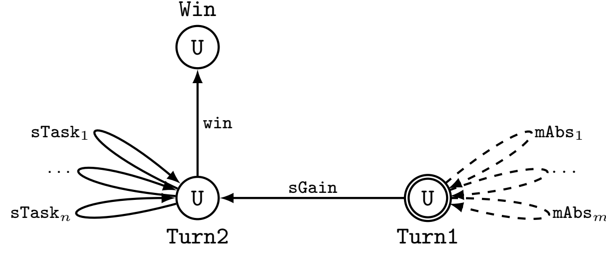
The workflow in Figure 7.1 is also statically, decrementally and dynamically resilient up to 1 user. Note that static resiliency is not a matter of which order is chosen (meeting the restrictions imposed by $\prec$) to execute tasks as the users are removed before starting. Dynamic resiliency *is* also a matter of order. Take `Log` and `PrepC` in Figure 7.1 as an example. The ACWF is resilient provided that we execute `Log` before `PrepC`. Assume we don't and reverse the order of execution (both orders are fine with respect to $\prec$). Assume also the best case in which no user is absent before starting and thus Charlie processes the request (in this scenario, all users authorized for `PrepC` and `Log` will be available). `Player2` breaks the execution by making Bob absent before `PrepC`. Then, since `PrepC` and `Log` must be executed by two different users, `Player2` will remove David (if Emma prepared the contract `PrepC`) or Emma (if David did so). In this way, the only user remaining for `Log` will be equal to the one who prepared the contract. This problem does not happen if we execute `Log` first.

7.2 Workflow Resiliency via Controller Synthesis

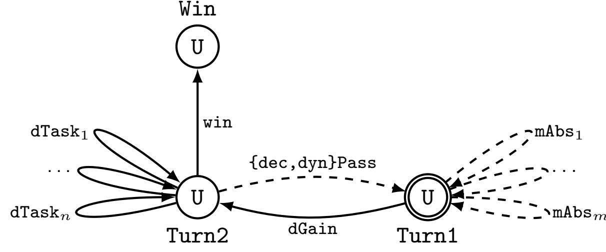
In this section, I encode a general ACWF $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$ into (three slightly different) TGAs to check static, decremental and dynamic resiliency. These encodings are similar and, of course, that for dynamic resiliency subsumes that for decremental resiliency, which in turn subsumes that for static resiliency. All encodings have the following components in common (Figure 7.3).

The "core" of the TGA consists of 3 *urgent* locations: `Turn1` (initial), `Turn2` and `Win` (which, if reached, implies resiliency). Each task $T \in \texttt{Tasks}$ is assigned a unique incremental integer starting from 0. So is each user $u \in \texttt{Users}$. Mapping tasks and users to integers allows me to employ them as indexes over array data structures I am going to use in UPPAAL-TIGA. Without loss of generality, when intended as indexes, I abuse notation and write u and T meaning their assigned integers.

I model the availability of users as a Boolean vector *Avail* indexed on users. Hence, if $Avail[u] = \top$, it means that u is available, and absent otherwise. The initial state of *Avail* consists of all elements set to true, meaning that all users are available. Likewise, I keep track of which user executed which task by means of a task vector *task* whose index ranges over tasks. If $task[T] = u$, then task T was executed by user u. The initial state of *task* consists of all elements set to -1, meaning that all tasks have not been executed yet. The set of clocks is



(a) Skeleton of the TGA for the static resiliency checking.



(b) Skeleton(s) of the TGA(s) for decremental (`dec`) and dynamic (`dyn`) resiliency checkings. The encodings only differ for the pass transition.

Fig. 7.3: Skeletons of the encodings for static (a) and decremental (if `decPass`) and dynamic (if `dynPass`) resiliency checking (b). All locations are urgent (`Turn1` is the initial one). $\mathtt{mAbs}_i$ transitions make users absent. $\mathtt{sTask}_i$ transitions model task executions. `sGain`, `dGain`, `{dec,dyn}Pass` model the game interplay. `win` allows `Player1` to move to `Win` when all tasks have been executed and all constraints are satisfied. `Player2` takes uncontrollable transitions (dashed edges), whereas `Player1` controllable ones (solid edges).

empty as I model resiliency as an instantaneous game. However, I keep using TGA reachability algorithms and prioritization of uncontrollable transitions to model the sudden absence of users. I maintain an integer variable k (initially set to 0) to model the current number of absent users and an integer constant $maxAbsent$ to model the maximum number of absent users.

7.2.1 Static Resiliency

Figure 7.3a shows the skeleton of the encoding of an ACWF into a TGA for the static resiliency checking, whereas Figure 7.4 shows the concretization in UPPAAL-TIGA for the ACWF in Figure 7.1. The encoding is as follows.

`Player2` (the environment) can make a user u absent by taking the corresponding *uncontrollable* self loop transition at `Turn1` labeled by `mAbs`. The TGA specifies as many `mAbs` transitions as the number of users in `Users` (i.e., one for each user). The guard and update of the transition making absent a general user u are:

Guard: $k < maxAbsent \wedge Avail[u] = \top$

Update: $k := k + 1, \mathit{Avail}[u] := \bot$

The guard says that u is available ($\mathit{Avail}[u] = \top$) and that it is still possible to remove users ($k < \mathit{maxAbsent}$), whereas the update makes u absent ($\mathit{Avail}[u] := \bot$) also incrementing the number of absent users ($k := k + 1$).

Since uncontrollable transitions have priority over controllable ones, `Player2` is always able to remove as many users as he wants (up to *maxAbsent*) before `Player1` gets control of the game by taking `sGain` whose guard and update are:

Guard: $\top$
Update: (no update)

The guard is always true, whereas the update makes no effect on the state of the system.

At `Turn2`, `Player1` assigns users to tasks, by taking the corresponding *controllable* self loop transitions labeled by `sTask`. For each task $T \in$ `Tasks` and authorized user $u \in \mathcal{A}(T)$, there is an `sTask` transition whose guard and update of are:

Guard: $\mathit{task}[T] = -1 \wedge \mathit{Avail}[u] = \top \wedge \Pi(T) \wedge \Delta(T, u)$
Update: $\mathit{task}[T] := u$

where

$$\Pi(T) = \bigwedge_{T' \prec T} \mathit{task}[T'] \neq -1$$

models the condition that all tasks preceding T have already been executed, and

$$\Delta(T, u) = \bigwedge_{T' \in \Pi(T) \wedge R_{\{T',T\}} \in \mathcal{C}} (\bot \bigvee_{(u',u) \in R_{\{T',T\}}} \mathit{task}[T'] = u'))$$

models, for each $T' \prec T$, which the allowed tuples matching $\mathit{task}[T] = u$ are.

As an example, consider the `sTask` transition modeling "Bob prepares the contract". From the formal specification of the ACWF in Figure 7.1 we have that

- $\Pi(\texttt{PrepC}) = \mathit{task}[\texttt{ProcR}] \neq -1$ as $\texttt{ProcR} \prec \texttt{PrepC}$ and that
- $\Delta(\texttt{PrepC}, \texttt{b}) = (\bot \vee \mathit{task}[\texttt{ProcR}] = \texttt{a} \vee \mathit{task}[\texttt{ProcR}] = \texttt{c})$ as Bob can prepare the contract if and only if either Alice or Charlie processed the request (Figure 7.2b).

This is an optimization to speed up the model-checking phase disabling this transition when neither Alice nor Charlie executed `ProcR`. Note that $\Delta(T, u)$ also contains the disjunct $\bot$ because when a pair (T, u) does not match any pair (u', T') where T' is a predecessor of T (i.e., when $T' \prec T$ and for all $u' \in \mathcal{A}(T')$ we have that $(u', u) \notin R_{\{T',T\}}$) it means that u is not going to be fine for T no matter who executed T'. In that case, $\bot$ creates a "dead end" falsifying the guard of the transition so that the model-checking phase will not search for a solution exploring impossible runs (a second optimization).

Finally, `Player1` moves to `Win` by taking the controllable `win` transition whose guard and update are:

Guard: $\mathit{over}() \wedge \mathit{sat}()$

Update: (no update)

where, $\mathit{over}() = \bigwedge_{T \in \texttt{Tasks}} \mathit{task}[T] \neq -1$ is a condition modeling that all tasks have been executed, and

$$\mathit{sat}() = \top \bigwedge_{R_{\{T_i,T_j\}} \in \mathcal{C}} \Big(\bigvee_{(u_i,u_j) \in R_{\{T_i,T_j\}}} (\mathit{task}[T_i] = u_i \wedge \mathit{task}[T_j] = u_j) \Big)$$

is another condition valuating to true if and only if all constraints are satisfied. In our example, we have that

$$\mathit{over}() = \top \wedge \mathit{task}[\texttt{ProcR}] \neq -1 \wedge \mathit{task}[\texttt{Log}] \neq -1 \wedge \mathit{task}[\texttt{PrepC}] \neq -1 \wedge \mathit{task}[\texttt{Sign}] \neq -1$$

whereas, e.g., the subpart of `sat` verifying $R_{\{\texttt{Log},\texttt{PrepC}\}}$ (Figure 7.2c) is $(\mathit{task}[\texttt{Log}] = \texttt{d} \wedge \mathit{task}[\texttt{PrepC}] = \texttt{b}) \vee (\mathit{task}[\texttt{Log}] = \texttt{d} \wedge \mathit{task}[\texttt{PrepC}] = \texttt{e}) \vee (\mathit{task}[\texttt{Log}] = \texttt{e} \wedge \mathit{task}[\texttt{PrepC}] = \texttt{b}) \vee (\mathit{task}[\texttt{Log}] = \texttt{e} \wedge \mathit{task}[\texttt{PrepC}] = \texttt{d})$.

7.2.2 Decremental and Dynamic Resiliency

The encodings for decremental and dynamic resiliency both extend that for static resiliency and differ from one another just for the guard of one single transition. I show the skeleton of the encodings in Figure 7.3b and the concretization in UPPAAL-TIGA in Figure 7.5 and Figure 7.6, respectively. The extensions are as follows.

I add a Boolean variable *done* initially set to $\bot$. I use this variable to guarantee that at any round (i) `Player1` executes one and only one task, and (ii) `Player2` waits for `Player1` to finish (Algorithm 3 and Algorithm 4). I extend the `sGain` transition (Figure 7.3a) into a `dGain` transition (Figure 7.3b) by refining the guard and update in

Guard: $\top$
Update: $\mathit{done} := \bot$

In this way, every time the run gets to `Turn2`, `Player1` can execute one task only. I extend `sTask` transitions (Figure 7.3a) into `dTask` transitions (Figure 7.3b) by refining the guard and update as follows

Guard: $\mathit{task}[T] = -1 \wedge \cdots \wedge \Delta(T,u) \wedge \mathit{done} = \bot$
Update: $\mathit{task}[T] := u, \mathit{done} := \top$

That is, each of these transitions can be taken only if $\mathit{done} = \bot$, and once taken *done* is set to $\top$ to prevent `Player1` from taking more than one.

I add an *uncontrollable* transition `decPass` going from `Turn2` to `Turn1` to regulate the turns in which the players play while the workflow executes (Figure 7.3b). This transition is the only difference between the encodings for decremental and dynamic resiliency.

In case of decremental resiliency the guard and update are:

Guard: $\mathit{done} = \top \wedge \neg \mathit{over}()$
Update: (no update)

whereas in case of dynamic resiliency they are:

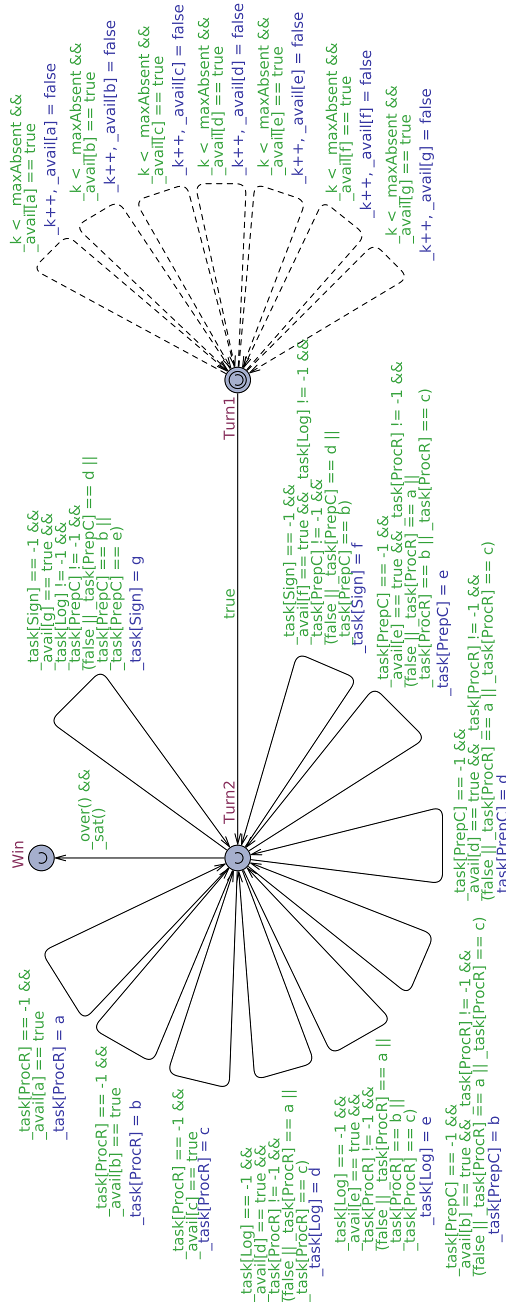


Fig. 7.4: Concretization in UPPAAL-TIGA for the static resiliency checking of the ACWF in Figure 7.1.

Guard: $done = \top \wedge \neg over()$
Update: $restore()$

where $restore()$ is an update function operating on the state of the system to, as Wang and Li said, reset the access control state [122, 123]. $restore()$ contains a statement $Avail[u] := \top$ for any $u \in \mathcal{A}(T)$ to make users available again and a statement $k := 0$ to reset the counter of absent users. All other components remain the same.

Finally, regardless of the type of resiliency we want to check, a controller can be synthesized by model checking the system with `control: A<> tga.Win`. If the workflow is resilient, a controller exists and the synthesized strategy *is* the dynamic plan for executing the workflow. If it does not, then the workflow is breakable and no strategy exists. In such a case, the analysis confirms that `Player2` has a counter-strategy to always break the execution no matter which users we decide to commit. We will always fail.

7.3 Correctness and Complexity of the Encodings

In this section, I first prove the correctness of the three encodings for static, decremental and dynamic resiliency, and then I analyze their complexity.

For the correctness part, I prove that each encoding generates a TGA such that any run of the TGA corresponds to a run of the corresponding game defined by Wang and Li, whereas for the complexity I prove that these encodings are generated in polynomial time.

7.3.1 Static resiliency

Theorem 7.1. *The encoding for static resiliency given in Section 7.2 correctly models Algorithm 2.*

Proof. The encoding for static resiliency given in Section 7.2 generates a TGA whose internal state consists of the following *components*:

Constants

1. n unique and consecutively integer constants $T_1, \ldots, T_n$ (starting from 0) to employ tasks as indexes.
2. m unique and consecutively integer constants $u_1, \ldots, u_n$ (starting from 0) to employ users as indexes.
3. An integer constant $maxAbsent$ hardcoding the maximum number of absent users.

Vectors

4. A Boolean vector $Avail$ indexed on users whose elements are initially all set to $\top$.
5. An integer vector $task$ indexed on tasks whose elements are initially all set to -1.

Variables

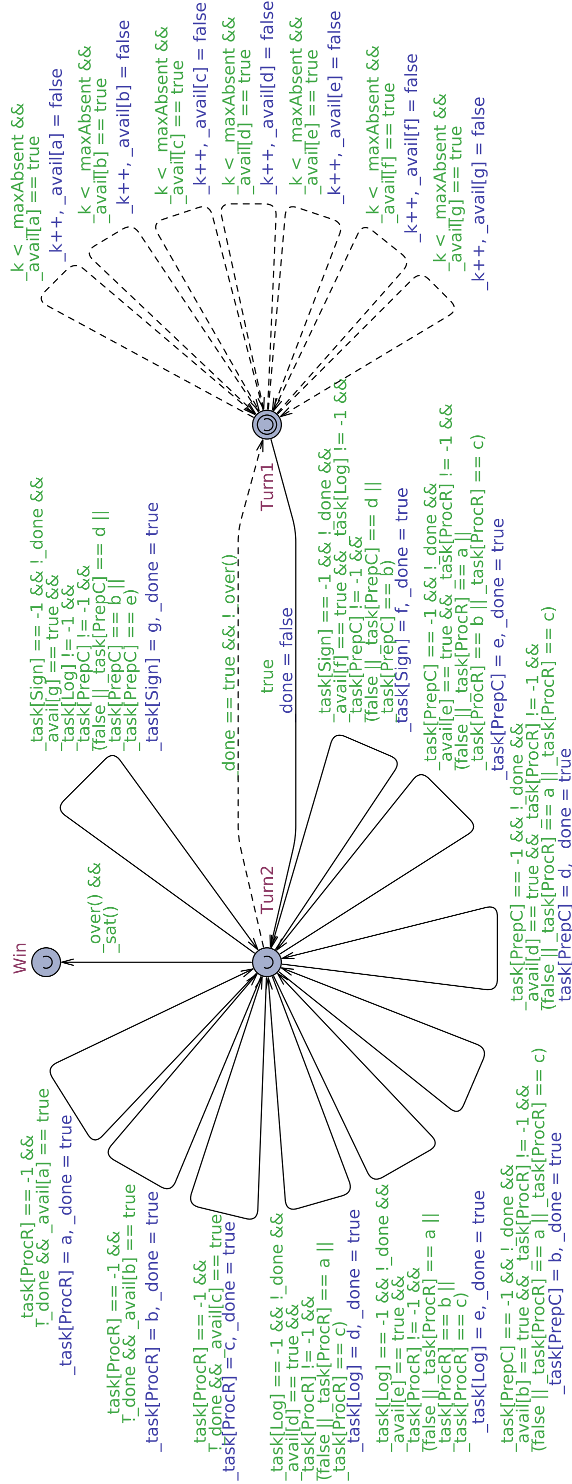


Fig. 7.5: Concretization in UPPAAL-TIGA for the decremental resiliency checking of the ACWF in Figure 7.1.

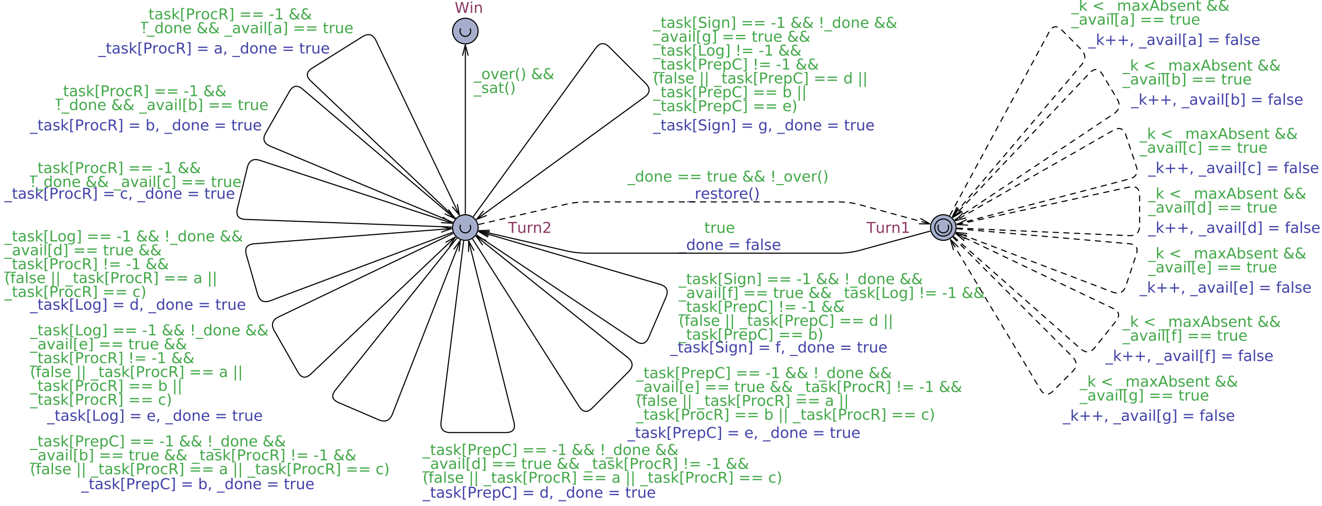


Fig. 7.6: Concretization in UPPAAL-TIGA for the dynamic resiliency checking of the ACWF in Figure 7.1.

7. An integer variable k initially set to 0, modeling the *current* number of absent users.

Functions

8. A function $over()$ modeling "all tasks executed".
9. A function $sat()$ modeling "all constraints satisfied".

The *state* of this TGA is a pair (L, V), where L is a location and V any data structure handling components (1)-(7).

I recall that all locations are urgent, thus any run of this TGA starts and also ends at 0. That is, all actions are instantaneous but their order of consideration is regulated by both the state of the TGA, which may falsify some guards, and the prioritization of the transitions, when controllable and uncontrollable transitions are taken simultaneously.

`Turn1` is the initial location, where `Player2` (i.e., the environment) plays. The initial state of the system is $(\texttt{Turn1}, V_0)$, where V_0 is

- $k = 0$,
- $Avail[u] = \top$ for all $u \in \texttt{Users}$,
- $task[T] = -1$ for all $T \in \texttt{Tasks}$.

When the run starts, `Player2` can make absent as many users as he likes (up to $maxAbsent$) by taking the `mAbs`-transitions. Every time `Player2` takes one of these transitions k is incremented by 1. Since all `mAbs` transitions are uncontrollable, `Player1` is unable to interrupt `Player2` by taking `sGain` (the unique controllable transition having source at `Turn1`). When `Player2` is done, `Player1` can take `sGain` to enter `Turn2`, the location in which he can assign (available) users to tasks. When `Player1` enters `Turn2`, the current state of the system is $(\texttt{Turn2}, V_1)$, where V_1 is such that

- $k \leq maxAbsent$,
- $Avail[u] = \bot$ for each user u turned absent,
- $task[T] = -1$ for all $T \in \texttt{Tasks}$.

This state corresponds to choosing $Absent \subset \texttt{Users}$ such that $|Absent| \leq k$ (Algorithm 2, lines 1-2). Now at `Turn2`, `Player2` will not play anymore. In this location, `Player1` can assign (available) users to (unexecuted) tasks by taking one and only one `sTask`-transition for each pair (u, T), where u is an available user authorized for the unexecuted task T. These transitions also specify conditions on predecessors (if any) and corresponding constraints related to them[1]. When `Player1` is done (i.e., when all `sTask` transitions are disabled), the run can be ended up in two possible states.

The first possibility is the state $(\texttt{Turn2}, V_2')$, where V_2' is:

- $k \leq maxAbsent$,
- $Avail[u] = \bot$ for each user u turned absent,
- $task[T] = -1$ for some $T \in \texttt{Tasks}$.

[1] This is an optimization to speed up the model checking phase pruning impossible runs. Removing this optimization would slow down the model checking phase but would not destroy the correctness of the approach.

If this is the case, it means that for some task T there is no authorized and available user u such that the partial plan extended with $T = u$ is locally consistent. In other words, the workflow is breakable if `Player2` removes all users u such that $Avail[u] = \bot$ before starting.

The second possibility is the state $(\texttt{Turn2}, V_2'')$, where V_2'' is:

- $k \leq maxAbsent$,
- $Avail[u] = \bot$ for each user u turned absent,
- $task[T] \neq -1$ for all $T \in \texttt{Tasks}$.

If this is the case, it means that all tasks have been assigned a user. `Player1` generates such an assignment according to $\prec$ and the related constraints. However, this assignment is not necessarily consistent as it verifies only part of the constraints as a speed up technique. Now, if `Player1` can take the `win`-transition it means that both the functions $sat()$ and $over()$ are true. That is, it means that a consistent plan π exists (Algorithm 2, lines 3-4). If he cannot, then it means that the workflow is breakable as there is no way to satisfy the constraints with any of the possible assignments under the absence of the users removed by `Player2` (Algorithm 2, lines 3 and 5-6). Therefore, the final state is $(\texttt{Win}, V_3)$, where V_3 is

- $k \leq maxAbsent$,
- $Avail[u] = \bot$ for each user u turned absent,
- $task[T] = u$ for each (u, T) such that $T \in \texttt{Tasks}$, $u \in \mathcal{A}(T)$, and $\pi(T) = u$,

where π is consistent. The model checking phase looks for a control strategy for `Player1` to always eventually enter `Win`, which is equivalent to saying that `Player1` can always eventually enter `Win` if and only if the ACWF is statically resilient.

7.3.2 Decremental resiliency

Theorem 7.2. *The encoding for decremental resiliency given in Section 7.2 correctly models Algorithm 3.*

Proof. For decremental resiliency the state of the TGA also contains a variable $done$ that is used to regulate the interplay of the game between `Player1` and `Player2`.

As for static, when the run is in `Turn1`, `Player2` can take as many `mAbs`-transitions as he likes before starting. However, since the encoding for decremental resiliency adds an uncontrollable transition from `Turn2` to `Turn1`, `Player2` can make users absent also during execution.

Therefore, all states discussed for static resiliency extend by adding the Boolean variable $done$. Therefore, $(\texttt{Turn1}, V_0)$, where V_0 is

- $k = 0$,
- $done = \bot$,
- $Avail[u] = \top$ for all $u \in \texttt{Users}$,
- $task[T] = -1$ for all $T \in \texttt{Tasks}$.

When `Player2` is done in his turn, `Player1` can take (the now renamed) `dGain` transition to enter `Turn2`.

At `Turn2`, before `Player1` starts playing, the current state of the system is $(\texttt{Turn2}, V_1)$, where V_1 is such that

- $k \leq \mathit{maxAbsent}$,
- $\mathit{done} = \bot$ (note that `dGain` sets done to $\bot$),
- $\mathit{Avail}[u] = \bot$ for each user u turned absent,
- $\mathit{task}[T] = -1$ for all $T \in \texttt{Tasks}$.

This corresponds to (extending) the set of absent users (Algorithm 3, lines 5-7).

At `Turn2`, `Player1` can make one assignment only as all `dTask` guards extend by adding the clause $\mathit{done} = \bot$ and all updates set done to $\top$. This corresponds to extending the partial plan π with one assignment only (Algorithm 3, line 8). Note that this assignment (if any) does not guarantee consistency. However, this is not a problem since if the run gets to `Win`, then it must have made an assignment such that π was locally consistent. Therefore, such an assignment models Algorithm 3, lines 12-13.

The `decPass` transition allows `Player2` to get back to `Turn1`. This is an uncontrollable transition. However, this transition cannot prevent `Player1` from making his assignment as `decPass` will be available only when `Player1` is done and there is still some task to execute. That is, as soon as done is set to $\top$ as a consequence of one `dTask` transition. At `Turn1`, `Player2` can again make absent a few more users (if any are left for this operation). When `Player2` is done with this turns, `dGain` allows `Player1` to make the next assignment. This run ends when assignments are no longer possible. As for static resiliency, this happens for two possibilities: $(\texttt{Turn2}, V_2')$ and $(\texttt{Turn2}, V_2'')$, where $(\texttt{Turn2}, V_2')$ is

- $k \leq \mathit{maxAbsent}$,
- $\mathit{done} = \bot$,
- $\mathit{Avail}[u] = \bot$ for each user u turned absent,
- $\mathit{task}[T] = -1$ for some $T \in \texttt{Tasks}$,

and $(\texttt{Turn2}, V_2'')$ is

- $k \leq \mathit{maxAbsent}$,
- $\mathit{done} = \top$,
- $\mathit{Avail}[u] = \bot$ for each user u turned absent,
- $\mathit{task}[T] \neq -1$ for all $T \in \texttt{Tasks}$.

In any of these two states `Player2` cannot go back to `Turn1` as $\mathit{done} = \bot$ (and no action of `Player1` sets it to $\top$). In the first case, the workflow is breakable, whereas in the second case, the workflow might be resilient (if the synthesized plan is always consistent). If this is the case, then the final state is $(\texttt{Win}, V_3)$ where V_3 is

- $k \leq \mathit{maxAbsent}$,
- $\mathit{done} = \top$,
- $\mathit{Avail}[u] = \bot$ for each user u turned absent,
- $\mathit{task}[T] = u$ for each (u, T) such that $T \in \texttt{Tasks}$, $u \in \mathcal{A}(T)$, and $\pi(T) = u$.

7.3.3 Dynamic resiliency

Theorem 7.3. *The encoding for dynamic resiliency given in Section 7.2 correctly models Algorithm 4.*

Proof. The encoding for dynamic resiliency refines that for decremental by adding the restore of the absent users at the end of any turn. This is implemented by a function *restore*() in the update of `dynPass`. Everything else remains the same.

Theorem 7.4. *Each encoding discussed in Section 7.2 is generated in polynomial time.*

Proof. I start by discussing the encoding for static resiliency, then I extend the discussion to dynamic resiliency passing through decremental resiliency.

Given an ACWF $W = \langle \texttt{Tasks}, \texttt{Users}, UA, \prec, \mathcal{C} \rangle$, the encoding for static resiliency builds a TGA consisting of 3 locations, $|\texttt{Users}|$ (uncontrollable) `mAbs`-transitions, 1 `sGain` transition, $\sum_{T \in \texttt{Tasks}} |\mathcal{A}(T)|$ `sTask` transitions, where in the worst case there are $|\texttt{Tasks}| \times |\texttt{Users}|$ transitions (all users authorized for all tasks), and 1 `win` transition. Also, users and tasks are mapped to as many integer constants as them, *over*() contains exactly $|\texttt{Tasks}|$ conjuncts $task[T] \neq -1$ and *sat*() contains $|\mathcal{C}|$ conjuncts[2], where each conjunct represents a relation $R_{\{T_i, T_j\}} \in \mathcal{C}$ by means of a disjunction[3] of pair-conjunctions $(task[T_i] = u_p \wedge task[T_j] = u_z)$ for each $(u_p, u_z) \in R_{\{T_i, T_j\}}$. In other words, the size of `sat` is $\sum_{R_{\{T_i, T_j\}} \in \mathcal{C}} |R_{\{T_i, T_j\}}|$, where abusing notation $|R_{\{T_i, T_j\}}|$ represents the number of tuples contained in the relation. The worst case is that in which all relations have equal size $S = |R_{\{T_i, T_j\}}| = \cdots = |R_{\{T_o, T_p\}}|$ and thus $sat()$ has length of $S \times |\mathcal{C}|$. The guards and updates of `mAbs`, `sGain` and `win` transitions specify a fixed number of statements. `sTask` transitions might specify conditions on predecessors and related constraints. The worst case is that in which we degenerate $\prec$ such that $T_1 \prec T_2$ for any $T_1, T_2 \in \texttt{Tasks}$ (unexecutable ACWF). In this way each transition would have $|\texttt{Tasks}|$ predecessors and related conditions which are a polynomial number of pairs involving the task whose execution is modeled by the transitions and the predecessors.

Since all subcomponents are generated in polynomial time, the overall complexity that sums them all up is polynomial.

Decremental resiliency "worsens" this encoding by adding *done* in some of the the guards of the transitions and an uncontrollable transition going from `Turn2` to `Turn1`. Therefore, the upper bound of the complexity coincides with that of the encoding for static resiliency.

Dynamic resiliency "worsens" the encoding for decremental resiliency by adding *restore*() in the update of `dynPass`. The length of this function is $|\texttt{Users}| + 1$ as it resets the availability of all users and resets k to 0. Therefore, the upper bound of the complexity coincides with that of the encoding for static and decremental resiliency.

7.4 Erre: A Tool for Workflow Resiliency

I developed Erre, a tool for workflow resiliency that takes as input a specification of an ACWF (Definition 2.35) and acts both as a solver for the three kinds of resiliency and as an executor simulator. Listing 7.1 shows Erre's help screen.

[2] I neglect the unique $\top$ at the beginning of the conjunction.
[3] I neglect the unique $\perp$ at the beginning of the disjunction.

Listing 7.1: Erre's help screen.

```
Usage: java -jar erre.jar <Workflow.wf> <Action> <static|decremental|
       dynamic> <k> <Workflow.s> [N] [--silent]

<Action>:
  --check    internally encodes the workflow in input into an UPPAAL-TIGA
             specification ready to check static, decremental or dynamic
             resiliency up to k users (saves the strategy to Workflow.s)

  --execute  performs [N] (default 1) executions of the workflow in input
             (if resilient) according to the strategy (.s) synthesized by
             UPPAAL-TIGA.

  --silent   suppresses output (optional)

Examples:
  java -jar erre.jar CaseStudy.wf --check dynamic 2 CaseStudy.s
  java -jar erre.jar CaseStudy.wf --execute dynamic 2 CaseStudy.s 10
```

The input language of Erre comprises the following three main sections. The section `Users`

```
Users {
  u1 ... un
}
```

specifies the set `Users` and provides here an example of `Users` $= \{u_1, \ldots, u_n\}$. The section `Tasks`

```
Tasks {
  ...
  (T1 : u1 u2 ...)
  ...
}
```

specifies the set `Tasks` and the authorization relation UA and provides here an example of `Tasks` $= \{T_1, \ldots\}$ such that $UA = \{(u_1, T_1), (u_2, T_1), \ldots\}$. The section `Precedence`

```
Precedence {
  ...
  (T1 < T2)
  ...
}
```

specifies $\prec$ and provides an example of $T_1 \prec T_2$. The section `Constraints`

```
Constraints {
  ...
    (T1 T2 : (u1 u2) ...)
  ...
}
```

specifies the set $\mathcal{C}$ and provides here an example of (R_S), where $S = \{T_1, T_2\}$, $R = \{(u_1, u_2), \dots\}$.

Given an ACWF specification file `Workflow.wf`, we check, for example, static (1), decremental (2) and dynamic (3) resiliency up to 1 users by running

```
1 $ java -jar erre.jar Workflow.wf --check static 1 Workflow.s
2 $ java -jar erre.jar Workflow.wf --check decremental 1 Workflow.s
3 $ java -jar erre.jar Workflow.wf --check dynamic 1 Workflow.s
```

If the ACWF is proved resilient, Erre saves to file the *dynamic plan* needed to later execute it. We execute (once) a resilient ACWF by running

```
$ java -jar erre.jar Workflow.wf --execute static 1 Workflow.s [N]
```

where `[N]` (default 1) is the number of simulations we want to carry out. For static resiliency, Erre randomly removes a subset of users whose cardinality is up to k *before* starting the execution. For decremental resiliency Erre also removes random users during execution, whereas for dynamic resiliency Erre randomly removes and puts back users at any time.

