Research Article

Formal Verification of a Change Control Process in Project Management

Ramon Brena[1], Alejandro Vazquez-Nava[1], Juan A. Nolazco[1], Jose I. Icaza[1], James A. Fangmeyer[1]

1. Tec de Monterrey, Monterrey, Mexico

Compliance of processes in enterprises to internal policies and external regulations could be critical because failing to follow them could result in great losses, but manual compliance auditing is difficult and prone to errors and oversight. In this paper, we present a method for formally verifying the properties of Integrated Change Control processes using temporal logic. We express the process in terms of states, and then we formulate some of its key properties, such as prerequisites, reachability, definiteness, and cycles, using a temporal logic called Computation Tree Logic. The properties to verify in the case study we present are taken from actual change control process auditing practice in a large business in the food industry. We formally verify those properties using a model-checking tool. We end up with a formally verified Integrated Change Control process and more robust assurance of its correctness than can be reached for its informal counterpart. To the best of our knowledge, this has not been done before.

Corresponding authors: Ramon Brena, ramon.brena@tec.mx; Alejandro Vazquez-Nava, a00767414@itesm.mx; Juan A. Nolazco, jnolazco@tec.mx; Jose I. Icaza, jicaza@tec.mx; James A. Fangmeyer, james.fangmeyer@itesm.mx

1. Introduction

Project management (PM) is a widely studied area (Project Management Institute, 2017), mostly from the point of view of methodological guidelines. In particular, in this paper we focus on the "*Integrated Change Control*" process (ICC) described in the Project Management Institute (PMI) Project Management Body of Knowledge (PMBOK). ICC is key for "controlling changes and recommending corrective or preventive action in anticipation of possible problems" [PMI, 17] during the progress of a project. Depending on the project area, the change control process should be tailored for complexity, contract requirements, and the context in which the project is controlled [PMI, 17]. Effective processes are supposed to have some desirable properties, such as reachability, liveness, compliance with prerequisites, and so on; but whether these properties are ensured by the workflow is something that can be extremely difficult to prove for large projects typical of big companies.

Methodology guides such as the PMBOK [PMI, 17] are generally composed of criteria and processes explained in plain English. While this is perfectly good for most design purposes, there are some scenarios in which we need a more precise –even mathematical– expression of the process such that we can verify if its key properties are guaranteed. The costlier are errors and mistakes, the more interest there is in achieving a high level of correctness assurance: "[erroneously] designed workflow models can result in failed workflow processes, execution errors, and disgruntled customers and employees" [Bi, 04].

According to Awad [Awad, 08], compliance rules have different origins, and change through time, and they include: a) Business processes, like the order of execution of activities, or the inclusion of certain control or supervision activities; b) Policies that produce either a competitive advantage or to protect the business from failures; c) Quality standards like ISO 9000; d) External mandatory regulations and laws, like the 2002 Sarbanes-Oxley Act. As Awad states, the consequences of failing to follow the policies could not only decrease quality or competitiveness, but also to lead to penalties and reputation loss.

To ensure the compliance to regulations mentioned above, experts perform manual audits of key business process models. This involves carrying out process and procedure walkthroughs, as well as the design of the corresponding check tests. This consumes considerable time and human effort, and even worse, is not guaranteed to detect every mistake or unwanted condition, because in practice it is very difficult to manually consider all the consequences of both the internal or external normativity and the procedures.

One of the authors of this paper oversees Information Technology (IT) Governance, Risk and Compliance in one of the largest alimentary industry in Mexico. He is responsible for supervising the technological IT platform, applications, IT processes and information systems, to ensure that the IT area supports in an effective way the business processes and corporate goals. His responsibilities about compliance with IT policies and regulations, as well as the design of adequate check and control mechanisms so that risks associated with IT processes are minimized, were one of the motivations to try a formal and automated way of proving properties, with methods like the ones we present in this papers.

The goal of this work was to develop a methodology that could leverage formal verification techniques to check the design of change processes, and to pinpoint errors when they exist, and to give a guarantee when there is none. This, of course, would raise the level of certainty that IT processes comply with regulations. The challenge for building such a methodology is that we must translate informal best practices and criteria, such as the rules a human expert is currently analyzing, into formal and even automated procedures and notation. Procedures must comply with certain properties such as: every change should have an authorization prior to its processing, duplicate changes should be avoided, identify changes that make the project inviable, changes should have successful tests and the acceptance of the involved parties, and so on. The detailed properties to be checked will be presented in sections 3 and 4.

In the context of ICC, a process designer or auditor should be interested in validating that the process under consideration has or does not have some key properties, and so it can be concluded confidently that the change control process is correctly designed. This was the motivation for using Formal Methods (FM) [Groefsema, 13] to reach mathematical-level property assurance. The use of formal methods in informal disciplines has the potential benefit of establishing with certainty that a process has (or not) key properties it is supposed to have, which translates in the correctness and integrity assurance. The use of FM for correctness assurance could be seen a form of “mathematical auditing” of the proposed ICC process. Obviously, a verified formal assurance would give an additional authority to a proposed ICC process. In the cases where the ICC process under study does not pass the mathematical validation, a “formal debugging” phase must ensue to correct the defects, and this debugging process is indeed one of the benefits of using formal methods for validation, as it allows a process designer to avoid the following risks associated with a poorly designed change control process: not tracking changes, developing changes (including unauthorized changes) impacting project plans (and project deliverables) with unforeseen effects, lack of priorities in change management, processing duplicate changes, developing changes that may turn the project unfeasible, etc.

We are interested in developing a mathematical formalization of both the ICC process and the properties it is supposed to hold in order to verify that the former actually exhibit the latter. We do so by using a temporal logic formalism, the “Computation Tree Logic” (CTL) [Bérard, 01] [Lamport, 83], and we run experiments using a model-based implementation of CTL named NuSMV [Cimatti, 00]. While other papers have previously used CTL and model checking for business process verification [Groefsema, 13], our proposal is specific for ICC processes.

In short, the contribution of this paper is a method for formally analyzing a change control process by applying temporal logic (in particular CTL) in a way that allows verification of the process’s key properties. We do not make contributions regarding the elements that ought to be part of a change control process.

The structure of the rest of the paper is as follows. In section 2 we present some background and related work. In section 3 we present the methodology for applying CTL to ICC. In section 4 our verification experiments are presented, and in section 5 we present the conclusions.

2. Background

In this section, we present previous work which gives the context and prerequisites for the application of our change process formal verification method.

2.1. PMI and Integrated Change Control

The ICC process consists of reviewing change requests, approving change requests, and updating the integrated plan for the project, the subsidiary plans, the product specifications, and the baselines of time, cost, and project quality [PMI, 17].

Every change request must be approved or rejected by some authority belonging to the project management team or an external organization [Kerzner, 05] [Stackpole, 09]. The approval of a change may require revision of activity sequences, cost estimates, scheduled dates, or resource requirements, as well as analysis of alternatives in response to the risks associated with the project. These changes can prompt adjustments to the entire project management plan [Phillips, 11] [Schwalbe, 09] [Lewis, 05] [Larson, 11]. We notice that the PMI PMBOK does not provide specific workflows for ICC, thus it is necessary to incorporate workflows from other sources, like ITIL [Rance, 11] in order to develop specific ICC workflows. Figure 1 shows the processes that have information relationships as inputs and outputs to PMI's ICC process.

2.2. Formal Methods and Specifications

System design (as in our ICC case) can generate or save big costs because errors propagate across stages in a system, and if they are detected in the last stages, error correction could imply reworking all the way from the error source. In software development this situation has been addressed by several studies [Anderson, 98] [Kelly, 95].

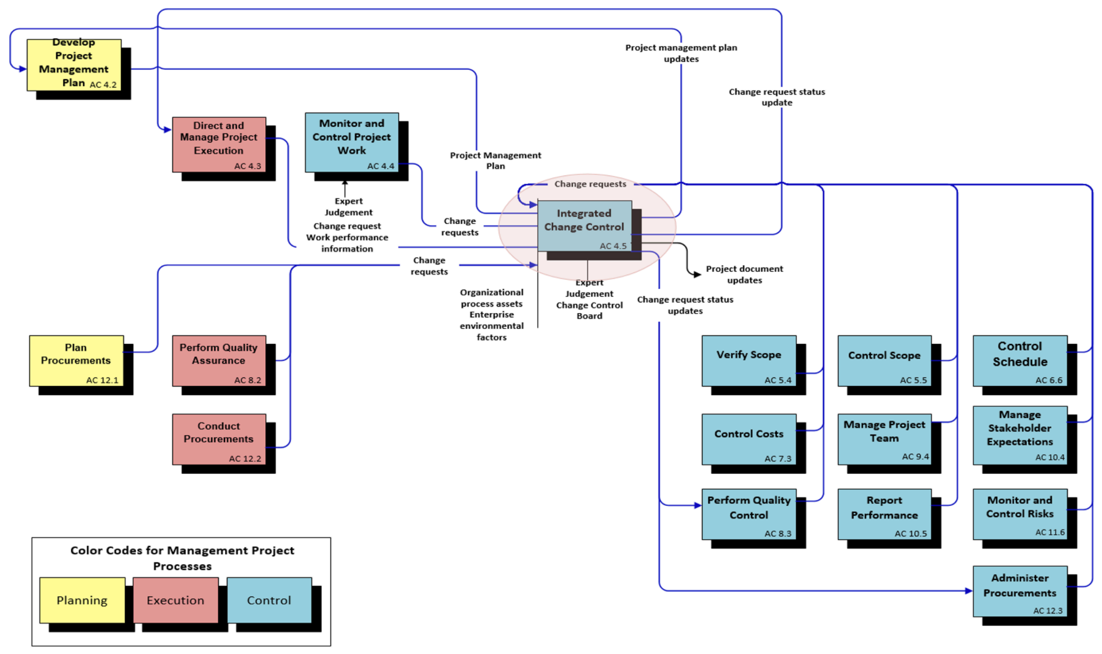

Develop Project Management Plan AC 4.2
Direct and Manage Project Execution AC 4.3
Monitor and Control Project Work AC 4.4
Project management plan updates
Change request status update
Project Management Plan
Change requests
Integrated Change Control AC 4.5
Expert Judgement
Change request
Work performance information
Change requests
Change requests
Project document updates
Organizational process assets
Enterprise environmental factors
Expert Judgement
Change Control Board
Change request status updates
Plan Procurements AC 12.1
Perform Quality Assurance AC 8.2
Conduct Procurements AC 12.2
Verify Scope AC 5.4
Control Scope AC 5.5
Control Schedule AC 6.6
Control Costs AC 7.3
Manage Project Team AC 9.4
Manage Stakeholder Expectations AC 10.4
Perform Quality Control AC 8.3
Report Performance AC 10.5
Monitor and Control Risks AC 11.6
Administer Procurements AC 12.3
Color Codes for Management Project Processes
Planning
Execution
Control

Figure 1. ICC process relationships with other processes of PM according to PMI

Indeed, we can learn valuable lessons on specification and modeling from the case of software. One approach for taming software errors is to rely on mathematical precision and tools, that is, the use of FM [Almeida, 11] [Bjørner, 14] [Burstall, 69] [Clarke, 96] [Monin, 03]. Formal methods consist of a set of techniques and tools based on mathematical modeling and formal logic used to specify and verify requirements and designs for software and computer systems. They can predict the logical properties of a system based on a mathematical model of the system using logic calculations, and can make it possible to find out whether a certain description of a system is internally consistent, to verify if certain properties are consequences of proposed requirements, and to check if the requirements have been interpreted correctly in the system design. In this work, we are primarily involved in the formal modeling of the Integrated Change Control process and the proof of this model's properties. This excludes many other applications of formal models, such as the refinement of specifications, the generation of actual code, the proof of correctness of an implementation, and many others.

