ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ&
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Ηλεκτρικών Βιομηχανικών Διατάξεων και
Συστημάτων Αποφάσεων

Διπλωματική Εργασία

ΑΝΑΛΥΣΗ ΚΑΙ ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗΣ ΜΕ ΧΡΟΝΙΣΜΕΝΑ ΑΥΤΟΜΑΤΑ

Χρυσόστομος Γιαννάκος

Επιβλέπων: Καθ. Δημήτριος Ασκούνης

Συνεπιβλέπων: Δρ. Δημήτριος Πανόπουλος (ΕΔΙΠ)

ΑΘΗΝΑ 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ&
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Ηλεκτρικών Βιομηχανικών Διατάξεων και
Συστημάτων Αποφάσεων

Διπλωματική Εργασία

ΑΝΑΛΥΣΗ ΚΑΙ ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗΣ ΜΕ ΧΡΟΝΙΣΜΕΝΑ ΑΥΤΟΜΑΤΑ

Χρυσόστομος Γιαννάκος

Επιβλέπων: Καθ. Δημήτριος Ασκούνης

Συνεπιβλέπων: Δρ. Δημήτριος Πανόπουλος (ΕΔΙΠ)

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26/10/2020

- - -

ΑΘΗΝΑ 2020

ΠΡΟΛΟΓΟΣ

Η παρούσα διπλωματική εργασία εκπονήθηκε στο τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου στο πλαίσιο των δραστηριοτήτων του εργαστηρίου Συστημάτων Αποφάσεων του τομέα Ηλεκτρικών Βιομηχανικών Διατάξεων και Συστημάτων Αποφάσεων. Η διπλωματική εργασία ολοκληρώθηκε τον Οκτώβρη του 2020.

Αντικείμενο μελέτης της διπλωματικής αυτής εργασίας αποτέλεσε η ανάλυση και μοντελοποίηση αλγορίθμων χρονοδρομολόγησης σε διαφορετικά προβλήματα με τη χρήση χρονισμένων αυτομάτων.

Στο σημείο αυτό αισθάνομαι την ανάγκη να εκφράσω τις ειλικρινείς και θερμές ευχαριστίες μου σε όσους συνέβαλλαν στην ολοκλήρωση αυτής της προσπάθειας.

Το μεγαλύτερο «ευχαριστώ» είναι στους γονείς μου, που στάθηκαν δίπλα μου υπό όλες τις συνθήκες, εύκολες και δύσκολες, αλλά και για την ατέλειωτη υποστήριξή τους σε όλες μου τις επιλογές. Επιπλέον θέλω να ευχαριστήσω τους φίλους μου που για όλο αυτό τον καιρό στάθηκαν και αυτοί δίπλα μου.

Τέλος, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή της εργασίας Καθ. Δ. Ασκούνη καθώς και τον συνεπιβλέποντα Δρ. Δ. Πανόπουλο για τη συνεχή καθοδήγηση, την αμέριστη υποστήριξη, τις ουσιώδεις συμβουλές καθώς και την εμπιστοσύνη που μου έδειξαν σε όλο αυτό το διάστημα.

Χρυσόστομος Γιαννάκος,

Οκτώβριος 2020

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΕΡΙΛΗΨΗ

Προβλήματα χρονοδρομολόγησης απαντώνται σε πολλούς κλάδους και ένα εύρος περιπτώσεων, βρίσκοντας εφαρμογές σε πεδία που εκτείνονται από τον προγραμματισμό παραγωγής μέχρι την κατανομή των εργασιών ενός υπολογιστικού συστήματος. Άξονα αυτών αποτελεί η βέλτιστη αξιοποίηση μιας σειράς περιορισμένων πόρων (resources) στους οποίους βασίζεται η λειτουργία του εκάστοτε συστήματος προκειμένου να εκτελεστούν συγκεκριμένες εφαρμογές.

Αν αντιμετωπίσουμε τέτοιες εφαρμογές ως ένα άθροισμα αλληλεξαρτώμενων δραστηριοτήτων (interdependent tasks), θα παρατηρήσουμε πως προκειμένου να εξασφαλίσουμε την αποτελεσματικότερη και «ομαλότερη» εκτέλεσή τους παράλληλα με την βέλτιστη αξιοποίηση των πόρων μας, εφαρμόζονται ορισμένες αρχές χρονοδρομολόγης που κατευθύνουν την κάθε αρμοδιότητα της εφαρμογής , με σκοπό την βέλτιστη αυτή απόδοση. Βέβαια, για κάποιες φυσικές εφαρμογές που τίθενται προς έλεγχο, είναι πολύ πιθανό να απαιτούνται προθεσμίες χρονισμού (timing deadlines) είτε για ολόκληρη την εφαρμογή (καθολικά) είτε για κάθε interdepend task ξεχωριστά. Η πρόκληση λοιπόν έγκειται στη σωστή εφαρμογή των scheduling principles ώστε να εξασφαλίζουν ότι πληρούνται όλες οι προθεσμίες χρονισμού της κάθε διεργασίας προς ανάλυση (schedulability analysis).

Σκοπός λοιπόν αυτής της εργασίας αποτελεί αρχικά η μοντελοποίηση τέτοιων περιπτώσεων χρονοδρομολόγησης σε κατάλληλο simulation tool. Η μοντελοποίηση των διαφόρων scheduling scenarios πραγματοποιήθηκε στο UPPAAL (4.15) [12,24] , το οποίο αποτελεί ένα εργαλείο για modelling, validation και verification [4,21] σε συστήματα πραγματικού χρόνου με την χρήση χρονισμένων αυτομάτων [κεφ3]. Συγκεκριμένα στα μοντέλα συμπεριλήφθηκαν ποικίλα scheduling policies [κεφ1], πληθώρα χαρακτηριστικών και εξαρτήσεων για το κάθε interdependent task, δυνατότητα ανάθεσης συγκεκριμένων resource σε συγκεκριμένα tasks (bus, processor) αλλά και δυνατότητα διακοπτόμενης χρονοδρομολόγησης.