Listing 7.2 shows the specification of Figure 7.1 into Erre's input language.

Listing 7.2: Specification of Figure 7.1.

```
1  Users {
2    a b c d e f g
3  }
4
5  Tasks {
6    (ProcReq : a b c)
7    (Log : d e)
8    (PrepContr : b d e)
9    (Sign : f g)
10 }
11
12 Precedence {
13   (ProcReq < Log)
14   (ProcReq < PrepContr)
15   (Log < Sign)
16   (PrepContr < Sign)
17 }
18
19 Constraints {
20   (ProcReq Log : (a d) (a e) (b e) (c d) (c e))
21   (ProcReq PrepContr : (a b) (a d) (a e) (b e) (c b) (c d) (c e))
22   (Log PrepContr : (d b) (d e) (e b) (e d))
23   (PrepContr Sign : (b f) (b g) (d f) (d g) (e g))
24 }
```

I ran Erre on the ACWF in Figure 7.1 to check each type of resiliency up to 1 absent user. I used a FreeBSD virtual machine run on top of a VMWare ESXi Hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM. The VM was assigned 16GB of RAM and full CPU power. Erre proved in

394ms that the ACWF in Figure 7.1 is statically resilient saving a dynamic plan of 36Kb (152-action strategy). Then, Erre proved in about 394ms that the ACWF is decrementally resilient saving a dynamic plan of 36Kb (147-action strategy). Finally, Erre proved in 410ms that the ACWF is also dynamically resilient saving a dynamic plan of 60Kb (239-action strategy). For each kind of resiliency, the ACWF was correctly executed. This example is available at http://regis.di.univr.it/ExampleResiliency.tar.bz2.

Listing 7.3 shows a few execution simulations of Figure 7.1 for static, decremental and dynamic resiliency.

Listing 7.3: Execution simulations for static, decremental and dynamic resiliency.

```
1  $ java -jar erre.jar ACWF.wf --execute static 1 ACWF.dyn.1.s 1
2  Resiliency level = 1 (static)
3  maxAbsent = 1
4  Execution 1
5  ------------------
6  Absent = {f}
7  ProcR = a
8  Log = d
9  PrepC = b
10 Sign = g
11 ------------------
12 Verifying ... SAT!
13
14 $ java -jar erre.jar ACWF.wf --execute decremental 1 ACWF.dec.1.s 1
15 Resiliency level = 1 (decremental)
16 maxAbsent = 1
17 Execution 1
18 ------------------
19 Absent = {}
20 ProcR = a
21 Absent = {}
22 Log = d
23 Absent = {c}
24 PrepC = b
25 Absent = {c}
26 Sign = f
27 ------------------
28 Verifying ... SAT!
29
30 $ java -jar erre.jar ACWF.wf --execute dynamic 1 ACWF.dyn.1.s 1
31 Resiliency level = 1 (dynamic)
32 maxAbsent = 1
33 Execution 1
34 ------------------
35 Absent = {d}
36 ProcR = a
37 Absent = {}
38 PrepC = b
39 Absent = {e}
40 Log = d
```

Algorithm 16: `ACWF-Gen`(n, u, c, k)

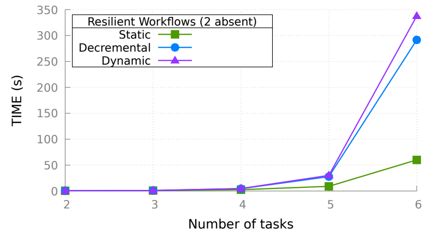
```
Input: An exact number of tasks n, and exact number of users u, a maximum
       number of constraints c, a maximum number of tuples k per constraint.
Output: An ACWF according to Definition 2.35.
1  W ← ⟨Tasks, Users, UA, ≺, C⟩                          ▷ Empty ACWF
   ▷ Generate tasks
2  Tasks ← {T_i | 1 ≤ i ≤ n}
3  Users ← {u_i | 1 ≤ i ≤ u}
   ▷ Generate authorization relation
4  for T ∈ Tasks do
5      A(T) = Users
   ▷ Generate constraints
6  for i = 1 to c do
7      T_1 ← Random(Tasks)
8      T_2 ← Random(Tasks)
9      R_{T_1,T_2} ← ∅
10     for j = 1 to k do
11         t ← Random(u_1, u_2)          ▷ For (u_1, u_2) ∈ A(T_1) × A(T_2)
           R_{T_1,T_2} ← R_{T_1,T_2} ∪ {t}
12     C ← C ∪ R_{T_1,T_2}
13 return W
```

```
41 Absent = {}
42 Sign = f
43 ------------------
44 Verifying ... SAT!
```

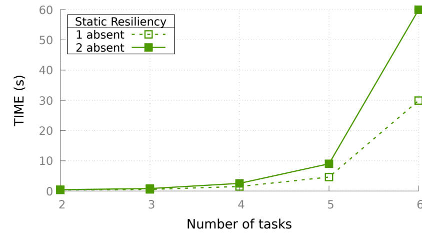
For static resiliency (first simulation) Frank is absent before the execution starts. For decremental resiliency (second simulation) nobody is absent until `Log` finishes and Charlie becomes absent after `Log`. For dynamic resiliency (third simulation) David is absent before starting but arrives after `ProcR` and Emma becomes absent after `PrepC` but comes back after `Log`.

Having Erre also allowed me to carry out an experimental evaluation against a set of random ACWFs.

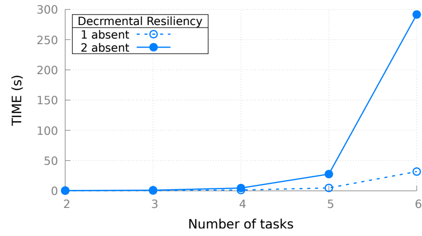
I generated 1000 ACWFs partitioned in 5 sets of benchmarks each one containing 100 dynamically resilient workflows (up to 2 absent users) and 100 dynamically breakable workflows (for 2 absent users). These sets of benchmarks are available at `http://regis.di.univr.it/EE_Resiliency2018.tar.bz2`. The first set (`2Tasks`) specifies workflows with 2 tasks, the second set (`3Tasks`) specifies workflows with 3 tasks and so on, up to the fifth one (`6Tasks`) that specifies workflows with 6 tasks. Regardless of the set, each workflow has exactly 6 users authorized for all tasks and specifies a maximum number of relational constraints of 10% of $|\texttt{Tasks}|^2$, where each relation $R_{\{T_1 T_2\}}$ has a maximum number of tuples of 60% of $|\mathcal{A}(T_1)| \times |\mathcal{A}(T_2)|$. Furthermore, each workflow does not specify any partial order. Again, authorizing all users for all tasks and not restricting the partial order contributes to generate hard instances. Algorithm 8 shows the pseudo-code of the



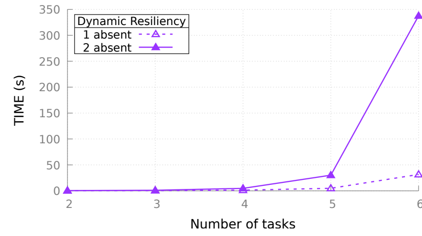
(a) *Static, decremental and dynamic resiliency checking time on (dynamically) resilient ACWFs.*



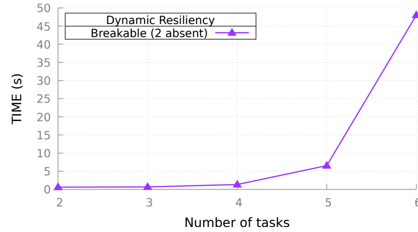
(b) *Static resiliency checking time on statically resilient ACWFs*



(c) *Decremental resiliency checking time on decrementally resilient ACWFs*



(d) *Dynamic resiliency checking time on dynamically resilient ACWFs*



(e) *Dynamic resiliency checking time on breakable ACWFs*

Fig. 7.7: Experimental evaluation with ERRE (time).

algorithm I developed to generate the networks. In the following I discuss the data I collected (time and space).

Figure 7.7 shows the graphical results of the experimental evaluation in which I focused on time, where x-axes always represent the number of tasks (i.e., the set of benchmarks under analysis) and y-axes represent the average time elapsed when analyzing the instances in that set. I ran ERRE on these sets of benchmarks without imposing any timeout. For each set, I ran the analysis for static, decremental and dynamic resiliency on dynamically resilient workflows for both 1 and 2 absent

users and I also ran the analysis for dynamic resiliency on workflows dynamically breakable for 2 absent users.

Figure 7.7a compares the time performances of the analysis for static, decremental and dynamic resiliency run on all workflows dynamically resilient up to 2 absent users. I recall that a dynamically resilient workflow (up to k absent users) is also decrementally and statically resiliency (up to the same k) as dynamic resiliency $\Rightarrow$ decremental resiliency $\Rightarrow$ static resiliency. The figure confirms, as I expected, that dynamic resiliency is harder than decremental resiliency which in turn is harder than static resiliency.

Figure 7.7b, Figure 7.7c and Figure 7.7d show the time differences when running, on the same workflows, the analysis for 1 absent user (dashed line) and 2 absent users (solid line) for static, decremental and dynamic resiliency, respectively. The results confirm that incrementing the number of absent users can only worsen the validation phase (particularly for workflows specifying more than 3 tasks).

Figure 7.7e shows how long it takes on average to break a workflow which is not dynamically resilient up to 2 absent users. The results show that the analysis worsens significantly for workflows specifying more than 3 tasks.

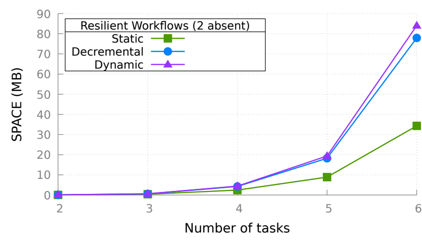
Instead, Figure 7.8 shows the graphical results of the experimental evaluation in which I focused on space, where x-axes still represent the number of tasks (i.e., the set of benchmarks under analysis) and y-axes represent the average space consumed by the synthesized strategies when analyzing the instances in that set.

Figure 7.8a compares the space consumption when synthesizing memoryless execution strategies for static, decremental and dynamic resiliency run on all workflows resilient up to 2 absent users. The figure confirms that "dynamic strategies" consume more space than "decremental strategies" which in turn consume more space than "static ones". However, for the analyzed instances there is no a huge difference between decremental and dynamic strategies.

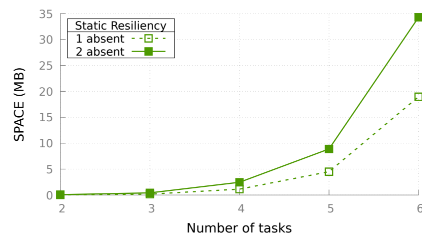
Figure 7.8b, Figure 7.8c and Figure 7.8d show the space differences when synthesizing strategies for statically, decrementally and dynamically resilient workflows up to 1 absent user (dashed line) and 2 absent users (solid line), respectively. The results confirm that incrementing the number of absent users results in synthesizing bigger strategies (particularly for workflows specifying more than 3 tasks).

Instead, Figure 7.8e, Figure 7.8f and Figure 7.8g show the differences in size of the strategies saved by UPPAAL-TIGA (dashed lines) and those saved by ERRE (solid lines) for statically, decrementally and dynamically resilient workflows, respectively, each up to 2 absent users. Differently from the experimental evaluation that I carried out for CSTNUDs, where there the strategies saved by ESSE consumed more space than those saved by UPPAAL-TIGA (Section 5.6), here the results show that the strategies saved by ERRE are smaller. The main reason is that the encodings for static, decremental and dynamic resiliency do not employ any clocks. Therefore, the synthesized strategies do not have any condition on (conjunction of) clock constraints (where in ESSE, one clock constraint means one more object to serialize to file). When dealing with these encodings, the strategies returned by UPPAAL-TIGA only specify discrete states and corresponding actions.

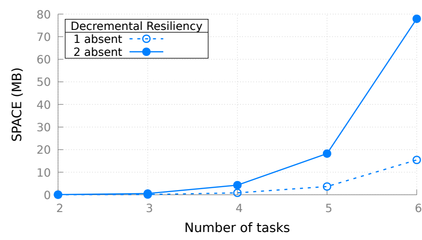
Finally, I executed 1000 times each resilient workflow for each type of resiliency. Each execution removed randomly up to 1 or 2 users according to the level of



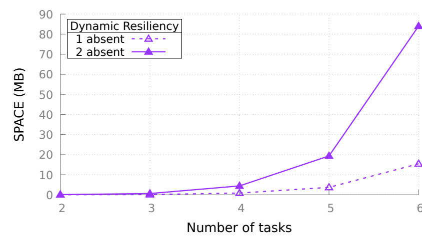
(a) *Strategy space for resilient workflows.*



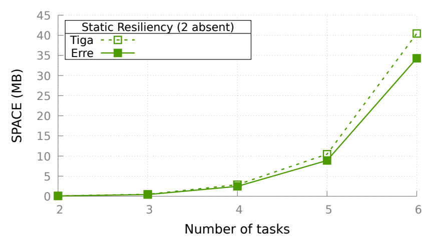
(b) *Strategy space for statically resilient workflows.*



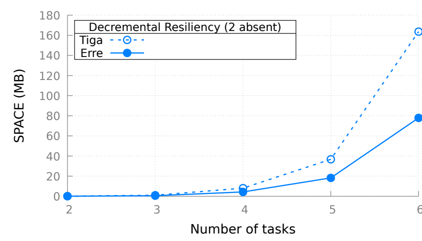
(c) *Strategy space for decrementally resilient workflows.*



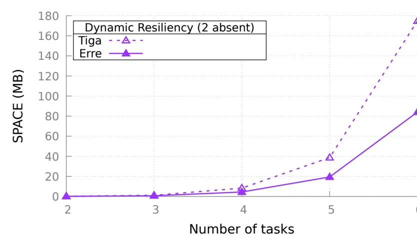
(d) *Strategy space for dynamically resilient workflows.*



(e) *Strategy space difference for statically resilient workflows*



(f) *Strategy space difference for decrementally resilient workflows*



(g) *Strategy space difference for dynamically resilient workflows*

Fig. 7.8: Experimental evaluation with Erre (space).

resiliency of the synthesized strategy. Overall, Erre simulated 3,000,000 of random executions. No one crashed.

7.5 Conclusions

I addressed static, decremental and dynamic resiliency. I provided three encodings into extended timed game automata to model these games and proved that my encodings are correct and run in polynomial time. I developed Erre, a tool for workflow resiliency and carried out an experimental evaluation in which I discussed how I generated random access controlled workflows.

The work in this chapter differs from [95,96] as it is not probabilistic. It differs from [47, 48] as I do not propose a “max-SAT” approach. It differs from [91, 107] as those works deal with static resiliency only. It differs from [77] as workflow feasibility (availability in some state) is the dual of workflow resiliency (thus incomparable). It also differs from [92] which deals with static and decremental resiliency only.

Part III

Temporal And Resource Controllability Together

Introduction

In Chapter 5 I used CSTNUDs to model temporal workflows. This new type of temporal network is able to address uncertain task durations, conditional constraints, decisions, classic temporal constraints such as delays and deadlines and relative constraints. However, CSTNUDs fail to address resource controllability as they do not deal with users nor with authorization constraints and they are thus unsuitable to underly a temporal workflow for which access control must be considered.

In Chapter 6 I used CNCUs to model access controlled workflows. This new type of constraint network is able to address authorization constraints and conditional uncertainty. However, CNCUs fail to address temporal controllability as they do not deal with temporal constraints.

In this part I merge some of the contributions of Part I with some of the contributions of Part II to deal with temporal and resource controllability together. I extend CSTNUs by first injecting users and authorization constraints and then I adapt the DC checking. Such an extension allows for the modeling of the *temporal workflow satisfiability problem (TWSP)*. I have already discussed that the WSP is the problem of finding an assignment of tasks to users so that the execution gets to the end without violating any authorization constraint (e.g., [123]). In a temporal context, in addition to saying which user we commit, we must also say when we do so. In other words, such an extension allows me to deal with situations in which users, authorization and temporal constraints *must* be considered all together and not orthogonal at all.

In fact workflows and access control have traditionally been approached as orthogonal, independent formalisms, but there is a large number of relevant cases in which they cannot be considered to be so as I discussed in Section 1.1.

Some work has been done (although not in a temporal context) to specify the access control model and the workflow in two levels by means of logical theories, such that the intersection of these two theories provides the means to operate on the access-controlled workflow; see, e.g., [7]. However, approaches relying on simply abstracting a workflow as a logical formula are not suitable for my aim. Indeed, doing so would not allow me to reason on uncontrollable parts and therefore on the runtime execution.

Contributions

Towards the modeling, validation and execution of plans dealing with resources, temporal and conditional uncertainty, my contributions in this part are the following.

1. First, I define *Access-Controlled Temporal Networks (ACTNs)* as an extension of CSTNUs by the injection of users and authorization constraints, motivated by the need to handle workflows subject to both temporal constraints and access control simultaneously.
2. I give the execution semantics of ACTNs in terms of *real-time execution decisions* (*RTEDs*, [25, 26, 28, 71]).

3. I extend the encoding for CSTNUs into TGAs for, again, a sound and complete dynamic controllability checking. I use a concrete, real-world case study from the e-health domain. I prove that the encoding of ACTNs into TGAs runs in polynomial time and it is correct.
4. I provide *Conditional Simple Temporal Network with Uncertainty and Resources (CSTNURs)* by extending CSTNUs with resources, temporal expressions and runtime resource constraints (RRCs).
5. I give the semantics of CSTNURs in RTEDs.
6. I extend the encoding into TGAs given for CSTNUs in order to model resource commitments for time point executions and RRCs and I model and validate an example from the air transport domain.
7. I prove that CSTNURs can also be encoded into Conditional Disjunctive Temporal Networks with Uncertainty (CDTNUs) and discuss pro and cons.
8. I provide a translation from periodic time into STNs as well as a connection mapping connecting this STN to the ACTN or the CSTNUR describing a temporal access controlled plan. In this way, I enforce a TRBAC model on top of ACTNs and CSTNURs.

Organization

Chapter 8 introduces *access controlled temporal networks (ACTNs)* along with their semantics in real time execution decisions (RTEDs) and an encoding from ACTNs into TGAs to check the dynamic controllability and synthesize execution strategies. It also discusses correctness results of the proposed algorithms. Each authorization constraint expresses a temporal range, (i.e., a real interval within which the occurrences of the events it connects are constrained to happen), a label modeling the conditional part and an authorization policy expressing which the authorized users are. In general, an ACTN is a temporal network disjunctive in its nature since its *time points* can be executed by different users and authorization constraints between time points may be different depending on the executing users. In Section 9.6 I discuss the relation between CSTNURs and CDTNUs. Chapter 9 introduces *conditional simple temporal networks with uncertainty and resources (CSTNURs)* along with their semantics in RTEDs and an encoding into TGAs to check the dynamic controllability and synthesize execution strategies. It also discusses correctness results of the proposed algorithms. CSTNURs differ from ACTNs for two main aspects. First, each user is associated to a temporal expression for each time point he is authorized for. Second, runtime resource constraints work on these temporal expressions and their effect might be different depending on which order the time points are executed. Section 9.8 introduces a translation from periodic time to STNs to model the temporal constraints of a given TRBAC instance. Then, it provides a connection mapping to enforce the temporal constraints on role enabling and disabling on CSTNURs and also on ACTNs.

8

Access Controlled Temporal Networks

In this chapter I extend CSTNUs by injecting users and authorization constraints, motivated by the need to handle workflows subject to both temporal constraints and access control simultaneously. Each authorization constraint expresses a temporal range, a label modeling the conditional part, and an authorization policy expressing which the authorized users are. I call this new kind of temporal network: *Access Controlled Temporal Network* (ACTN).

I start by considering a simple but real-world (thus fully realistic) case study which is a simplification of the official guidelines for the treatment of STEMI patients (i.e., patients with ST-Elevation Myocardial Infarction), published by the American College of Cardiology/American Heart Association (see the discussion in [34]). None of the previous approach discussed in Part I and Part II can handle such a "simple" example as they handle either temporal or resource controllability in isolation.

8.1 Motivating Example

Figure 8.1 shows the workflow modeling the STEMI guidelines. There, task durations, delays and temporal ranges are in minutes. The workflow starts with a patient evaluation (`PatEv`), in which an `EmergencyDoctor` establishes if the patient is in need of immediate medical attention. Should this be the case the Boolean variable `urgent` is set to $\top$ (i.e., true), and $\bot$ (i.e., false), otherwise. `PatEv` lasts minimum 5 and maximum 10 minutes. After minimum 1 and within 3 minutes since `PatEv` has finished, the workflow management system executes the conditional split connector (i.e., the first diamond component encountered).

The conditional split connector splits the flow of the execution in order to decide in which branch the execution must continue according to what truth value `urgent` has been assigned. Once the branch to follow has become known, the first task in it starts after 1 and within 5 minutes since the join connector has terminated. If the patient is urgent, a surgery intervention (`SurInt`) takes place. As soon as the intervention has finished, a `Surgeon` has to manage the patient's stay in the Intensive Care Unit (`ICUStayM`). This task lasts at least 2 minutes and at most 4. In addition, following the *Temporal Separation of Duties (TSoD)* policy,

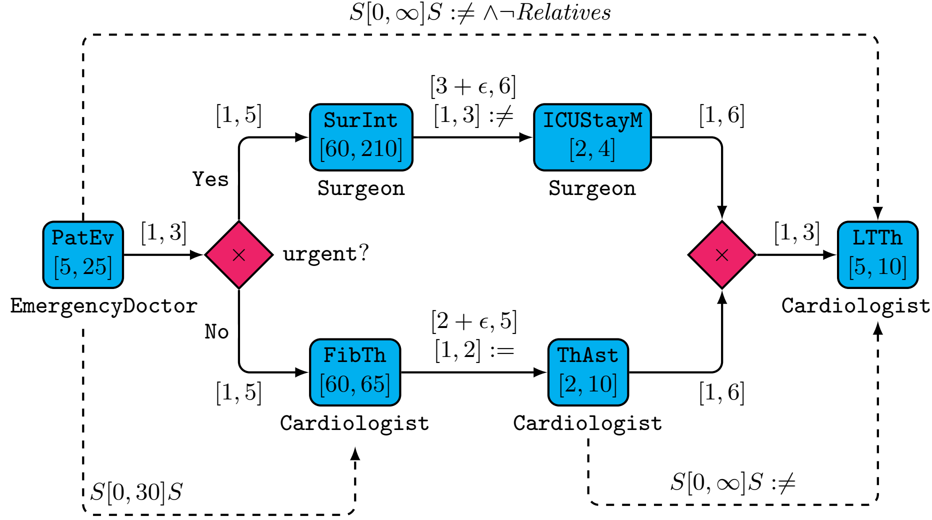


Fig. 8.1: Official STEMI guidelines (excerpt). I model TSoD as $[1, 3] :\neq$, TBoD as $[1, 2] :=$. Ranges $[x, y]$ and $[x + \epsilon, y]$ model all durations t s.t. $x \leq t \leq y$ and $x < t \leq y$, respectively.

whenever `ICUStayM` starts within 3 minutes since the end of `SurInt`, the `Surgeons` executing the two tasks must be different. TSoD does not apply if this temporal distance is greater than 3.

If the patient is not urgent, a fibrinolytic therapy (`FibTh`) takes place. This task lasts at least 60 and at most 65 minutes, and according to the guidelines, it has to start within 30 minutes since `PatEv` has started. After `FibTh` has terminated, a therapy assessment (`ThAst`) starts with minimal and maximal allowed durations of 2 and 10 minutes, respectively. Following the *Temporal Binding Of Duties (TBoD)* policy, if the start of `ThAst` is within 2 minutes since the end of `FibTh`, then the same `Cardiologist` must execute the two tasks. TBoD does not apply if this temporal distance is greater than 2 minutes.

Regardless of the chosen branch, after the last task in that branch has finished, a long term therapy assessment (`LTTh`) starts. The minimal and maximal durations for this task are 5 and 10 minutes, respectively. A security policy requires that the `Cardiologist` executing `LTTh` must be (i) different from and not a relative of the one who executed `PatEv`, and different from the one who executed `ThAst` if the patient was not urgent.

I consider a *role-based access control environment* in which `EmergencyDoctor` contains the users `john` and `lara`, `Surgeon` contains `bob`, `tom` and `sara`, and `Cardiologist` contains `lara`, `kate` and `rick`.

8.2 Syntax

An *Access-Controlled Temporal Network (ACTN)* extends a CSTNU by adding a set of users $\mathcal{U}$ and turning $\mathcal{C}$ into a set of sets each one containing (possibly several

disjunctive) *authorization constraints*. Users are in charge of executing time points, whereas authorization constraints say which users are authorized to execute the time points they connect.

Given a finite set $\mathcal{R} = \{\rho_1, \ldots, \rho_n\}$ of relations, an *authorization policy* α is any conjunction of relations drawn from $\mathcal{R}$, each one appearing as ρ or $\neg\rho$ if different from the neutral relation $*$ (i.e., that allowing all tuples). If a relation appears as ρ (respectively, $\neg\rho$), then we say that ρ is *positive* (respectively, *negative*). The neutral authorization policy, i.e., that consisting of $*$ only, is denoted by $\circledast$.

A pair (u_1, u_2) *satisfies* an authorization policy α iff (u_1, u_2) satisfies each $\rho \in \alpha$ and does not satisfy each ρ such that $\neg\rho \in \alpha$. An authorization policy α_1 *entails* another authorization policy α_2 (written $\alpha_1 \Rightarrow \alpha_2$) iff α_1 contains all the relations appearing in α_2. Two authorization policies α_1 and α_2 are *consistent* if $\alpha_1 \wedge \alpha_2$ is satisfiable, and *equivalent* if both α_1 entails α_2 and α_2 entails α_1 (i.e., $\alpha_1 \Rightarrow \alpha_2$ and $\alpha_2 \Rightarrow \alpha_1$).

For example, suppose that $\mathcal{U} = \{$`john`, `lara`, `tom`, `bob`, `sara`, `rick`, `kate`$\}$ and $\mathcal{R} = \{*, \neq, \mathit{Relatives}\}$, with $\neq$ denoted as $\{(u_1, u_2) \mid u_1, u_2 \in \mathcal{U} \wedge u_1 \neq u_2\}$ and $\mathit{Relatives} = \{(\mathtt{john}, \mathtt{kate}), (\mathtt{kate}, \mathtt{john})\}$. Consider the authorization policy α defined as $\neq \wedge \neg \mathit{Relatives}$ given in Figure 8.1 between `PatEv` and `LTTh` saying that the physician giving the patient the long term therapy must be different and not a relative of the physician who did the initial evaluation. If `john` carries out the initial `PatEv`, then `lara` can execute `LTTh` as $(\mathtt{john}, \mathtt{lara})$ satisfies α. That is, $(\mathtt{john}, \mathtt{lara})$ does not satisfy $=$ nor *Relatives*.

Definition 8.1. *An* Access-Controlled Temporal Network (ACTN) *is a tuple* $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{U}, \mathit{UA}, \mathcal{R}, \mathcal{C} \rangle$*, where:*

- $\mathcal{T}, L, \mathcal{OT}, O, \mathcal{P}, \mathcal{L}$ *are the same as for a CSTNU (Definition 2.15).*
- $\mathcal{U}$ *is a non-empty finite* set of users.
- $\mathit{UA} \subseteq \mathcal{U} \times \mathcal{T}$ *is the* authorization relation. $\mathcal{A}(X) = \{u \mid (u, X) \in \mathit{UA}\}$ *is the* set of users authorized for X.
- $\mathcal{R}$ *is a finite set of relations* $\rho \subseteq \mathcal{U} \times \mathcal{U}$*, with* $* = \mathcal{U} \times \mathcal{U}$ *the neutral relation.*
- $\mathcal{C}$ *is a set of sets such that each set* $\mathcal{C}_{XY} \in \mathcal{C}$ *consists of a (possibly many) authorization constraints of the form* $(x \leq Y - X \leq y, \ell : \alpha)$[1] *where* $x, y \in \mathbb{R} \cup \pm\infty$*,* $Y, X \in \mathcal{T}$*,* $\ell \in \mathcal{P}^*$*,* α *is an authorization policy. Also, when* $|\mathcal{C}_{YX}| > 1$ *then each contained constraint represents a disjunct (only one will be considered runtime).*
- $\ell = L(X) \wedge L(Y)$ *for each* $(x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY}$*, and* $\ell, L(X), L(Y)$ *entail* $L(P?)$ *for each* $p, \neg p \in \ell, L(X)$ *and* $L(Y)$*.*
- *For each pair* $(x_1 \leq Y - X \leq y_1, \ell_1 : \alpha_1), (x_2 \leq Y - X \leq y_2, \ell_2 : \alpha_2)$ *belonging to the* same *set* $\mathcal{C}_{XY}$*, if* $[x_1, y_1] = [x_2, y_2]$*, then* $\alpha_1 \neq \alpha_2$ *(note that* $\ell_1 = \ell_2$*).*
- *For each* $(A, x, y, C) \in \mathcal{L}$*,* A *is non-contingent and* $\mathcal{A}(A) = \mathcal{A}(C)$*.*
- *For each* $(A_1, x_1, y_1, C_1), (A_2, x_2, y_2, C_2) \in \mathcal{L}$*,* $A_1 \not\equiv A_2$ *and* $C_1 \not\equiv C_2$*.*

The *ACTN-graph* extends the CSTNU-graph by labeling requirement links (recall the notation used in Section 5.7.4) $X \rightarrow Y$ by (possibly many) labels of the form $[x, y], \ell : \alpha$ each one corresponding to the authorization con-

[1] $(x \leq Y - X \leq y, \ell : \alpha)$ shortens $(Y - X \leq y, \ell : \alpha)$ and $(X - Y \leq -x, \ell : \alpha)$. I assume $\mathcal{C}_{XY} = \mathcal{C}_{YX}$.

straint $(x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY}$. Figure 8.2 shows the ACTN modeling the workflow in Figure 8.1 considering access control. `PatEv` is mapped to $(A_1, 5, 25, C_1)$ and its authorized users are $\mathcal{A}(A_1) = \mathcal{A}(C_1) = \{\texttt{john}, \texttt{lara}\}$ who belong to the role `EmergencyDoctor`. `SurInt` and `ICUStayM` are mapped to $(A_2, 60, 210, C_2)$ and $(A_3, 2, 4, C_3)$, respectively, and their authorized users are $\mathcal{A}(A_2) = \mathcal{A}(C_2) = \mathcal{A}(A_3) = \mathcal{A}(C_3) = \{\texttt{bob}, \texttt{sara}, \texttt{tom}\}$, who belong to the role `Surgeon`. `FibTh`, `ThAst` and `LTTh` are mapped to $(A_4, 60, 65, C_4)$, $(A_5, 2, 10, C_5)$, and $(A_6, 5, 10, C_6)$, respectively, and their authorized users are $\mathcal{A}(A_4) = \mathcal{A}(C_4) = \mathcal{A}(A_5) = \mathcal{A}(C_5) = \mathcal{A}(A_6) = \mathcal{A}(C_6) = \{\texttt{lara}, \texttt{kate}, \texttt{rick}\}$, who belong to the role `Cardiologist` (note that `lara` is a specialist in two medical fields, so she belongs to two roles).

I model the conditional split and join connectors (diamonds) by means of $U?$ and E, and I connect all the components by means of authorization constraints, which also impose relative constraints among two non-sequential tasks specifying authorization policies. For example, I model the TSoD between tasks `SurInt` and `ICUStayM` by means of the authorization constraint[2] $C_2 \xrightarrow{[1,3], u:\neq} A_3$ saying that once C_2 has been executed (i.e., `SurInt` has been completed), say by `bob` at time $t = 90$, then either `tom` or `sara` can execute A_3 (i.e., start `ICUStayM`) from time 91 to 96. The second authorization constraint $C_2 \xrightarrow{[3+\epsilon,6], u:\circledast} A_3$ ensures that TSoD does not apply otherwise allowing all users to execute A_3 from time $93 + \epsilon$ to 96 (I use ϵ to make sure that no overlap exists between $[1, 3]$ and $[3 + \epsilon, 6]$).

8.3 Semantics

I give the execution semantics of ACTNs in terms of *real-time execution decisions (RTEDs)*. Intuitively, an RTED is a decision to commit some users(s) to executing a set of time points, or a decision to wait for something to happen or an instantaneous reaction.

In what follows, I adapted the RTEDs for CSTNUs given in [26] by also renaming a few symbols and adding details for explanatory purposes. For an ACTN, the *controller* (`ctrl`) seeks a strategy for committing available users to executing all relevant control time points (i.e., the part under control) such that all relevant constraints in $\mathcal{C}$ will eventually be satisfied no matter what durations the *environment* (`env`) chooses for contingent links and truth values for propositions. Thus, RTEDs exist for both `ctrl` and `env`[3]. I study the interplay between these two RTEDs in terms of partial and full outcomes, which determine how the state of the whole system evolves.

I assume that each contingent link (A, x, y, C) is (i) tied to one range $[x, y]$ only, and (ii) executed by one user only who, once he has executed A, remains blocked until the execution of C. I leave as future work the generalization of the approach to handle situations in which users are given less (or more) time for the same task. However, I point out that most of these assumptions adapt straightforwardly to

---

[2] $C_2 \xrightarrow{[1,3], u:\neq} A_3$ models $(1 \leq A_3 - C_2 \leq 3, u : \neq) \in \mathcal{C}_{A_3 C_2}$.

[3] Here, I use `ctrl` and `env` instead of `Player1` and `Player2`.