A formal specification is a "concise description of the behavior and properties of a system written in a mathematically-based language, specifying what a system is supposed to do as abstractly as possible, thereby eliminating distracting detail and providing a general description resistant to future system modifications" [Kelly, 95]. Formal specifications are a translation of a non-mathematical description of the high-level behavior and properties of a system (diagrams, tables, natural language) into a formal specification language, which gives a concise and precise description [Kelly, 95]. Formal specifications describe a system and its desired properties, which may include functional behavior, timing behavior, performance characteristics, or internal structure [Clarke, 96]. The language used for the specification must have a well-defined semantics based on mathematics in order to support deductions.

2.3. Discrete Processes and Temporal Logic

Most processes in organizations fall in the category of discrete event systems [Cassandras, 09], in which processes are characterized by a start event, a duration and a termination event. Of course, the start of a process goes before the end. This is called a dependency. There are also more complex dependencies, like for instance that one process goes altogether after another one, so the finish time for the latter goes before the start time for the former. Time dependencies have been studied in much detail in many forms of temporal logic [Fisher, 11] [Lamport, 83] [Rozier, 10]. Temporal logic is related to Modal Logic [Chellas, 80] and is specifically used to describe the temporal ordering of events.

Discrete event systems have discrete states, and a sequence between these states. Understanding this sequence is critical for process design and audit and can lead to critical questions. For instance, if we have been in a state of the system, is it possible to fall in that state again? Are there states from which it is not possible to exit? These execution properties are of great interest for process auditing purposes because they could imply that a given process is or is not correct. Expressing states and sequences in a way that helps us answer these questions is possible with a temporal logic called Computation Tree Logic (CTL).

2.4. CTL

We are going to consider one particular form of temporal logic which we found suited to change control specification, namely, Computation Tree Logic (CTL). CTL has been very well studied and there are computer implementations of it that allow experiments and verifications. We will take advantage of these implementations in this work. In CTL, each moment in time can be divided into several possible futures. Essentially, “this logic sees the structure of time as a tree, rooted in the present time, with a series of branching paths at each node of the tree” [Rozier, 10].

Regardless of the original form of the considered process, eventually it should be translated into a state transition system [Harel, 98]. A state transition system can be visualized as a graph, called an automaton or Kripke structure (Figure 2) that represents the behavior of a system [Rozier, 10], by means of a collection of states linked by arrows that represent the changes from one state to another.

The Kripke structure is a tuple M = (S, I, R, L), where S is a finite set of states, I ⊆ S is a set of initial states, and R ⊆ S S is the transition relation from one state to the next [Cimatti, 00]. Additionally, there is a labeling function L that attaches properties to each state. The properties take logical values that can be true or false.

CTL is a mathematical language for declaring properties about the Kripke structure [Fisher, 11]. The declared properties refer not to the automaton itself, but to the possible executions (sequences of transitions) that could happen in the automaton. CTL makes it possible to extract a static mathematical entity from the dynamics of the automaton. In CTL, the computation tree is this static entity. Consider, for instance, the automaton in Figure 2 [Rozier, 10], which includes *p*, *q* and *r* conditions attached to states s_0, s_1 and s_2. Taking s_0 as the initial state, it is possible to transition to s_1 or s_2; this is represented in the computation tree in Figure 3 [Cimatti, 00] as a tree with root s_0, and branches to s_1 and s_2. As the execution continues, there will be branches extending and eventually dividing. In many cases, branches could have infinite length.

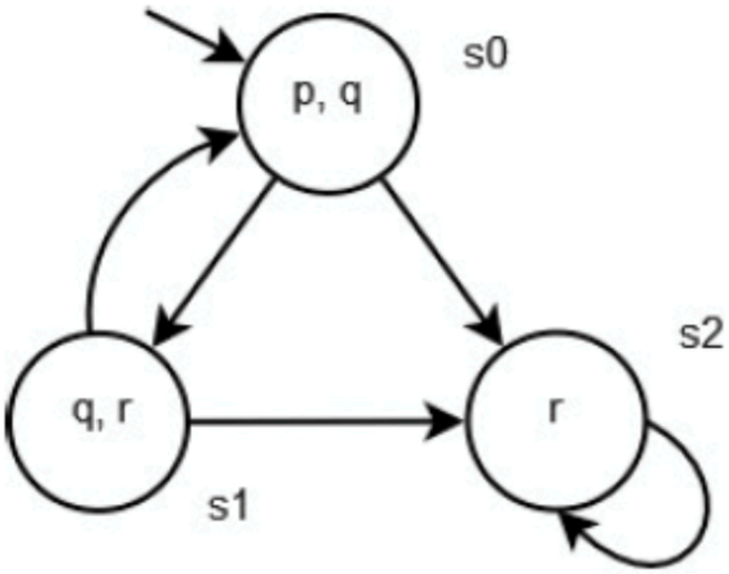


Figure 2. Kripke structure (based on [Cimatti, 00])

For the computation tree in fig 2b, we could consider the following property: “whenever the condition *r* is established, it is never lost.” This property could be true or false (in this particular case it is false, as can be shown by starting in s_0, moving left to s_1 and then left again to s_0); of course this example is a simple automaton that we can visually verify, but there could be much more complex automata and properties (imagine you are dealing with an automaton with hundreds of conditions and states), so specialized languages and methods have been proposed, one of which is CTL.

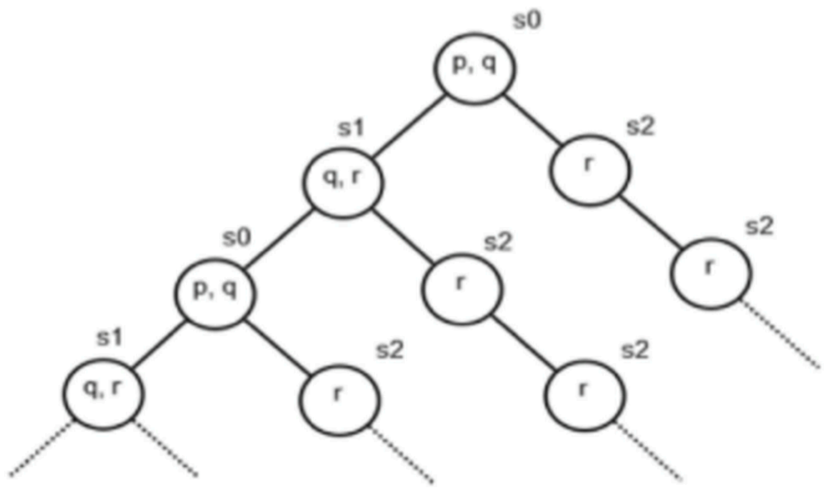


Figure 3. Computation tree (based on [Cimatti, 00])

The CTL language uses two combined special quantifiers: the first one may take value **A** (property should hold along **A**ll paths) or **E** (there **E**xists at least one path where it holds), while the second one may take value F (at some particular point in the Future), **G** (**G**lobally, standing from the present all the way to the future), **X** (in the ne**X**t state) or **U** (**U**ntil a condition is met). So, the combined possible quantifiers are **AG**, **AF**, **AX**, **AU**, **EG**, **EF**, **EX**, **EU**, and they could be nested in formulas.

For instance, the statement “whenever the property r is established, it is never lost” could be represented by the CTL formula: **AG** ($r \Rightarrow AG\ r$). A counterexample, showing that this formula is false for the considered Kripke structure, would take s_1 as the present state, and go from s_1 to s_0, which contradicts the property, because in s_0 property *r* does not hold. Consult a CTL introduction for additional details [Fisher, 11].

The specific CTL “dialect” we are going to use is NuSMV [Cimatti, 00] (see below).

3. Modeling and Verification Method

In this section, we present our proposed formal verification method for CCI processes in project management. The method we propose is composed of the following 5 steps:

1. Start with a detailed change control process suited for a specific context.
2. Express the change control process as a state transition system, in the notation of a Kripke structure.
3. Translate the Kripke structure to a program in the NuSMV language.
4. Express the key properties we are interested in validating as CTL formulas in the NuSMV language.
5. Test the properties in the NuSMV tool.

Now we explain how we apply each step, following the case study of the ICC process. The ICC process considered is based on the integration of criteria from the PMI and other organizations, as mentioned before.

3.1. Start with a detailed change control process

As commented before, the general ICC process from PMI does not consider the specific activities that are involved in change management. In the case study, we consider activities proposed by the Construction Industry Institute (CII) [Ibbs, 01]. By combining concepts about information flows from the Integrated Change Control process (according to PMI) [PMI, 17] and the subprocesses and control activities defined by the CII, we can define a process that contains sufficient detail for simulation and analysis.

Figure 4 shows a flowchart fragment associated with the proposed Integrated Change Control process. This flowchart shows analysis activities that apply to any change request. Figure 4 considers the activities indicated by the process proposed by Ibbs [Ibbs, 01] and adds other activities inspired in ITIL [Rance, 11]. The first activities are associated with identifying the need for a change, followed by those that have to do with registering and analyzing the corresponding change request. If the change does not have urgent priority, time is taken to check whether the information registered in the change request is complete, if the request is duplicated, or if the change can be postponed. This process finally shows the treatment given to a change request that is detected as duplicated. The process must be detailed and appropriate for the context. To begin understanding the process as a state transition system, states in the process must be identified. Black dots show key states in the detailed Integrated Change Control process.

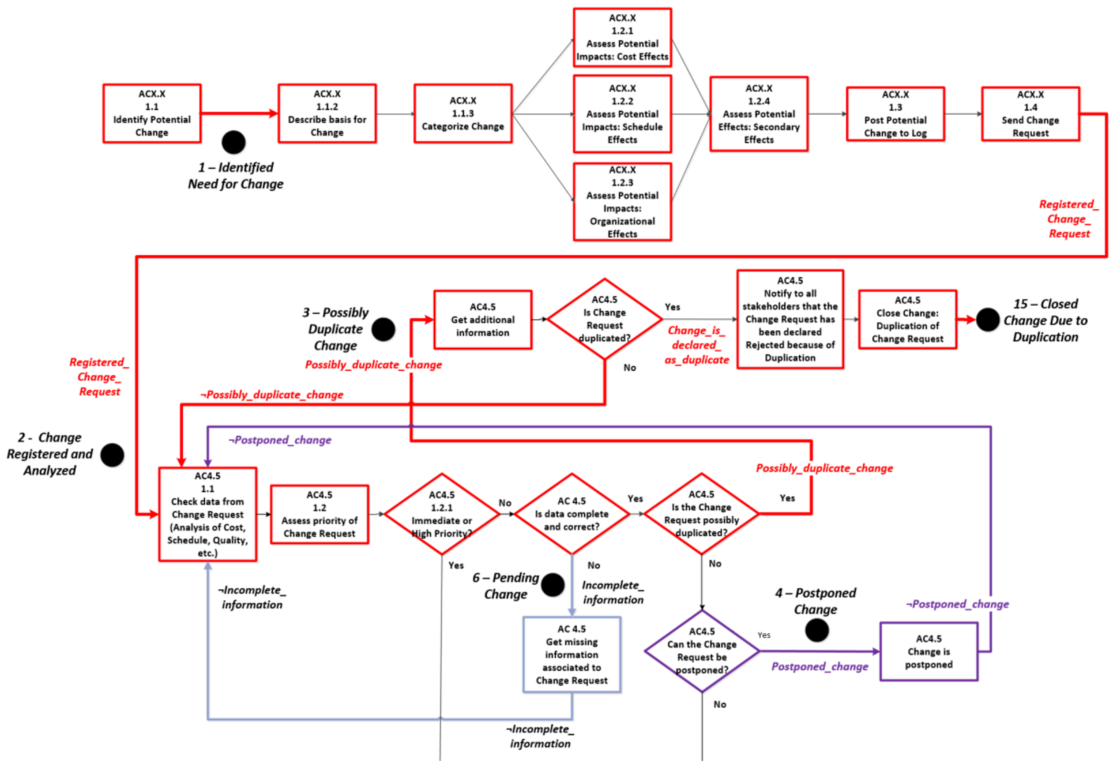

ACX.X 1.1 Identify Potential Change
ACX.X 1.1.2 Describe basis for Change
ACX.X 1.1.3 Categorize Change
ACX.X 1.2.1 Assess Potential Impacts: Cost Effects
ACX.X 1.2.2 Assess Potential Impacts: Schedule Effects
ACX.X 1.2.3 Assess Potential Impacts: Organizational Effects
ACX.X 1.2.4 Assess Potential Effects: Secondary Effects
ACX.X 1.3 Post Potential Change to Log
ACX.X 1.4 Send Change Request
1 – Identified Need for Change
Registered_ Change_ Request
3 – Possibly Duplicate Change
AC4.5 Get additional information
AC4.5 Is Change Request duplicated?
Yes
No
Change_is_ declared_ as_duplicate
AC4.5 Notify to all stakeholders that the Change Request has been declared Rejected because of Duplication
AC4.5 Close Change: Duplication of Change Request
15 – Closed Change Due to Duplication
Possibly_duplicate_change
¬Possibly_duplicate_change
¬Postponed_change
2 - Change Registered and Analyzed
AC4.5 1.1 Check data from Change Request (Analysis of Cost, Schedule, Quality, etc.)
AC4.5 1.2 Assess priority of Change Request
AC4.5 1.2.1 Immediate or High Priority?
AC 4.5 Is data complete and correct?
AC4.5 Is the Change Request possibly duplicated?
6 – Pending Change
Incomplete_ information
¬Incomplete_ information
AC 4.5 Get missing information associated to Change Request
4 – Postponed Change
AC4.5 Can the Change Request be postponed?
Postponed_change
AC4.5 Change is postponed

Figure 4. Flowchart for initial steps associated with the analysis of a change request

3.2. Express the change control process as a state transition system

Figure 5 shows the state transition system obtained from the flowchart shown in Figure 4. The transformation from regular flowchart to the corresponding state transition system is treated by [Bjørner, 70], as every flowchart can be represented in a state transition system [29]. The six black dots in Figure 4 have become the six highlighted ovals in Figure 5.

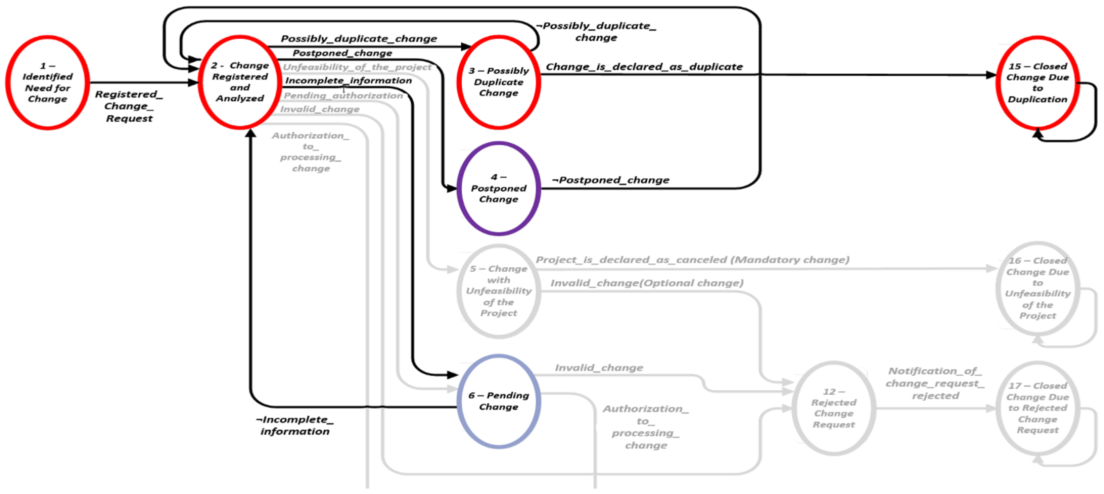

1 – Identified Need for Change
Registered_ Change_ Request
2 - Change Registered and Analyzed
Possibly_duplicate_change
Postponed_change
Unfeasibility_of_the_project
Incomplete_information
Pending_authorization
Invalid_change
Authorization_ to_ processing_ change
3 – Possibly Duplicate Change
¬Possibly_duplicate_ change
Change_is_declared_as_duplicate
15 – Closed Change Due to Duplication
4 – Postponed Change
¬Postponed_change
5 – Change with Unfeasibility of the Project
Project_is_declared_as_canceled (Mandatory change)
Invalid_change(Optional change)
16 – Closed Change Due to Unfeasibility of the Project
6 – Pending Change
Invalid_change
Authorization_ to_ processing_ change
¬Incomplete_ information
12 – Rejected Change Request
Notification_of_ change_request_ rejected
17 – Closed Change Due to Rejected Change Request

Figure 5. Partial state transition system obtained from flowchart in Figure 4.

A complete state transition system that models the ICC process is defined by exhaustively considering all possible states and transitions. It consists of 20 states and 35 transitions, represented visually as a Kripke structure (automaton) in Figure 6 and Figure 7. Figures 6 and 7 are constructed the same way as Figure 5, but they include the entire state transition system. Figures 6 and 7 illustrate the states of the ICC process and the state variables that trigger transitions from one state to another. The figures ought to be interpreted starting in Figure 6 at state #1: *"IdentifiedNeedForChange"*. Figure 6 connects to Figure 7 via state #7: *"AuthorizedChange"*. Altogether, Figures 6 and 7 are simply an exhaustive state transition system of the ICC process.

The sub-sequence in Figure 8 shows one possible path of states and transitions for a particular change. The change begins in state #1: *"IdentifiedNeedForChange"*. Once a need for change has been identified, the change request is registered and analyzed by the Change Control Board (state #2: *"ChangeRegisteredAndAnalyzed"*). In this particular scenario, the Board identifies that the change is possibly a duplicate change, and the change passes to state #3: *"PossiblyDuplicateChange"*. This state represents activities that check if the request is indeed a duplicate. If the change is not confirmed as a duplicate it returns to state #2: *"ChangeRegisteredAndAnalyzed"*. If the change is confirmed as a duplicate, it is declared a duplicate and enters the terminal state #15: *"ClosedChangeDueToDuplication"*.

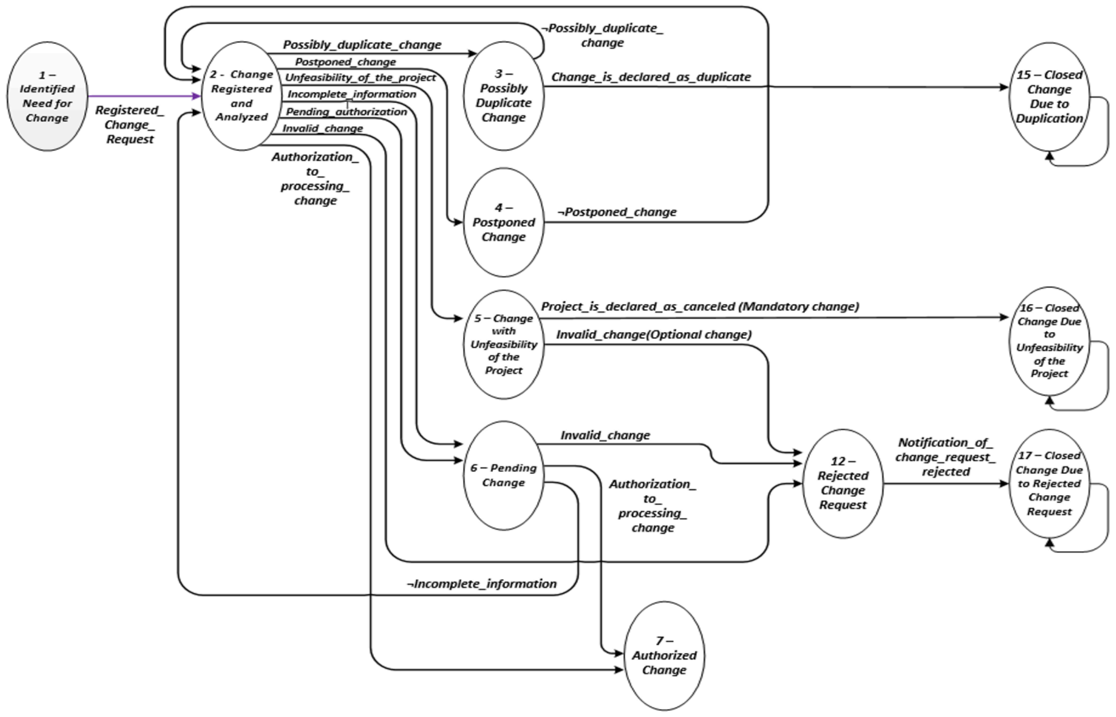

1 – Identified Need for Change
Registered_ Change_ Request
2 - Change Registered and Analyzed
Possibly_duplicate_change
Postponed_change
Unfeasibility_of_the_project
Incomplete_information
Pending_authorization
Invalid_change
Authorization_ to_ processing_ change
3 – Possibly Duplicate Change
¬Possibly_duplicate_ change
Change_is_declared_as_duplicate
15 – Closed Change Due to Duplication
4 – Postponed Change
¬Postponed_change
5 – Change with Unfeasibility of the Project
Project_is_declared_as_canceled (Mandatory change)
Invalid_change(Optional change)
16 – Closed Change Due to Unfeasibility of the Project
6 – Pending Change
Invalid_change
Authorization_ to_ processing_ change
¬Incomplete_information
12 – Rejected Change Request
Notification_of_ change_request_ rejected
17 – Closed Change Due to Rejected Change Request
7 – Authorized Change