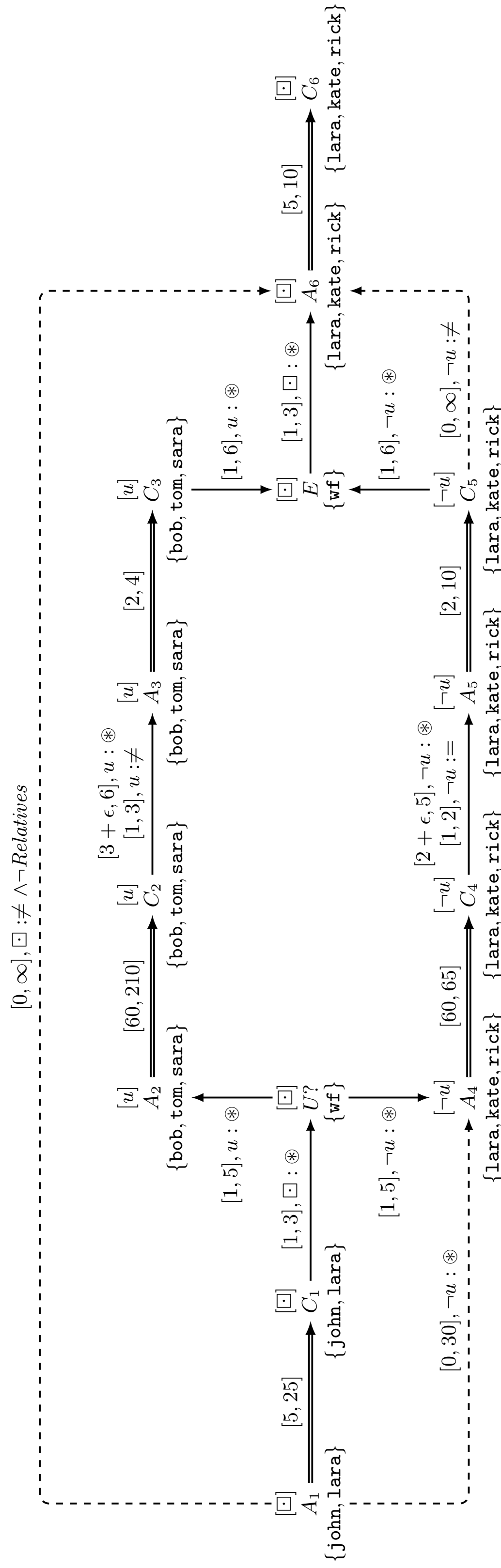


Fig. 8.2: ACTN modeling the workflow given in Figure 8.1.

sensitive workflows such as our running example where we do not want a `Surgeon` to be multi-tasking putting at risk patients' life.

The *set of all available users* is $Avail \subseteq \mathcal{U}$, so that $u \in Avail$ means that u is available and busy otherwise. The initial state is $Avail = \mathcal{U}$ meaning that all users are available. Blocking users guarantees that the same user can be committed for more than one task in a parallel block ensuring this user will be always executing one task at a time.

Definition 8.2. *A* partial schedule *for an ACTN is a pair*

$$\mathcal{PS} = (OccTP, KnownProp)$$

where OccTP is a set of triples (u, X, k) *meaning that user u executed time point X at time k, and KnownProp is a set of pairs* (p, b) *meaning that p has been assigned* $b \in \{\top, \bot\}$.

I represent the current partial scenario *(i.e., the scenario being generated upon the execution of the observation time points) by means of the label* ℓ_{cps} *consisting of the conjunction of the already known literals. That is,*

$$\ell_{cps} = \{p \mid (p, \top) \in KnownProp\} \cup \{\neg q \mid (q, \bot) \in KnownProp\}$$

$Executed(OccTP) = \{X \mid (u, X, k) \in OccTP\}$ *is the* set of executed time points *and* $Instants(OccTP) = \{k \mid (u, X, k) \in OccTP\}$ *is the* set of time instants in which the time points in $Executed(OccTP)$ were executed*. For each* $X \in Executed(OccTP)$*, I represent the time instant k in which X was executed as* $time(X)$ *and the user who executed it as* $user(X)$.

I represent the time instant of the last executed time point as

$$last = \max\{v \mid v \in Instants(OccTP)\}$$

if $OccTP = \emptyset$*, then* $last = -\infty$. $\mathcal{PS}$ *is called* locally consistent *if all* $(u, X, k) \in OccTP$ *satisfy the constraints in* $\mathcal{C}$*. The set of all possible partial schedules is represented by* $\mathcal{PS}^*$.

The solution generated *dynamically* (i.e., the real-time execution decisions adding triples (u, X, k) to $\mathcal{PS}$) answers the workflow satisfiability problem without needing to do such an assignment before starting the execution of the workflow. If the network is proven to be dynamically controllable, we are guaranteed that a dynamic execution strategy generating such an assignment runtime, depending on what is going on, always exists.

Definition 8.3. *An* RTED for `ctrl` *has two forms:* `wait` *or* $(t, ControlTP)$.

- $\Delta_{\texttt{ctrl}} = \texttt{wait}$ *is applicable only if a contingent time point has been activated (i.e., the activation time point A has been executed but the related contingent C has not).*
- $\Delta_{\texttt{ctrl}} = (t, ControlTP)$ *represents the conditional constraint "if* `env` *does nothing before time t, then for each pair* $(u, X) \in ControlTP$*, commit the user u to execute the time point X at time t". Such an RTED is applicable iff* $t > last$*, ControlTP is a non empty set of pairs of available users and unexecuted time*

points. That is, for each $(u, X) \in \mathit{ControlTP}$, $X \notin \mathit{Executed}(\mathit{OccTP})$ *and* $u \in \mathit{Avail}$. *Finally, for each* $(u_1, X_1), (u_2, X_2) \in \mathit{ControlTP}$, *if* X_1 *and* X_2 *are (different) activation points, then* $u_1 \neq u_2$. *If* X_1 *is an activation point and* X_2 *is not, then we execute* X_2 *first. This is because activation points block users.*

An RTED for the environment `env` *(in symbols,* $\Delta_{\mathtt{env}}$*) has two forms:* `wait` *or* $(t, \mathit{ContingentTP})$.

- $\Delta_{\mathtt{env}} = \mathtt{wait}$ *is applicable only if no contingent point* C *has been activated.*
- $\Delta_{\mathtt{env}} = (t, \mathit{ContingentTP})$ *represents the conditional constraints "if* `ctrl` *does nothing before or at time* t*, then execute the time points in* $\mathit{ContingentTP}$ *at time* t*". Such a decision is applicable iff* $t > \mathit{last}$, $\mathit{ContingentTP}$ *is a non empty subset of unexecuted contingent points whose ranges of allowed durations contain* t.

The sets of all RTEDs for `ctrl` and `env` *are represented by* $\Delta^*_{\mathtt{ctrl}}$ *and* $\Delta^*_{\mathtt{env}}$, *respectively.*

In other words, $\Delta_{\mathtt{ctrl}}$ deals with the parts under control: users and non-contingent time points. $\Delta_{\mathtt{env}}$ deals with the parts out of control: contingent time points and truth value assignments.

Definition 8.4. *Let* $\mathcal{PS} = (\mathit{OccTP}, \mathit{KnownProp})$ *be a partial schedule in which at least one contingent time point* C *has been activated and is allowed to be executed at* last. *An* instantaneous reaction $\mathcal{IR}$ *is a decision*

(1) to execute a set $\mathcal{I}_C$ *of such time points at time* last, *or*
(2) to assign truth values for each proposition associated to the observation time points in OccTP *that has not been assigned yet (I model such an assignment as a set* $\mathcal{I}_B$ *of pairs* (p, b) *with* $p \in \mathcal{P}$ *and* $b \in \{\top, \bot\}$*), or*
(3) to do both actions.

I represent an instantaneous reaction $\mathcal{IR}$ *as a pair* $(\mathcal{I}_C, \mathcal{I}_B)$. *The* set of all instantaneous reactions *is represented by* $\mathcal{IR}^*$.

If *last* happens to be the last possible time at which a currently-activated contingent time point C can execute, then the instantaneous reaction must include C. Similarly, if t is the time in which an observation time point $P?$ was executed, the instantaneous reaction must include a truth value assignment for p.

Of course, `env` can carry out more than one $\mathcal{IR}$. I now define how $\Delta_{\mathtt{ctrl}}$ and $\Delta_{\mathtt{env}}$ are handled. Since both players can either wait or conditionally commit to executing a set of time points, there are four possible cases (I point out that a `wait` decision is not applicable for both `ctrl` and `env` simultaneously).

Definition 8.5. *The* partial outcome *of* $\Delta_{\mathtt{ctrl}}$ *and* $\Delta_{\mathtt{env}}$ *are modeled as* $\mathcal{O}_p(\mathit{OccTP}, \Delta_{\mathtt{ctrl}}, \Delta_{\mathtt{env}})$ *neglecting any* $\mathcal{IR}$. *There are four possible cases.*

(1) $\mathcal{O}_p(\mathit{OccTP}, \mathtt{wait}, (t, \mathit{ContingentTP})) = \mathit{OccTP} \cup \{(\mathit{user}(A), C, t) \mid C \in \mathit{ContingentTP}\}$ *and for each* $C \in \mathit{ContingentTP}$, $\mathit{Avail} = \mathit{Avail} \cup \mathit{user}(C)$, *where* A *is the activation time point that activated* C.

(2) $\mathcal{O}_p(OccTP, (t_1, ControlTP), (t_2, ContingentTP)) = OccTP \cup \{(user(A), C, t_2) \mid C \in ContingentTP\}$ *if* $t_2 < t_1$ *and for each* $C \in ContingentTP$, $Avail = Avail \cup user(C)$*, where* A *is the activation time point that activated* C*.*

(3) $\mathcal{O}_p(OccTP, (t, ControlTP), \texttt{wait}) = OccTP \cup \{(u, X, t) \mid (u, X) \in ControlTP\}$ *and for each* $(u, A) \in ControlTP$, $Avail = Avail \setminus u$*.*

(4) $\mathcal{O}_p(OccTP, (t_1, ControlTP), (t_2, ContingentTP)) = OccTP \cup \{(u, X, t_1) \mid (u, X) \in ControlTP\}$ *if* $t_1 \le t_2$ *and for each* $(u, A) \in ControlTP$, $Avail = Avail \setminus u$*.*

For example, $\mathcal{O}_p(OccTP, (0, \{(\texttt{john}, A_1)\}), \texttt{wait}) = OccTP \cup \{(\texttt{john}, A_1, 0)\}$.

(1) says that `env` can execute the time points in *ContingentTP* at time t if `ctrl` decides to do nothing at that time. (2) says that `env` can execute the time points in *ContingentTP* if he decided to do so before `ctrl` executes his. (3) is similar to (1) but with respect to `ctrl`. (4) says that when `ctrl` decides to execute a set of time points before or at the same time of those `env` has decided to execute, `ctrl` moves first to allow `env` to react instantaneously (Definition 8.6).

Definition 8.6. *The* full outcome *of* $\Delta_{\texttt{ctrl}}$ *and* $\Delta_{\texttt{env}}$ *are modeled as* $\mathcal{O}(OccTP, \Delta_{\texttt{ctrl}}, \Delta_{\texttt{env}}, \mathcal{IR})$ *and defined as I did for* $\mathcal{O}_p(OccTP, \texttt{wait}, (t, ContingentTP))$ *except that in cases (3) and (4) OccTP is augmented with* $\{(C, t) \mid C \in \mathcal{I}_C\}$ *(applicable if C has been activated), and KnownProp is augmented with* $\{(p, b) \mid (p, b) \in \mathcal{I}_B\}$ *(applicable if P? has been executed). Either way* $t = last$*.*

The full outcome says how the state of the system (i.e., $\mathcal{PS}$) evolves according to the interplay of `ctrl`'s and `env`'s RTEDs.

Definition 8.7. *An* RTED-based strategy for `ctrl` *is a mapping* $\sigma_{\texttt{ctrl}} : \mathcal{PS}^* \to \Delta^*_{\texttt{ctrl}}$ *from partial schedules to RTEDs. An* RTED-based strategy for `env` *is a pair of mappings* $\sigma_{\texttt{env}} = (\mu_{\Delta_{\texttt{env}}}, \mu_{IR})$*, where* $\mu_{\Delta_{\texttt{env}}} : \mathcal{PS}^* \to \Delta^*_{\texttt{ctrl}}$ *is a mapping from partial schedules to RTEDs, and* $\mu_{IR} : \mathcal{PS}^* \to \mathcal{IR}^*$ *is a mapping from partial schedules to instantaneous reactions.*

Definition 8.8. *The* one-step outcome *of the game modeling the execution of the network by* `ctrl` *and* `env` *is defined as*

$$\mathcal{O}^1(OccTP, \sigma_{\texttt{ctrl}}, \sigma_{\texttt{env}}) = \mathcal{O}(OccTP, \sigma_{\texttt{ctrl}}(OccTP), \mu_{\Delta_{\texttt{env}}}(OccTP), \mu_{IR}(OccTP_p))$$

where $OccTP_p = \mathcal{O}_p(OccTP, \sigma_{\texttt{ctrl}}(OccTP), \mu_{\Delta_{\texttt{env}}}(OccTP))$*.*

The terminal outcome $\mathcal{O}^*(\sigma_{\texttt{ctrl}}, \sigma_{\texttt{env}})$ *is the complete schedule that results from the recursive definition:* $OccTP_0 = \emptyset$*,* $OccTP_{i+1} = \mathcal{O}^1(OccTP_i, \sigma_{\texttt{ctrl}}, \sigma_{\texttt{env}})$*.*

Definition 8.9. *An ACTN is DC if there exists an RTED-based strategy* $\sigma_{\texttt{ctrl}}$ *such that for all RTED-based strategies* $\sigma_{\texttt{env}}$*, the variable assignments* (u, X, k) *in the complete schedule* $\mathcal{O}^*(\sigma_{\texttt{ctrl}}, \sigma_{\texttt{env}})$ *satisfy all constraints in* $\mathcal{C}$*.*

8.4 Encoding ACTNs into TGAs

In this section, I extend the encoding given in Section 2.2.2, I use UPPAAL-TIGA as an off-the-shelf model checker and discuss a few optimizations.

Assume that $\mathcal{G}$ in Figure 8.3 is the TGA equivalent to the ACTN in Figure 8.2. The core of the TGA remains the same of that discussed in Section 2.2.2.

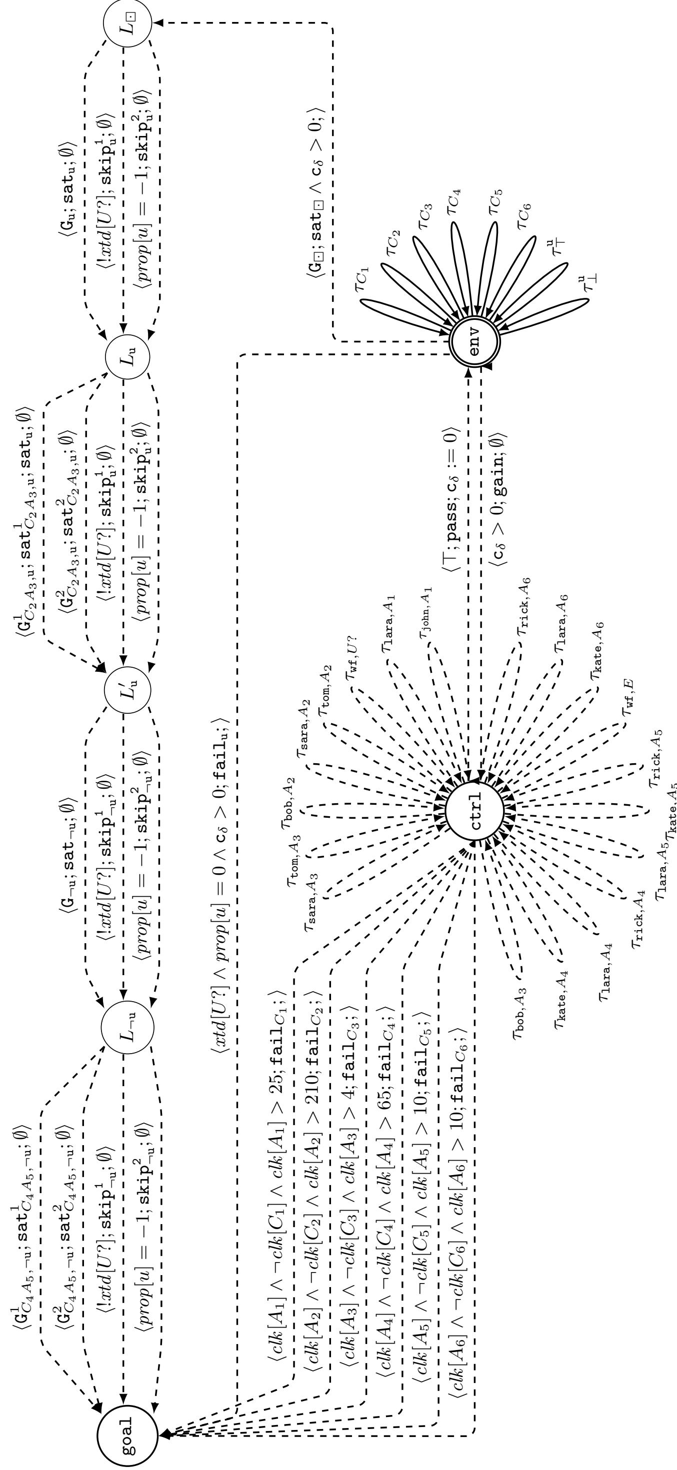


Fig. 8.3: TGA equivalent to the ACTN in Figure 8.2.

8.4.1 Internal state

I represent relations, users and their availability by extending the internal states of the TGA with a new piece of information. I assign a unique incremental integer starting from 0 to each time point, and proceed similarly for each user and proposition as I did for workflow resiliency in Section 7.2. Once again, when intended as indexes, I abuse notation and write u, X, and p meaning their assigned integers.

I model availability of users as a Boolean vector[4] Avail indexed on users. Hence, $\mathit{Avail}[u] = \top$ means that u is available and busy otherwise. The initial state of Avail consists of all elements set to $\top$ meaning that all users are available.

I keep track of *who did what* by means of a system trace vector $\mathit{SysTrace}$ whose index ranges over time points. If $\mathit{SysTrace}[X] = u$, then time point X has been executed by user u. The initial state of $\mathit{SysTrace}$ consists of all elements set to -1, meaning that all time points have not been executed yet.

For instance, if `john` has executed A_1, then $\mathit{SysTrace}[A_1] = \texttt{john}$. I do not need to keep track of *when* a time point X has been executed since its value is given by the difference between global time and its clock.

I reorganize clocks associated to time points by means of a vector clk indexed on time points, i.e., $\mathit{clk}[X]$ is the clock associated to the time point X. The vector clk does not contain clocks $\hat{\mathtt{c}}$ (representing global time) and $\mathtt{c}_\delta$ (used in the guards and updates of the transitions modeling the interplay between `ctrl` and `env`). Likewise, I reorganize Boolean variables associated to time points by means of a vector xtd indexed on time points whose elements are initialized to $\bot$. Finally, I reorganize propositions by means of an integer vector prop whose elements are initialized to 0.

8.4.2 Predecessors and transition range limitations

I adapt the enforcement of predecessor (already discussed for CSTNUD in Chapter 5) for it to support disjunctive constraints. I proceed as follows.

Definition 8.10. *Let Y be a non-contingent time point. For any $X \in \mathcal{T}$ different from Y, X is a* predecessor *of Y (in symbols $X \in \Pi(Y)$) if all authorization constraints between X and Y have a positive lower bound ($x \geq 0$) and label ℓ identical to that of Y. Formally:*

$$\Pi(Y) = \{X \mid (x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY} \wedge x \geq 0 \wedge \ell = L(Y)\}$$

If $X \in \Pi(Y)$ it follows that $Y - X \in [x_{min}, y_{max}]$, where

- $x_{min} = \min\{x \mid (x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY}\}$, *and*
- $y_{max} = \max\{y \mid (x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY}\}$

That is, if all authorization constraints between X and Y have a positive lower bound and label equal to $L(Y)$ (i.e., at least one of them must be satisfied when considering Y), then Y definitely executes after X. Moreover, once we have executed X, we can execute Y after at least the minimum among the lower bounds

[4] Uppaal-TIGA does not provide data structures for sets.

(x_{min}) and within the maximum of the upper bounds (y_{max}) of the authorization constraints in $\mathcal{C}_{XY}$, to consider all the possible disjunctions.

As an example, consider Figure 8.2. $\Pi(A_3) = \{C_2\}$ since $\mathcal{C}_{C_2A_3} = \{(1 \leq A_3 - C_2 \leq 3, u :\neq), (3 + \epsilon \leq A_3 - C_2 \leq 6, u : \circledast)\}$ and $L(A_3) = u$. Moreover, once env executes C_2, ctrl can execute A_3 such that $A_3 - C_2 \in [1, 6]$ ($x_{min} = 1$ and $y_{max} = 6$). Likewise, $\Pi(A_5) = \{C_4\}$, $A_5 - C_4 \in [1, 5]$. $\Pi(U?) = \{C_1\}$ and $U? - C_1 \in [1, 3]$. $\Pi(A_6) = \{E, A_1\}$, $A_6 - E \in [1, 3]$ and $A_6 - A_1 \in [0, \infty]$. $\Pi(A_2) = \{U?\}$ and $A_2 - U? \in [1, 5]$. $\Pi(A_4) = \{U?, A_1\}$ and $A_4 - U? \in [1, 5]$, $A_4 - A_1 \in [0, 30]$.

Instead, I cannot do much for E, since each constraint having E as a target has a label that is not entailed by $\boxdot$ (i.e., $L(E)$). That is, for $(1 \leq E - C_3 \leq 6, u : \circledast) \in \mathcal{C}_{C_3E}$ and $(1 \leq E - C_5 \leq 6, \neg u : \circledast) \in \mathcal{C}_{C_5E}$, we have that neither $L(E) \not\Rightarrow u$ nor $L(E) \not\Rightarrow \neg u$. I recall that E always executes since its label, $\boxdot$, is entailed by any scenario. But, if scenario u (respectively, $\neg u$) holds, the guard of the transition enforcing the partial order (i.e., that considering $\Pi(E)$) would become conditional depending on the current partial scenario. For E, I would generate three transitions: one for $\boxdot$ with $\Pi(E) = \emptyset$, one for u with $\Pi(E) = \{C_3\}$ and $C_3 - E \in [1, 6]$, and another for $\neg u$ with $\Pi(E) = \{C_5\}$ and $C_5 - E \in [1, 6]$.

Although such an approach risks making the encoding exponential, I should also prove that the generated transitions do not prevent any scenario from being explored by the model-checking phase. For this reason, each $\Pi(Y)$ contains a predecessor only when I am sure that $X \in \Pi(Y)$ always comes before Y (similarly, I do not consider the case "C_5 is before A_6").

Note that activation time points are trivially before their related contingent and already handled with a similar approach in the guards of the transitions executing them at env.

8.4.3 Time Point Transitions

For each user authorized for X (where $X \notin CT$), there exists an uncontrollable self-loop transition having the form:

$$(\texttt{ctrl}; \mathit{Guard}(u, X); \texttt{uExX}; \mathit{Update}(u, X, \mathit{setBusy}); \texttt{ctrl})$$

where:

- $\mathit{Guard}(u, X) = \neg \mathit{xtd}[X] \bigwedge_{p \in L(X)} (\mathit{prop}[p] = 1) \bigwedge_{\neg q \in L(X)} (\mathit{prop}[q] = -1) \wedge \mathit{Avail}[u] \bigwedge_{Y \in \Pi(X)} (\mathit{xtd}[Y] \wedge \mathit{clk}[Y] \geq x_{min} \wedge \mathit{clk}[Y] \leq y_{max})$. This function formalizes the mandatory part in the guard: X has not been executed yet, $L(X)$ is true, the authorized user u is available, all $Y \in \Pi(X)$ have been executed and $\mathit{clk}[Y] \in [x_{min}, y_{max}]$ (as discussed before).
- $\mathit{Update}(u, X, \mathit{setBusy}) = \mathit{xtd}[X] := \top$, $\mathit{clk}[X] := 0$, $\mathit{SysTrace}[X] := u$, $\mathit{Avail}[u] := \neg \mathit{setBusy}$. This function formalizes the mandatory part in the update: X has been executed by u at time $\hat{\mathtt{c}} - \mathit{clk}[X]$, and u is now busy ($\mathit{setBusy}$) if X is an activation time point.

For concreteness, consider the self loop $\tau_{\texttt{kate}, A_5} = (\texttt{ctrl}; \mathit{Guard}(\texttt{kate}, A_5); \texttt{kateExA}_5; \mathit{Update}(\texttt{kate}, A_5, \top); \texttt{ctrl}\rangle$ in Figure 8.3. It says that if the patient is not urgent, kate can start ThAst (i.e., kate executes A_5) after 1 and within 5

minutes, after `FibTh` has been completed (i.e., $xtd[C_4] \wedge clk[C_4] \geq 1 \wedge clk[C_4] \leq 5$). $\tau_{\mathtt{kate},A_5}$ along with other 18 similar self loops $\tau_{u,X}$ model the authorized executions of all non contingent time points.

For each contingent time point, I only extend the update part of the transition given in Section 2.2.2 by adding the two statements $SysTrace[C] := SysTrace[A]$, and $Avail[SysTrace[C]] := \top$. That is, first, I save that the user executing C is the same of that who executed A, and second, I release such a user. Note that the guard does not contain conditions on availability on purpose. Adding such conditions would result in `env` shrinking the range of C.

8.4.4 Game Interplay, Failing Transitions, and Winning path considering disjunctive constraints

Failing transitions and transitions regulating the game interplay and assigning the truth values to propositions remain the same as those discussed in Section 5.3 (adapted to the new array-reorganization).

The winning path is extended by adding intermediate locations to guarantee that whenever disjunctive authorization constraints exist, I always take into consideration one of them. I proceed by first introducing the set of core constraints (i.e., those I must always consider for the traditional winning path).

Definition 8.11. *Let $\ell \in \mathcal{P}^*$ be a label. The set of* core constraints *labeled by ℓ is given by:*

$$Core(\ell) = \bigcup_{\mathcal{C}_{XY} \in \mathcal{C}} \{(x \leq Y - X \leq y, \ell : \alpha) \mid |\mathcal{C}_{XY}| = 1 \wedge \\ (x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY} \wedge \ell = L(X) \wedge L(Y)\}$$

For the example I am discussing,

$$Core(u) = \{(1 \leq A_2 - U? \leq 5, u : \circledast), (1 \leq E - C_3 \leq 6, u : \circledast)\}$$

Note that I do not take into consideration

$$\mathcal{C}_{C_2 A_3} = \{(1 \leq C_3 - C_2 \leq 3, u : \neq), (3 + \epsilon \leq C_3 - C_2 \leq 6, u : \circledast)\}$$

as $|\mathcal{C}_{C_2 A_3}| = 2$.

$\mathtt{G_u}$ in Figure 8.3 verifies both that all time points labeled by u have been executed and that the core constraints and their related authorization policies they express are satisfied. $\mathtt{skip}_1^{\mathtt{u}}$ and $\mathtt{skip}_2^{\mathtt{u}}$ are the same as those discussed for classic CSTNUDs in Section 5.3.

After generating the location and the set of transitions for the core constraints I now deal with the disjunctive authorization constraints. I first compute the set of non-core constraints (dual to the set core), i.e., the set containing all the disjunctions with respect to a given label ℓ. Then, I generate the locations and transitions to verify that the run satisfies *at least* one authorization constraint for each disjunction avoiding the combinatorial explosion.

Definition 8.12. *Let $\ell \in \mathcal{P}^*$ be a label. The set of* non-core constraints *labeled by ℓ for each $\mathcal{C}_{XY}$ is given by:*

$$NonCore(\mathcal{C}_{XY}, \ell) = \{(x \leq Y - X \leq y, \ell : \alpha) \mid |\mathcal{C}_{XY}| > 1 \wedge \\ (x \leq Y - X \leq y, \ell : \alpha) \in \mathcal{C}_{XY} \wedge \ell = L(X) \wedge L(Y)\}$$

For this example,

$$NonCore(\mathcal{C}_{A_3C_2}, u) = \{(1 \leq A_3 - C_2 \leq 3, u :\neq), (3 + \epsilon \leq A_3 - C_2 \leq 6, u : \circledast)\}$$

That is, either A_3 is executed such that $A_3 - C_2 \in [1, 3]$ and TSoD ($\neq$) must hold, or A_3 is executed such that $A_3 - C_2 \in [3 + \epsilon, 6]$ and any user can do so ($\circledast$).

To handle the disjunctions with respect to the label ℓ for each $NonCore(\mathcal{C}_{XY}, \ell)$, I create an intermediate location $L_{\mathcal{C}_{XY},\ell}$ to avoid generating an exponential number of transitions. For each $L_{\mathcal{C}_{XY},\ell}$, there is a set of transitions containing the same `skip` transitions as those for L_ℓ and there is a `sat` transition for each disjunct in $NonCore(\mathcal{C}_{XY}, \ell)$. Concretely, for $NonCore(\mathcal{C}_{A_3C_2}, u)$, I generate $L_{\mathcal{C}_{C_2A_3},u}$ (shortened as $L'_{\mathtt{u}}$ in Figure 8.3), $\mathtt{sat}^1_{C_2A_3,\mathtt{u}}$ (verifying the first disjunction) and $\mathtt{sat}^2_{C_2A_3,\mathtt{u}}$ (verifying the second one).

Likewise, I generate $L_{\mathcal{C}_{C_4A_5},\neg u}$ (i.e., `goal`), $\mathtt{sat}^1_{C_4A_5,\neg\mathtt{u}}$ and $\mathtt{sat}^2_{C_4A_5,\neg\mathtt{u}}$ for $NonCore(\mathcal{C}_{C_4A_5}, \neg u)$.

The ACTN in Figure 8.2 is dynamically controllable. One of the possible executions is the following. `ctrl` starts the workflow in Figure 8.1 by assigning `PatEv` to `john`. As soon as `john` finishes (i.e., `env` executes C_1), `ctrl` commits the workflow management system `wf` to execute the conditional split connector (modeled by U?). If the patient is urgent (i.e., if u is assigned $\top$), then `SurInt` is assigned to `tom` and exactly 1 minute after `tom` is done, `ICUStayM` is assigned to `bob` (since a TSoD must hold). If the patient is not urgent, `FibTh` and `ThAst` are both carried out by `lara` because the start of `ThAst` occurs one minute after `FibTh` and thus a TBoD must hold. Regardless of which branch has been taken, `ctrl` commits `wf` to execute the join connector 1 minute after the last task of the chosen branch terminated. Finally, `ctrl` commits `rick`, who is different from and not related to `lara` or `john`, to executing `LTTh` 1 minute after the join connector ended. This execution strategy satisfies all authorization constraints.

8.5 Correctness and complexity of the encoding

In this section I discuss the correctness and complexity of the encoding.

Theorem 8.1. *Encoding ACTNs into TGAs has polynomial-time complexity.*

Proof. The main components having a role in the complexity analysis of the encoding of an ACTN are: (i) time points, (ii) authorized users for each control time point, (iii) authorization constraints, and (iv) different labels in the ACTN.

For each control time point, there is a self-loop transition for any authorized user. These transitions contain in their guard the scenario in which the time point has to be executed and may contain additional conditions on the predecessors.

The complexity of finding the predecessors for a time point X is linear in the number of all constraints (Section 5.3). Thus, the generation of these transitions is a polynomial-time task.

Transitions modeling the execution of the contingent time points and assigning truth values are the same as those given for CSTNUs (I just extend the update part), and they are not even replicated for the authorized users. Fail transitions and transitions regulating the game interplay remain exactly the same as well. Thus, the time complexity of the generation of this set of transitions remains polynomial.

I am left to prove that the generation of the winning path is a polynomial-time task too. The main problem is that such a path consists of intermediate locations that must allow one to consider all possible combinations of the disjunctions expressed as authorization constraints. The generation of the winning path is the same of that for CSTNUs (augmented with the checks for the authorization policies) when considering the set of core constraints (polynomial). When considering the non core authorization constraints I create as many intermediate locations as the number of disjunctions. However, I remark that my encoding does not generate any combination of the `sat` transitions nor a combination of the locations. The model-checking phase will take care of exploring them all. Thus, the generation of the winning path has linear complexity in the number of the different labels with respect to the number of the possible disjunctions.

Since all operations have polynomial complexity, the overall complexity is polynomial as well.

Theorem 8.2. *Encoding ACTNs into TGAs correctly captures the execution semantics given in Section 8.3.*

Proof. I show that any sequence of partial schedules that can be generated for any ACTN according to the execution semantics given in Section 8.3 corresponds to a run for the equivalent TGA that can be generated by following its transitions according to the classic TGA semantics. I extend the proof of correctness given in [26] to accommodate users and authorization constraints. The proof is by induction.

Each respectful partial schedule that can be generated for the ACTN corresponds to a state of the TGA in which: the location is `env`, $\mathtt{c}_\delta = 0$, $last = \hat{\mathtt{c}}$, and for each executed time point X, $time(X) = \hat{\mathtt{c}} - clk[X]$, $xtd[X] = \top$, $user(X) = SysTrace[X]$, whereas $time(X) = \hat{\mathtt{c}}$, $xtd[X] = \bot$ and $user(X)$ if X has not been executed yet. If $P?$ is an observation time point, then $KnownProp$ contains (p, b) for $b \in \{\top, \bot\}$ if $P?$ has been executed, and does not contain it otherwise. Also, if some activation point has been executed by the user u and the related contingent has not, then $Avail[u] = \bot$, else $Avail[u] = \top$. Note that $Avail[u] = \top$ and $Avail[u] = \bot$ (in the TGA) mean $u \in Avail$ and $u \notin Avail$ in the execution semantics given in Section 8.3 (where $Avail$ is a set).

Base case. The initial $\mathcal{PS}$ corresponds to the initial state of the TGA in which the location is `env`, $\hat{\mathtt{c}} = 0$ all clocks $clk[X] = 0$, all $xtd[X] = \bot$, all $SysTrace[X] = -1$, all $prop[p] = 0$, and $Avail[u] = \top$. This partial schedule is trivially consistent.

Inductive step. Suppose that $\mathcal{PS}$ is a (locally) consistent partial schedule that can be generated according to the execution semantics for ACTNs, and that

$\mathcal{PS}$ satisfies the invariant that I required of each locally consistent partial schedule at the beginning of this proof.