Figure 6. State transition system for the ICC process (part 1)

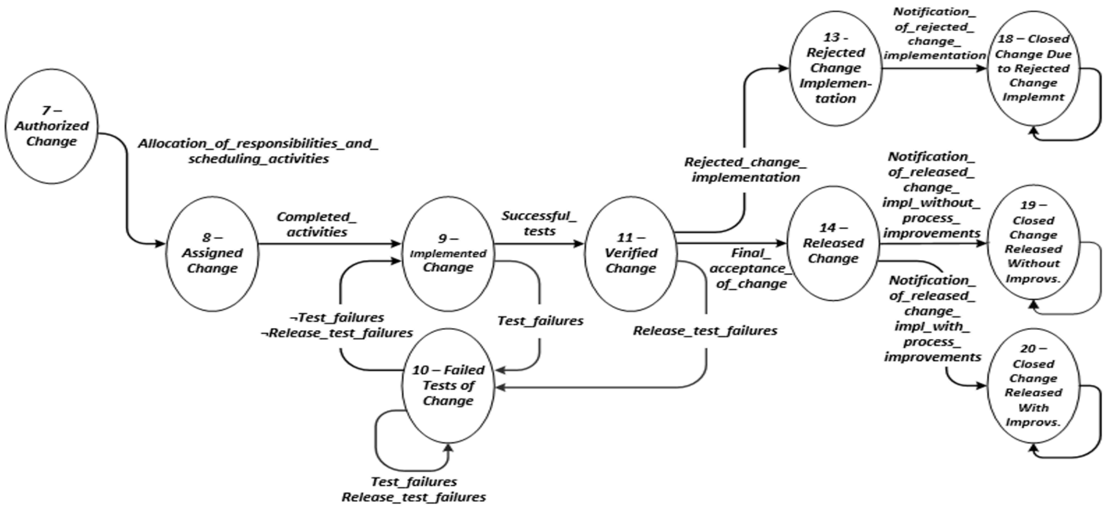

7 – Authorized Change
Allocation_of_responsibilities_and_ scheduling_activities
8 – Assigned Change
Completed_ activities
9 – Implemented Change
Successful_ tests
11 – Verified Change
¬Test_failures
¬Release_test_failures
Test_failures
Release_test_failures
10 – Failed Tests of Change
Test_failures
Release_test_failures
Rejected_change_ implementation
13 - Rejected Change Implemen- tation
Notification_ of_rejected_ change_ implementation
18 – Closed Change Due to Rejected Change Implemnt
Final_ acceptance_ of_change
14 – Released Change
Notification_ of_released_ change_ impl_without_ process_ improvements
19 – Closed Change Released Without Improvs.
Notification_ of_released_ change_ impl_with_ process_ improvements
20 – Closed Change Released With Improvs.

Figure 7. State transition system for the ICC process (part 2)

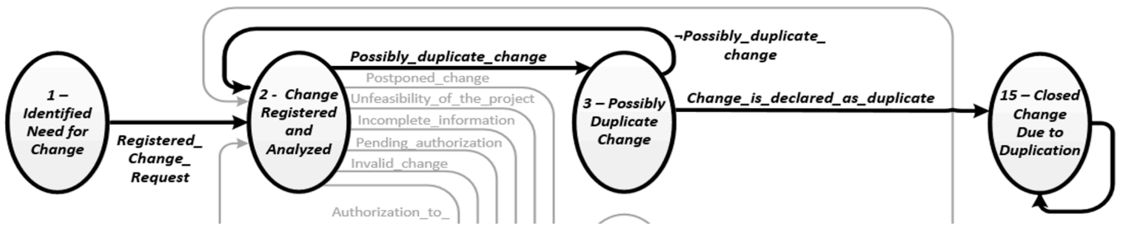

1 – Identified Need for Change
Registered_ Change_ Request
2 - Change Registered and Analyzed
Possibly_duplicate_change
Postponed_change
Unfeasibility_of_the_project
Incomplete_information
Pending_authorization
Invalid_change
Authorization_to_
3 – Possibly Duplicate Change
¬Possibly_duplicate_ change
Change_is_declared_as_duplicate
15 – Closed Change Due to Duplication

Figure 8. Example of state transitions. Sequence for duplicate change.

Figures 6 and 7are the state transition system visualized as a Kripke structure. Without altering the state transition system of the ICC process, we can "unweave" it using CTL, and this will facilitate declaring its desired properties in temporal logic.

In CTL there exists the idea of multiple possible futures. Multiple possible futures are the different possible states realizable from a current state, which branch from the present time. For instance, suppose the change process is at the present time in state #2 in Figure 8. From there, there are 7 different transitions, all of which could be taken; so the next state from state #2 could be state #3, #4, #5, #6, #7 or #12. These alternatives can be represented as branches from state #2 to these other states. Then, from state #3 there are two different possibilities, either to go back to state #2 or go to state #15, so two branches are going out from state #3, and so on. Figure 9 shows the computation tree that is obtained by unfolding the sequence of state transitions associated with the case of the duplicate change in Figure 8.

The properties stated using the CTL language refer to the unfolded computation tree, rather than the original Kripke structure. There is a labeling function that attaches properties to states. We have introduced a collection of auxiliary variables that will be useful for dealing with the formulation of properties.

A group of auxiliary variables refers to declarations concerning past states. Keeping track of past states is fundamental in the practice of change control. For example: "Whenever a change is assigned (state #8), previously it has been registered and analyzed (state # 2)." CTL by construction, however, solely permits users to define properties concerning present or future events. For example: "In all possible executions from the present state, at some future point the property X will be true." So auxiliary variables are introduced to keep track of history. In this case the variable *"RegisteredChangeRequest"* keeps track of passing through state #2. Other auxiliary variables remember global situations that we need to track, like for instance *"PostponedChange"* or *"InvalidChange"*. Table 1 shows the relationship between the state variables and the values they have in each of the states of the ICC process.

3.3. Express the ICC state transition system in the computational platform

Moving the ICC process from a state transition system to a coded model requires declaring the states with their variables and transitions. Table 1 shows the entire ICC process with states in the rows and state variables in the columns.

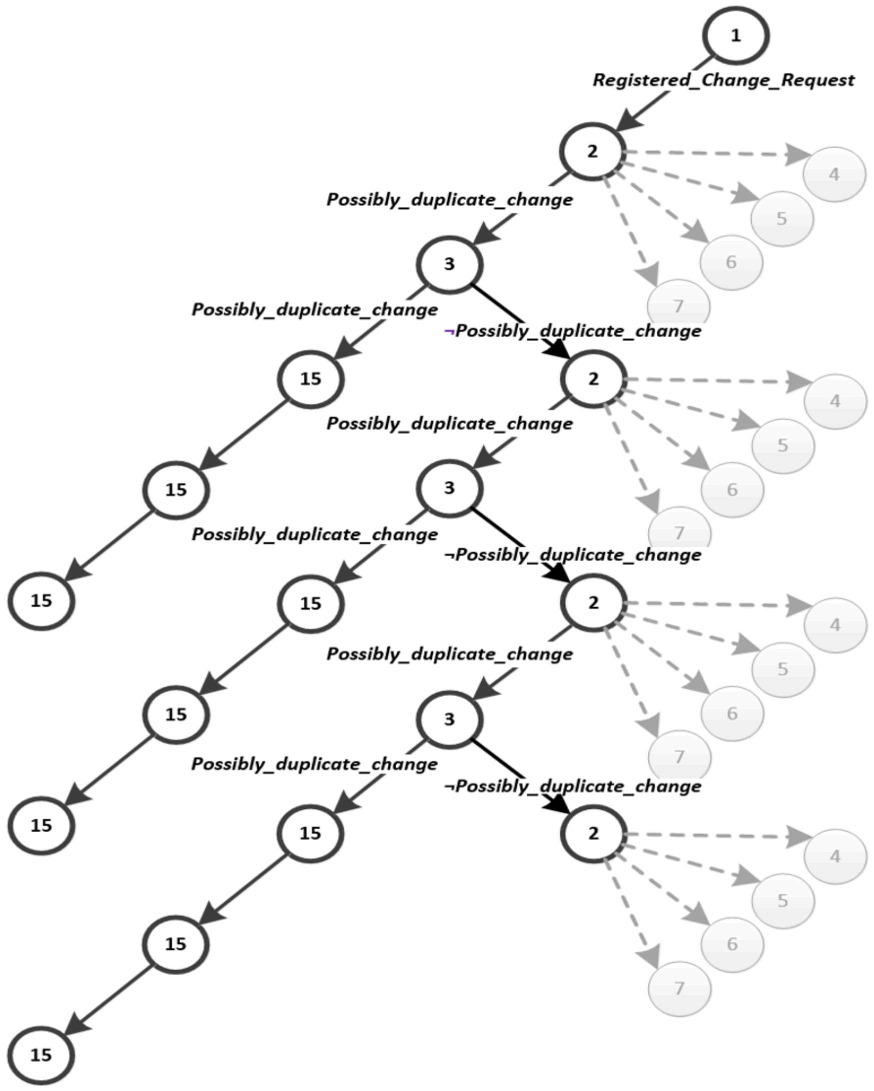
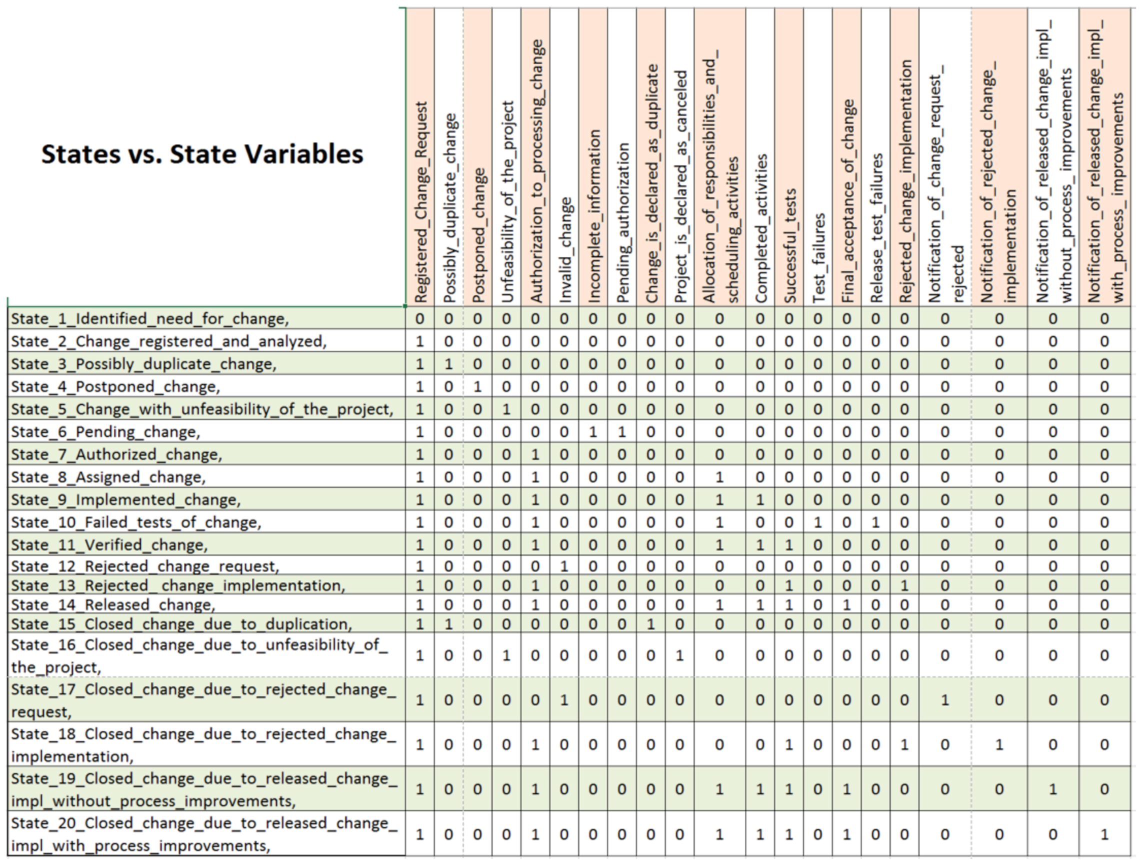