Let θ be the corresponding state of the TGA. Since $\mathtt{c}_\delta = 0$, the only transitions that are immediately enabled are those handling contingent time point executions and truth value assignments. These transitions, if taken, correspond to `env`'s instantaneous reactions, in which a set of one or more contingent time-points can be executed simultaneously or some proposition can be assigned a value. Suppose that `env` does not take any transition when $\mathtt{c}_\delta = 0$. As soon as $\mathtt{c}_\delta > 0$, both `ctrl` and `env` have transitions that they could take at any time with respect to the enforced condition over the predecessors. For example, `env` might decide to execute one or more contingent time-points $C_1, \ldots, C_n$ when $\mathtt{c}_\delta = 3$. That would correspond to $\Delta_{\mathtt{env}} = (k, \{C_1, \ldots, C_n\})$, where $k = \mathit{last} + 3$.

Since each time `env` takes a transition $\mathtt{c}_\delta$ is reset to 0, `ctrl` is unable to interrupt `env` while `env` is executing contingent time points and assigning truth values to propositions. Thus, at those time instants $\Delta_{\mathtt{ctrl}} = \mathtt{wait}$ and the resulting outcomes are exactly the cases 1-2 of Definition 8.5. The guard of `env`'s transition, enforcing the duration bounds for a contingent link (A, x, y, C), ensures that the resulting partial schedule is respectful as C can only be executed such that $C - A \in [x, y]$ (analogous to predecessors). Likewise, for a truth value assignment the fail transition that `ctrl` can take (if $\delta > 0$) ensures that `env` can assign a truth value to a proposition instantaneously after the execution of the observation time point (otherwise, `ctrl` could trivially move to `goal`).

Also, when `env`'s sequence of "simultaneous" transitions completes, $\hat{\mathtt{c}}$ equals the time of the most recent execution (e.g., $\mathit{last} + 3$). In addition, for each newly executed time-point C, the clock $\mathit{clk}[C]$ is reset (ensuring that $\hat{\mathtt{c}} - \mathit{clk}[C]$ equals the execution time of C), $\mathit{xtd}[C]$ is set to $\top$, $\mathit{SysTrace}[C]$ is set to $\mathit{SysTrace}[A]$ (ensuring that the user executing A is the same of that executing C), and $\mathit{Avail}[\mathit{SysTrace}[C]] = \top$ (ensuring that user u is free when C executes). Since $\mathit{clk}[C]$ is reset only once, $\hat{\mathtt{c}} - \mathit{clk}[C]$ remains fixed forever.

Instead, suppose that `ctrl` has decided to commit a set of users to execute a set of control time points before `env` executes his, say at time $\mathit{last} + 2$. This situation results in `ctrl` taking the `gain` transition to take back control and then, once in its location, instantaneously commit the users to execute the time points at that time, blocking all users executing an activation time point, and immediately returning to the `env` location by means of the `pass` transition. Since the location of `ctrl` is urgent, $\hat{\mathtt{c}} = \mathit{last} + 2$ when the `pass` transition is taken. This sequence of transitions corresponds to the partial outcome in Definition 8.5 (cases 3-4) where $\Delta_{\mathtt{ctrl}} = (t, \{(u_1, X_1), \ldots, (u_n, X_n)\})$, $t = \mathit{last} + 2$, and for each $(u, X) \in \mathit{ControlTP}$ (of $\Delta_{\mathtt{ctrl}}$), $u \in \mathcal{A}(X)$. Moreover, if `env` chooses to instantaneously execute some contingent time point at the same time $\mathit{last}+2$, that will correspond to an instantaneous reaction.

Finally, if at time last, `ctrl` and `env` both decide to execute some time points at time $\mathit{last}+1$, then the ACTN semantics (inheriting the CSTNU semantics) ensures that `ctrl`'s time points are executed first, and that `env` is able to instantaneously react if he decides to do so (equivalent to `ctrl`'s transitions having priority over `env`'s). As soon as the execution returns to the location of `env`, $\hat{\mathtt{c}}$ will still be $\mathit{last} + 1$ (because, again, time has not elapsed at `ctrl`). Since, in all cases, the

resulting state of the TGA satisfies the desired invariant property, the result is proven.

Theorem 8.3. *Let* $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ *be any ACTN,* $\mathcal{G}$ *be the encoding of* $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$*, and* $\sigma_{\mathcal{G}}$ *be a winning TGA counter-strategy for* `ctrl`*. Then there is an equivalent RTED-based strategy* $\sigma_{\texttt{ctrl}}$ *for* `ctrl` *that will ensure the satisfaction of all authorization constraints in* $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$ *whatever the contingent durations and truth value assignments.*

Proof. If $\langle \mathcal{T}, \mathcal{OT}, \mathcal{P}, O, L, \mathcal{L}, \mathcal{C} \rangle$, $\mathcal{G}$, $\sigma_{\mathcal{G}}$ are as assumed, then $\sigma_{\mathcal{G}} \colon Loc \times V \to Act \cup \texttt{wait}$, with Act the set of `ctrl`'s actions (equivalently the set of uncontrollable transitions) and V abstracts the internal state of the TGA.

Suppose the TGA has just got into the state $(\texttt{env}, V)$. As I have already said, for any time point X, associated clock $clk[X]$ and Boolean variable $xtd[X]$, we have: $xtd[X] = \bot$, $clk[X] = \hat{\texttt{c}}$, and $SysTrace[X] = -1$ before X executes, and $xtd[X] = \top$, $clk[X] < \hat{\texttt{c}}$, and $SysTrace[X] = u$ after X executed. For each observation time point $P?$, the associated proposition modeled by $prop[p]$ is 0 (i.e., unknown) before $P?$ executes, and either 1 (i.e., $\top$) or -1 (i.e., $\bot$) after $P?$ executed. Thus, V specifies a partial schedule.

Now, suppose that $last < \hat{\texttt{c}}$ (i.e., that some positive time has elapsed since the last execution event in $\mathcal{PS}$). If nothing has happened, it means that there has been a sequence of `gain` and `pass` transitions going back and forth between `env`'s and `ctrl`'s locations. In such a loop, `ctrl` has not executed any control point, and `env` has just waited. Let $(\texttt{env}, V')$ be the state immediately preceding such loop. Then, for some positive $\epsilon > 0$, all the clocks in V equal those in V' plus ϵ, and by construction, $last$ refers to the clocks in V'. I abuse notation and write $V + k$ meaning that all values of the clocks in V are augmented by k. Next, let $d = \min\{d \mid \sigma_{\mathcal{G}}(\texttt{env}, V' + d) \neq \texttt{wait} \wedge \sigma_{\mathcal{G}}(\texttt{env}, V' + d) \neq \texttt{pass}\}$ be the minimum time that can elapse from V before the strategy $\sigma_{\mathcal{G}}$ recommends a transition different from `gain` and `pass`, and let $V_0 = V_0 + d$. The unique sequence of execution transitions at `ctrl` is $\tau_1 = \sigma_{\mathcal{G}}(\texttt{ctrl}, V_0), \ldots, \tau_n = \sigma_{\mathcal{G}}(\texttt{ctrl}, V_n)$, where each $V_{i+1} = V_i$, except for the $clk[X]$ with X the time point executed by τ_i. The termination of this sequence of transitions is guaranteed since time points are finite and can only be executed once. If τ_n is the last execution transition, then $\texttt{pass} = \sigma_{\mathcal{G}}(\texttt{ctrl}, V_n)$. That transition leads back to the state (env, V_n), where V_n is the same as V_0, except that the clocks for the time points executed by the transitions, $\tau_1, \ldots, \tau_n$, are all 0 in V_n.

Next, let t be the time at which $\sigma_{\mathcal{G}}$ recommends `ctrl` a non-trivial transition, and $ControlTP$ be the set of time-points corresponding to the execution transitions, $\tau_1, \ldots, \tau_n$. Then $(t, ControlTP)$ is a $\Delta_{\texttt{ctrl}}$ corresponding to what the strategy recommends at $(\texttt{env}, V_0)$. Note that `env` may decide to instantaneously react by executing some contingent points at time t too, an outcome that is prevented by the execution semantics for ACTNs (Definition 8.5, cases 3-4). Finally, `env` may decide to intervene before time t arrives, by executing one or more contingent time-points and effectively generating a new partial schedule $\mathcal{PS}'$. In that case, the same procedure could be applied to $\mathcal{PS}'$ to generate an appropriate $\Delta_{\texttt{ctrl}}$. Since the guard on the transition from `env` to `ctrl` requires a positive time delay, that $\Delta_{\texttt{ctrl}}$ is properly prohibited from any kind of instantaneous reaction (by

ctrl). This procedure gives a mapping from any (env, V) state that is reachable following $\sigma_{\mathcal{G}}$. The sequences of partial schedules generated by following the RTEDs correspond to runs that can be produced by $\sigma_{\mathcal{G}}$. Thus, the complete schedules generated by the RTEDs satisfy all authorization constraints assuming that env respects the bounds on all contingent links and assigns instantaneously all truth values to the propositions.

8.6 Conclusions

I defined *access controlled temporal networks (ACTNs)* as an extension of CSTNUs in order to take into consideration users and (temporal conditional) authorization constraints simultaneously. I provided an encoding from ACTNs into TGAs as an extension of that given for CSTNUs to accommodate users and authorization constraints and I also discussed a few optimizations to speed up the model-checking phase. I discussed the correctness and complexity of the encoding. I used ACTNs to analyze the official STEMI guidelines.

ACTNs differ from all temporal networks discussed in Part I as those formalisms do not employ resources. ACTNs differ from CNCUs as ACTNs do not employ temporal constraints and they also differ from the contributions in Chapter 7 as they do not deal with the uncertain availability of resources.

9

CSTNUs with Resources

In this chapter I provide *CSTNUs with Resources (CSTNURs)*. The temporal plans I am interested in modeling augment again the plans I discussed in Part I and Part II by injecting resources, in charge of executing the time points, and *runtime resource constraints (RCCs)* saying *when* and *which* resources are committable to execute a task according to when and which other resources have been committed for some other tasks in the past. In other words, in a CSTNUR, each time point is executed by committing a resource that can be chosen from a set associated to the time point. Each resource is associated to a temporal expression representing its temporal availability during the execution. A resource can be committed to execute a time point at a certain time instant t if and only if the valuation of its associated temporal expression is true with respect to t. Moreover, the availability of a resource can be constrained by means of special kind of constraints, runtime resource constraints, in order to refine its availability in real time with respect to the execution time of previous time points or previous resource commitments.

9.1 Motivating Example

As a motivating example, I consider (a simplification of) a temporal plan modeling a round-trip flight from Anchorage, Alaska to Frankfurt, Germany (direct flights). I show its graphical workflow-representation in Figure 9.1. I focus on the part involving pilots and engineers. Once boarding is complete, the take off could be delayed for extreme weather conditions and related safety procedures such as, for example, deicing. Deicing is the process of removing snow and ice from the plane surfaces (especially wings) by "power washing" the aircraft with chemicals which also remain on the surfaces in order to prevent the reformation of the ice. This (uncontrollable) condition is modeled by a conditional split connector (diamond labeled by `deicing?`). If `deicing?` $= \top$ (i.e., `deicing?` is true) then the `Deicing` process starts after minimum 5 and within 10 minutes (`Yes` branch). This task lasts from 1 to 3 hours[1]. After `Deicing` has finished, the plane takes off after minimum 5 and within 10 minutes. `alice` and `bob` are two specialized workers who can be

[1] Actually, deicing an aircraft does not take 3 hours, but since all leaving aircrafts have to do so following the departure scheduling, each plane queues for its turn.

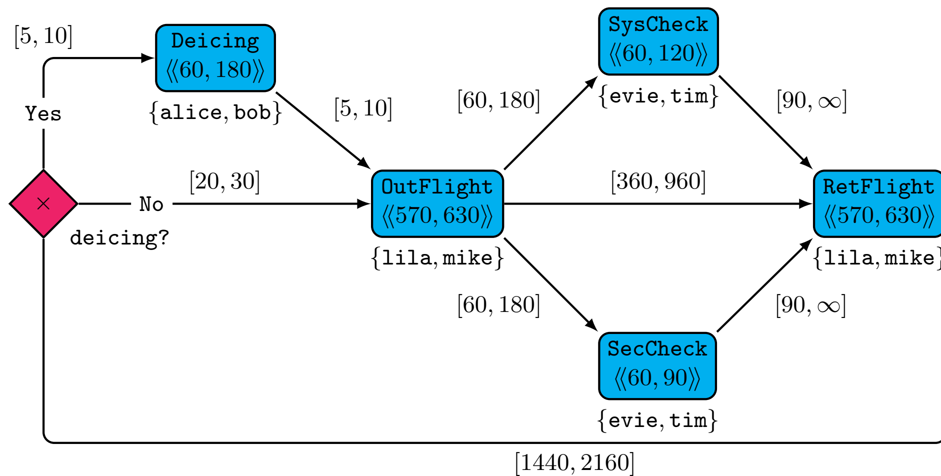


Fig. 9.1: Example of a temporal workflow for a round-trip flight. All ranges are in minutes. Pilots must rest at least 14 hours before piloting again. No engineer carries out `SysCheck` and `SecCheck` simultaneously.

commited for this task. Instead, if `deicing?` $= \bot$ (i.e., `deicing?` is false), the plane just takes off after 20 and within 30 minutes modeling the time needed to provide passengers with the safety instructions and reach the runway (`No` branch). Note that in case of deicing, there is plenty of time for the safety instructions. Once the aircraft has taken off, the outbound flight (`OutFlight`) lasts from 9 hours and 30 minutes to 10 hours and 30 minutes. `lila` and `mike` are pilots who can be committed for this task. Once the aircraft has landed, a system check (`SysCheck`) and a security check (`SecCheck`) start after minimum 1 and within 3 hours. `evie` and `tim` are two engineers who can be committed for these two tasks. `SysCheck` lasts 1 to 2 hours, whereas `SecCheck` lasts 1 to 1 hour and a half. Once both these two tasks are done, the plane can take off again after minimum 6 and maximum 16 hours since its landing (`RetFlight`) with the same pilots available for this task. The whole process lasts minimum 24 and maximum 36 hours.

This process employs users as resources and enforces two safety properties. First, the process enforces the FAA regulations for flight time limitations and rest requirements[2] saying that after a 10-12 hour (multi-time zone) flight, a pilot must rest from 14 to 18 hours before piloting again. Second, the process also requires that if `SysCheck` and `SecCheck` are executed in parallel, they are not executed by the same engineer (who can however execute both sequentially). For the sake of simplification, in this chapter I assume that one resource only is committed for executing each task (equivalently, each task is executed by a single user).

[2] https://www.ecfr.gov/cgi-bin/text-idx?view=text&node=14:2.0.1.3.10#se14.2.91_11059

9.2 Syntax

In what follows, I introduce some preliminary concepts and definitions before giving the formal definition of CSTNUR.

A *temporal expression* represents an assertion with respect to an instant—in this thesis, it is always the execution time—and a possible time point; a temporal expression is useful to characterize the temporal availability of resources in a CSTNUR in a compact way.

Definition 9.1 (Temporal Expression). *A* temporal expression (TE) τ *is a (temporal) assertion defined according to the grammar:*

$$\tau ::= \odot \mid \square k \mid \square X + k \mid \tau_1 \wedge \tau_2$$

where $\square \in \{>, <, \geq, \leq, =\}$*,* X *is a time point,* $k \in \mathbb{N}$[3]*,* $\odot$ *is the empty expression, and* τ_1, τ_2 *are two TEs. There are 4 types of TEs:*

- Type 0 *when* $\tau = \odot$
- Type 1 *when* $\tau = \square k$
- Type 2 *when* $\tau = \square X + k$
- Type 3 *when* $\tau = \tau_1 \wedge \tau_2$

The set of all possible TEs is denoted by $\mathcal{TE}$*.*

Every TE of Type 2, $\tau = \square X + k$, is equivalent to a Type 1 once X has been executed. When X is unexecuted I assume that $X = \infty$. The interpretation of a TE τ, with respect to a particular instant t, is as follow.

Definition 9.2 (TE interpretation). *The* interpretation of a TE τ *with respect to a temporal instant* $t \in \mathbb{R}^{\geq 0}$ *is defined as follows:*

1. $t \models \odot$
2. $t \models \square k$ *iff* $t \square k$*, where* $k \in \mathbb{N}$ *and* $\square \in \{>, <, \geq, \leq, =\}$*.*
3. $t \models X + k$ *if* $t \models (t_X + k)$*, where* $t_X = \infty$ *if* X *is unexecuted and* $t_X \neq \infty$ *is the time at which* X *was executed otherwise.*
4. $t \models \tau_1 \wedge \tau_2$ *iff* $t \models \tau_1$ *and* $t \models \tau_2$*.*

By using temporal expressions, I can represent the concept of *availability* of resources at run time in a compact way.

Definition 9.3 (Temporal availability). *Given a set of resources* $\mathcal{R}$*, a set of time points* $\mathcal{T}$ *and the set of all possible temporal expressions* $\mathcal{TE}$*, I model the* temporal availability *of resources as a pair* (RA, RE)*, where:*

- $RA \subseteq \mathcal{R} \times \mathcal{T}$ *models the resource-time point association relation (i.e., which resources can be committed for which time points). I abuse notation and write* $\mathcal{R}(X) = \{r \mid (r, X) \in RA\}$ *to represent the* resources *committable for* X *and I impose that for each contingent link* $(A, x, y, C) \in \mathcal{L}$*,* $\mathcal{R}(A) = \mathcal{R}(C)$ *and for each* $X \in \mathcal{T}$*,* $\mathcal{R}(X) \neq \emptyset$*.*

[3] Since I am going to translate temporal expressions into clock constraints I avoid using reals here.

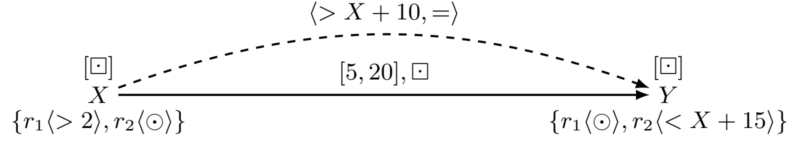


Fig. 9.2: An augmented CSTNU. Each time point has an associated set consisting of elements having the form $r\langle\tau\rangle$ meaning $(r, X) \in RA \wedge RE(r, X) = \tau$.

- $RE\colon RA \to \mathcal{TE}$ *represents the temporal expression associated to each pair* (r, X)*, where* $r \in \mathcal{R}(X)$*.*

A resource $r \in \mathcal{R}(X)$ *is* committable *at time* t *if* $t \models RE(u, X)$*, not committable otherwise.*

Figure 9.2 shows an augmented CSTNU consisting of two time points X and Y connected by two edges: a classic one representing the temporal constraint with range $5 \le Y - X \le 20$ (solid), and a new one representing a runtime resource constraint (dashed). Time points are also labeled by the set of committable resources, where, by abusing notation, if $r\langle\tau\rangle$ appears in X's label it means that $(r, X) \in RA \wedge RE(u, X) = \tau$. In other words, r_1 and r_2 are two resources committable for executing both X and Y but with different temporal expressions. For r_1, the temporal expression associated to X (i.e., $RE(u_1, X)$) is > 2 meaning that r_1 can be committed to execute X when its execution time is greater than 2. For r_2, the temporal expression with respect to Y is $< X + 15$ meaning that r_2 can be committed to execute Y until global time is smaller than $t_X + 15$, where t_X is the execution time of X (again ∞ if X is unexecuted).

Runtime resource constraints (RRCs) refine in real time the temporal expressions of resources by appending other temporal expressions as conjuncts. Each RRC is defined between two time points, *firing time point* and *target time point*. When the firing time point is executed, the effect of RRC is to append new TEs to the existing ones associated to the resources of the target time point (if such a time point is still unexecuted) considering the relation defined in the RRC. In this way, it is possible to adjust, for example, the committable resources of the target time point considering which resource was committed for the firing time point.

Definition 9.4 (Runtime Resource Constraint). *A* Runtime Resource Constraint (RRC) *is a 4-tuple* $\langle X, \tau, Y, \rho\rangle$*, where:*

- $X, Y \in \mathcal{T}$ *are the firing and target time points, respectively, such that* $X \neq Y$*,* Y *is non contingent, and* $L(X)$ *is consistent with* $L(Y)$ *(and* $L(Z)$ *if* τ *is a Type 2 TE).*
- $\tau \in \mathcal{TE}$ *is a temporal expression.*
- $\rho \subseteq \mathcal{R} \times \mathcal{R}$ *is a binary relation over resources. As usual,* $=$ *shortens* $\{(r, r) \mid r \in \mathcal{R}\}$*,* $\neq$ *shortens* $\{(r_1, r_2) \mid r_1, r_2 \in \mathcal{R} \wedge r_1 \neq r_2\}$*, and* $*$ *shortens* $\{(r_1, r_2) \mid r_1, r_2 \in \mathcal{R}\}$ *(the universal relation).*

I interpret each RRC $\langle X, \tau, Y, \rho\rangle$ as follows: when a resource r_X is committed to execute X, then τ is *instantaneously* appended to all temporal expressions of those resources r_Y committable for Y such that $(r_X, r_Y) \in \rho$. In symbols,

$$\forall r_Y \in \mathcal{R}(Y).\, (r_X, r_Y) \in \rho \implies RE(r_Y, Y) = RE(r_Y, Y) \wedge \tau$$

where $r_X \in \mathcal{R}(X)$ is the resource committed for X.

An RRC does not imply an execution order among time points. An RRC $\langle X, \tau, Y, \rho\rangle$ has effect on Y, only when, once we execute X, Y has not been executed yet (refining the availability of already committed resources does not make sense). That is, when a resource is committed for a time point, its temporal expression (with respect to that time point) no longer changes. Therefore, if there is an RRC between X and Y and the two time points have to be executed at the same time, it is necessary to fix an execution order between them to decide whether the RRC applies. Moreover, if there is an RRC between a contingent time point C and a non-contingent time point X and the two time points occur at the same time t (because `env` decided to execute C after `ctrl` had decided to execute X at time t), then the RRC is ignored because C is assumed to be executed after X even if the two time points are executed at the same instant.

In Figure 9.2, the RRC $\langle X, > X + 10, Y, =\rangle$ — drawn as a dashed edge $X \to Y$ labeled by $\langle > X + 10, =\rangle$ — represents the fact that the resource committed for X can also be committed for Y if the execution time of Y is greater than 10 time units since X was executed. Note that the temporal constraint between X and Y allows for the execution of Y just 5 time units after X. Suppose that r_2 is committed for X at time 1. $\mathcal{R}(Y)$ is instantaneously updated considering $\langle X, > X + 10, Y, =\rangle$. Since the RRC has now become $\langle X, > 11, Y, =\rangle$ (as we know the value of X), its TE part > 11 is appended to all TEs associated to the same resource r_2 committable for Y (because ρ is '$=$'). Here, there is only one TE for r_2. Therefore, the application of the RRC results in evolving the "state" of the temporal expressions of the resources in $\mathcal{R}(Y)$ as follows:

$$\overbrace{\{r_1\langle\odot\rangle, r_2\langle< X + 15\rangle\}}^{\mathcal{R}(Y)\text{ for } t<1} \quad \rightsquigarrow \quad \overbrace{\{r_1\langle\odot\rangle, r_2\langle> 11 \wedge\ < 16\rangle\}}^{\mathcal{R}(Y)\text{ for } t\geq 1}$$

The allowed delay for executing Y after X is $[5, 20]$. If `ctrl` decides to fix the execution of Y at $5 \leq t' \leq 11$ or $16 \leq t' \leq 20$, then the only committable resource is r_1, and also r_2 otherwise. For example, if we fix $t' = 13$, it is simple to verify that r_2 is committable for Y as

$$t' \models (> 11 \wedge < 16) = t' \models (> 11) \text{ and } t' \models (< 16) = \top \wedge \top = \top$$

Now, I can give the formal definition of a CSTNUR putting together everything I have discussed so far.

Definition 9.5 (CSTNUR). *A* Conditional Simple Temporal Network with Uncertainty and Resources (CSTNUR) *is a tuple* $\langle \mathcal{T}, \mathcal{P}, L, \mathcal{OT}, O, \mathcal{C}, \mathcal{LR}, RA, RE, \mathcal{RRC}\rangle$, *where:*

1. $\langle \mathcal{T}, \mathcal{P}, L, \mathcal{OT}, O, \mathcal{C}, \mathcal{L}\rangle$ *is a CSTNU.*
2. $\mathcal{R} = \{r_0, r_1, \dots\}$ *is a finite set of resources.*
3. *The pair* (RA, RE) *specifies temporal availability according to Definition 9.3.*
4. $\mathcal{RRC}$ *is a set of runtime resource constraints according to Definition 9.4.*

Figure 9.3 shows the CSTNUR modeling the temporal plan in Figure 9.1 considering access control. There are seven users, `alice`, `bob`, `lila`, `mike`, `evie`, `tim` and `wf`, where `wf` represents the WfMS. $\langle C_2, \geq C_2 + 840, A_5, = \rangle$ (RRC_1) (dashed edge $C_2 \to A_5$) models a *temporal separation of duties (TSoD)* meaning that the *same* pilot ($=$) who executes C_2 (i.e., piloted the aircraft in the `OutFlight`) will return available to pilot again after 14 hours (FAA regulations). $\langle A_3, > C_3, A_4, = \rangle$ (RRC_2) and $\langle A_4, > C_4, A_3, = \rangle$ (RRC_3) (dashed edges $A_3 \to A_4$ and $A_4 \to A_3$, respectively) model a "no multi-tasking" policy for `SysCheck` and `SecCheck` requiring that either different resources are committed for those tasks when executed in parallel, or the same resource can be committed for both provided that these tasks are executed sequentially.

Indeed, RRC_2 specifies that the user who executes A_3 will return available for executing A_4 as soon as C_3 has executed. Likewise, RRC_3 specifies that the user who starts A_4 will return available for executing A_3 as soon as C_4 has executed.

9.3 Semantics

In a CSTNUR, resources are committed to execute the time points. Resources committed for contingent time points are the same that were committed for the corresponding activation time points. However, `env` is still free to schedule these time points when he wants.

Since CSTNURs extend CSTNUs, we still have that the truth values of propositions and the duration of contingent links are incrementally revealed over time as the corresponding observation/contingent time points are executed, respectively. Again, a *dynamic execution strategy* reacts to observations and contingent time points in real time also saying which resources are committed for which time points. A *viable* and *dynamic execution strategy* for a CSTNUR is a strategy executing all non-contingent time points such that all relevant constraints about temporal distances and resource commitments will be satisfied no matter which truth values for propositions and durations for contingent links are incrementally revealed over time. A CSTNUR with such a strategy is called *dynamically controllable.*

A more formal description of the execution semantics of CSTNURs can be given in terms of extended RTEDs. In what follows, like I did for ACTNs in Chapter 8, I extend the RTEDs given for CSTNUs [26] to also consider resources and RRCs.

For a CSTNUR, `ctrl` seeks a strategy for scheduling all relevant non-contingent time points such that all relevant temporal constraints involving resources and time points are eventually satisfied no matter what `env` does.

A *partial schedule* for a CSTNUR is still a pair $\mathcal{PS} = (\mathit{Executed}, \mathit{Assigned})$, where *Executed* renames *OccTP* in Section 8.3 and here it is a set of triples (r, X, t), where r is the resource committed for X at time t. Instead, *Assigned* just renames *KnownProp* in Section 8.3.

$\mathit{Executed}_{\mathcal{T}}$ shortens the set of time points in *Executed* (without any other information). For each $X \in \mathit{Executed}_{\mathcal{T}}$, $\mathit{time}(X)$ still queries *Executed* to get information about when X was executed, whereas $\mathit{res}(X)$ does the same but with respect to the committed resource. $\mathcal{PS}$ is *locally consistent* if *Executed* satisfies all temporal constraints of the underlying CSTNU and for each $(r, X, t) \in \mathit{Executed}_{\mathcal{T}}$,

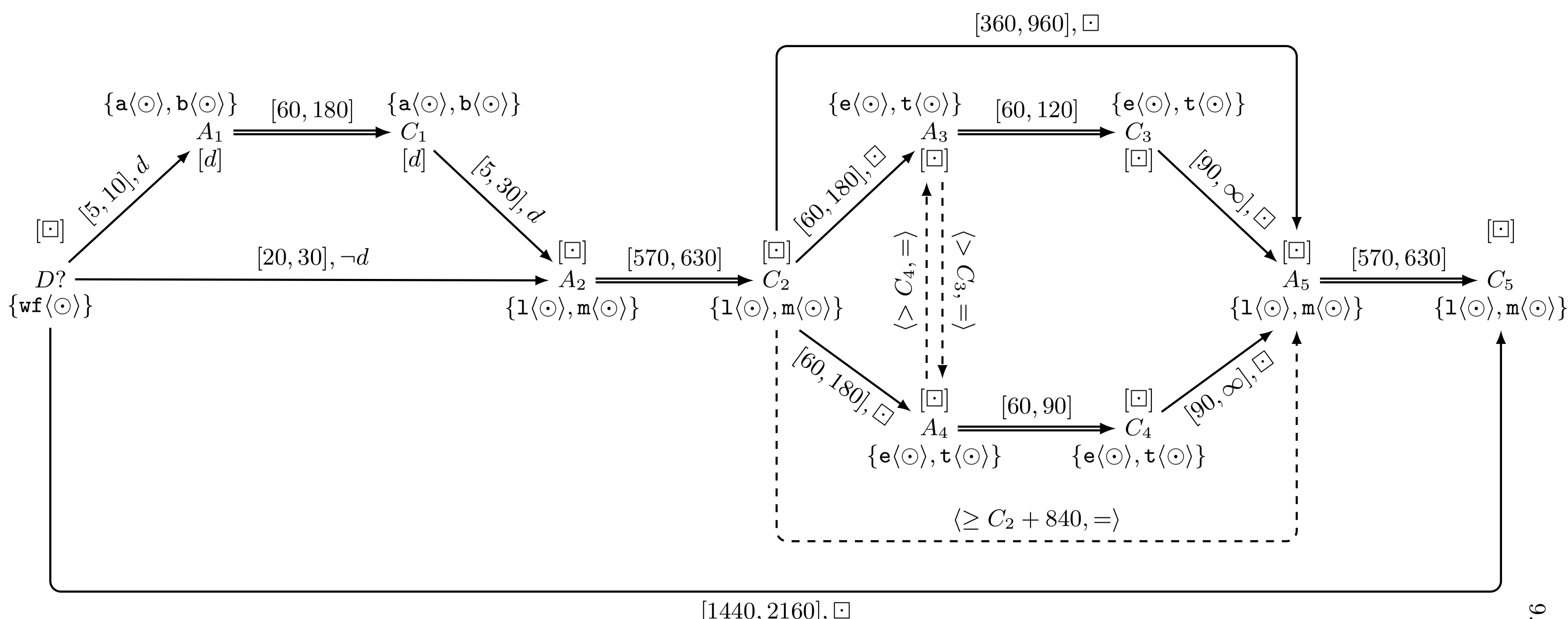


Fig. 9.3: CSTNUR modeling the temporal plan in Figure 9.1 with access control. $D?$ model the conditional split connect, whereas $A_1 \Rightarrow C_1$, $A_2 \Rightarrow C_2$, $A_3 \Rightarrow C_3$, $A_4 \Rightarrow C_4$ and $A_5 \Rightarrow C_5$ model `Deicing`, `OutFlight`, `SysCheck`, `SecCheck` and `RetFlight`.

$r \in \mathcal{R}(X)$ and $t \models RE(r, X)$, where $RE(r, X)$ is the temporal expression associated to the pair (r, X).[4] The set of all possible partial schedules remains represented by $\mathcal{PS}^*$.

In the following, I fully formalize the execution semantics of CSTNURs in terms of extended RTEDs for `env` and `ctrl`. Moreover, *NonContingent* is now a set of pairs (r, X), where r is the resource that `ctrl` wants to commit for X.

Definition 9.6 (RTED for `ctrl`). *An* RTED *for the controller* `ctrl`*,* $\Delta_{\texttt{ctrl}}$*, specifies which action has to be performed by* `ctrl` *during an execution (represented by* $\mathcal{PS}$*). It has two forms:* `wait` *or* $(t, \mathit{NonContingent})$*.*

- $\Delta_{\texttt{ctrl}} = \texttt{wait}$ *is the same as that given in Definition 8.3.*
- $\Delta_{\texttt{ctrl}} = (t, \mathit{NonContingent})$ *represents the conditional constraint: "if* `env` *does nothing* before *time* t*, then for each pair* $(r, X) \in \mathit{NonContingent}$*, commit the resource* r *to execute time point* X *at time* t*." Such a decision is applicable if and only if* $t > \mathit{last}$*,* $\mathit{NonContingent}$ *is a (non empty)* ordered *set of pairs* (r_i, X_i) $i = 1, \ldots, k$*, where* r_i *is a resource and* X_i *is a non-contingent unexecuted time point such that* $\ell_{cps} \Rightarrow L(X)$*.*

Definition 9.7 (RTED for `env`). *An* RTED *for the environment* `env`*,* $\Delta_{\texttt{env}}$*, specifies which action has to be performed by* `env` *during an execution. It has two forms:* `wait` *or* $(t, \mathit{Contingent})$*.*

- $\Delta_{\texttt{env}} = \texttt{wait}$ *is the same as that given in Definition 8.3.*
- $\Delta_{\texttt{env}} = (t, \mathit{Contingent})$ *is the same as that given in Definition 8.3 committing* $\mathit{res}(A)$ *to execute* C*. In other words, such a decision is applicable if and only if* $t > \mathit{last}$*,* $\mathit{Contingent}$ *is a non-empty subset of pairs* (r, C)*, where* C *is the contingent time point such that* $(A, x, y, C) \in \mathcal{L}$ *and* $A \in \mathit{Executed}_{\mathcal{T}}$*,* $r = \mathit{res}(A)$ *is the resource that was committed for* A *and* $t \in [\mathit{time}(A){+}x, \mathit{time}(A){+}y]\}$*.*

$\Delta^*_{\texttt{ctrl}}$ *and* $\Delta^*_{\texttt{env}}$ *still denote the sets of all RTEDs for* `ctrl` *and* `env`*. Furthermore, since the association of resources to contingent time points is implicit and the assignment of truth values to propositions does not involve resources, the instantaneous reactions definition is the same as the one given for CSTNUs in Section 8.3.*

I am now ready to extend the concept of the partial and full outcome between $\Delta_{\texttt{ctrl}}$ and $\Delta_{\texttt{env}}$.

Definition 9.8 (Partial Outcome). *Let* $\mathcal{PS}$ *be a locally consistent partial schedule. Let* $\Delta_{\texttt{ctrl}}$ *be an RTED for* `ctrl` *and* $\Delta_{\texttt{env}}$ *and RTED for* `env`*. I model the* partial outcome *of* $\Delta_{\texttt{ctrl}}$ *and* $\Delta_{\texttt{env}}$ *as a mapping* $PO(\mathit{Executed}, \Delta_{\texttt{ctrl}}, \Delta_{\texttt{env}})$ *neglecting any instantaneous reaction* $\mathcal{IR}$*. There are four possible cases:*

(1) $PO(\mathit{Executed}, \texttt{wait}, (t, \mathit{Contingent})) = \mathit{Executed} \cup \{(\mathit{res}(A), C, t) \mid C \in \mathit{Contingent}\}$*. Also, for any* $\langle C, \tau, Y, \rho\rangle \in \mathcal{RRC}$ *such that* $Y \notin \mathit{Executed}_{\mathcal{T}}$ *we have that* $\forall r_Y \in \mathcal{R}(Y), (\mathit{res}(A), r_Y) \in \rho \implies RE(r_Y, Y) = RE(r_Y, Y) \wedge \tau$*.*

[4] Once r has been committed for X at time t, no RRC will ever be able to restrict r's associated temporal expression for X as RRCs only apply to unexecuted target time points. Therefore, the valuation of $t \models RE(u, X)$ will remain fixed forever.

(2) $PO(Executed, (t_1, NonContingent), (t_2, Contingent)) = Executed \cup \{(res(A), C, t_2) \mid C \in Contingent\}$ *if* $t_2 < t_1$. *Also, for any* $\langle C, \tau, Y, \rho \rangle \in \mathcal{RRC}$ *such that* $Y \notin Executed_{\mathcal{T}}$ *we have that* $\forall r_Y \in \mathcal{R}(Y), (res(A), r_Y) \in \rho \implies RE(r_Y, Y) = RE(r_Y, Y) \wedge \tau$.

(3) $PO(Executed, (t, NonContingent), \mathtt{wait}) = Executed \cup \{(r_i, X_i, t) \mid (r_i, X_i) \in NonContingent$ *for* $i = 1, \ldots, k\}$. *Also, every time we add* (r_i, X_i, t) *to* $Executed$ *we fire the related RRCs (if any). That is, for any* $\langle X_i, \tau, Y, \rho \rangle \in \mathcal{RRC}$ *such that* $Y \notin Executed_{\mathcal{T}}$ *we have that* $\forall r_Y \in \mathcal{R}(Y), (r_i, r_Y) \in \rho \implies RE(r_Y, Y) = RE(r_Y, Y) \wedge \tau$.

(4) $PO(Executed, (t_1, NonContingent), (t_2, Contingent)) = Executed \cup \{(r_i, X_i, t) \mid (r_i, X_i) \in NonContingent$ *for* $i = 1, \ldots, k\}$ *if* $t_1 \leq t_2$. *Again, every time we add* (r_i, X_i, t) *to* $Executed$ *we fire the related RRCs (if any). That is, for any* $\langle X_i, \tau, Y, \rho \rangle \in \mathcal{RRC}$ *such that* $Y \notin Executed_{\mathcal{T}}$ *we have that* $\forall r_Y \in \mathcal{R}(Y), (r_i, r_Y) \in \rho \implies RE(r_Y, Y) = RE(r_Y, Y) \wedge \tau$.

The explanations are similar to those given for ACTNs in Section 8.3. The definitions of full outcome and RTED-based strategies for `ctrl` and `env` are the same as those given for ACTNs. The definition of dynamic controllability thus refines to:

Definition 9.9 (Dynamic Controllability of a CSTNUR). *A CSTNUR is* dynamically controllable (DC) *if there exists an RTED-based strategy* $\sigma_{\mathtt{ctrl}}$ *such that for all RTED-based strategies* $\sigma_{\mathtt{env}}$*, the variable assignments* (r, X, k) *in the complete schedule* $FO^*(\sigma_{\mathtt{ctrl}}, \sigma_{\mathtt{env}})$ *satisfy all temporal constraints of the underlying CSTNU, and each assignment* (r, X, k) *satisfies both* $r \in \mathcal{R}(X)$ *and* $k \models RE(r, X)$.

9.4 Encoding CSTNURs into TGAs

In this section, I extend the encoding into TGAs given in Section 5.3 for CSTNUDs in order to represent the DC checking of CSTNURs as a two-player game between `ctrl` and `env` according to the semantics I gave in Section 9.3. Despite CSTNURs do not employ decision time points, I rely on the recent encoding for CSTNUDs which enforces time point label honesty and predecessors in the transition guards modeling the execution of non-contingent time points (Section 5.3). Then, I proceed by encoding committable resources into dedicated clocks and resource commitments for time point executions consideing RRCs into circular paths.

9.4.1 Encoding Committable Resources into Dedicated Clocks

As I have already pointed out, in a CSTNUR resources are committed for time points according to the RA relation. Therefore, to model which resource has been committed for which time point, I start by associating a dedicated clock `rX` to each element $(r, X) \in RA$ and interpreting the value of such clocks as follows.

1. If $\mathtt{rX} > \mathtt{cX}$, where `cX` is the clock associated to time point X, it means r was committed for X at time $\hat{\mathtt{c}} - \mathtt{cX}$.

2. If $\mathtt{rX} = \mathtt{cX} = \hat{\mathtt{c}}$, it means that X has not been executed yet and r is committable for X.
3. If $\mathtt{rX} = \mathtt{cX} < \hat{\mathtt{c}}$, it means that X has been executed and r is not the resource committed for X.
4. If $\mathtt{rX} < \mathtt{cX}$, it means that X has not been executed yet and r cannot be committed for X because r is not available. In details, $\mathtt{rX}$ becomes less than $\mathtt{cX}$ by a reset action. The reset of $\mathtt{rX}$ occurs when it is necessary to prevent r from being committed for X since $\hat{\mathtt{c}} \not\models RE(r, X)$.

Differently from what I have done for clocks associated to time points, here $\mathtt{rX}$ clocks may be reset more than once. If r is committable for X and $\mathtt{rX}$ has never been reset, then r can be committed to execute X. To determine which resource was committed for a time point X, I only need to find the unique clock $\mathtt{rX}$ such that $\mathtt{rX} > \mathtt{cX}$. It is guaranteed that all other clocks $\mathtt{r}_j\mathtt{X}$ where $r_j \in \mathcal{R}(X)$ and $r_j \neq r$ are equal to $\mathtt{cX}$.

Getting back to the example, the resource association relation RA specified for the CSTNUR in Figure 9.3 implies the following clocks.

- $\mathtt{wD}$ models $\mathcal{R}(D?) = \{\mathtt{wf}\}$
- $\mathtt{aA}_1$, $\mathtt{bA}_1$ model $\mathcal{R}(A_1) = \{\mathtt{alice}, \mathtt{bob}\}$
- $\mathtt{aC}_1$, $\mathtt{bC}_1$ model $\mathcal{R}(C_1) = \{\mathtt{alice}, \mathtt{bob}\}$
- $\mathtt{lA}_2$, $\mathtt{mA}_2$ model $\mathcal{R}(A_2) = \{\mathtt{lila}, \mathtt{mike}\}$
- $\mathtt{lC}_2$, $\mathtt{mC}_2$ model $\mathcal{R}(C_2) = \{\mathtt{lila}, \mathtt{mike}\}$
- $\mathtt{eA}_3$, $\mathtt{tA}_3$ model $\mathcal{R}(A_3) = \{\mathtt{evie}, \mathtt{tim}\}$
- $\mathtt{eC}_3$, $\mathtt{tC}_3$ model $\mathcal{R}(C_3) = \{\mathtt{evie}, \mathtt{tim}\}$
- $\mathtt{eA}_4$, $\mathtt{tA}_4$ model $\mathcal{R}(A_4) = \{\mathtt{evie}, \mathtt{tim}\}$
- $\mathtt{eC}_4$, $\mathtt{tC}_4$ model $\mathcal{R}(C_4) = \{\mathtt{evie}, \mathtt{tim}\}$
- $\mathtt{lA}_5$, $\mathtt{mA}_5$ model $\mathcal{R}(A_5) = \{\mathtt{lila}, \mathtt{mike}\}$
- $\mathtt{lC}_5$, $\mathtt{mC}_5$ model $\mathcal{R}(C_5) = \{\mathtt{lila}, \mathtt{mike}\}$

9.4.2 Encoding Resource Commitments into Circular Paths

In the encoding for CSTNUDs given in Section 5.3 the executions of time points are modeled as self-loop transitions at L_1. In CSTNURs, I must extend this part in order to model that resources are committed to execute time points. A resource r is committable to execute a time point Y at time t if $r \in \mathcal{R}(Y)$ and t satisfies its associated TE. The latter condition entails validating all fired RRCs (if any) targeting the time point I am trying to execute. I achieve all this as follows.

Instead of having a (possibly) exponential number of self-loop transitions modeling all possible executions with respect to all possible combinations of RRCs, I model the commitment of a resource r for a time point X by means of a *circular path* of *urgent locations* starting and ending at L_1 (see Figure 9.4a). All transitions involving these locations are uncontrollable. The first location has the same name of the time point to execute (i.e., Y). A run of the TGA enters Y if and only if the corresponding time point has not been executed yet (the guard is exactly the same of that given for the self-loop transitions for CSTNUDs). Then, the run goes through a set of n locations $Y_1, \dots, Y_n$, where n is the number of RRCs targeting

Y. Moving from Y_{i-1} to Y_i means validating the ith RRC targeting Y. In Figure 9.4a, each thick edge connecting Y_{i-1} to Y_i abstracts a DAG with Y_{i-1} as a source and Y_i as a sink. Such a DAG has two possible forms according to the type of TE contained in an RRC (see below). Validating RRCs may result in blocking some resources—those for which the current time does not satisfy their associated TEs (refined by the RRC itself)—by resetting their associated clocks. The validation of several different RRCs could block the same resource r more than once by keeping on resetting $\texttt{rY}$.

Finally, a set of m transitions connect Y_n to Y_r, where m is the number of resources associated for Y. Taking one of these transitions means to commit one and only one *committable* resource among those associated to Y. Recall that, at any time, $r \in \mathcal{R}(Y)$ is committable if and only if $\texttt{rY} = \hat{\texttt{c}}$. Therefore, none, a few or all of these transitions could be disabled as their guards might be false.

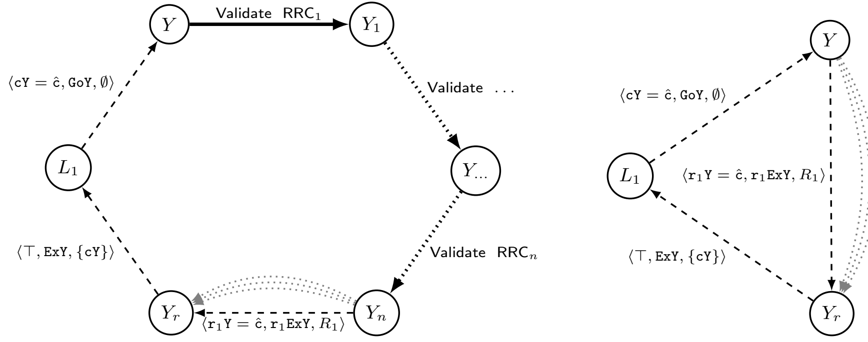
If r is not committable, it means that r's current associated TE is violated. Since in the previous locations $Y_1, \ldots, Y_n$ all RRCs having Y as a target have been validated, at least one of these transitions must have blocked r by resetting $\texttt{rY}$. If the run can enter Y_r, it means that at least one resource for Y is committable (i.e., survived the blocking process). Again, none, a few or all resources can survive this process. Each one of these transitions verifies that r_i is committable (having $\texttt{r}_i\texttt{Y} = \hat{\texttt{c}}$ as the unique clock constraint in its guard) and resets all clocks $\texttt{r}_j\texttt{Y}$ associated to all other resources $r_j \neq r_i$ committable for the same time point. The last (single) transition connecting Y_r to L_1 fixes the execution time of Y by resetting $\texttt{cY}$. Eventually, if r is committed for Y at time $\hat{\texttt{c}} - \texttt{cY}$, the following three conditions hold:

1. $r \in \mathcal{R}(Y)$
2. $\hat{\texttt{c}} - \texttt{cY} \models RE(r, Y)$
3. $\texttt{rY} > \texttt{cY}$ and $\texttt{r}_i\texttt{Y} = \texttt{cY}$ for all $r_i \in \mathcal{R}(Y)$ such that $r \neq r_i$.

I now discuss more in detail how I encode RRCs and block resources. The simplest case is when no RRC targets a time point Y (Figure 9.4b). In such a case the circular path reduces to three locations only: L_1, Y and Y_r. Again, a run enters Y if the corresponding time point has not been executed yet, and then moves (for sure) to Y_r since no resource has been blocked. Finally, it fixes the execution time of Y. Instead, when a time point X appears as target in some RRC, I must make sure that, if this RRC has fired and the related TEs of the resources committable for X have been updated, the global time $\hat{\texttt{c}}$ must satisfy at least one of the TEs associated to different resources. This way, at least one resource can be committed to execute the time point.

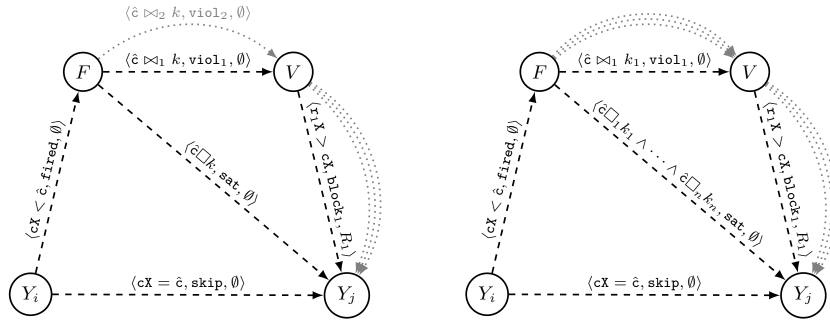
In the rest of the paper I will only consider RRCs whose embedded TEs are either conjunctions of Type 1 TEs (i.e., a subset of all possible Type 3) or Type 2 only. I do not consider Type 0 TEs as current time (no matter its value) always satisfies them. I *normalize* Type 3 TEs containing conjuncts of Type 2 by removing such conjuncts and generate new RRCs containing single Type 2 TEs. This can be done in linear time with respect to the number of conjuncts. For example, the RRC

$$\langle X, \underbrace{\geq 3 \wedge \leq 7}_{1} \wedge \underbrace{\geq C}_{2} \wedge \underbrace{\leq Z + 5}_{3}, Y, \rho \rangle$$



(a) Circular path modeling the execution of Y. A run enters this circular path only if Y is unexecuted ($L_1 \to Y$). Then, it verifies all the RRCs targeting Y (if any) ($Y \to \cdots \to Y_n$). After that, it commits one and only one committable resource among those associated to Y (if any) ($Y_n \to Y_r$). Finally, it fixes the execution time of Y going back to L_1 ($Y_r \to L_1$). Thick edges labeled by "Validate r_i" are a compact representation of the DAGs depicted in Figure 9.4(c)-(d).

(b) Circular path modeling the general execution of Y when no RRC targets Y. A run just commits one and only one committable resource among those associated to X and fixes the time of such an execution.



(c) Encoding RRCs specifying a single TE of Type 1 $\langle X, \Box k, Y, \rho\rangle$. A run enters F if only if the RRC has fired ($Y_i \to F$), otherwise it skips the verification ($Y_i \to Y_j$). At F, either $\Box k$ is satisfied, and then no resource is blocked ($F \to Y_j$), or $\Box k$ is violated ($F \to V$), and a few resources may eventually be blocked ($V \to Y_j$).

(d) Encoding RRCs specifying conjunctions of TEs of Type 1 $\langle X, \bigwedge_i \Box_i k_i, Y, \rho\rangle$. `skip`, `fired` and `block`$_i$ transitions remain the same of those given in Figure 9.4c. `sat` extends by checking that the entire conjunction is true. Finally, there are as many `viol`$_i$ transitions as the number of disjuncts arising from $\neg(\bigwedge_i \hat{c}\Box_i k_i)$.

Fig. 9.4: Modeling resource commitments: (a) shows the general circular path, (b) shows the case in which a time point does not appear as a target in any RRC, (c) and (d) give the encodings for (conjunctions of) TEs of Type 1.

is normalized to

$$\{\langle X, > 3 \wedge \leq 10, Y, \rho\rangle, \langle X, > C, Y, \rho\rangle, \langle X, \leq Z+5, Y, \rho\rangle\}$$

I now proceed by discussing how I encode RRCs in the circular path. As I discussed at the beginning of this section, I validate the jth RRC r_j by going from Y_i to Y_j. There are two possible cases: (1) the considered RRC contains a conjunction of TEs of Type 1, or (2) the considered RRC contains a TE of Type 2.

Encoding RRCs containing (a conjunction of) TEs of Type 1

I encode an RRC having the form $\langle X, \Box k, Y, \rho\rangle$ by means of four locations Y_i, Y_j, F, and V (see Figure 9.4c) and I connect such locations as discussed below. Entering F means that the RRC has *fired* (i.e., that X has been executed), whereas entering V means that both the RRC has fired and the TE of Type 1 specified in it is *violated*. Therefore, starting from Y_i, there are two possible transitions: one going to F (`fired`) and one going to Y_j (`skip`).

The `fired` transition represents the fact that X has been executed and, therefore, the associated RRC whose firing time point is X has to be considered (i.e., the RRC has fired). The `skip` transition represents the fact that X has not been executed yet and, therefore, the associated RRC has to be ignored.

At F I have two transitions: either $t \models \Box k$ or not. In the first case, no resource will be blocked since current time sarisfies the TE. In such a case, the run moves to Y_j (`sat` transition) and goes ahead. The guard of `sat` contains the clock constraint $\hat{\mathtt{c}}\Box k$ modeling $t\Box k$. From Definition 9.2, it holds that $t \models \Box k$ iff $t\Box k$ whatever $\Box$ is. Since in a CSTNUR TEs are evaluated with respect to global time (modeled by $\hat{\mathtt{c}}$), I just need to substitute $\hat{\mathtt{c}}$ for t getting $\hat{\mathtt{c}}\Box k$.

In the second case, $t \not\models \tau$ and thus I must block all resources having τ associated. To achieve this aim, the run first enters V. Either one or two transitions connect F to V according to which $\Box$ τ specifies.

If τ is $= k$ (i.e., the corresponding clock constraint is $\hat{\mathtt{c}} = k$), then there are two transitions, one having guard $\hat{\mathtt{c}} < k$ and the other having guard $\hat{\mathtt{c}} > k$. Since $\neg\tau$ is true, then one of these two transitions must be true.

If τ is $= \Box k$, where $\Box \in \{<, >, \leq, \geq\}$, then I just need to specify a unique transition going from F to V whose guard is $\neg(\hat{\mathtt{c}}\Box k)$. In Figure 9.4c, such transitions have labels $\mathtt{viol}_1$ and $\mathtt{viol}_2$, where $\mathtt{viol}_2$ (drawn in gray) exists if and only if $\Box$ is $=$ in τ. $\bowtie_1$ and $\bowtie_2$ model the new $\Box$ operators arising from the negation of τ. Finally, a set of transitions connects V to Y_i. There exists one transition for each resource $r_X \in \mathcal{R}(X)$ saying that if r_X was committed for X, then all resources r_Y associated to Y such that the pair (r_X, r_Y) belongs to the relation ρ expressed in the considered RRC must be blocked by resetting their associated clocks. That is, the guard of each transition $\mathtt{block}_i$ is $\mathtt{r}_i\mathtt{X} > \mathtt{cX}$ (i.e., r_i was committed for X), whereas each update specifies the set R_i of clocks to reset computed as follows:

$$R_i = \{\mathtt{r}_j\mathtt{Y} \mid r_j \in \mathcal{R}(Y) \wedge (r_i, r_j) \in \rho\}$$

I do not leave "anything behind" as all of these transitions are mutually-exclusive.

Now that I have discussed how to encode an RRC containing a single TE of Type 1 I consider RRCs containing TEs of Type 3 where each conjunct is of Type 1. That is, RRCs having the form $\langle X, \bigwedge_i \Box_i k_i, Y, \rho\rangle$. Figure 9.4d shows the encoding of such an RRC, where rather than normalizing $\langle X, \bigwedge_i \Box_i k_i, Y, \rho\rangle$ and obtaining $\langle X, \Box_1 k_1, Y, \rho\rangle$, ..., $\langle X, \Box_n k_n, Y, \rho\rangle$ and then encoding each RRC according to Figure 9.4c resulting in a circular path $Y_1 \rightarrow \cdots \rightarrow Y_n \rightarrow Y_r$ of n DAGs, I generate a refined shorter path $Y_1 \rightarrow Y_u$, consisting of one DAG only, able to deal with the entire conjunction of TEs. This encoding substantially extends the $\mathtt{sat}$ and $\mathtt{viol}_i$ transitions in Figure 9.4c. All other transitions remain the same. I refine $\mathtt{sat}$ so that it verifies the clock constraint $\hat{\mathtt{c}}\Box_1 k_1 \wedge \cdots \wedge \hat{\mathtt{c}}\Box_n k_n$. To generate $\mathtt{viol}_i$ transitions, I proceed exactly as I did in Figure 9.4c. That is, I compute $\neg(\hat{\mathtt{c}}\Box_1 k_1 \wedge \cdots \wedge \hat{\mathtt{c}}\Box_n k_n)$ resulting in $\bigvee_j \neg(\hat{\mathtt{c}}\Box_i k_j)$, where, again, if $\Box$ is $=$ in some conjunct of the initial TE, I generate two disjuncts.

As an example, consider $\langle X, \leq 7 \wedge = 6 \wedge > 5, Y, \rho\rangle$. It follows that, $\leq 7 \wedge = 6 \wedge > 5)$ becomes the clock constraint $\hat{\mathtt{c}} \leq 7 \wedge \hat{\mathtt{c}} = 6 \wedge \hat{\mathtt{c}} > 5$ ($\mathtt{sat}$), and $\neg(\hat{\mathtt{c}} \leq 7 \wedge \hat{\mathtt{c}} = 6 \wedge \hat{\mathtt{c}} > 5)$ becomes $(\hat{\mathtt{c}} > 7) \vee (\hat{\mathtt{c}} < 6) \vee (\hat{\mathtt{c}} > 6) \vee (\hat{\mathtt{c}} \leq 5)$ from which I generate $\mathtt{viol}_1, \mathtt{viol}_2, \mathtt{viol}_3, \mathtt{viol}_4$ connecting F to V. In this way, the circular path is made of one DAG only having the same number of locations and in general more transitions.

Encoding an RRC containing a TE of Type 2

I encode an RRC having the form $\langle X, \Box Z + k, Y, \rho\rangle$ by building a DAG consisting of the locations Y_i, Y_j, F, V, Z_e and Z_u (see Figure 9.6a) and connecting such locations as discussed below. The meaning of F and V as well as that of the transitions from Y_i to F and to Y_j are the same of those given for RRCs expressing TEs of Type 1 (Figure 9.4c). At F, the RRC has fired and the TEs associated to some resources associated to Y have been updated (possibly differently depending on whether or not Z has been executed). Indeed, if Z has been executed, then $\Box Z + k$ is equivalent to $\Box(t_Z + k)$, where t_Z is the time at which Z executed. Otherwise, $t\Box Z + k$ is either $\top$ or $\bot$ depending on $\Box$. Therefore, instead of connecting F to both V and Y_j as I have done before, I connect F to Z_e and Z_u, modeling the fact that Z has, or has not, been executed, respectively. A run moves to Z_e if and only if Z has been executed ($\mathtt{cZ} < \hat{\mathtt{c}}$), whereas it moves to Z_u if and only if Z has not ($\mathtt{cZ} = \hat{\mathtt{c}}$).

At Z_u we might block some resource(s) according to $\Box$. If $\Box \in \{=, >, \geq\}$, then $t \not\models \Box Z + k$. Therefore, from Z_u to Y_n, there are as many blocking transitions as the number of resources in $\mathcal{R}(X)$. Each one specifies the set R_i as I have done for RRCs having TEs of Type 1. If $\Box \in \{<, \leq\}$, then the run moves to Y_n not blocking any resource (as $t \models \Box Z + k$ implying $R_i = \emptyset$).

At Z_e, we must valuate $\Box Z + k$, therefore this TE becomes the clock constrait $\hat{\mathtt{c}}\Box(\hat{\mathtt{c}} - \mathtt{cZ}) + k$ which simplifies to $\mathtt{cZ}\Box k$ as $\Box Z + k$ is actually $\Box(t_Z + k)$ since Z was executed at $t_Z = \hat{\mathtt{c}} - \mathtt{cZ}$. If $\mathtt{cZ}\Box k$ is true, no resource will be blocked and a $\mathtt{sat}$ transition allows the run to move to Y_n. If $\mathtt{cZ}\Box k$ does not hold, then $\neg(\mathtt{cZ}\Box k) = \mathtt{cZ} \bowtie k$ and, therefore, there are one or two $\mathtt{viol}_i$ transitions allowing the run to move to V. At V the run moves to Y_j by (possibly) blocking some resources for Y and generating again a blocking transition for each resource associated to the firing time point X as I did for RRCs having TEs of Type 1.

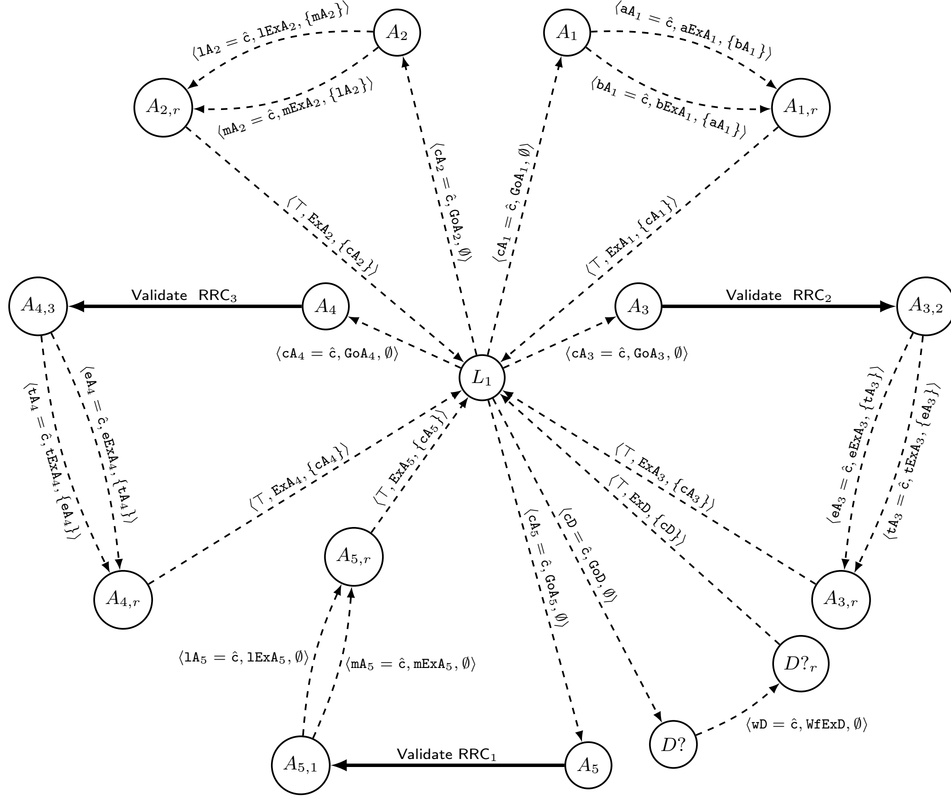
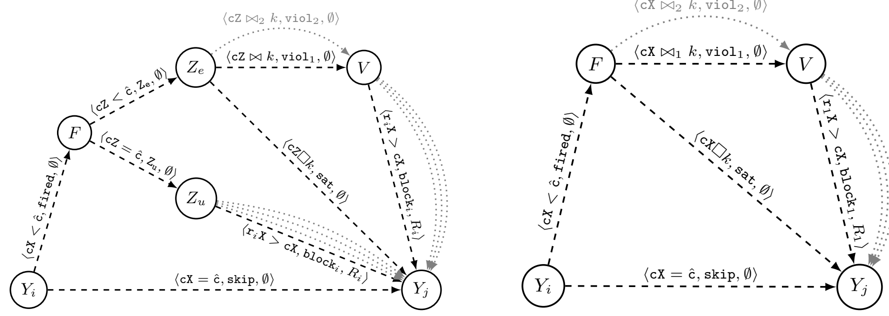


Fig. 9.5: Circular paths modeling the authorized execution of the non-contingent time points of the CSTNUR in Figure 9.3.

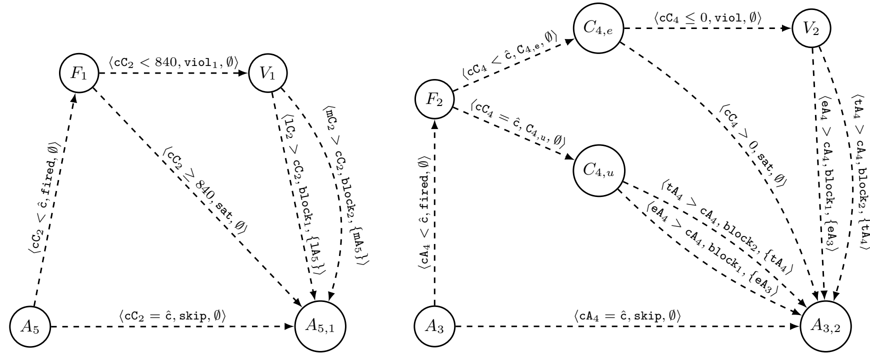
Now that I have discussed the general encoding for RRCs whose TEs are of Type 2, I consider those RRCs where the firing time point and the time point belonging to the embedded TE are the same. That is, those having the form $\langle X, \Box Z + k, \rho, Y \rangle)$ where $Z = X$. For this case, I can remove redundancy of the original encoding given in Figure 9.6a (locations Z_e and Z_u). Figure 9.6b shows such an encoding. Differently from what I discussed for the first case above, this encoding does not result in reducing the number of DAGs, but it avoids generating Z_e and Z_u for each DAG going from Y_{i-1} to Y_i (as done in Figure 9.6a). Indeed, when $X = Z$, keeping Z_e and Z_u would correspond to checking twice if X has been executed or not. Therefore, this encoding first removes Z_e and Z_u along with the transitions to enter these locations and, then, it connects F directly to Y_i and to V, maintaining the same `sat` and $\mathtt{viol}_i$ transitions.

Figure 9.5 shows the circular paths modeling the authorized executions of the non-contingent time points $D?$, A_1, A_2, A_3, A_4 and A_5 of the CSTNUR in Figure 9.3, whereas Figure 9.6c and Figure 9.6d detail the validation of the related RRCs (thick edges in Figure 9.5) for the circular paths modeling the authorized execution of A_3, A_5, respectively. I do not discuss the encoding for A_4 as it is similar to that for A_3.



(a) Encoding normalized RRCs whose TEs are of Type 2 $\langle X, \Box Z + k, Y, \rho \rangle$. A run enters F if only if the RRC has fired ($Y_i \to F$), otherwise it skips the verification ($Y_i \to Y_j$). At F, either Z has been executed ($F \to Z_e$) or not ($F \to Z_u$). At Z_u a few resources might eventually be blocked ($Z_u \to Y_j$). At Z_e either $\Box Z + k$ is satisfied and thus no resource is blocked ($Z_e \to Y_j$), or $\Box Z + k$ is violated ($Z_e \to V$) and a few resources might eventually be blocked ($V \to Y_j$).

(b) Optimizing the encoding of normalized RRCs whose TEs of Type 2 have the form $\langle X, \Box X + k, Y, \rho \rangle$. `skip`, `fired`, `sat`, $\texttt{viol}_i$ and $\texttt{block}_i$ transitions remain the same of those given in Figure 9.6a (they just connect different locations). In fact, at F, the run already knows that X (i.e., Z in Figure 9.6a) has already been executed.



(c) Validation of $\langle C_2, \geq C_2 + 840, A_5, = \rangle$ through the path $A_3 \to A_{3,1}$.

(d) Validation of $\langle A_4, > C_4, A_3, \neq \rangle$ through the path $A_3 \to A_{3,2}$.

Fig. 9.6: Encoding RRCs specifying TEs of Type 2 $\langle X, \Box Z + k, Y, \rho \rangle$, (a) shows the general encoding, (b) refines it for the case $X = Z$ (c) shows the encoding of the RRC going from C_2 to A_5 in Figure 9.3, whereas (d) that of the RRC going from A_4 to A_3. All locations are urgent and all transitions are uncontrollable.

9.4.3 Encoding Contingent Time Points into Contingent Circular Paths

In the encoding for CSTNUs, transitions modeling the execution of the contingent time points are *controllable* self-loop transitions at L_0. In the semantics I gave for CSTNURs in Section 9.3, the two time points of a contingent link (A, x, y, C) must be executed by committing the same resource. Moreover, contingent time points cannot appear as targets in any RRC. Therefore, for each contingent time point the encoding generates a *contingent circular path* similar to that depicted in Figure 9.4b neglecting the validation of RRCs.

The path starts and ends at L_0 and contains two internal *urgent* locations C and C_r (see Figure 9.7a). A unique transition `GoC` goes from L_0 to C, whereas a set of $|\mathcal{R}(A)|$ transitions connects C to C_r. A run can enter C as soon as the clock constraint for contingent time points (i.e., $\mathtt{cA} < \hat{\mathtt{c}} \land \mathtt{cC} = \hat{\mathtt{c}} \land \mathtt{cA} \geq x \land \mathtt{cA} \leq y$) becomes true. `GoC` is unique and does not reset any clock. After that, the resource that was committed for A is committed for C as well by means of a transition $\mathtt{r}_i\mathtt{ExC}$ going from C to C_r. The only enabled transition $\mathtt{r}_i\mathtt{ExC}$ is the one whose guard contains the clock constraint $\mathtt{r}_i\mathtt{C} > \mathtt{cA}$, i.e., the transition associated to the resource r_i committed for A.