1
Registered_Change_Request
2
4
5
6
7
Possibly_duplicate_change
3
Possibly_duplicate_change
¬Possibly_duplicate_change
15

Figure 9. Computation tree for duplicate change shown in Fig 5.

The computational platform we used for implementing our CTL models is called *NuSMV*. NuSMV [Cimatti, 00] is a model-checking tool for verifying CTL properties over automata. To do so, NuSMV explores systematically and exhaustively all relevant execution paths in order to determine with certainty if a property is true or false. Even if some paths in the execution tree are infinite, model checking is guaranteed to terminate. When a property is found not to hold, a counterexample is presented as evidence. NuSMV provides ways for representing the Kripke structure using a programming language, testing it interactively, and checking properties written in CTL and other temporal logic variants. Thus, change control processes in project management can be formally modelled by starting with a visualization such as a Kripke structure, creating a computation tree using CTL, and verifying the model with a tool such as NuSMV.

We used a global state variable that represents the current state in the automaton as a way to make direct references to states when validating model properties (see section 4). This variable is called "*State*" and takes the value of each state of the model according to the following statement expressed in the specification language of NuSMV tool.



States vs. State Variables

	Registered_Change_Request	Possibly_duplicate_change	Postponed_change	Unfeasibility_of_the_project	Authorization_to_processing_change	Invalid_change	Incomplete_information	Pending_authorization	Change_is_declared_as_duplicate	Project_is_declared_as_canceled	Allocation_of_responsibilities_and_ scheduling_activities	Completed_activities	Successful_tests	Test_failures	Final_acceptance_of_change	Release_test_failures	Rejected_change_implementation	Notification_of_change_request_ rejected	Notification_of_rejected_change_ implementation	Notification_of_released_change_impl_ without_process_ improvements	Notification_of_released_change_impl_ with_process_ improvements
State_1_Identified_need_for_change,	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_2_Change_registered_and_analyzed,	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_3_Possibly_duplicate_change,	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_4_Postponed_change,	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_5_Change_with_unfeasibility_of_the_project,	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_6_Pending_change,	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
State_7_Authorized_change,	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_8_Assigned_change,	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
State_9_Implemented_change,	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
State_10_Failed_tests_of_change,	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0
State_11_Verified_change,	1	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
State_12_Rejected_change_request,	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State_13_Rejected_ change_implementation,	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
State_14_Released_change,	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0
State_15_Closed_change_due_to_duplication,	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
State_16_Closed_change_due_to_unfeasibility_of_ the_project,	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
State_17_Closed_change_due_to_rejected_change_ request,	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
State_18_Closed_change_due_to_rejected_change_ implementation,	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0
State_19_Closed_change_due_to_released_change_ impl_without_process_improvements,	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0
State_20_Closed_change_due_to_released_change_ impl_with_process_improvements,	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	1

Table 1. States vs. state variables in the Integrated Change Control model

```
State :{
State_1_IdentifiedNeedForChange,
State_2_ChangeRegisteredAndAnalyzed,
State_3_PossiblyDuplicateChange,
...
State_20_ClosedChangeReleasedWithProcessImprovements};
```

The declaration of the transitions shown in Figure 8 is done using an operator “next” as follows; the initial state is specified with the operator “init”:

```
Init(State) := State_1_IdentifiedNeedForChange;
next(State) := case
    State = State_1_IdentifiedNeedForChange:
              State_2_ChangeRegisteredAndAnalyzed;
    State = State_2_ChangeRegisteredAndAnalyzed:
              State_3_PossiblyDuplicateChange;
    State = State_3_PossiblyDuplicateChange :{
             State_2_ChangeRegisteredAndAnalyzed,
             State_15_Closed_change_due_to_duplication};
```

The statement above is only part of the automaton. It represents the sequences of transitions associated with the case of a duplicate change. The declaration of the entire automaton should consider all possible transitions from one state to all next states that are considered in the model. In order to obtain a complete declaration, it is necessary to inspect each state and declare all its possible transitions. Thus the change control process has been codified in programming language.

We notice in the last line of the example code that expressions in NuSMV do not necessarily evaluate to a unique value as a result. In general, the terms take a value in a non-deterministic way from a set of possible values.

The values adopted by state variables in each state can be expressed in the NuSMV tool. From the information about states and state variables presented in Table 1, it is easy to declare the updating rules for state variables. A '1' at the intersection of a state and state variable in Table 1 is coded as 'TRUE' in NuSMV. So, for example, in the case of the variable "*AuthorizationForProcessingChange*", the specification that governs its value based on the current state is declared as follows:

```
AuthorizationForProcessingChange:= case
    State = State_1_IdentifiedNeedForChange: FALSE;
            State = State_2_ChangeRegisteredAndAnalyzed:
            FALSE;
    ...
    State = State_19_ClosedChangeDueToReleasedChangeWithoutProcessImprovements : TRUE;
    State = State_20_ClosedChangeReleasedWithProcessImprovements: TRUE;
     TRUE: FALSE;
            esac;
```

The NuSMV tool can randomly generate samples containing execution traces of submitted changes, simulating many changes within the programmed model. These sample changes follow the programmed model as the written code allows. We simulated 120 changes to confirm that the programmed change control process model follows the behavior seen in the state transition system. One-hundred twenty random changes were generated for the test, 112 of which reached one of the final states within the maximum of 20 state transitions that we imposed; the other 8 changes would have required more than 20 state transitions to reach a final state and therefore were truncated. On manual inspection, in all cases, the sequences of the state transitions behaved following the ICC process state transition system (including the 8 cases that required more than 20 state transitions, until truncation).

Of course, a finite number of runs does not give absolute surety of the correctness of the change control process, and this is indeed one of the main reasons to carry formal proofs, rather than solely simulations, in the first place. Nevertheless, we actually detected some flaws in the model by using this simulated checking. When we simulated changes based on an earlier version of the ICC state transition model, we identified a case with an execution sequence of states including 6-7-12 (see Figure 10). Now, it makes no sense that a pending change (state #6), receives authorization (state #7) and subsequently the corresponding change request is rejected (state #12).

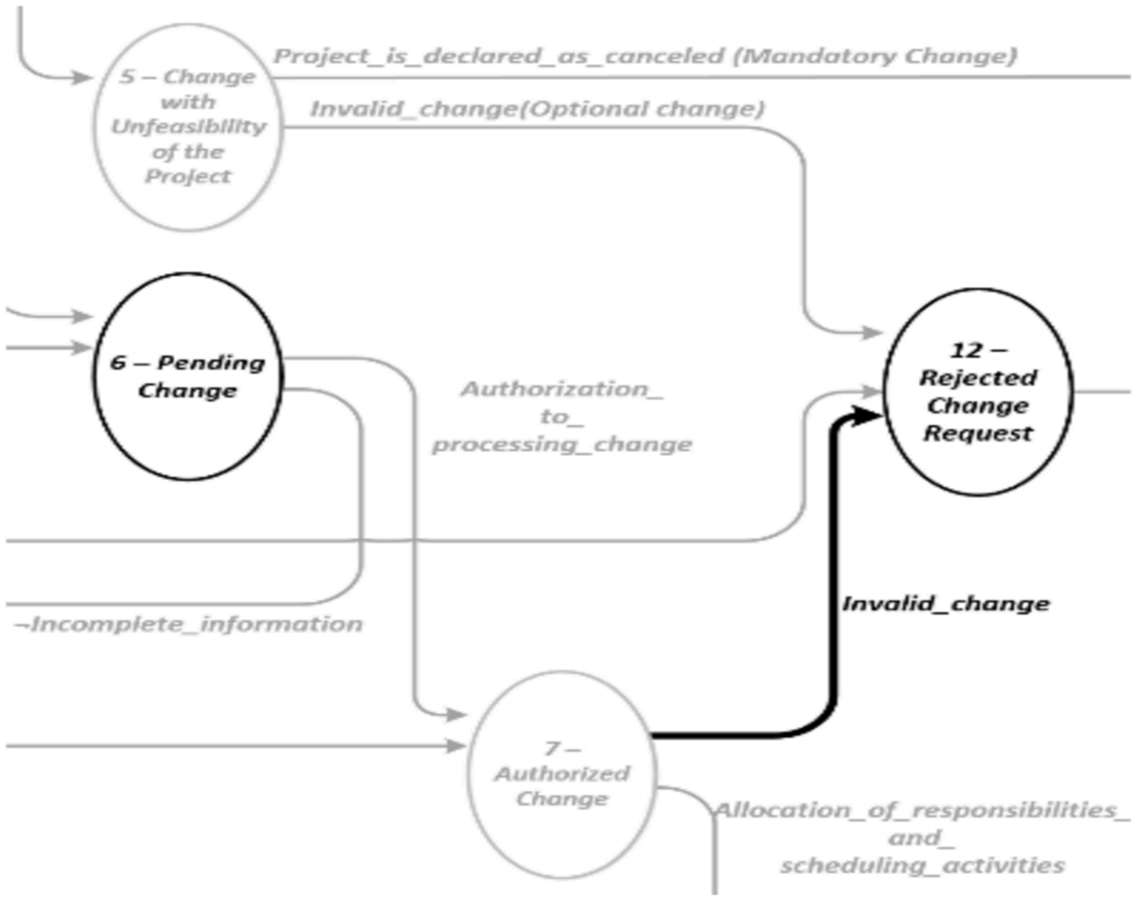

5 – Change with Unfeasibility of the Project
Project_is_declared_as_canceled (Mandatory Change)
Invalid_change(Optional change)
6 – Pending Change
Authorization_ to_ processing_change
12 – Rejected Change Request
¬Incomplete_information
Invalid_change
7 – Authorized Change
Allocation_of_responsibilities_ and_ scheduling_activities

Figure 10. Invalid transition from state #7 to state #12 - before model correction.