Indeed, for all other $r_j \in \mathcal{R}(A)$ where $r_j \neq r_i$, it holds that $\mathtt{r}_j\mathtt{C} = \mathtt{cA}$. Finally, the run moves back to L_0 by resetting `cC`, i.e., fixing the execution time for C.

Figure 9.7 sums up the general pattern for modeling the execution of contingent time points (Figure 9.7a) and the application of such pattern to the contingent time points of the CSTNUR in Figure 9.3 (Figure 9.7b).

9.5 Correctness and Complexity of the Encoding

In this section, I discuss the complexity of encoding a CSTNUR

$$\mathcal{S} = \langle \mathcal{T}, \mathcal{P}, L, \mathcal{OT}, O, \mathcal{C}, \mathcal{LR}, RA, RE, \mathcal{RRC} \rangle$$

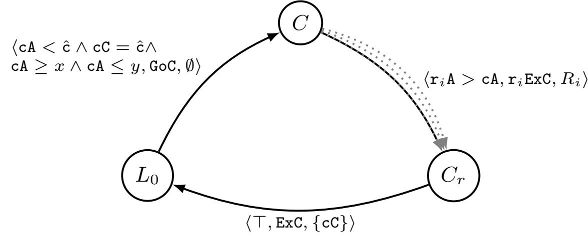
into a TGA. In Section 9.4 I already proved that rewriting RRCs in the suitable form I need to encode them in the corresponding DAGs has linear complexity with respect to the number of conjuncts. The worst case is that in which all RRCs contain TEs τ consisting of the same number of conjuncts where each conjunct is a Type 2 TE. Given any TE τ, I shorten the number of its conjuncts as $|\tau|$. Therefore, the normalization process has complexity

$$\mathit{Normalization}(\mathcal{RRC}) = \mathcal{O}(|\mathcal{RRC}| \times |\tau|)$$

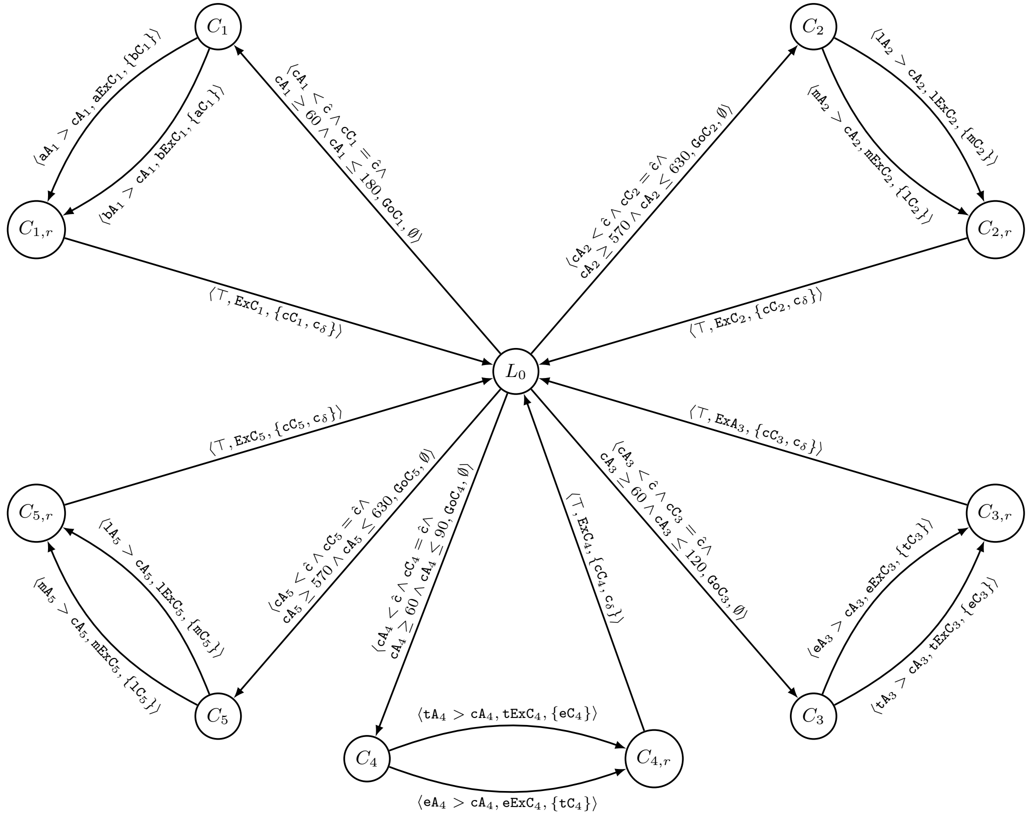
Theorem 9.1. *Any CSTNUR can be encoded into a corresponding TGA in polynomial time.*

Proof. In order to give an estimation of time complexity, I need to estimate the number of generated TGA locations and transitions.

For each non-contingent time point Y, the encoding generates a circular path $Y \to Y_1 \to \cdots \to Y_n \to Y_r$ (cf. Figure 9.4a), where each $Y_{i-1} \to Y_i$ is a DAG handling the validation of the ith RRC containing a conjunction of TEs of Type



(a) Modeling the resource commitment for a generic contingent time point C. This path just commits the same resource r_i that was committed for the corresponding activation time point A. Once A has been executed, only one transition connecting $C \to C_r$ will be enabled. For each of these transitions $R_i = \{\mathtt{r}_j\mathtt{C} \mid r_j \in \mathcal{R}(C) \wedge r_j \neq r_i\}$.



(b) The encoding of C_1, C_2, C_3, C_5 and C_5 of Figure 9.3.

Fig. 9.7: Modeling the executions of contingent time points: (a) shows the pattern for modeling the execution of a contingent time point C, whereas (b) shows the encoding of C_1, C_2, C_3 and C_4 of Figure 9.3.

1 or a TE of Type 2. In the former case, the DAG consists of 4 locations (cf. Figure 9.4d), whereas in the latter it consists of 6 (cf. Figure 9.6a).

In the worst case, all time points different from Y specify RRCs having Y as a target and containing a Type 2 TE. Recall that a lot of RRCs having the same firing and target time points can be specified (e.g., as a result of the normalization process). Therefore, let

$$R(X, Y) = |\{\langle X, \tau, Y, \rho\rangle \mid \langle X, \tau, Y, \rho\rangle \in \mathcal{RRC}\}|$$

be the number of RRCs having X as a firing time point and Y as a target time point, and let, by abusing notation,

$$R(Y) = \sum_{X \in \mathcal{T} \setminus \{Y\}} R(X, Y)$$

be the *total* number of RRCs targeting Y. Further, let

$$M(Y) = \max\{R(X, Y) \mid X \in \mathcal{T} \setminus \{Y\}\}$$

be the size of the biggest $R(X, Y)$. It holds that $R(Y) \leq (|\mathcal{T}| - 1)M(Y) \leq |\mathcal{T}| \times |\mathcal{RRC}|$, where $\mathcal{T}$ is the set of time points. ow, given a time point Y, the number of generated TGA locations is

$$nLoc(Y) = 2 + 5R(Y) \leq 2 + 5((|\mathcal{T}| - 1)M(Y)) = \mathcal{O}(|\mathcal{T}| \times |\mathcal{RRC}|)$$

where the initial 2 takes into account Y and Y_r, whereas each RRC is validated by adding a DAG of 5 (instead of 6) since the source of each DAG coincides with the sink of the previous one (the source of the first DAG is Y).

The number of transitions in each circular path depends (mainly) on the sum of the transitions of each DAG in the path. Since I have already discussed how to compute the set of RRCs involving a time point Y, I continue by discussing how many transitions a DAG validating an RRC expresses. Again, the worst case is when there is an RRC having the form $\langle X, = Z + k, Y, \rho\rangle$, where $X \neq Z$. Such a DAG (cf. Figure 9.6a) consists of 1 `fired` transition, 1 `skip`, 2 transitions to go from F to Z_e and Z_u, 2 `viol` transitions and $|\mathcal{R}(X)|$ `block` transitions connecting V to Y_j and other $|\mathcal{R}(X)|$ `block` transitions connecting Z_u to Y_n. Therefore, the total number of transitions is

$$T(\langle X, = Z + k, Y, \rho\rangle) = 7 + 2|\mathcal{R}(X)| \leq 7 + 2|\mathcal{R}| = \mathcal{O}(|\mathcal{R}|)$$

Now, if all resources are committable for all time points (i.e., $|\mathcal{R}(X)| = |\mathcal{R}|$ for all $X \in \mathcal{T}$), it follows that the above inequality holds for all RRCs belonging to $R(Y)$.

From $R(Y) \leq (|\mathcal{T}| - 1)M(Y)$, it follows that a circular path has the following number of transitions

$$\begin{aligned} nTr(Y) = 2 + |\mathcal{R}(Y)| + \sum_{\langle X, \tau, Y, \rho\rangle \in R(Y)} T(\langle X, \tau, Y, \rho\rangle) &\leq 2 + |\mathcal{R}| + |R(Y)|(7 + 2|\mathcal{R}|) \\ \leq 2 + |\mathcal{R}| + 7|R(Y)| + 2|\mathcal{R}| \times |R(Y)| &= \mathcal{O}(|\mathcal{R}| \times |\mathcal{T}| \times |\mathcal{RRC}|) \end{aligned}$$

The constant 2 is related to `GoY` and `ExY`, whereas $|\mathcal{R}(Y)|$ is the number of all the transitions $Y_n \to Y_r$ for committing a resource (cf. Figure 9.4a).

The above analysis focuses on the number of locations and transitions considered the worst case; that is, a CSTNUR not containing any contingent links and having all time points labeled by $\boxdot$. It is possible to show that if a CSTNUR contains some observation time points or contingent ones, the number of locations and transitions generated by the encoding is smaller. This is because contingent time points cannot appear as targets in any RRCs and RRCs cannot be expressed for a firing time point X and a target time point Y such that $L(X)$ and $L(Y)$ are inconsistent. It follows naturally that restricting the number of possible RRCs results in generating fewer locations and consequently transitions.

The final number of TGA locations is given by the sum of all locations belonging to all circular paths, plus the number of locations W_{loc} encoding the winning path (already proved to be linear in the number of distinct labels present in the input network [26]). Therefore, for any CSTNUR $\mathcal{S}$, the total number of locations is:

$$Locations(\mathcal{S}) = 2 + W_{\text{loc}} + \sum_{X \in \mathcal{T}} nLoc(X) = \mathcal{O}(W_{\text{loc}} + |\mathcal{T}|^2 \times |\mathcal{RRC}|)$$

where 2 is given by the presence of L_1 and L_0. The final number of TGA transitions is given by the total number of transitions for encoding circular paths plus the transitions for encoding the winning path:

$$\begin{aligned} Transitions(\mathcal{S}) &= 2 + W_{\text{tr}} + \sum_{X \in \mathcal{T}} nTr(X) \\ &\leq 2 + W_{\text{tr}} + 2 + |\mathcal{R}| + 7|R(Y)| + 2|\mathcal{R}| \times |R(Y)| \\ &= \mathcal{O}(W_{\text{tr}} + |\mathcal{T}| \times |\mathcal{R}| \times |\mathcal{RRC}|) \end{aligned}$$

Since the number of locations and transitions is polynomial with respect the length of the input network, and the generation of each location and transition takes constant time, the overall time complexity of the encoding is polynomial.

I prove the correctness of the encoding given in Section 9.4 by means of the following two theorems. Such theorems extend those given in [26] proving the correctness of the CSTNU-to-TGA encoding. My extension takes into account resources and RRCs. I first prove prove the equivalence between the execution semantics of the resulting TGA and the semantics of RTEDs (Theorem 9.2), and then that any counter-strategy for `ctrl` synthesized by reachability analysis of the resulting TGA corresponds to an RTED-based strategy (Theorem 9.3).

Theorem 9.2. *Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{P}, L, \mathcal{OT}, O, \mathcal{C}, \mathcal{LR}, RA, RE, \mathcal{RRC} \rangle$ be a CSTNUR and let $\mathcal{G}$ be the encoding of $\mathcal{S}$ into a TGA, as described in Section 9.4. Then $\mathcal{G}$ correctly captures the execution semantics for $\mathcal{S}$ in the sense that any sequence of partial schedules that can be generated for $\mathcal{S}$ according to the execution semantics for CSTNURs corresponds to a run for $\mathcal{G}$ that can be generated by following its transitions according to the TGA semantics.*

Proof. I show that any sequence of partial schedules that can be generated for any CSTNUR according to the execution semantics given in Section 9.3 corresponds

to a run for the equivalent TGA that can be generated by following its transitions according to the classic TGA semantics.

Before introducing the invariant, it is useful to fix the notation about the clocks and state of $\mathcal{G}$. Let $\vec{c} = (\hat{\mathtt{c}}, \mathtt{c}_\delta, \mathtt{cX}, \ldots, \mathtt{cY}, \mathtt{bP}, \ldots, \mathtt{r}_0\mathtt{X}, \ldots, \mathtt{r}_0\mathtt{Y}, \ldots)$ be the vector containing all clocks of $\mathcal{G}$. I write $\vec{c} = k$ and $\vec{c} + k$ meaning that all clocks in $\vec{c}$ are equal to k and are advanced of k, respectively. A state of the TGA $\mathcal{G}$ is a pair $(L, \vec{c})$, where L is a location and $\vec{c}$ represents the values of all clocks. The proof shows by induction that the following invariant holds.

Invariant

Each locally consistent partial schedule that can be generated for $\mathcal{S}$ corresponds to a state $(L, \vec{c})$ of the TGA $\mathcal{G}$ in which:

- $L = L_0$, $\mathtt{c}_\delta = 0$, $\mathit{last} = \hat{\mathtt{c}}$.
- For each executed time point X, $\mathit{time}(X) = \hat{\mathtt{c}} - \mathtt{cX}$ and $\mathit{res}(X) = \mathtt{rX}$, where $\mathtt{rX}$ is the only $\mathtt{r}_i\mathtt{X}$ clock not reset.
- For each unexecuted time point X, $\mathit{time}(X) = \hat{\mathtt{c}}$ and $\mathit{res}(X)$ is not defined (i.e., all $\mathtt{r}_i\mathtt{X} = \hat{\mathtt{c}}$).
- For each executed observation time point $P?$, if $p = \top$ then $\mathtt{bP} = \hat{\mathtt{c}}$, and if $p = \bot$ then $\mathtt{bP} < \hat{\mathtt{c}}$. If the time point is not executed, the value of $\mathtt{bP}$ means nothing.

Base case.

The initial $\mathcal{PS}$ corresponds to the initial state $(L_0, \vec{c})$ where $\vec{c} = 0$. This partial schedule is trivially locally consistent.

Inductive step.

Suppose that $\mathcal{PS}$ is a locally consistent partial schedule that has been generated according to the execution semantics for CSTNURs, and that $\mathcal{PS}$ satisfies the above invariant. Let $(L_i, \vec{c_i})$ be the corresponding state of the TGA. Since $\mathtt{c}_\delta = 0$, the only transitions that are immediately enabled are those entering the contingent circular paths and those that set truth values to propositions. In case a run enters a contingent circular path corresponding to the executing of a contingent time point C, it enters the location C and then it must move to location C_r, representing the event that a resource has been committed, picking the transition having the same resource that was committed for the related activation time point A. Finally, it must move back to L_0. Since the homonymous location C and the location C_r are urgent, time does not elapse. All transitions executed during a walk through contingent circular paths and the transitions modeling the truth value assignments represent the instantaneous reactions of $\mathtt{env}$, in which a set of one or more contingent time points and/or proposition assignments can be executed simultaneously. Suppose that $\mathtt{env}$ does not take any transition when $\mathtt{c}_\delta = 0$. As soon as $\mathtt{c}_\delta > 0$, both $\mathtt{ctrl}$ and $\mathtt{env}$ may execute enabled transitions (i.e., those with true guards). For example, $\mathtt{env}$ might decide to execute one or more contingent time-points $C_1, \ldots, C_n$ when $\mathtt{c}_\delta = 3$. That would correspond to $\Delta_{\mathtt{env}} = (k, \{C_1, \ldots, C_n\})$, where $k = \mathit{last} + 3$.

Each time env takes a transition pFalse to reset the clock associated to transition p (i.e., setting p to $\bot$) or a transition ExC to execute a contingent time point, c_δ is reset to 0, making ctrl unable to interrupt env during the execution of the initiated transition.

Thus, at these time instants, it holds that $\Delta_{\texttt{ctrl}} = \texttt{wait}$ and the resulting outcomes are exactly the ones described in cases 1-2 of Definition 9.8. The guard of the env transition, enforcing the duration bounds for a contingent link (A, x, y, C), ensures that the resulting partial schedule is respectful as C can only be executed in an instant such that $C - A \in [x, y]$. Likewise, for a truth value assignment, the fail transition that ctrl can take (if $\delta > 0$) ensures that env assigns a truth value to a proposition instantaneously after the execution of the observation time point.

Also, when env's sequence of "simultaneous" transitions completes, $\hat{\texttt{c}}$ equals the time of the most recent execution (e.g., $\mathit{last} + 3$). In addition, for each newly executed time-point C, the clock cC is reset and for each $r_i \in \mathcal{R}(C)$, if $\texttt{r}_i\texttt{C} = \hat{\texttt{c}}$ then $\texttt{r}_i\texttt{A} = \hat{\texttt{c}}$ and if $\texttt{r}_i\texttt{C} < \hat{\texttt{c}}$ then $\texttt{r}_i\texttt{A} < \hat{\texttt{c}}$ ensuring that $\hat{\texttt{c}} - \texttt{cC}$ equals the execution time of C. Analogously, for each pFalse taken, it holds that $\texttt{bP} < \hat{\texttt{c}}$. Since cC is reset only once and each proposition is assigned only once, the values of $\hat{\texttt{c}} - \texttt{cC}$ and bP remain fixed forever.

Instead, suppose that ctrl has decided to commit a set of resources to execute a set of non-contingent time points before env executes some contingent time points at time $\mathit{last} + 2$. This situation results in ctrl taking the gain transition to take back control and then, once in its location, instantaneously go through the circular paths (for non-contingent time points) to commit the resources to execute those time points at that time, and immediately returning to the env location by means of the pass transition. Since all the locations but L_0 are urgent, $\hat{\texttt{c}}$ still has value $\mathit{last} + 2$ when the pass transition is taken. The sequence of transitions to go through the circular paths corresponds to the partial outcome in Definition 9.8 (cases 3-4) where $\Delta_{\texttt{ctrl}} = (t, \{(r_1, X_1), \ldots, (r_n, X_n)\})$, $t = \mathit{last} + 2$, and for each $(r, X) \in \mathit{NonContingent}$ (of $\Delta_{\texttt{ctrl}}$), $r \in \mathcal{R}(X)$.

Finally, if at time *last*, ctrl and env both decide to execute some time points at time $\mathit{last} + 1$, then the CSTNUR semantics (inheriting the CSTNU semantics) ensures that ctrl time points are executed first, and that env is able to instantaneously react if it decides to do so (equivalent to ctrl transitions having priority over env). As soon as the execution returns to the location of env, $\hat{\texttt{c}}$ has still the same value $\mathit{last} + 1$ because, again, time has not elapsed. Since, in all cases, the resulting state of the TGA satisfies the desired invariant property, the result is proven.

Theorem 9.3. *Let $\mathcal{S}$ be a CSTNUR, $\mathcal{G}$ be the encoding of $\mathcal{S}$, and $\sigma_{\mathcal{G}}$ be a winning TGA counter-strategy for* ctrl. *Then, there is an equivalent RTED-based strategy $\sigma_{\texttt{ctrl}}$ for* ctrl *that will ensure the satisfaction of all temporal constraints in $\mathcal{S}$ and all RRCs, if fired, whatever the contingent durations and truth value assignments.*

Proof. If $\mathcal{S}$, $\mathcal{G}$, $\sigma_{\mathcal{G}}$ are as assumed, then $\sigma_{\mathcal{G}} \colon S \to \mathit{Act} \cup \texttt{wait}$, where S is the state of the TGA and *Act* the set of ctrl actions (equivalently the set of uncontrollable transitions).

Suppose the TGA has just got into the state $(L_0, \vec{c})$. As I have already remarked, for any time point X associated to clock cX, it holds that:

- $\mathtt{cX} = \hat{\mathtt{c}}$,
- all $\mathtt{r}_i\mathtt{X}$ have value $\hat{\mathtt{c}}$ before X executes, and
- all $\mathtt{r}_i\mathtt{X}$ but one have value $\mathtt{cX} < \hat{\mathtt{c}}$ after X executes. The clock $\mathtt{r}_i\mathtt{X}$ remaining equal to $\hat{\mathtt{c}}$ represents that the corresponding user has been committed to execute X.

For each observation time point $P?$, the associated proposition modeled by $\mathtt{bP}$ is $\mathtt{cP} = \hat{\mathtt{c}}$ (i.e., unknown before $P?$ executes), and either $\mathtt{cP} = \hat{\mathtt{c}}$ (i.e., $\top$) or $\mathtt{cP} < \hat{\mathtt{c}}$ (i.e., $\bot$) after $P?$ has been executed. Thus, $(L_0, \vec{c})$ specifies a partial schedule.

Now, suppose that $\hat{\mathtt{c}} > \mathit{last}$, i.e., that some positive time has elapsed since the last execution event in $\mathcal{PS}$. If nothing has happened, it means that there has been a sequence of `gain` and `pass` transitions going back and forth between `env` and `ctrl` locations. In such a loop, `ctrl` has not executed any non-contingent time point, and `env` has just waited. Let $(L_0, \vec{c}_\prime)$ be a state preceding such loop. Then, for some $\epsilon > 0$, all the clocks in $\vec{c}$ equal those in $\vec{c}_\prime + \epsilon$, and by construction, *last* refers to the clocks in $\vec{c}_\prime$. Next, let $d = \min\{d \mid \sigma_{\mathcal{G}}(L_0, \vec{c}_\prime + d) \neq \mathtt{wait} \wedge \sigma_{\mathcal{G}}(L_0, \vec{c}_\prime + d) \neq \mathtt{pass}\}$ be the minimum time that can elapse from $\vec{c}_\prime$ before the strategy $\sigma_{\mathcal{G}}$ recommends a transition different from `gain` and `pass`, and let $\vec{c}_0 = \vec{c}_\prime + d$. The unique sequence of execution transitions at `ctrl` is $\sigma_{\mathcal{G}}(L_1, \vec{c}_0), \ldots, \sigma_{\mathcal{G}}(L_1, \vec{c}_n)$, where each $\vec{c}_{i+1} = \vec{c}_i$, except for $\mathtt{cX}$ with X the time point executed by $\sigma_{\mathcal{G}}(L_1, \vec{c}_i)$. The termination of this sequence of transitions is guaranteed since time points are finite and can only be executed once. If $\sigma_{\mathcal{G}}(L_1, \vec{c}_n)$ is the last execution transition, then $\mathtt{pass} = \sigma_{\mathcal{G}}(L_1, \vec{c}_n)$. That transition leads back to the state $(L_0, \vec{c}_n)$, where $\vec{c}_n$ is the same as $\vec{c}_0$, except that the clocks for the time points executed by the transitions plus those for resources for those time points, $\sigma_{\mathcal{G}}(L_1, \vec{c}_0), \ldots, \sigma_{\mathcal{G}}(L_1, \vec{c}_n)$, are all 0 in $\vec{c}_n$.

Next, let t be the time at which $\sigma_{\mathcal{G}}$ recommends `ctrl` a non-trivial transition, and *NonContingent* be the set of pairs (resource, time point) corresponding to the execution transitions, $\sigma_{\mathcal{G}}(L_1, \vec{c}_0), \ldots, \sigma_{\mathcal{G}}(L_1, \vec{c}_n)$. Then $(t, \mathit{NonContingent})$ is a $\Delta_{\mathtt{ctrl}}$ corresponding to what the strategy recommends at $(\mathtt{env}, \vec{c}_0)$. Note that `env` may decide to instantaneously react by executing some contingent point at time t too, an outcome that is prevented by the execution semantics for CSTNURs (Definition 9.8, cases 3-4). Finally, `env` may decide to intervene before time t arrives, by executing one or more contingent time-points and effectively generating a new partial schedule $\mathcal{PS}'$. In that case, the same procedure could be applied to $\mathcal{PS}'$ to generate an appropriate $\Delta_{\mathtt{ctrl}}$. Since the guard of `gain` requires a positive time delay, that $\Delta_{\mathtt{ctrl}}$ is properly prohibited from any kind of instantaneous reaction (by `ctrl`). This procedure gives a mapping from any $(L_0, \vec{c})$ state that is reachable following $\sigma_{\mathcal{G}}$.

9.6 Encoding CSTNURs into CDTNUs

Since resources can basically be seen as controllable (discrete) choices, one could fairly wonder if a CSTNUR can be encoded into a *Conditional Disjunctive Temporal Networks with Uncertainty (CDTNU, [26])*, a formalism able to deal with uncontrollable choices, disjunctive uncontrollable durations and disjunctive constraints. The answer is yes and I prove it in the following.

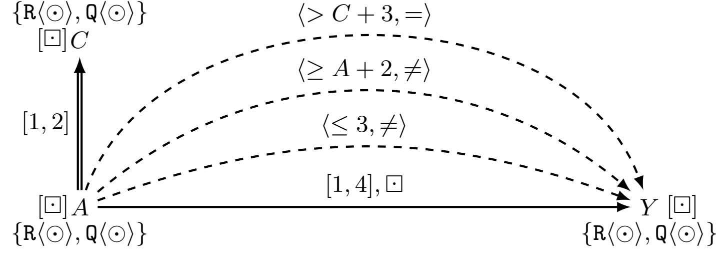


Fig. 9.8: Fragment of CSTNUR.

In what follows I will consider streamlined models of the temporal networks under analysis obtained in a similar way of those for CSTNs (I just consider a bigger horizon to address upper bounds of contingent links).

Definition 9.10. *A* Conditional Disjunctive Temporal Network with Uncertainty (CDTNU) *is a tuple* $\langle \mathcal{T}, \mathcal{C}, L, \mathcal{OT}, \mathcal{O}, \mathcal{P}, \mathcal{L} \rangle$*, where*

- $\mathcal{T}, L, \mathcal{OT}, \mathcal{O}, \mathcal{P}$ *are the same as those given for a CSTNU (Definition 2.15)*
- $\mathcal{C}$ *is a set of constraints* (ϕ, ℓ) *where* ϕ *is an arbitrary Boolean combination of atoms of the form* $Y - X \leq k$ *for* $X, Y \in \mathcal{T}$ *and* $k \in \mathbb{R}$ *and* $\ell \in \mathcal{P}^*$
- $\mathcal{L}$ *is a set of contingent links* $(A, \mathcal{B}, C)$*, where* $A, C \in \mathcal{T}$ *and* $\mathcal{B}$ *is a finite set of (disjoint) ranges* $[x, y]$ *such that* $0 < x < y < \infty$

When each contingent link specifies exactly one range and all constraints (ϕ, ℓ) *are such that* ϕ *does not contain any disjunction, then the CDTNU boils down to a classic CSTNU.*

For example, if $\mathcal{C}$ contains $\phi = ((Y - X \leq -3) \vee (Y - X \geq 4), p\neg q)$ it means that whenever p is true and q is false, then the controller must schedule Y and X in a way that must satisfy *either* $(Y - X \leq -3)$ *or* $(Y - X \geq 4)$. Instead, a contingent link $(A, \{[2, 3] \vee [5, 7]\}, C)$ means that once the controller executes A, the environment first chooses to assign *either* $[2, 3]$ *or* $[5, 7]$ to the contingent link and then it schedules C such that $C - A$ belongs to the chosen range.

I compact the notation for constraints and write $Y = X$ as a short for $Y - X = 0$, $Y - X = k$ as a short for $Y - X \leq k$ and $X - Y \leq -k$, and $X = k$, $X \leq k$, ... as shorts for $X - Z = k$, $X - Z \leq k$, ..., where Z is the zero time point. I might also use $Y - X > k, Y - X < k, Y - X \geq k$ or $Y \geq X + k$ and negations of all these atoms.

Consider the fragment of CSTNUR in Figure 9.8, where $h = 4$ and $h^* = 5$.

The first problem we come across is that of encoding resources and their commitment for time point executions. All *unlabeled* time points (observation ones included), contingent links and *labeled* constraints in the streamlined CSTNUR belong to the CDTNU too (note that contingent links do not turn disjunctive in the CDTNU). I assume to have an extra time point Z in the CSTNUR such that $\mathcal{R}(Z) = \{r^*\}$ and that no RRC will have Z neither as firing nor target time point. To model time points' associated resources, for each time point X (in the CSTNUR) such that $\mathcal{R}(X) = \{r_1, \ldots, r_n\}$, I add $|\mathcal{R}(X)|$ time points

$$X_{r_1}, \ldots, X_{r_n}$$

to the CDTNU such that $L(X_{r_1}) = \cdots = L(X_{r_n}) = \boxdot$. Each of these X_{r_i} can only be assigned two values: the same value that X gets during execution or h^*. If $X_{r_i} = X$, then it means that r_i is committed to execute X, whereas if $X_{r_i} = h^*$, then it means that r_i is not committed. Since I do not have a way to *exclude* some time point X_{r_i} from the execution, I follow the ideas of streamlined models and distinguish between executed or not executed time points by reasoning on the horizon. If a time point X_{r_i} is executed *within the horizon*, then r_i was committed for X, whereas if X_{r_i} is executed *after the horizon*, then r_i was not committed.

Now, I must enforce that for each *non-contingent* time point X in the CSTNUR one and only one associated resource is committed for its execution. I model this condition in the CDTNU with the constraint

$$(\overbrace{(X_{r_1} = X \wedge \cdots \wedge X_{r_j} = h^*)}^{r_1 \text{ is committed for } X} \vee \cdots \vee \overbrace{(X_{r_1} = h^* \wedge \cdots \wedge X_{r_j} = X)}^{r_n \text{ is committed for } X}, \boxdot)$$

In Figure 9.8 I have that $\mathcal{R}(A) = \mathcal{R}(Y) = \{\mathtt{R}, \mathtt{Q}\}$, thus I add to the CDTNU the time points $Z, A_R, A_Q, C_R, C_Q, Y_R, Y_Q$ (recall that, $L(Z) = L(A_R) = L(A_Q) = L(C_Q) = L(C_Q) = L(Y_R) = L(Y_Q) = \boxdot$) and the constraints

- $(\underbrace{Z_{r^*} = Z \vee Z_{r^*} = h^*}_{\phi_1}, \boxdot)$ (but this constraint really doesn't matter)
- $(\underbrace{\overbrace{(A_R = A \wedge A_Q = h^*)}^{\mathtt{R} \text{ is committed for } A} \vee \overbrace{(A_R = h^* \wedge A_Q = A)}^{\mathtt{Q} \text{ is committed for } A}}_{\phi_2}, \boxdot)$
- $(\underbrace{\overbrace{(Y_R = Y \wedge Y_Q = h^*)}^{\mathtt{R} \text{ is committed for } Y} \vee \overbrace{(Y_R = h^* \wedge Y_Q = Y)}^{\mathtt{Q} \text{ is committed for } Y}}_{\phi_3}, \boxdot)$

to model the resource commitment for Z, A and Y

For contingent time points I must commit the same resource that was committed for the related activation. I do so as follows. For each $(A, x, y, C) \in \mathcal{L}$ such that $\mathcal{R}(A) = \mathcal{R}(C) = \{r_1, \ldots, r_n\}$ I add the following constraint.

$$(\overbrace{(A = A_{r_1} \wedge C = C_{r_1}) \vee \cdots \vee (A = A_{r_n} \wedge C = C_{r_n})}^{\text{Commit the same resource for } C}, \boxdot)$$

In Figure 9.8, I have the contingent link $(A, 1, 2, C)$, therefore I add

$$(\underbrace{(A = A_R \wedge C = C_R) \vee (A = A_Q \wedge C = C_Q)}_{\phi_4}, \boxdot)$$

I am left to model RRCs in the CDTNU. I shorten the discussion focusing on the RRCs of Figure 9.8. Since an RRC between two time points X, Y either fires or do not fire depending on the order of execution of X and Y, I must hard code a condition to understand which time point executes first. Note that this is necessary to handle limit cases where a temporal constraint $[0, 0]$ is specified between X and Y. Therefore, for every RRC $\langle X, \tau, \rho, Y\rangle$ I add a time point X_Y ($L(X_Y) = \boxdot$) and

add the disjunctive constraint $(X_Y = X \vee X_Y = h^*, \boxdot)$. If $X_Y = X$, then X is executed before Y (even when $Y - X = 0$). If I have another RRC $\langle Y, \tau, \rho, X\rangle$, I will add a Y_X and $(Y_X = X \vee Y_X = h^*, \boxdot)$ and also $((X_Y = X \wedge Y_X = h^*) \vee (X_Y = h^* \wedge Y_X = Y), \boxdot)$ (either X is before Y or the contrary). If X executes before Y, then the temporal constraint $X - Y \leq 0$ must hold. Therefore for each $\langle X, \tau, \rho, Y\rangle$ I add the pair of constraints

- $(X_Y = X \Rightarrow X - Y \leq 0, \boxdot)$
- $(X_Y = h^* \Rightarrow X - Y \geq 0, \boxdot)$

Consider the RRC $A \to Y$ labeled by $\langle \leq 3, \neq\rangle$ in Figure 9.8. The condition I need to model is: *If A is executed before Y and the resources committed for A and Y are different, then Y must occur within global time* 3. That is, I add A_Y and the constraints

- $(\underbrace{A_Y = A \vee A_Y = h^*}_{\phi_5}, \boxdot)$
- $(\underbrace{A_Y = A \Rightarrow A - Y \leq 0}_{\phi_6}, \boxdot)$
- $(\underbrace{A_Y = h^* \Rightarrow A - Y \geq 0}_{\phi_7}, \boxdot)$
- $(\underbrace{A_Y = A \wedge \overbrace{((A_R = A \wedge Y_Q = Y) \vee (A_Q = A \wedge Y_R = Y))}^{\text{Resources committed for } A \text{ and } Y \text{ are different}} \Rightarrow Y \leq 3}_{\phi_8}, \boxdot)$

Consider the RRC $A \to Y$ labeled by $\langle \geq A+2, =\rangle$ in Figure 9.8. The condition I need to model is: *If A is executed before Y and the resources committed for A and Y are equal, then Y must be executed after minimum 2 since A.* That is, I add the constraint

$$(\underbrace{A_Y = A \wedge \overbrace{((A_R = A \wedge Y_R = Y) \vee (A_Q = A \wedge Y_Q = Y))}^{\text{Resources committed for } A \text{ and } Y \text{ are equal}} \Rightarrow Y \geq A + 2}_{\phi_9}, \boxdot)$$

Finally, consider the RRC $A \to Y$ labeled by $\langle > C + 3, =\rangle$ in Figure 9.8. The condition I need to model is: *If A is executed before Y and the resources committed for A and Y are equal, then ((if C has already been executed, then Y must be executed after 3 since C), whereas (if C has not been executed, then no solution exists).* Since once fired, RRCs involve the execution of Y, I encode "C has already been executed" as "C is executed *before* Y", and "C has not been executed yet" as "C is executed *after* Y" ("whereas" here means "and"). Moreover, the "no solution exists" is because if C is still unexecuted by the time Y executes, then equal resources have associated temporal expressions which are not satisfied by the current time. Therefore, I add C_Y and the constraints

- $(\underbrace{C_Y = C \vee C_Y = h^*}_{\phi_{10}}, \boxdot)$
- $(\underbrace{C_Y = C \Rightarrow C - Y \leq 0}_{\phi_{11}}, \boxdot)$

- $(\underbrace{C_Y = h^* \Rightarrow C - Y \geq 0}_{\phi_{12}}, \boxdot)$
- (ϕ_{13}, ℓ) is $((A_Y = A \wedge \overbrace{((A_R = A \wedge Y_R = Y) \vee (A_Q = A \wedge Y_Q = Y))}^{\text{Resources committed for } A \text{ and } C \text{ are equal}}) \Rightarrow$ $((\underbrace{C_Y = C \Rightarrow Y - C \leq 3}_{\text{First case}}) \wedge \underbrace{(C_Y = h^* \Rightarrow \underbrace{Y - Y \leq -1}_{\text{No sol. exists}})}_{\text{Second case}}), \boxdot)$

Note that "no solution exists" is modeled as negative self loops. In this example I used -1 as weight for $Y \to Y$, but any negative real value or any unsatisfiable constraint fulfills this purpose: "break the execution by using this".

Similar encodings apply for the other cases of RRCs with respect to the specific $\square$.

Finally, $\mathcal{C}$ consists of the native temporal constraints of the initial CSTNUR plus $(\phi_1, \boxdot) \wedge \cdots \wedge (\phi_{13}, \boxdot)$ that can be compacted as $(\mathtt{CNF}((\phi_1) \wedge \cdots \wedge (\phi_{13})), \boxdot)$.

After that, dynamic controllability of the CDTNU can be checked by using the methods in [26]. I point out that since resource commitments are Boolean conditions (any resource for any time point is either committed or not), when computing the conjunctive normal forms, I can safely impose that $\neg(X_{r_i} = X)$ is equivalent to $X_{r_i} = h^*$ (like I did for clocks modeling Boolean propositions in the TGA encodings where $\neg(\mathtt{bP} < \hat{\mathtt{c}})$ becomes $\mathtt{bP} = \hat{\mathtt{c}}$).

Such an encoding keeps a polynomial number of time points and constraints with respect to the size of the initial CSTNUR.

9.7 A possible implementation with UPPAAL-TIGA

As a proof of concept, I wrote the specification of the TGA encoding the CSTNUR depicted in Figure 9.3 and ran UPPAAL-TIGA to answer to the decision problem of dynamic controllability. The example is available at `http://regis.di.univr.it/FlightExample.tar.bz2`. I took advantage of Boolean variables to represent propositions and the RA relation[5]

I used a FreeBSD virtual machine running on top of a VMWare ESXi hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM for the experimental evaluation. The VM was assigned 16GB of RAM and full CPU power.

I verified that the CSTNUR in Figure 9.3 is dynamically controllable. The model checking phase took 207 minutes and 28 seconds to synthesize a 1.6MB memoryless execution strategy as a *certificate of YES* for this decision problem. Such a strategy consists of statements like *state* $\to$ *action*, where *state* abstracts conditions over clock constraints (and Boolean variables), whereas *action* says either to take a specific transition or to wait. Figure 9.9 shows what the TGA encoding the CSTNUR in Figure 9.3 looks like in UPPAAL-TIGA.

[5] For each proposition p, $p = \top$ (resp., $p = \bot$) means $\mathtt{bP} = \hat{\mathtt{c}}$ (resp., $\mathtt{bP} < \hat{\mathtt{c}}$). For each $(u, X) \in RA$, $uX = \top$ means $\mathtt{uX} > \hat{\mathtt{c}}$ if $\mathtt{cX} < \hat{\mathtt{c}}$ (u executed X), $\mathtt{uX} = \hat{\mathtt{c}}$ if $\mathtt{cX} = \hat{\mathtt{c}}$ (u is available), whereas $uX = \bot$ means $\mathtt{uX} = \mathtt{cX}$ if $\mathtt{cX} < \hat{\mathtt{c}}$ (u did not execute X) or $\mathtt{uX} < \mathtt{cX}$ if $\mathtt{cX} = \hat{\mathtt{c}}$ (u has been blocked).

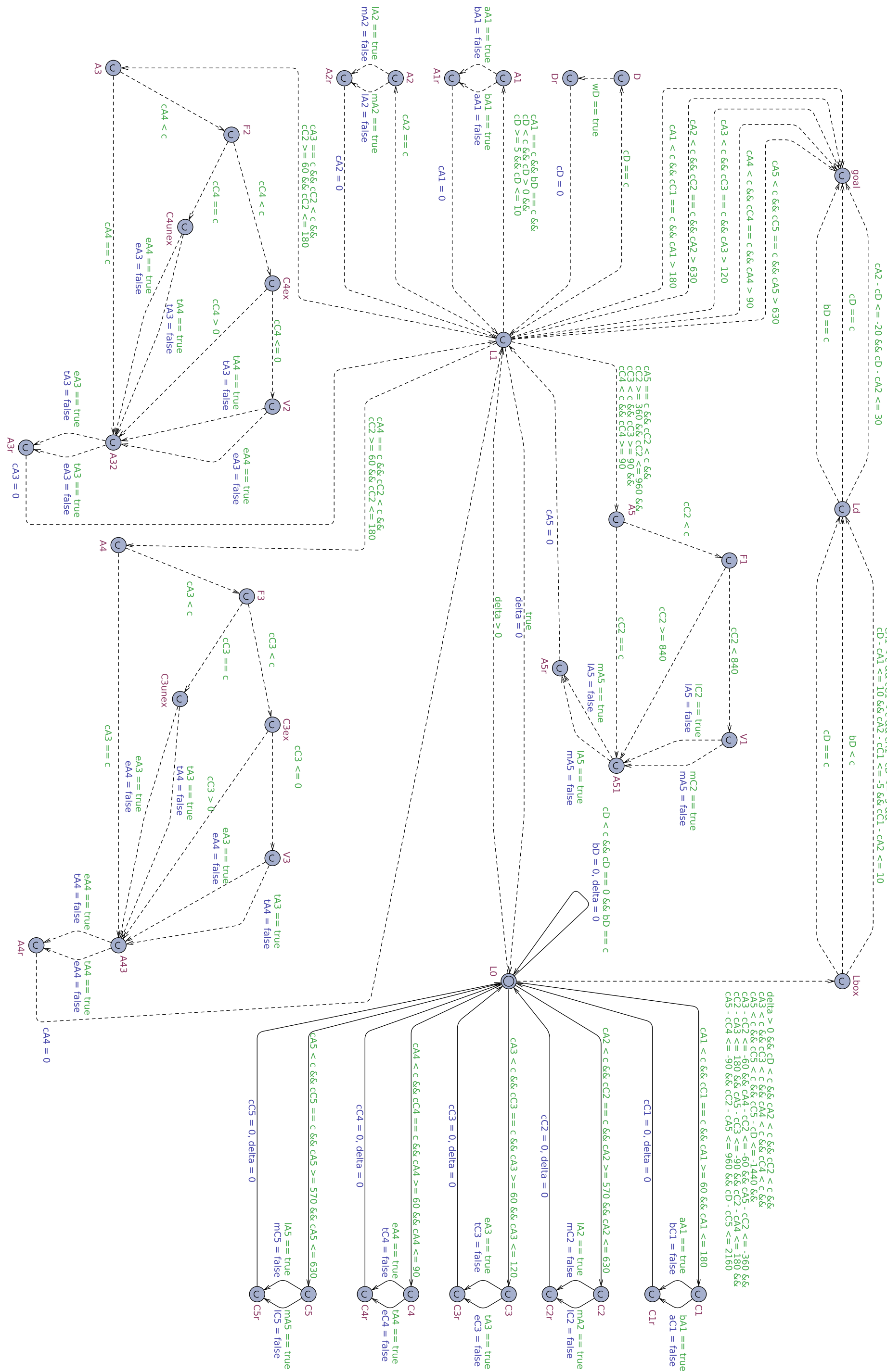


Fig. 9.9: Modeling and validation of Figure 9.3 with UPPAAL-TIGA.

9.8 TRBAC on top of CSTNURs (and ACTNs)

In Chapter 8 and Chapter 9, I provided two different formalisms of temporal networks to deal with temporal and resource controllability simultaneously. ACTNs and CSTNURs mainly differ from each other for the type of employed constraints. ACTNs define disjunctive temporal authorization constraints between pairs of non contingent time points, whereas CSTNURs associate temporal expressions to resources and provide runtime resource constraints to operate on these temporal expressions. However, in a workflow context, users (belonging to roles) might also be subject to further temporal constraints on their "daily or weekly availability". For example, some users could only work during a night shift, whereas some others during the day. To model this further condition, we need to understand whether a workflow can be executed with respect to a given fragment of TRBAC. Once again, in such a model, the set of permissions consists of the workflow tasks, and the interpretation of the role-permission assignment relation $(R, T) \in PA$ is "all users belonging to the role R are authorized to execute the task T".

9.8.1 From granules to real time instants

Section 2.4 showed how to convert a symbolic periodic expression in a temporal constraint consisting of a periodic constraint that models the periodicity of the expression plus a gap-order constraint which limits its applicability.

However, each element of this temporal constraint (an integer) represents a granule in the minimal granularity (e.g., "an hour"). The problem is that time points in temporal networks model instants and not intervals. Therefore, it would not make any sense to talk about an event arrived in the first hour, since this way of reasoning would model that the handling of such an event took one entire hour.

To give another example, assume that a user has to execute a task T in a workflow. Normally, from a temporal point of view the execution of a task can be viewed as the execution of a start point plus the execution of an end point. What if that task lasted exactly 1 hour? We could not simply execute its starting point at the nth hour and its finishing point at the $(n+1)^{\text{th}}$ hour, because it would mean that the task took 2 hours, but worse still (to be more detailed) that the start took 1 hour (whatever it means) and the end another one. Of course, it does not make sense to execute both the start and the end at the same hour either, since we would not be able to understand which event has come first, thus leading to consider situations such as "a task ends before starting" or "a task starts after ending".

We need to see a granule as an indivisible interval which has a start and an end point. For example, we can interpret that the first hour of the first day of 2015 starts at real time $t = 0$ and lasts up to $t = 1$, the second from $t = 1$ (coinciding with the end of the first) to $t = 2$, and so forth. Likewise, the start of the second week of 2015 is $t = 240$ as that week starts in the 241th hour.

To give a few examples I show a translation of the periodic expressions given in Table 9.1 limited to the first complete five days of 2015 (i.e., limited to `[01/01/15,05/01/15]`).

Assuming *Hours* as the minimum granularity, the corresponding granules of the the gap order constraint are $g_b = 1$ corresponding to the first hour of the

Table 9.1: Examples of periodic expressions.

Name	Periodic Expression
`NightTime`	$All \cdot Days + \{20\} \cdot Hours \triangleright \{12\} \cdot Hours$
`DayTime`	$All \cdot Days + \{8\} \cdot Hours \triangleright \{12\} \cdot Hours$
`EveryTwoHours`	$All \cdot Days + \{1,5,9,13,17,21\} \cdot Hours \triangleright \{2\} \cdot Hours$
`Mondays`	$All \cdot Weeks + \{1\} \cdot Days \triangleright \{1\} \cdot Days$

Table 9.2: Displacement, Periodicity and Granularity of the periodic expressions given in Table 9.1.

PE	*Displacement*(*PE*)	*Periodicity*(*PE*)	*Granularity*(*PE*)
`NightTime`	20	24	12
`DayTime`	8	24	12
`EveryTwoHours`	$\{1,5,9,13,17,21\}$	24	2
`Mondays`	97	168	24

first day of 2015 and $g_e = 120$ corresponding to the last hour of the fifth day. The corresponding instants are $g_b - 1$ (start) and g_e itself (end). Therefore, the gap-order constraint is given by the real time interval $[g_b - 1, g_e]$.

The characteristics of the periodicity constraints PCs for the periodic expressions are given in Table 9.2 where, for example `NightTime` has a displacement of 8 as the night starts at the 20th hour of the day (i.e. at time $t = 19$), a periodicity of 24 hours and a granularity of 12 granules (hours). I give a graphical representation of the periodic expressions of Table 9.1 in Figure 9.10. I recall that, $Periodicity(P)$ is the number of time units in which P repeats, $Granularity(P)$ the duration of each spanned interval and $Displacement(P)$ the set of integers representing the starting points of the spanned intervals.

9.8.2 From Periodic Expressions to STNs

In the previous section I discussed the need to work with instants and real time intervals instead of granules (sets of integers). I proceed to show how a periodic expression P can be unfolded and translated as an STN, where time points correspond to the start and end of the granules of P.

Once I have translated P in a periodicity constraint plus a gap order constraint, I can get real time instants delimiting the intervals spanned by P itself and finally generate an equivalent STN representing them. Note that for an STN to be generated it is important that the upper bound of the interval limiting the applicability of the expression P is $\neq \infty$.

Consider `DayTime` in Figure 9.10 and assume that the instant 0 refers to the starting instant of the first hour in 2015. Then, the corresponding real time intervals are:

$$\texttt{DayTime} = [7, 19] \cup [31, 43] \cup [55, 67] \cup [79, 91] \cup [103, 115]$$

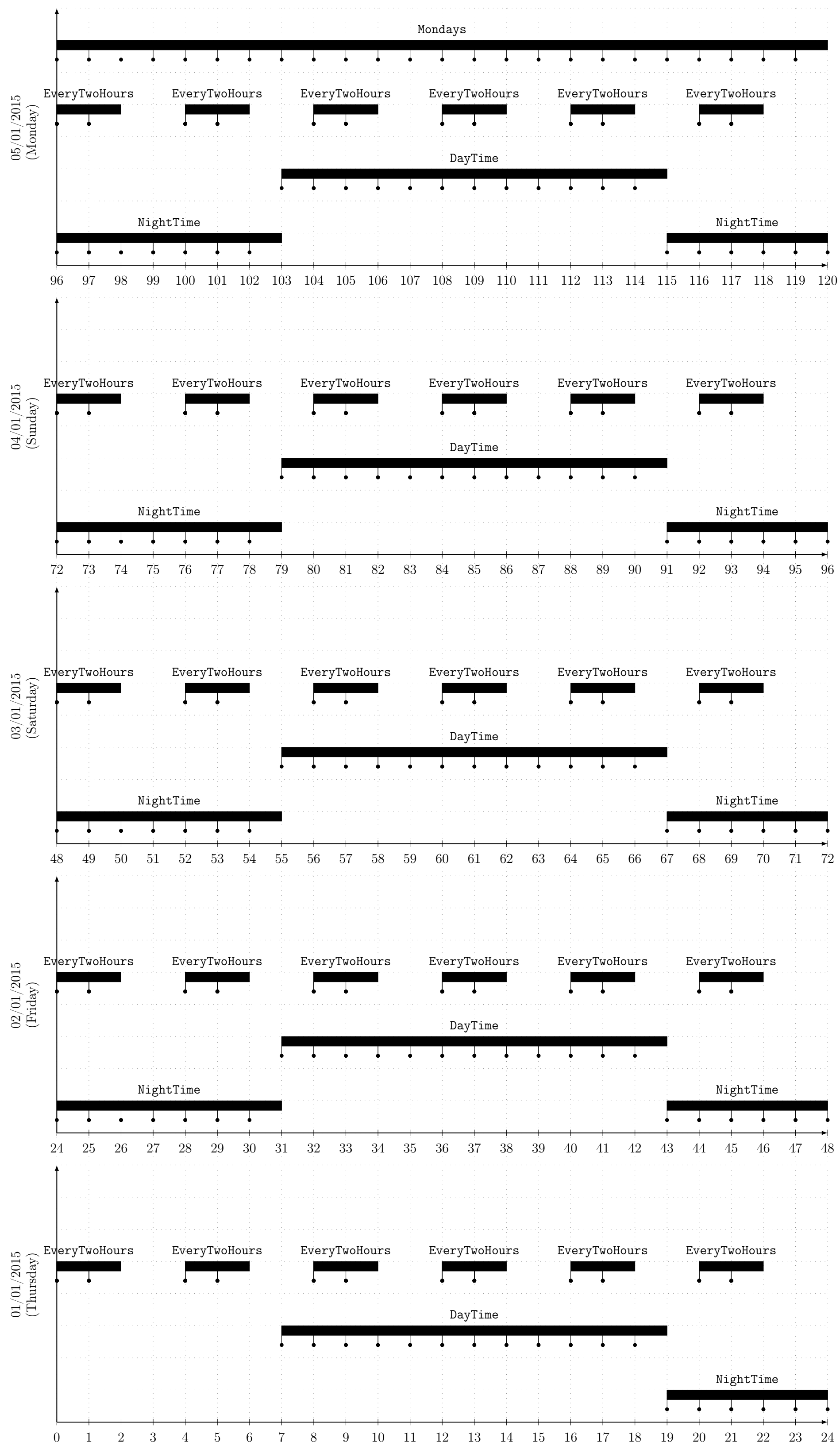


Fig. 9.10: Graphical representations of the periodic expressions in Table 9.1 within the first five days of 2015. Black bars represent calendars, bullets starting instants of granules.

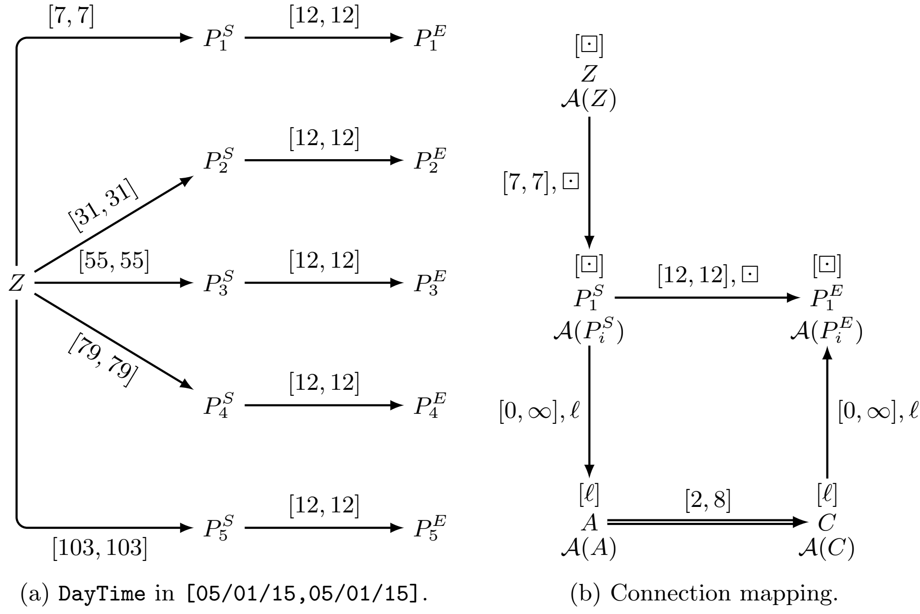


Fig. 9.11: Modeling role temporalities of a TRBAC by means of an STN (a), and enforcing tasks to be executed when the authorized roles are enabled (b).

and the resulting STN is represented in Figure 9.11a where time point Z is executed at time 0, each P_i^S is the starting instant of a `DayTime` interval and each P_i^E the ending instant. For example, the first real interval $[7, 19]$ starts at $P_1^S = 7$ and ends at $P_1^E = P_1^S + 12 = 19$.

Theorem 9.4. *Given any periodic expression P whose applicability is limited by a real interval whose upper bound is $\neq \infty$, the equivalent STN is (i) consistent and (ii) admits exactly one solution.*

Proof. It follows from the fact that the periodic events I consider are *non-conflicting* and *complementary*, so translating a periodic expression P generates an STN which does not contain any negative loop.

Suppose now that a role R is enabled during `DayTime` and it is authorized to execute a task T represented as a contingent link (A, x, y, C). To enforce that such a task is executed when R is enabled, I connect the STN describing the temporal constraints of TRBAC to the network describing the temporal plan.

All I have to do is to impose that the start of T has to occur *after* P_i^S, whereas the end *before* P_i^E for some interval $[P_i^S, P_i^E]$ in which R is enabled. In other words, role R cannot start T before getting `enabled`, and cannot end it after getting `disabled`. Figure 9.11b shows an example in which the contingent link $(A, 2, 8, C)$ (modeling some task T lasting from 2 to 8 hours) is constrained to be executed in the first interval spanned by `DayTime` (i.e., on January 1, 2015 from 7AM to 7PM). Since the contingent link lasts minimum 2 and maximum 8 hours, it is pretty clear that for the augmented network in Figure 9.11b to be

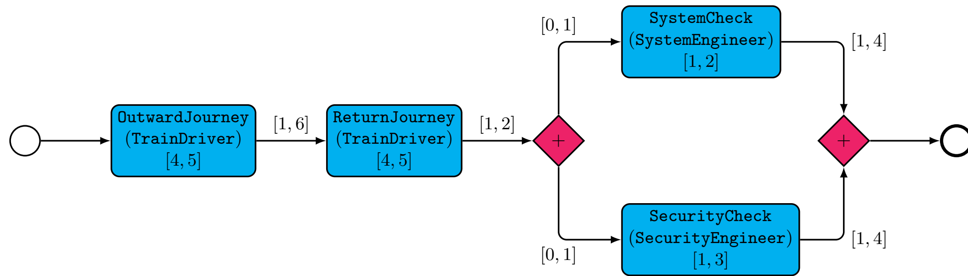


Fig. 9.12: Access-controlled workflow.

controllable, we must start T (i.e., execute A_1) within global time $t = 11$. If we don't, C could occur at time $t > 19$ if it takes its maximal duration. Note that since I am about to enforce a TRBAC on CSTNURs (and ACTNs), when I connect the STN representing the temporal constraints of the access control model to the CSTNUR or ACTN representing the workflow, I add a fresh user (`wf`) to execute the time points of this STN, and labels if the CSTNURs or ACTNs representing the workflow specifies conditional constraints. In this way, I turn the STN into a CSTNUR or into an ACTN. Either way, I do not specify RRCs (CSTNURs) nor disjunctive authorization constraints (ACTNs) between the time points of the part of the network modeling the TRBAC.

9.8.3 Case study

I consider a workflow modeling a round-trip from London to Edinburgh. It starts with the task `OutwardJourney` in which a train travels from London to Edinburgh. The journey takes from 4 to 5 hours to be completed. After the train has arrived to Edinburgh train station, the `ReturnJourney` to London starts within 5 hours since 1 hour after the arrival. Once the train has returned, before the next round trip starts, a `SecurityCheck` and a `SystemCheck` are done in parallel. The first check takes 1 to 2 hours, the second 1 to 3 hours. Figure 9.12 shows the workflow consisting of 4 tasks, 3 roles and 5 users. The instantiation of the TRBAC (for the workflow) is as follows:

- $\texttt{Users} = \{\texttt{Alice}, \texttt{Bob}, \texttt{Charlie}, \texttt{Eve}, \texttt{Kate}\}$
- $\texttt{Roles} = \{\texttt{TrainDriver}, \texttt{SystemEngineer}, \texttt{SecurityEngineer}\}$
- $\texttt{Tasks} = \{\texttt{OutwardJourney}, \texttt{ReturnJourney}, \texttt{SystemCheck}, \texttt{SecurityCheck}\}$
- $UA = \{(\texttt{Alice}, \texttt{TrainDriver}), (\texttt{Bob}, \texttt{TrainDriver}), (\texttt{Charlie}, \texttt{SystemEngineer}),$ $(\texttt{Charlie}, \texttt{SecurityEngineer}), (\texttt{Eve}, \texttt{SecurityEngineer}), (\texttt{Kate}, \texttt{SystemEngineer})\}$
- $TA = \{(\texttt{TrainDriver}, \texttt{OutwardJourney}), (\texttt{TrainDriver}, \texttt{ReturnJourney}),$ $(\texttt{SystemEngineer}, \texttt{SystemCheck}), (\texttt{SecurityEngineer}, \texttt{SecurityCheck})\}$

Figure 9.13 shows the role enabling base. I recall that each line represents a periodic event as described in Section 2.5[6]. The periodic events in Figure 9.13 say that:

[6] I also assume that $\mathcal{R}$ contains the complementary expressions to disable the roles.

$\mathcal{R}$

$\texttt{PE}_1 : (\texttt{[01/01/15,}\infty\texttt{]},\ all \cdot Days + \{9\} \cdot Hours \triangleright \{12\} \cdot Hours\texttt{,}\ \texttt{enable TrainDriver})$
$\texttt{PE}_2 : (\texttt{[01/01/15,}\infty\texttt{]},\ all \cdot Days + \{16\} \cdot Hours \triangleright \{9\} \cdot Hours\texttt{,}\ \texttt{enable SystemEngineer})$
$\texttt{PE}_3 : (\texttt{[01/01/15,}\infty\texttt{]},\ all \cdot Days + \{16\} \cdot Hours \triangleright \{12\} \cdot Hours\texttt{,}\ \texttt{enable SecurityEngineer})$

Fig. 9.13: The Role Enabling Base of the case study.

$\texttt{PE}_1$ `TrainDriver` is enabled every day from 8AM to 8PM
$\texttt{PE}_2$ `SystemEngineer` is enabled every day from 3PM to 12AM (midnight)
$\texttt{PE}_3$ `SecurityEngineer` is enabled every day from 3PM until 3AM of the day after

Let me consider the time window `[01/01/15:01,02/01/15:03]` so that `TrainDriver` is enabled during $[8, 20]$, `SystemEngineer` during $[15, 24]$ and `SecurityEngineer` during $[15, 27]$. Furthermore, this example enforces the following two security policies.

$\texttt{SP}_1$ A user is allowed to execute no more than one task at a time
$\texttt{SP}_2$ If the train driver from Edinburgh to London is the same as the one who drove the train from London to Edinburgh, he must rest at least 2 hours before driving again.

The corresponding CSTNUR modeling this case study is given in Figure 9.14, whereas the ACTN in Figure 9.15. In both networks, Z and E are the starting and ending time points of the workflow. Authorized users are derived the same way I did in Section 6.6 (i.e., as the union of all users belonging to the roles authorized for the tasks). RRCs for the CSTNUR and disjunctive authorization constraints for the ACTN are detailed in the corresponding captions. Since both the CSTNUR and the ACTN embed the STN modeling the TRBAC part of the example, and adding a TRBAC part does not require to extend CSTNURs nor ACTNs[7], I can exploit the previous algorithms discussed in Chapter 8 and Chapter 9 to check dynamic controllability of these networks.

9.9 Conclusions

I defined *conditional simple temporal networks with uncertainty and resources (CSTNURs)* by extending CSTNUs with resources, temporal expressions and run-time resource constraints (RRCs). RRCs are a new class of constraints able to refine in real time the temporal expressions associated to the resources depending on the specific execution. Resources are associated to time points and must be committed for their execution, whereas RRCs enforce (temporal) security policies such as temporal separation and binding of duties. I extended the encoding in [26] (with the optimizations in Section 5.3) from CSTNUs into TGAs in order to do the DC-checking. I also discussed a few optimizations and proved that any CSTNUR

[7] Because the STN becomes a CSTNUR or ACTN depending on the specific case, all labels are $\boxdot$ and the fresh user for the time points is, for example, `wf`.

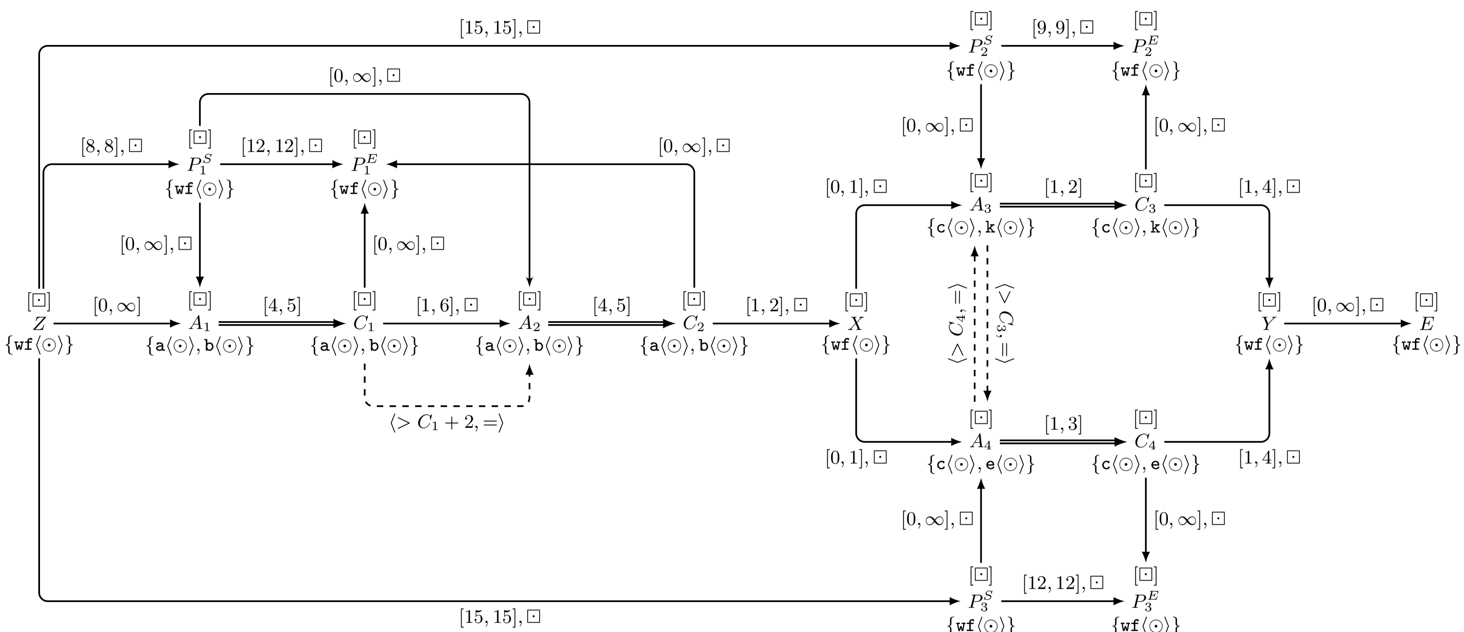


Fig. 9.14: CSTNUR equivalent to the access-controlled workflow depicted in Figure 9.12. Users `a`, `b`, `c`, `e`, `k` represent `Alice`, `Bob`, `Charlie`, `Eve`, and `Kate`, respectively. Since CSTNURs do not automatically prevent the same user from executing more than one task simultaneously, I model $\texttt{SP}_1$ with the pair of RRCs $\langle A_3, > C_3, A_4, = \rangle$ and $\langle A_4, > C_4, A_3, = \rangle$, whereas $\texttt{SP}_2$ with $\langle C_1, > C_1 + 2, A_2, = \rangle$.

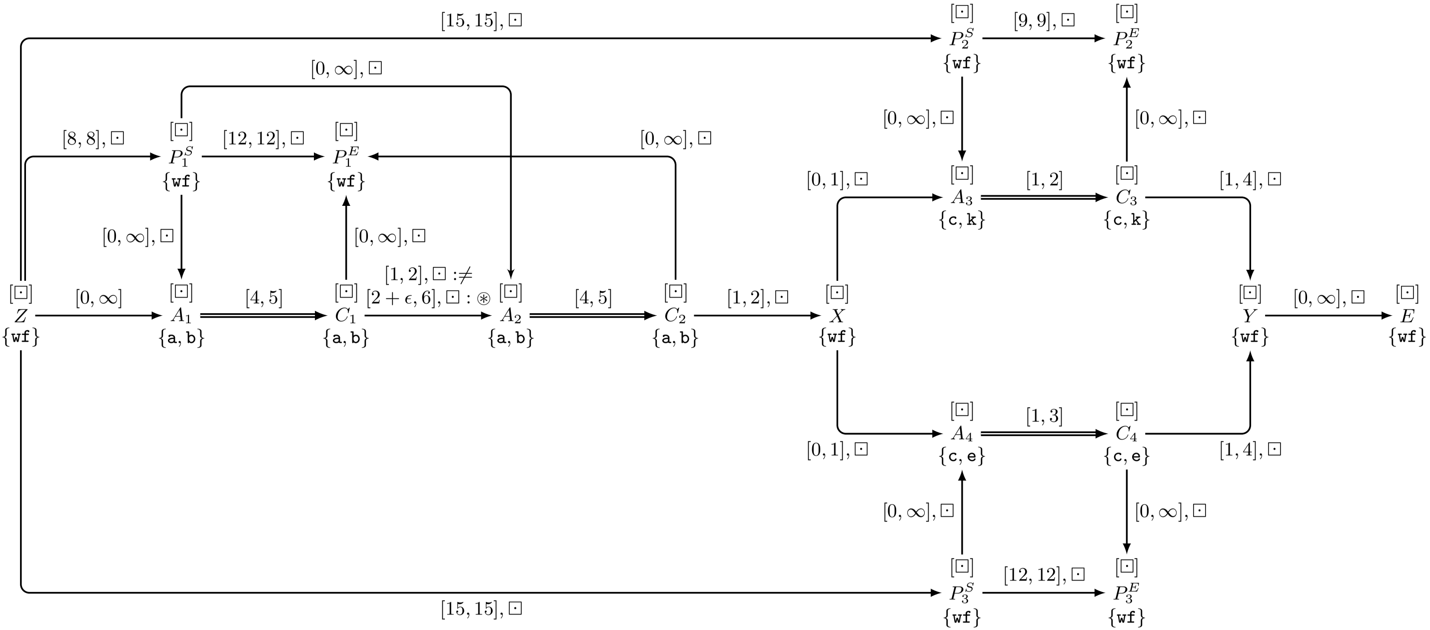


Fig. 9.15: ACTN equivalent to the access-controlled workflow depicted in Figure 9.12. Users a, b, c, e, k represent Alice, Bob, Charlie, Eve, and Kate, respectively. $\mathtt{SP}_1$ holds by default as ACTNs prevent the same user from carrying out more than one task simultaneously. I model $\mathtt{SP}_2$ with a pair of constraints $[1, 2], \boxdot :\neq$ and $[2 + \epsilon, 6], \boxdot : \circledast$ labeling $C_1 \to A_2$.