Not only did we find an error in the computer-programmed model, but we realized a deficiency in the state transition model: there was not transition from state #6: “PendingChange” to state #12: “RejectedChangeRequest”. We determined that the model should be adjusted so that the pending change (state #6) could receive additional information (and transitioning to state #2), or be authorized (transitioning to state #7), or be rejected (transitioning state #12). So, we fixed the coded model by eliminating the transition from state #7 to state #12, and we improved the state transition system by adding the transition from state #6 to #12 associated to the activation of the variable “InvalidChange”, which was then also reflected in the computer model. This is shown in Figure 11.

In the following section, we are going to present the two last steps in our method: expressing key properties in CTL using the NuSMV language and finally validating those key properties or disproving them.

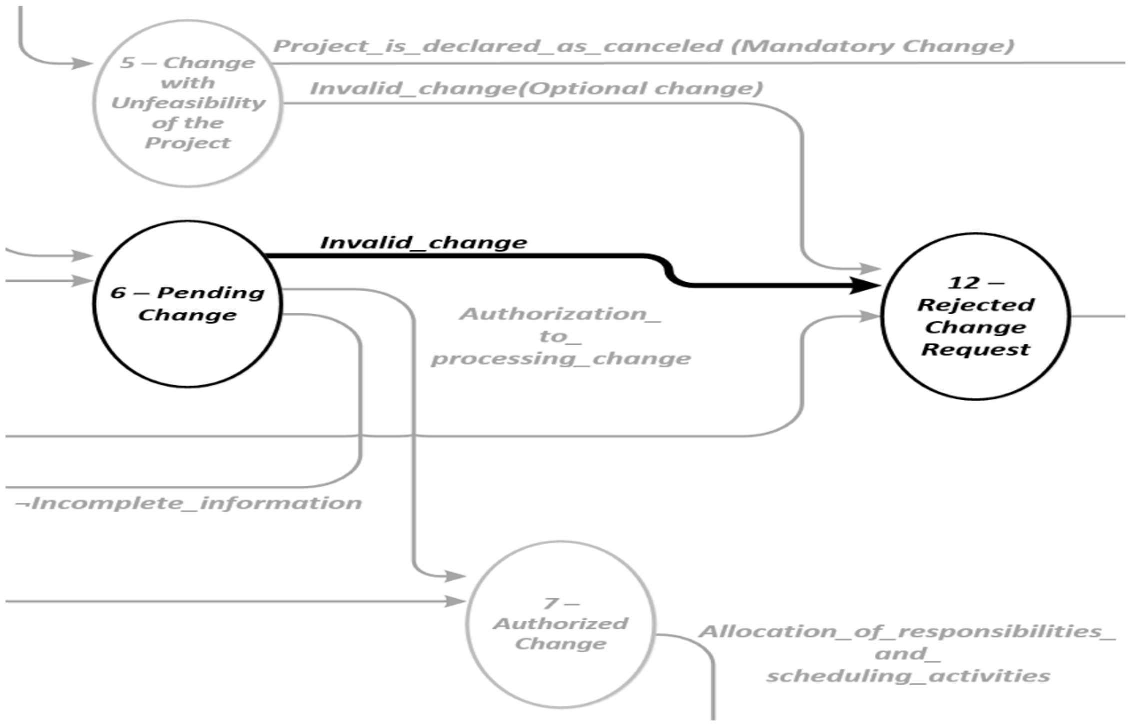

Project_is_declared_as_canceled (Mandatory Change)
5 – Change with Unfeasibility of the Project
Invalid_change(Optional change)
Invalid_change
6 – Pending Change
Authorization_ to_ processing_change
12 – Rejected Change Request
¬Incomplete_information
7 – Authorized Change
Allocation_of_responsibilities_ and_ scheduling_activities

Figure 11. Model corrected by adding transition from state #6 to state #12 and removing transition from state #7 to state #12.

4. Results: Specification and experimental verification of key properties

After coding the change control process, we arrive at the point where we can declare the key properties that the process must satisfy. In our experience, CTL temporal logic proved to be very adequate for representing and testing the kind of properties that we wanted the process to have. In the NuSMV tool [Cimatti, 00], properties are declared using the keyword "SPEC" followed by a CTL formula, as we will see in the examples in this section.

4.1. Reachability properties

A property that is checked in many systems is the *reachability* of final states. This is important because if a final state is not reachable from the initial or other states through transitions in the automaton, the final state becomes irrelevant. In our model, we seek to validate that, from the initial state, we could arrive to a state where improvements have been approved and the ticket is closed. Taking our model with the states listed in Table 1, one final state is state #20: *"ClosedChangeReleasedWithProcessImprovements"*. We must ensure that it is indeed possible to get to state #20 from the initial state and possibly from other states as well. The following specification declares the possibility of getting to final state #20 from the initial state:

```
SPEC State = State_1_IdentifiedNeedForChange ->
EF (State = State_20_ClosedChangeReleasedWithProcessImprovements)
```

That is, if we are currently in state #1, then there is a path that in the future leads to state #20. Indeed, this property was successfully verified in NuSMV. Needless to say, in our small model this could be verified by hand, but in a model with thousands of states the task is far from trivial.

4.2. Prerequisite properties

One set of properties that we are going to verify refers to prerequisites that are supposed to have been fulfilled when we arrive at a certain point in the process. We are now trying to validate that, in all possible configurations, every change assigned for implementation should previously be registered and authorized. In terms of our states, this means that we want to be sure that whenever we arrive at state #8, we have already been in states #1, #2 and #7, and possibly state #6 as well.

One difficulty for specifying a property like this, as we have commented before, is that CTL can state properties of *present and future* states in the system, not past states. We solved this problem by introducing some "history markers," logical variables that are set to true when the event we want to remember happens. When a state we want to remember is reached, we set a corresponding variable to true, for instance the variable "*ItHasGoneThroughState2*". As the process evolves through the transitions, we keep track of the corresponding changes in the history marker variables, using both "init" and "next" operators in the NuSMV tool.

Take for instance the property: "All change requests should always be recorded and previously authorized in order to be allocated for implementation." To verify this property, we first translate it to the corresponding states and history markers.

Taking state #8: *"AssignedChange"* as the current state, we can see this state is a future state from prerequisite states such as #1, #2, #6 and #7. Thus, the possible state transitions to reach state #8, are the following: 1-2-7 and 1-2-6-7. The specification requiring state #8 to have these possible prerequisites is coded as follows, and has been proven true by the NuSMV tool.

```
 SPEC AG ((State = State_8_AssignedChange) ->
  (ItHasGoneThroughState1 & ItHasGoneThroughState2 & ItHasGoneThroughState7 ) |
  (ItHasGoneThroughState1  &  ItHasGoneThroughState2  &  ItHasGoneThroughState6  &
ItHasGoneThroughState7 ))
```

4.3. Definitiveness properties

Another set of properties establishes that negative resolutions are definitive, for instance in the case that “Invalid change” has been decided. Whenever a request for an optional change turns the project unfeasible, the Change Control Board should reject the change request and not assign it for processing. In other words, an optional change request cannot be assigned if it turns the project unfeasible, and therefore the change request should be rejected. We want to declare that after an optional change is judged invalid because it turns the project unfeasible, there is no way in the process that the declaration of “Invalid change” could be overturned.

This property can be expressed by means of the state variable “*MandatoryChange*” (if the state variable “*MandatoryChange*” is false, then the change is optional) and the state variable “*UnfeasibilityOfProject*” (if this variable is true, then the change turns the project unfeasible. In the combination of an optional change and an unfeasible project the change request is considered invalid (the variable “*InvalidChange*” is set true) and it is not possible to set true the other state variables “*AuthorizationForProcessingChange*” and “*AllocationOfResponsibilities_SchedulingActivities*”. The coded specification states that under these circumstances, it is only possible to reach the state #12: “RejectedChangeRequest”, and impossible to reach state #8: *“AssignedChange”*. Since these actions must always be carried out when the situation described occurs, the CTL AG operator is used.

The *specification* of this property is then stated as follows:

```
SPEC (!MandatoryChange & UnfeasibilityOfProject ) -> AG (InvalidChange &
!AuthorizationForProcessingChange & !AllocationOfResponsibilities_SchedulingActivities )
```

An important remark about the expression of a property like this is that the "informal" formulation could be ambiguous. "Negative resolutions are definitive" can be interpreted differently according to one's tolerance for negativity and level of authority. On the contrary, once the specification formula is stated in a formal model, its meaning is precise and completely unambiguous.

4.4. Mutual exclusion

Another interesting property is mutual exclusivity. Figure 12 shows three mutually exclusive paths that can follow from state #11: *"VerifiedChange"*. Only one of the following three situations may occur (mutually exclusive):

- The Change Control Board accepts the change for release and updates subsidiary plans, the comprehensive plan, or associate baselines (state #14 *"ReleasedChange"*).
- The Change Control Board identifies failures in release tests and the change moves to state #10: *"FailedTestsOfChange".*
- The Change Control Board refuses to release the implementation of the change and decides to reject it and close the case (state #13: "*RejectedChangeImplementation*").

This property can be represented by two equivalent specifications expressed in CTL:

- One solution uses the variables associated with the model states: the current state is "*State_11_VerifiedChange*" and immediately the next state may be "*State_14_ReleasedChange*", "*State_10_FailedTestsOfChange*", or "*State_13_RejectedChangeImplementation*".

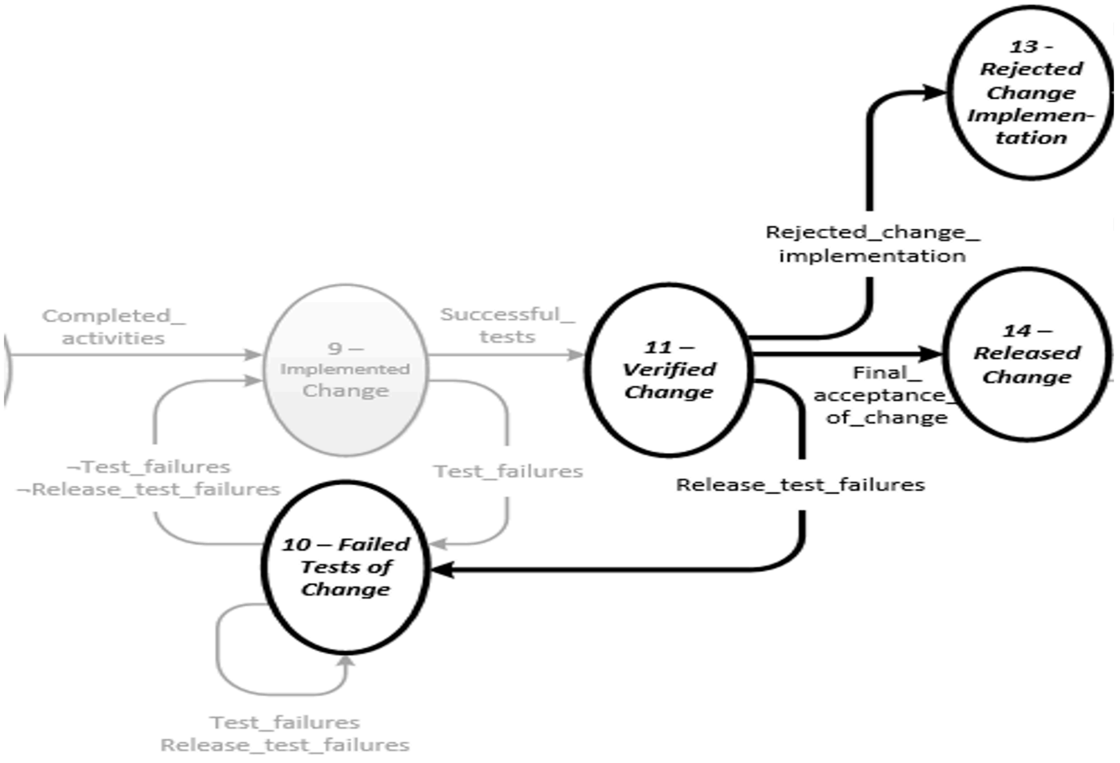


Figure 12. Example of mutually exclusive states and transitions.

- The other solution uses the state variables associated with the possible paths that lead from the initial state (*State_1_IdentifiedNeedForChange*) to the current state (*State_11_VerifiedChange*).

In both cases, a mutually exclusive relationship exists between states immediately following the current state (*State_11_VerifiedChange*); this is expressed using the AX operator, which gives the idea of a mandatory situation.

This specification is stated as follows; it has been proved true by the NuSMV tool:

```
SPEC     ((RegisteredChangeRequest & AuthorizationForProcessingChange &
AllocationOfResponsibilities_SchedulingActivities & CompletedActivities &
SuccessfulTests) -> AX (
((State = State_13_RejectedChangeImplementation) &
!(State = State_14_ReleasedChange) &
!(State = State_10_FailedTestsOfChange ))|
(!(State = State_13_RejectedChangeImplementation) &
 (State = State_14_ReleasedChange) &
!(State = State_10_FailedTestsOfChange ))|
(!(State = State_13_RejectedChangeImplementation) &
!(State = State_14_ReleasedChange) &
 (State = State_10_FailedTestsOfChange)))
```

4.5. Cycles

Yet another set of properties refers to the case of cyclic sequences of state transitions. For instance, it should be true for all situations where a test fails, that a correcting cycle leads to new tests and eventually to approval. In the state model, we have the cycles involving states #9 and #10 as shown in Figure 13.

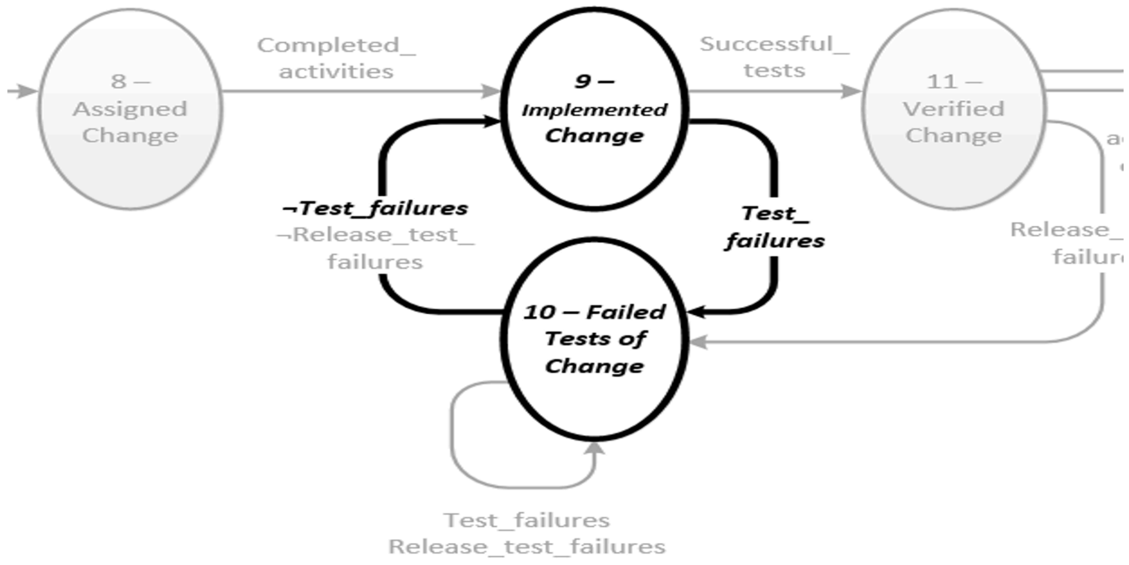


Figure 13. Example of property associated to cyclic paths

The example shows the case of transitions between state #9: “*ImplementedChange*” and state #10: “*FailedTestsOfChange*”. When testing failures occur (*TestFailures*), a transition from state #9 to #10 is triggered, and once the adjustments for the change implementation are performed and no test failures occur (*!TestFailures*), the change being processed returns to state #9. In this case, the variable "*TestFailures*" changes from true to false indefinitely. This can be graphically observed by unfolding the cycle between state #9 and state #10 in a computation tree as shown in Figure 14.

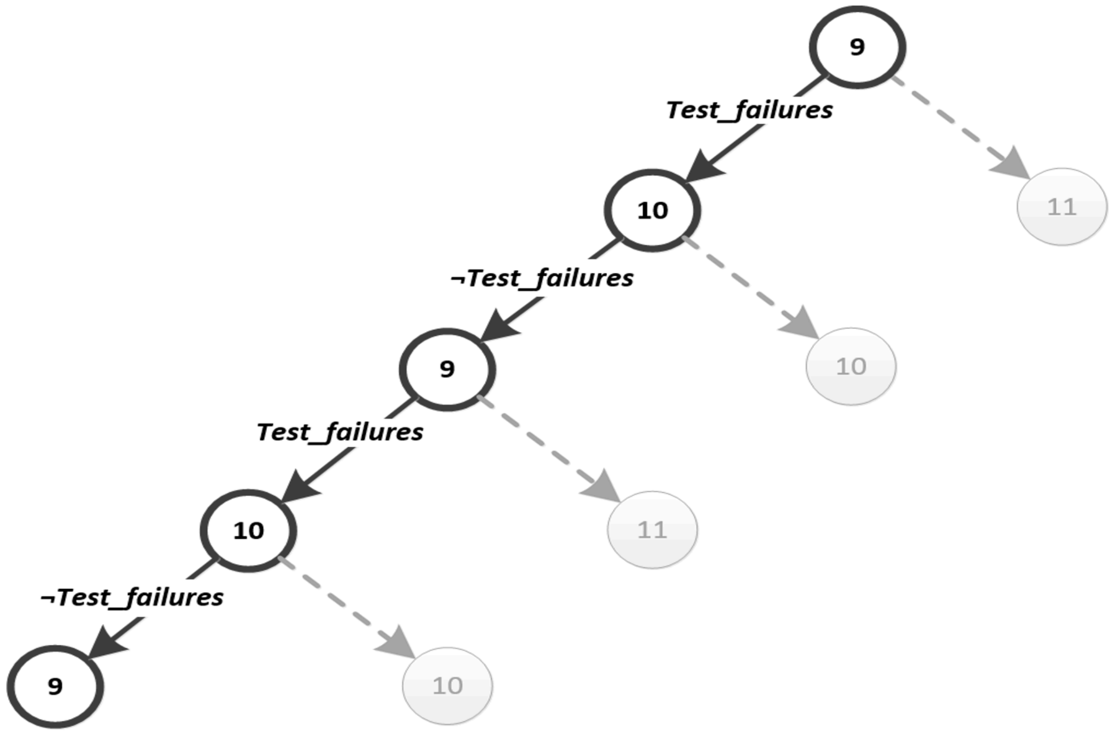


Figure 14. Computation tree for a cyclic path (states #9 and #10)

The specification of this property is then stated as follows:

```
SPEC AG  (((State = State_9_ImplementedChange) & !TestFailures) -> EX (
(State = State_10_FailedTestsOfChange) & TestFailures) &
((State = State_10_FailedTestsOfChange) & TestFailures) -> EX (
(State = State_9_ImplementedChange) & !TestFailures))
```

This formula states that there is a way, after leaving state #9 for state #10, to go back to state #9 in the following step. As a result of a NuSMV run, this property has been proven true.

4.6. False properties and counterexamples

The NuSMV model checking tool is able to navigate the model in its entirety and check for compliance with all of our desired specifications. During verification of the model, the automated tool will indicate whether each property is true, or it will generate a counterexample for each property that did not comply.

The following example will show how NuSMV generates counterexamples to show the violation of a desired property, and how this allows the process to be corrected. It will be shown that counterexamples identify root causes of these problems and therefore help process designers correct their processes.