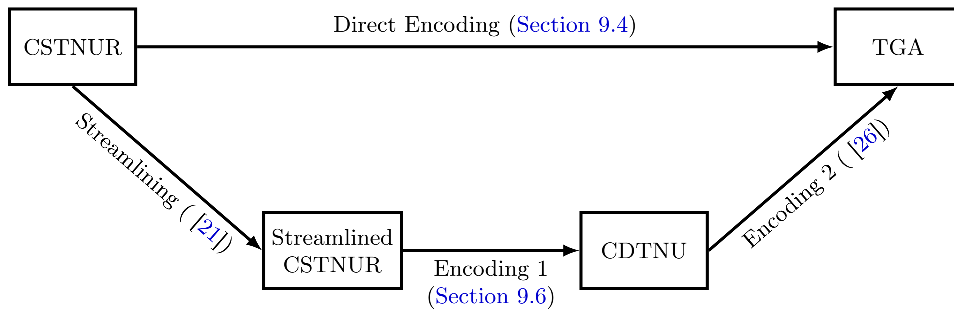


Fig. 9.16: Deciding dynamic controllability of CSTNURs via TGAs: direct encoding (path above), reduction to CDTNUs (path below).

can be encoded into a TGA in polynomial time. I used UPPAAL-TIGA for the concretization phase and also discussed further software optimizations provided by the tool. I showed how to translate the temporal constraints of a time window of TRBAC into an equivalent STN to be connected to the temporal network describing a workflow. I provided a connection mapping to connect such an STN to both ACTNs and CSTNURs.

Like ACTNs, CSTNURs differ from classic temporal networks discussed in Part I because those formalisms do not employ resources. CSTNURs also differ from CNCUs in Part I because CNCUs do not employ temporal constraints. CSTNURs do not deal with the uncertain availability of resources, therefore they cannot do much for workflow resiliency. CSTNURs differ from ACTNs for the type of constraints they specify. CSTNURs associate temporal expressions to resources associated to time points and employ RRCs to operate on these temporal expressions. RRCs might not be fired depending on the execution order of the time points. ACTNs employ static constraints that must always be satisfied. CSTNURs may also not block resources when they are executing contingent links. Instead, ACTNs just block them. CSTNURs differ from [41] as CSTNURs are based on CSTNUs and not on STNUs and because [41] does not provide a dynamic controllability algorithm for the proposed extension of the network.

CSTNURs are not more expressive than CDTNUs. However, they provide a compact language to model temporal plans with resources and a direct encoding into TGAs which skips two intermediate steps before getting to the TGA (Figure 9.16).

10

Conclusions and Future Work

In this thesis I have addressed temporal and resource controllability of workflows under uncertainty first in isolation and then simultaneously.

In Part I, I dealt with temporal (consistency and) controllability by addressing temporal plans subject to both temporal and conditional uncertainty. I provided the following contributions.

- I defined simple temporal networks with decisions (STNDs) and provided two hybrid consistency checking algorithms (HSCC) for seeking a single or all consistent scenarios. These novel algorithms rely on a SAT solver and well-known shortest paths algorithms for directed weighted graphs, thus they are sound and complete by default. `STND-HSCC1` tests STN-projections for negative cycles by iterating on *complete* models returned by the SAT solver, whereas `STND-HSCC2` exploits partial truth value assignments to hunt down negative cycles in STN-projections as early as possible. If the projected STN is inconsistent, I add a clause to the SAT solver to exclude the relevant part of that scenario, else I let the solver go. The more inconsistent STN-projections an STND admits, the better `STND-HSCC2` performs. I provided KAPPA, a tool for STNDs. I also discussed how to generate random temporal networks and carried out an experimental evaluation. In general `STND-HSCC2` beats `STND-HSCC1`. I proved that STNDs and DTPs are equivalent.
- I defined *conditional simple temporal networks with uncertainty and decisions* (CSTNUDs) as a unified formalism. CSTNUDs implicitly embed all minor temporal network formalisms based on STNs (see Figure 5.4 for a hierarchy of simple temporal networks). I modeled the DC-checking of a CSTNUD as a two-player game where `Player1` models the controller and `Player2` models the environment and gave the execution semantics in move-based strategies. At any time t, `Player1` and `Player2` make their moves in their turns $T_1(t)$ and $T_2(t)$. `Player1` always plays first. Each player can make many moves in its turn at the same time instant provided that these moves respect an order. A player waits in a turn when it does not make any move. Both the players can wait at the same time. I provided an encoding from CSTNUDs into timed game automata (TGAs) as an optimized extension of that given for CSTNUs in [25,26,28] and discussed the correctness and complexity of such an encoding which results in a sound and complete approach for the dynamic controllability

checking. I provided Esse, a tool for CSTNUDs. If a CSTNUD is DC, Esse saves to file a memoryless execution strategy to later carry out an arbitrary number of execution simulations. I also discussed how to generate random temporal networks and carried out two experimental evaluations. The first, comparing with the initial one done in [126] discovering that Esse performs better that the first prototype. The second, with respect to CSTNUDs in which all components but one were fixed, where the free component ranged from a minimum to a maximum value.

- I showed how temporal networks can be employed for the modeling, validation and execution of temporal workflows under conditional and temporal uncertainty. I provided a high level language to specify TWFs under temporal and conditional uncertainty, defined weak, strong and dynamic controllability of such workflows and provided an encoding from TWFs (expressed in a fragment of BPMN) into CSTNUDs. It is pretty clear that depending on the modeled workflow, different kinds of temporal networks may arise. For example, if I model a process not having any conditions or decisions but specifying uncontrollable task durations, an STNU is enough (in that case validation and execution are in PTIME). If I model a process not having any decisions or uncontrollable durations but employing uncontrollable conditions a CSTN is enough. In that case I can exploit algorithms targeted for the specific kind of network. However, using algorithms for higher classes of temporal networks (e.g., those for CSTNs applied to STNs) is not optimal but not wrong either (provided these algorithms are sound and complete). I modeled, validated and executed a case study for a goods delivery process in which customers must receive the goods they ordered within one day or after one but within three days.

In Part II, I dealt with resource controllability by addressing plans subject to conditional uncertainty and subject to the uncertain availability of resources. I provided the following contributions.

- I defined *constraint networks under conditional uncertainty* (CNCUs) to address a kind of CSP under conditional uncertainty. CNCUs implicitly embed classic CNs (if $\mathcal{OV} = \emptyset$ and $\prec = \emptyset$). I defined weak, strong and dynamic controllability of a CNCU and provided algorithms to check each type of controllability. Currently, I only deal with CNCUs that are controllable with respect to a total ordering for the variables. I discussed the correctness and complexity of the algorithms I proposed and I provided Zeta, a tool for CNCUs that acts as a solver for the three kinds of controllability as well as an execution simulator. I provided an experimental evaluation against a set of benchmarks of CNCUs and I also discussed an algorithm to generate random CNCUs. Strong controllability is the easiest type of controllability to check, followed by weak and dynamic, which is currently the hardest one. CNCUs not admitting any total ordering on the variables are uncontrollable for all three kinds of controllability. Dynamic controllability is a matter of order as the same CNCU could be controllable or uncontrollable depending on which total order of the variables is chosen. Strong and dynamic controllability provide usable strategies for executing workflows under conditional uncertainty.

- I showed how CNCUs can be employed for the modeling, validation and execution of access controlled workflows under conditional uncertainty. I provided a high level language to specify ACWFs under conditional uncertainty, defined weak, strong and dynamic controllability and provided an encoding from ACWFs (expressed in a restriction of BPMN) into CNCUs. It is pretty clear that depending on the modeled workflow, different kinds of constraint network may arise. For example, if I model a process not having any condition, a CN (plus a partial order) arises. In that case, classic algorithms for CNs can be used. However, using WC-CHECKING, SC-CHECKING and DC-CHECKING could be not optimal but not wrong either. I modeled, validated and executed a case study for a loan origination process.
- I addressed the uncertain availability of resources by addressing workflow resiliency. I started from the definitions of the corresponding games provided by Wang and Li in [122, 123] for static, decremental and dynamic workflow resiliency. I provided three encodings into extended timed game automata to model these games, and I proved that my encodings are correct and run in polynomial time. ACWFs that are resilient are (dynamically) satisfiable, the vice versa does not hold in general. I employed UPPAAL-TIGA as an off the shelf model checker for TGA reachability properties. With this approach, I only need to query the TGA (which the ACWF has been encoded into) by asking for a control strategy for `Player1` allowing him to win the game if and only if he *always eventually* enters a location of interest. If the ACWF is resilient, i.e., if a winning strategy for `Player1` exists, UPPAAL-TIGA returns in output such a strategy. If the ACWF is breakable, UPPAAL-TIGA returns a *counter-strategy* for `Player2` allowing him to always *break* the execution (i.e., prevent `Player1` from entering `Win`). I developed ERRE, a tool for workflow resiliency. ERRE allows for an automated model generation by encoding a specification of an ACWF taken as input into a TGA. ERRE internally relies on UPPAAL-TIGA to prove that the workflow is either resilient or breakable. If the ACWF is resilient, ERRE compresses (online) the strategy returned in output by UPPAAL-TIGA and saves it to file. ERRE also allows one to carry out random execution simulations. I carried out an experimental evaluation against a set of benchmarks and also discussed an algorithm to generate random ACWFs. On the one hand, UPPAAL-TIGA guarantees that the algorithms employed to answer the *decision* problem of workflow resiliency are sound and complete, on the other hand ERRE guarantees that the approach is fully automated from analysis to simulation. In this way, I provided a *usable* approach even for designers and/or security officers with little or no knowledge on TGAs. As I expected, the experimental evaluation confirmed that checking static resiliency is easier than checking decremental resiliency which, in turn, is easier than checking dynamic resiliency (when the ACWF is resilient for all the three kinds of resiliency).

In Part III, I dealt with temporal and resource controllability simultaneously by addressing plans subject to temporal and conditional uncertainty. I provided the following contributions.

- I defined *access controlled temporal networks (ACTNs)* as an extension of CSTNUs in order to take into consideration users and (temporal conditional) authorization constraints simultaneously. Users are in charge of executing contingent links, whereas requirement links express (temporal) authorization constraints. In general, a user u_Y can execute a time point Y if for each requirement link $X \to Y$ labeled by $[x, y], \ell : \alpha$ such that $x \geq 0$, ℓ is entailed by the current partial scenario, and the pair (u_X, u_Y) satisfies α, where u_X is the user who executed X. I gave the execution semantics for dynamic controllability in real time execution decisions (RTEDs) by extending that for CSTNUs to take into account users and authorization constraints. The main difference is in $\Delta_{\texttt{ctrl}}$, which, in addition to waiting, can now commit users to execute time points at certain time instants. I provided an encoding from ACTNs into TGAs as an extension of that given for CSTNUs to accommodate users and authorization constraints and I also discussed a few optimizations to speed up the model-checking phase. I discussed the correctness of the encoding and proved that the encoding runs in polynomial time. As a result, since I use the reachability algorithms of TGAs (TCTL model checking), I provided a *sound* and *complete* approach for checking the dynamic controllability of an ACTN. I used ACTNs to analyze the official STEMI guidelines as a concrete example.
- I defined *conditional simple temporal networks with uncertainty and resources (CSTNURs)* by extending CSTNUs with resources, temporal expressions and runtime resource constraints (RRCs). RRCs are a new class of constraints able to refine in real time the TEs associated to the resources depending on the specific execution. Resources are committed for time point executions, whereas RRCs enforce (temporal) security policies such as temporal separation and binding of duties. RRCs are fired whenever their precondition, a time point called firing time point, is executed. The firing of an RRC results in appending the TE it contains to all resources associated to the target time point, provided that these resources along with that who executed the firing point satisfy the specified relation. An RRC is not fired if the target time point has already been executed when the firing time point is executed. A contingent time point cannot appear as a target in an RRC, otherwise it could interfere with the uncontrollable actions of the environment (i.e., it could, for example, modify its uncontrollable duration). Furthermore, for each contingent time point, I assumed that the resource committed for the activation time point, is committed for the contingent too. I gave the execution semantics in real time execution decisions (RTEDs). I extended the encoding proposed in [26] (with the optimizations in Section 5.3) from CSTNUs into TGAs in order to do the DC-checking. I translated TEs into equivalent clock constraints in order to employ them in some specific TGA guards, I encoded the *RA* relation into dedicated clocks, I encoded resource commitments for non-contingent time points in circular paths validating all RRCs targeting those time points and I blocked resources (if not available because of some RRC) by resetting their associated clocks. All clocks surviving this validation phase say which resources are committable for the time point. Likewise, I encoded the execution of a contingent time point into a contingent circular path, which commits for C the same resource that was committed for A. I proved that encoding any CSTNUR into

a TGA runs in polynomial time. I used UPPAAL-TIGA for the concretization phase and also discussed further software optimizations provided by the tool.
- I showed how to translate the temporal constraints of a time window of TRBAC into an equivalent STN to be connected to the temporal network describing a workflow. I refined the concept of periodic set of integers in periodic real intervals by considering the starting and ending instants of a granule. I translated periodic expressions into real time intervals and generated a corresponding consistent STN. I provided a connection mapping to connect such an STN to both ACTNs and CSTNURs. I modeled an example for a train round trip with both ACTNs and CSTNURs augmented with a TRBAC part. This approach allows a designer to understand if the temporal constraints of an access control model and the temporal constraints of a workflow are coherent. That is, if an access controlled workflow can be executed with respect to a given instance of a TRBAC.

As future work, this thesis has paved the way for plenty of directions worth following. Here are a few of them.

Temporal controllability

- A metric suggesting when it is better to use `STND-HSCC1` or `STND-HSCC2` depending on the form of the STND in input is currently missing. Recall that for STNDs that have all consistent scenarios `STND-HSCC1` must perform better than `STND-HSCC2`. What about those with a fairly high number of consistent scenarios? What is this limit? Also, since STNDs are equivalent to DTPs a comparison with DTP solvers and SMT solvers is missing too.
- Weak and strong controllability of CSTNUDs (along with their related complexities) remain unexplored. A possible (albeit not optimal) approach is to extend the methods in [29, 30] to exploit SMT solvers and work with quantifiers.
- Constraint-propagation algorithms must perform better than TGAs as they mainly do not suffer from ordering problems during the checking phase (recall that the controller synthesis for TGAs encoding temporal networks sped up as soon as I restricted the partial order between the transitions modeling the execution of time points).
- HSCC algorithms can be extended to hybrid SAT-based *controllability* checking algorithms to deal with STNUDs, CSTNDs and CSTNUDs whenever decisions must be made offline. In these cases instead of using Bellman-Ford on the arising projections we should use the dedicated algorithm for the specific class of networks.

Resource controllability

- Working on the *all topological sort* phase of DC-CHECKING for CNCUs in order to contain the explosion of this step and also investigating if CNCUs classified as non-DC with respect to all possible total orderings might turn DC for some ordering that refines dynamically during execution.

- The complexity of the decision problem of weak and dynamic controllability for CNCUs remains unexplored. Strong controllability is no different from classic consistency for CNs.
- The encodings for workflow resiliency may be adapted to support Type 2-3 entailment constraints as well as counting constraints. Also, it remains to understand if decremental resiliency is a matter of order.
- Addressing multiple workflows (or many instances of the same workflow) that run in parallel to deal with the sharing of resources, the specification of constraints between different workflows and, given a set of workflows, the computation of the maximum number of workflows or (instances of the same) that can run in parallel without running out of resources.

Temporal and resource controllability together

- Weak and strong controllability for both ACTNs and CSTNURs are yet to be addressed. Once again this part could be investigated via SMT. The complexity of dynamic controllability of ACTNs and CSTNURs is currently unknown and software tools (like Esse, Erre, Kappa or Zeta) are currently missing for both.
- A language to encode (temporal) ACWFs into ACTNs or CSTNURs is missing.

References

1. James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, May 1993.
3. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, London, UK, UK, 1995. Springer-Verlag.
5. Egon Balas. Disjunctive programming. In P.L. Hammer, E.L. Johnson, and B.H. Korte, editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 3 – 51. Elsevier, 1979.
6. Tomas Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.
7. Michele Barletta, Silvio Ranise, and Luca Viganò. A declarative two-level framework to specify and verify workflow and authorization policies in service-oriented architectures. *SOCA*, 5(2):105–137, 2011.
8. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
9. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *CAV 2007*. Springer Berlin Heidelberg, 2007.
10. Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
11. Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.
12. Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3):231–285, 1998.
13. Elisa Bertino, Piero A. Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.
14. Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, February 1999.

15. Elisa Bertino, Elena Ferrari, and Vijayalakshmi Atluri. A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, RBAC '97, pages 1–12, New York, NY, USA, 1997. ACM.
16. Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Distributed and Parallel Databases*, 11(3):269–306, May 2002.
17. Business process modeling notation 2.0. http://www.omg.org/spec/BPMN/2.0/.
18. Cristina Cabanillas, Manuel Resinas, Adela del-Río-Ortega, and Antonio Ruiz Cortés. Specification and automated design-time analysis of the business process human resource perspective. *Inf. Syst.*, 52:55–82, 2015.
19. Massimo Cairo, Carlo Combi, Carlo Comin, Luke Hunsberger, Roberto Posenato, Romeo Rizzi, and Matteo Zavatteri. Incorporating decision nodes into conditional simple temporal networks. In S. Schewe, T. Schneider, and J. Wijsen, editors, *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, LIPIcs, pages 9:1–9:17, 2017.
20. Massimo Cairo, Carlo Comin, and Romeo Rizzi. Instantaneous reaction-time in dynamic-consistency checking of conditional simple temporal networks. In *23rd International Symposium on Temporal Representation and Reasoning (TIME 2016)*, pages 80–89, 2016.
21. Massimo Cairo, Luke Hunsberger, Roberto Posenato, and Romeo Rizzi. A streamlined model of conditional simple temporal networks - semantics and equivalence results. In *24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16-18, 2017, Mons, Belgium*, pages 10:1–10:19, 2017.
22. Massimo Cairo and Romeo Rizzi. Dynamic controllability of conditional simple temporal networks is pspace-complete. In *23rd International Symposium on Temporal Representation and Reasoning, TIME 2016, Kongens Lyngby, Denmark, October 17-19, 2016*, pages 90–99, 2016.
23. Massimo Cairo and Romeo Rizzi. Dynamic controllability made simple. In *24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16-18, 2017, Mons, Belgium*, pages 8:1–8:16, 2017.
24. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 93–107, 2013.
25. Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, Roberto Posenato, and Marco Roveri. Sound and complete algorithms for checking the dynamic controllability of temporal networks with uncertainty, disjunction and observation. In *21st International Symposium on Temporal Representation and Reasoning (TIME 2014)*, pages 27–36, 2014.
26. Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, Roberto Posenato, and Marco Roveri. Dynamic controllability via timed game automata. *Acta Informatica*, 53(6-8):681–722, 2016.
27. Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2242–2249, 2014.
28. Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2242–2249, 2014.

29. Alessandro Cimatti, Andrea Micheli, and Marco Roveri. An SMT-based approach to weak controllability for disjunctive temporal problems with uncertainty. *Artif. Intell.*, 224, 2015.
30. Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Solving strong controllability of temporal problems with uncertainty using SMT. *Constraints*, 20(1), Jan 2015.
31. David D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pages 184–195, 1987.
32. Workflow Management Coalition. Terminology & glossary. http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf, 1996.
33. Carlo Combi, Mauro Gambini, and Sara Migliorini. The NestFlow Interpretation of Workflow Control-Flow Patterns. In *ADBIS 2011*, pages 316–332, 2011.
34. Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. Representing business processes through a temporal data-centric workflow modeling language: An application to the management of clinical pathways. *IEEE T. Systems, Man, and Cybernetics: Systems*, 44(9):1182–1203, 2014.
35. Carlo Combi, Matteo Gozzi, José M. Juárez, Barbara Oliboni, and Giuseppe Pozzi. Conceptual modeling of temporal clinical workflows. In *14th International Symposium on Temporal Representation and Reasoning (TIME 2007), 28-30 June 2007, Alicante, Spain*, pages 70–81, 2007.
36. Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,*, pages 144–156. INSTICC, ScitePress, 2013.
37. Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty - revisited. In *Agents and Artificial Intelligence*, volume 449 of *CCIS*, pages 314–331. 2014.
38. Carlo Combi and Roberto Posenato. Controllability in temporal conceptual workflow schemata. In *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, pages 64–79, 2009.
39. Carlo Combi and Roberto Posenato. Towards temporal controllabilities for workflow schemata. In *TIME 2010 - 17th International Symposium on Temporal Representation and Reasoning, Paris, France, 6-8 September 2010*, pages 129–136, 2010.
40. Carlo Combi, Roberto Posenato, Luca Viganò, and Matteo Zavatteri. Access controlled temporal networks. In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART 2017)*, pages 118–131. INSTICC, ScitePress, 2017.
41. Carlo Combi, Luca Viganò, and Matteo Zavatteri. Security constraints in temporal role-based access-controlled workflows. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY 2016. ACM, 2016.
42. Carlo Comin, Roberto Posenato, and Romeo Rizzi. Hyper temporal networks - A tractable generalization of simple temporal networks and its relation to mean payoff games. *Constraints*, 22(2):152–190, 2017.
43. Carlo Comin and Romeo Rizzi. Dynamic consistency of conditional simple temporal networks via mean payoff games: A singly-exponential time dc-checking. In *22nd International Symposium on Temporal Representation and Reasoning (TIME 2015)*, pages 19–28, 2015.
44. Jean-François Condotta, Souhila Kaci, and Yakoub Salhi. Optimization in temporal qualitative constraint networks. *Acta Inf.*, 53(2):149–170, 2016.

45. Patrick R. Conrad and Brian C. Williams. Drake: An efficient executive for temporal plans with choice. *J. Artif. Int. Res.*, 42(1):607–659, September 2011.
46. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.
47. Jason Crampton, Gregory Gutin, and Daniel Karapetyan. Valued workflow satisfiability problem. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 3–13, New York, NY, USA, 2015. ACM.
48. Jason Crampton, Gregory Gutin, Daniel Karapetyan, and Rémi Watrigant. The bi-objective workflow satisfiability problem and workflow resiliency. *Journal of Computer Security*, 25(1):83–115, 2017.
49. Jason Crampton, Gregory Gutin, and Anders Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur.*, 16(1):4:1–4:31, June 2013.
50. Jing Cui and Patrik Haslum. Dynamic controllability of controllable conditional temporal problems with uncertainty. In *27th International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 2017.
51. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.
52. Rina Dechter. *Constraint processing.* Elsevier, 2003.
53. Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.
54. Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Int.*, 34(1), 1987.
55. Daniel Ricardo dos Santos and Silvio Ranise. A survey on workflow satisfiability, resiliency, and related problems. *CoRR*, abs/1706.07205, 2017.
56. Daniel Ricardo dos Santos, Silvio Ranise, Luca Compagna, and Serena Elisa Ponta. Assisting the deployment of security-sensitive workflows by finding execution scenarios. In *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, pages 85–100, 2015.
57. Daniel Ricardo dos Santos, Silvio Ranise, Luca Compagna, and Serena Elisa Ponta. Automatically finding execution scenarios to deploy security-sensitive workflows. *Journal of Computer Security*, 25(3):255–282, 2017.
58. Johann Eder, Wolfgang Gruber, and Euthimios Panagos. Temporal modeling of workflows with conditional execution paths. In Mohamed Ibrahim, Josef Küng, and Norman Revell, editors, *Database and Expert Systems Applications*, pages 243–253, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
59. Johann Eder, Euthimios Panagos, Heinz Pozewaunig, and Michael Rabinovich. *Time Management in Workflow Systems*, pages 265–280. Springer London, 1999.
60. Johann Eder, Euthimios Panagos, and Michael Rabinovich. Time constraints in workflow systems. In Matthias Jarke and Andreas Oberweis, editors, *Advanced Information Systems Engineering*, pages 286–300, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
61. Johann Eder, Euthimios Panagos, and Michael Rabinovich. *Workflow Time Management Revisited*, pages 207–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
62. Hélène Fargier and Jérôme Lang. Uncertainty in constraint satisfaction problems: A probabilistic approach. In *ECSQARU '93.* Springer, 1993.

63. Hélène Fargier, Jérôme Lang, and Thomas Schiex. Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In *IAAI 96*, 1996.
64. Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003.
65. Eugene C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29, 1982.
66. Georg Gottlob. On minimal constraint networks. *Artif. Intell.*, 191-192, 2012.
67. Julius Holderer, Rafael Accorsi, and Günter Müller. When four-eyes become too much: a survey on the interplay of authorization constraints and workflow resilience. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1245–1248, 2015.
68. David Hollingsworth. The workflow reference model. http://www.wfmc.org/standards/model.htm, 1995.
69. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
70. Jinbo Huang, Jason Jingshi Li, and Jochen Renz. Decomposition and tractability in qualitative spatial and temporal reasoning. *Artificial Intelligence*, 195:140 – 164, 2013.
71. Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *TIME 2009, 16th International Symposium on Temporal Representation and Reasoning, Bressanone-Brixen, Italy, 23-25 July 2009, Proceedings*, pages 155–162, 2009.
72. Luke Hunsberger and Roberto Posenato. Checking the dynamic consistency of conditional simple temporal networks with bounded reaction times. In *26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, pages 175–183, 2016.
73. Luke Hunsberger and Roberto Posenato. A new approach to checking the dynamic consistency of conditional simple temporal networks. In *Principles and Practice of Constraint Programming (CP 2016)*, volume 9892 of *LNCS*, pages 268–286, 2016.
74. Luke Hunsberger, Roberto Posenato, and Carlo Combi. The Dynamic Controllability of Conditional STNs with Uncertainty. In *Workshop on Planning and Plan Execution for Real-World Systems (PlanEx) at ICAPS-2012*, pages 1–8, Atibaia, June 2012.
75. Luke Hunsberger, Roberto Posenato, and Carlo Combi. A sound-and-complete propagation-based algorithm for checking the dynamic consistency of conditional simple temporal networks. In *22st International Symposium on Temporal Representation and Reasoning (TIME 2015)*, pages 4–18, 2015.
76. Erez Karpas, Steven James Levine, Peng Yu, and Brian Charles Williams. Robust execution of plans for human-robot teams. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pages 342–346, 2015.
77. Arif Akram Khan and Philip W. L. Fong. *Satisfiability and Feasibility in a Relationship-Based Workflow Authorization Model*, pages 109–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
78. Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 431–445, London, UK, UK, 2000. Springer-Verlag.
79. Phil Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 487–493, 2001.

80. Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.
81. Andreas Lanz, Roberto Posenato, Carlo Combi, and Manfred Reichert. Controllability of time-aware processes at run time. In Robert Meersman, Hervé Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Deijing Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, pages 39–56, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
82. Andreas Lanz, Roberto Posenato, Carlo Combi, and Manfred Reichert. Simple temporal networks with partially shrinkable uncertainty. In *ICAART 2015 - Proceedings of the International Conference on Agents and Artificial Intelligence, Volume 2, Lisbon, Portugal, 10-12 January, 2015.*, pages 370–381, 2015.
83. Andreas Lanz, Manfred Reichert, and Barbara Weber. A formal semantics of time patterns for process-aware information systems. Technical Report UIB-2013-02, University of Ulm, January 2013.
84. Andreas Lanz, Barbara Weber, and Manfred Reichert. Workflow time patterns for process-aware information systems. In Ilia Bider, Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 94–107, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
85. Andreas Lanz, Barbara Weber, and Manfred Reichert. Time patterns for process-aware information systems. *Requirements Engineering*, 19(2):113–141, Jun 2014.
86. Ralf Laue and Jan Mendling. The impact of structuredness on error probability of process models. In *Proc. of 2nd UNISCON*, pages 585–590, 2008.
87. Thomas Léauté and Brian C. Williams. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 114–120, 2005.
88. Maria Leitner and Stefanie Rinderle-Ma. A systematic review on security in process-aware information systems - constitution, challenges, and future directions. *Information & Software Technology*, 56(3):273–293, 2014.
89. Richard Lenz and Manfred Reichert. It support for healthcare processes - premises, challenges, perspectives. *Data Knowl. Eng.*, 61(1):39–58, 2007.
90. Steven James Levine and Brian Charles Williams. Concurrent plan recognition and execution for human-robot teams. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014.
91. Ninghui Li, Qihua Wang, and Mahesh Tripunitara. Resiliency policies in access control. *ACM Trans. Inf. Syst. Secur.*, 12(4):20:1–20:34, April 2009.
92. Meghna Lowalekar, Ritesh Kumar Tiwari, and Kamalakar Karlapalem. *Security Policy Satisfiability and Failure Resilience in Workflows*, pages 197–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
93. Haibing Lu, Yuan Hong, Yanjiang Yang, Yi Fang, and Lian Duan. *Dynamic Workflow Adjustment with Security Constraints*, pages 211–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
94. Xudong Luo, Jimmy Ho-man Lee, Ho-fung Leung, and Nicholas R. Jennings. Prioritised fuzzy constraint satisfaction problems: Axioms, instantiation and validation. *Fuzzy Sets Syst.*, 136(2), 2003.
95. John C. Mace, Charles Morisset, and Aad van Moorsel. *Quantitative Workflow Resiliency*, pages 344–361. Springer International Publishing, Cham, 2014.
96. John C. Mace, Charles Morisset, and Aad van Moorsel. *WRAD: Tool Support for Workflow Resiliency Analysis and Design*, pages 79–87. Springer International Publishing, Cham, 2016.

97. Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1), 1977.
98. Oded Maler, Amir Pnueli, and Joseph Sifakis. *On the synthesis of discrete controllers for timed systems*, pages 229–242. Springer, Berlin, Heidelberg, 1995.
99. Olivera Marjanovic and Maria E. Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge and Information Systems*, 1(2):157–192, May 1999.
100. Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI 90*, 1990.
101. Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7, 1974.
102. Paul Morris. The mathematics of dispatchability revisited. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'16, pages 244–252. AAAI Press, 2016.
103. Paul Morris and Nicola Muscettola. Temporal Dynamic Controllability Revisited. In *AAAI*, pages 1193–1198. AAAI Pr., 2005.
104. Paul H. Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *IJCAI 2001*, pages 494–502, 2001.
105. M. Niezette and J. Stevenne. An Efficient Symbolic Representation of Periodic Time. In *Proceedings of CIKM'92*, pages 161–168. ISMM, 1992.
106. Andrea Orlandini, Alberto Finzi, Amedeo Cesta, and Simone Fratini. TGA-Based Controllers for Flexible Plan Execution. In Joscha Bach and Stefan Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence: 34th Annual German Conference on AI, Berlin, Germany, October 4-7,2011. Proceedings*, pages 233–245. Springer Berlin Heidelberg, 2011.
107. Federica Paci, Rodolfo Ferrini, Yuqing Sun, and Elisa Bertino. *Authorization and User Failure Resiliency for WS-BPEL Business Processes*, pages 116–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
108. Horst Pichler, Johann Eder, and Margareta Ciglic. Modelling processes with time-dependent control structures. In Heinrich C. Mayr, Giancarlo Guizzardi, Hui Ma, and Oscar Pastor, editors, *Conceptual Modeling*, pages 50–58, Cham, 2017. Springer International Publishing.
109. Hajo A. Reijers and Jan Mendling. Modularity in process models: Review and effects. In *BPM 2008*, pages 20–35, 2008.
110. Peter Z. Revesz. Safe stratified datalog with integer order programs. In *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, pages 154–169, 1995.
111. Ravi S. Sandhu. Separation of duties in computerized information systems. In *Database Security, IV: Status and Prospects. Results of the IFIP WG 11.3 Workshop on Database Security, Halifax, U.K., September 18-21, 1990*, pages 179–190, 1990.
112. Ravi S. Sandhu. Roles versus groups. In *ACM Workshop on Role-Based Access Control*, 1995.
113. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.
114. Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 248–253, 1998.
115. Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artif. Intell.*, 120(1):81–117, 2000.
116. David Toman and Jan Chomicki. Datalog with integer periodicity constraints. *J. Log. Program.*, 35(3):263–290, 1998.

117. Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artif. Intell.*, 151(1-2):43–89, 2003.
118. Ioannis Tsamardinos, Thierry Vidal, and Martha E. Pollack. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4):365–388, 2003.
119. Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
120. Panagiotis Vasilikos, Flemming Nielson, and Hanne Riis Nielson. Time Dependent Policy-Based Access Control. In Sven Schewe, Thomas Schneider, and Jef Wijsen, editors, *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, volume 90 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
121. Marc B. Vilain and Henry A. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science.*, pages 377–382, 1986.
122. Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow systems. In *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, pages 90–105, 2007.
123. Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4), 2010.
124. Peng Yu, Cheng Fang, and Brian Charles Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014.
125. Peng Yu and Brian Charles Williams. Continuously relaxing over-constrained conditional temporal problems through generalized conflict learning and resolution. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 2429–2436, 2013.
126. Matteo Zavatteri. Conditional simple temporal networks with uncertainty and decisions. In *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, volume 90 of *LIPIcs*, 2017.
127. Matteo Zavatteri, Carlo Combi, Roberto Posenato, and Luca Viganò. Weak, strong and dynamic controllability of access-controlled workflows under conditional uncertainty. In *Business Process Management (BPM 2017)*, 2017.
128. Matteo Zavatteri and Luca Viganò. Constraint networks under conditional uncertainty. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART 2018)*, pages 41–52. INSTICC, SciTePress, 2018.