Consider a process designer writing a specification for the mutually exclusive paths that follow state #11: *"VerifiedChange"*. State #11 is associated with a set of state variables. This set of variables is listed in row 11 of Table 1. Five variables are TRUE in state #11: "*RegisteredChangeRequest*", "*AuthorizationForProcessingChange*", "*AllocationOfResponsibilities_SchedulingActivities*", "*CompletedActivities*", and "*SuccessfulTests*". The process designer writes a specification using these variables from state #11, such that when this profile of variables is satisfied, one of three mutually exclusive states must follow: state #13, #14, or #10.

```
((RegisteredChangeRequest & AuthorizationForProcessingChange &
AllocationOfResponsibilities_SchedulingActivities & CompletedActivities &
SuccessfulTests) -> AX (
(  (State = State_13_RejectedChangeImplementation) &
 !(State = State_14_ReleasedChange) &
 !(State = State_10_FailedTestsOfChange ))|
( !(State = State_13_RejectedChangeImplementation) &
  (State = State_14_ReleasedChange) &
 !(State = State_10_FailedTestsOfChange ))|
( !(State = State_13_RejectedChangeImplementation) &
 !(State = State_14_ReleasedChange) &
  (State = State_10_FailedTestsOfChange )) )
```

Next the process designer adds the CTL operator AG to the beginning of the specification. Using AG requires that in every instance when the five specified variables are satisfied, the process must continue to one of the three states #10, #13, or #14. When running the NuSMV tool with this new specification, the following results are obtained:

```
-- specification AG (((((RegisteredChangeRequest & AuthorizationForProcessingChange)
& AllocationOfResponsibilities_SchedulingActivities) & CompletedActivities) &
SuccessfulTests) -> AX (
(((State = State_13_RejectedChangeImplementation &
!(State = State_14_ReleasedChange)) &
!(State = State_10_FailedTestsOfChange)) |
((!(State = State_13_RejectedChangeImplementation) &
State = State_14_ReleasedChange) &
!(State = State_10_FailedTestsOfChange))) |
((!(State = State_13_RejectedChangeImplementation) &
!(State = State_14_ReleasedChange)) &
State = State_10_FailedTestsOfChange)))  is false

-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
State = State_1_IdentifiedNeedForChange
-- All state variables are displayed for the initial state (all variables with value FALSE).
… ...
-> State: 1.7 <-
State = State_14_ReleasedChange
FinalAcceptanceOfChange = TRUE
ItHasGoneThroughState11 = TRUE
-> State: 1.8 <-
State = State_20_ClosedChangeReleasedWithProcessImprovements
NotificationOfReleasedChangeImplWithProcessImprovements = TRUE
ItHasGoneThroughState14 = TRUE
```

Adding the AG operator creates an error. The reason is that the same set of five true variables used to identify state #11 is also associated with other states, but the outcome of state #11 is not an appropriate outcome for these other states. Specifically, the counterexample shows the states #14 and #20. Reviewing the state variables considered in the specification, it is clear that each variable with value true in state #11 is also true in state #14. The specification, as it is written above, confuses state #14 for state #11 and requires that a change move from state #14 to either state #10, #13, or #14, but these are not viable transitions from state #14, and therefore the NuSMV tool produces state #14 as a counterexample to the above specification. A similar error happens in state #20. To correct the specification, we must be able to distinguish state #11 from #14. A variable that makes the difference between states #11 and #14 has not been considered. The variable "*FinalAcceptanceOfChange*" in state #11 has the value false, while in state

#14 it has the value true. Hence, the specification should be corrected by including the variable "*FinalAcceptanceOfChange*" with value false as indicated in state #11. The specification is corrected as follows:

```
AG ((RegisteredChangeRequest & AuthorizationForProcessingChange &
AllocationOfResponsibilities_SchedulingActivities & CompletedActivities &
SuccessfulTests & !FinalAcceptanceOfChange ) -> AX (<same as before>)
```

The corrected expression was successfully verified by the NuSMV tool. With this example it has been shown that when finding counterexamples (which means that the coded model does not comply with the desired properties), it is possible to identify the root causes of problems and to proceed to correct erroneous specifications in the model. These erroneous specifications in the model may reflect errors in the change control process, and thereby the formal model verification process can verify if a project management change control process is well designed.

5. Discussion and conclusions

This paper presents a methodology for formally verifying properties of a change control process. Starting from a process diagram, following the PMI's ICC guidelines, we express it as a state transition diagram, which is then encoded with CTL in the NuSMV formal modelling tool. Then, key properties of the ICC process should be expressed with CTL in the NuSMV tool. When verified with the NuSMV tool, the properties were either declared correct with complete assurance or counterexamples were presented. Thus, once proved, we can be confident that the programmed specifications hold the properties that the process is desired to have, avoiding the risks of getting a process with errors or failures. We consider that the results of automated tests done in NuSMV demonstrate a way to verify crucial properties in IT processes, at least according to the experience of one of the authors at the IT normativity office that he holds.

The contribution of this paper is the proposal of a methodology that applies temporal logic, and more specifically CTL and NuSMV to make a formal verification of a change control process, represented as a state automaton model. To the best of our knowledge, this has not been done before. Further, this model was programmed in the NuSMV to automatically verify desirable properties of effective process, such as *reachability*, *liveness*, *compliance* with prerequisites, among others.

The process tested here is based on the framework of the Project Management Institute's *Project Management Body of Knowledge* [PMI, 17]. In the review of literature sources, no similar works were found applying this technique to process change models based on the PMI or a related framework. We conclude that formal modeling of change control processes makes it possible to prove if a change control process has or does not have some desired properties. The Integrated Change Control model considered in this paper is a simple one that expresses a change control process for projects according to the PMI's PMBOK concepts, and it is complemented by the activities considered by the Construction Industry Institute. This is a limited model as it does not handle simultaneous changes or non-independent changes. Therefore, the model would not be able to identify possible relationships or dependencies between simultaneous changes. Another limitation is that the model does not include bounds on repetitive sequence cycles (e.g. cycles between state #9: "*ImplementedChange*" and state #10: "*FailedTestsOfChange*"), so the model would not be adequate for handling cases that exceed a reasonable number of cycle repetitions. Further, the model proposed in this paper does not consider changes with varying levels of urgency.

5.1. Future Work

Improvements or additions to be considered in future work would tend to overcome the limitations described above. Also, concerning the actual deployment of the presented method in actual businesses, we think it is not just a matter of getting permission; indeed, the change control process modeled here should be implemented in software, which is, of course, a major endeavor. As presented, this work is just a proof-of-concept, and in order to be widely applicable, we would have to adapt the method to a wide range of applications, as well as develop software for providing automated support for the method application.

References

- [Almeida, 11] Almeida, J.B., Frade, M.J., Pinto, J.S., de Sousa, S.M., Rigorous software development: an introduction to program verification, Springer, 2011.
- [Anderson, 98] Anderson, S.O., Bloomfield, R.E., Cleland, G.L.. Guidance on the use of Formal Methods in the Development and Assurance of High Integrity Industrial Computer Systems. Parts I and II. European Workshop on Industrial Computer Systems (EWICS), 1998.
- [Awad, 08] Awad, A., Decker, G., & Weske, M.. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In Business Process Management (Vol. 5240, pp. 326–341). Berlin, Heidelberg:

Springer Berlin Heidelberg, 2008.

- [Bérard, 01] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen P., Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001.
- [Bi, 04] Bi, H. H. and Zhao, J. L., Applying propositional logic to workflow verification. Information Technology and Management, 5:293–318, 2004.
- [Bjørner, 06] Bjørner, D.: Software Engineering 2: Specification of Systems and Languages (Texts in Theoretical Computer Science. An EATCS Series) Springer, Berlin, 2006.
- [Bjørner, 14] Bjørner, D., Havelund, K., 40 years of formal methods, in "FM 2014: Formal Methods", 42-61, Springer, 2014.
- [Bjørner, 70] Bjørner, D.: Flowchart-Machines. BIT 10(4) pp 415-442, 1970.
- [Burstall, 69] Burstall, R., Landin, P., Programs and their Proofs: An Algebraic Approach. In Machine Intelligence 4. Edinburgh University Press, 1969.
- [Cassandras, 09] Cassandras, Ch.G., Lafortune, S., Introduction to Discrete Event Systems. Springer Science & Business Media, 2009.
- [Chellas, 80] Chellas, B., Modal Logic: An Introduction. Cambridge University Press, 1980.
- [Cimatti, 00] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M., NuSMV: A new symbolic Model Checker. International Journal on Software Tools for Technology Transfer 2(4), 410-425, Springer-Verlag, 2000.
- [Clarke, 96] Clarke, E.M., Wing, M.J., Formal methods: State of the art and future directions. ACM Comput. Surv. 28(4), 626–643, 1996.
- [Fisher, 11] Fisher, M., An Introduction to Practical Formal Methods Using Temporal Logic, John Wiley & Sons, 2011.
- [Groefsema, 13] Groefsema, H., & Bucur, D., A survey of formal business process verification: From soundness to variability. In Proceedings of the Third International Symposium on Business Modeling and Software Design, pp. 198-203, 2013.
- [Harel, 98] Harel, D., & Politi, M., Modeling reactive systems with statecharts: the STATEMATE approach, McGraw-Hill, 1998.
- [Ibbs, 01] Ibbs, W. C., Wong, C. K., & Kwak, Y. H.: Project Change Management System. Journal of Management in Engineering, 159-165, 2001.
- [Kelly, 95] Kelly, J.C., Lead, T., Kemp, K., & Fairmont, W.V., Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Vol 1. NASA report (NASA-GB-002-95), 1995.

- [Kerzner, 05] Kerzner, H., Project Management: A Systems Approach to Planning, Scheduling, and Controlling (9 edition). Wiley, 2005.
- [Lamport, 83] Lamport, L., What good is temporal logic?, IFIP congress, 657-668, 1983.
- [Larson, 11] Larson, E. W., & Gray, C., Project Management: The Managerial Process (Fifth edition). New York, NY: McGraw-Hill/Irwin, 2011.
- [Lewis, 05] Lewis, J., Project Planning, Scheduling & Control, 4E: A Hands-On Guide to Bringing Projects in on Time and on Budget (4 edition). New York: McGraw-Hill, 2005.
- [Monin, 03] Monin, J., Understanding Formal Methods, Springer, 2003.
- [Phillips, 11] Phillips, J. J., Bothell, T. W., & Snead, G. L., The Project Management Scorecard (1 edition). Amsterdam; Boston: Routledge, 2011.
- [PMI, 17] PMI, A Guide to the Project Management Body of Knowledge (PMBOK Guide) - Sixth Edition. Project Management Institute Inc., Pennsylvania, USA, 2017.
- [Rance, 11] Rance, S., ITIL Service Transition 2011 Edition. London: The Stationery Office, 2011.
- [Rozier, 10] Rozier, K., Linear Temporal Logic Symbolic Model Checking. Computer Science Review 5(2), 163-203, Elsevier, 2010.
- [Schwalbe, 09] Schwalbe, K., Information Technology Project Management (6 edition). Boston, MA: Cengage Learning, 2009.
- [Stackpole, 09] Stackpole, C. S., A Project Manager's Book of Forms: A Companion to the PMBOK Guide (1 edition). Hoboken, N.J.; Newtown Square, PA: Wiley, 2009.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.