Στη συνέχεια επήλθε πειραματισμός και μελέτη πάνω στα γενικά μοντέλα μεταβάλλοντας τις αρχικές τιμές των παραμέτρων, ώστε να προκύψουν ξεχωριστά και διαφορετικά σενάρια χρονοδρομολόγησης των οποίων η εγκυρότητα ελέγχθηκε μέσω του verifier του UPPAAL [12,24]. Τα συμπεράσματα αλλά και μια πιθανή μελλοντική εργασία θα παρατεθούν στο τέλος.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: χρονοδρομολόγηση, διεργασία, μοντελοποιήση, αλληλοεξάρτηση διεργασιών, πόροι, προθεσμίες χρονισμού, μοντελοποιήση, επαλήθευση, επικύρωση

ABSTRACT

Scheduling problems nowadays are encountered in a wide range of cases and in various branches of occupation, while being applicable in fields such as production scheduling or even task distribution of computer systems. Axis of these is the optimal use of a series of limited resources on which the operation of each system is based in order to run specific applications.

If we treat such applications as a sum of interdependent tasks, it is easily observable that in order to ensure their most efficient and smooth execution, along with the optimal utilization of our resources, various timing principles must be applied in order to guide the competence of each implementation of applications, with an ultimate goal being performance optimization. However, certain physical applications that are about to be controlled may require timing deadlines for either the entire application, as a whole, or for each interdependent task separately. The final challenge constitutes the correct appliance of scheduling principles, to ensure that all scheduling timelines of each process are met.

Initial goal of this thesis is the modelling of scheduling scenarios in an appropriate simulation tool. Although there are many alternative methods, the implementation of scheduling scenarios took place in UPPAAL (4.15) [12,24], which is a fine modelling, validation and verification [4,21] tool for Real-Time Systems using Timed Automata [Chapter 3]. Notably, the models included a variety of scheduling policies [Chapter 1], a variety of features and dependencies for each interdependent task, the ability to assign specific resources to specific tasks (bus, processor) and the possibility of pre-emptive and non-pre-emptive scheduling.

Subsequently, we performed many case studies and experiments on the generic models by alternating and substituting the intial values of certain parameters, in order to obtain separate and unalike scheduling scenarios, whose validity was checked through UPPAAL verifier [12,24]. The conclusions as well as a possible future work of this thesis will be presented at the end.

KEYWORDS: scheduling, process, modelling, interdependent tasks, resources, timing deadlines, validation, verification

ΔΟΜΗ

Η δομή που εφαρμόστηκε για την οργάνωση και εκπόνηση της παρούσας διπλωματικής εργασίας έχει ως εξής:

Στο πρώτο κεφάλαιο γίνεται ορισμός του θεωρητικού υποβάθρου της έννοιας της χρονοδρομολόγησης. Αρχικά , παρουσιάζεται ο ορισμός της διεργασίας καθώς και η εκτενής ανάλυση και περιγραφή της κάθε δυνατής κατάστασης που μπορεί να μεταβεί. Επόμενο βήμα αποτελεί η παρουσίαση των χαρακτηριστικών μίας διεργασίας που εμείς έχουμε ορίσει άξια προς μελέτη για τα πειράματά μας ενώ ταυτόχρονα εξηγούμε τη χρηστικότητα αυτών των χαρακτηριστικών. Στη συνέχεια του κεφαλαίου εισάγουμε την έννοια της αλληλεξάρτησης διεργασιών, δηλαδή πως η εκτέλεση μίας διεργασίας επηρεάζεται από την ύπαρξη εξαρτήσεων με μία διαφορετική διεργασία, αλλά και παρουσιάζουμε την έννοια του πόρου καθώς η συμβολή τους είναι καταλυτική για την εκτέλεση των δρομολογημένων διεργασιών. Τέλος, παρουσιάζουμε τους διάφορους αλγορίθμους χρονοδρομολόγησης προς εξέταση συνοδευόμενους με ένα παράδειγμα για τον καθένα για την ορθή επεξήγησή τους. Οι αλγόριθμοι που παρουσιάζονται είναι οι FIFO, SJF+SRTF, FPS, RRκαι EDFμε την παραπάνω ακριβώς σειρά.

Στο δεύτερο κεφάλαιο εισαγάγουμε το μαθηματικό υπόβαθρο που χρειάστηκε για την μοντελοποίηση των προβλημάτων. Αρχικά δίνουμε έναν εύρη ορισμό ενός χρονισμένου αυτομάτου καθώς και μια συνοπτική περιγραφή των εφαρμογών του. Η επόμενη υποενότητα παρέχει έναν μαθηματικό ορισμό του γράφου αλληλεξαρτήσεων, ένα πολύ σημαντικό «σχηματικό» εργαλείο για την αλληλεπίδραση και εκτέλεση των διεργασιών σε προβλήματα χρονοδρομολόγησης. Στο τέλος του κεφαλαίου δίνονται τρείς ακριβείς μαθηματικοί ορισμοί , του χρονισμένου αυτομάτου ,της εκτίμησης χρόνου αλλά και της συνύπαρξης χρονισμένων αυτομάτων και διεργασιών.

Το τρίτο κεφάλαιο παρέχει μια εισαγωγή στο εργαλείο μοντελοποίησης του προβλήματος αλλά και στην εφαρμογή των ήδη ορισμένων θεωρητικών εννοιών των προηγούμενων κεφαλαίων. Πιο συγκεκριμένα στην αρχή του κεφαλαίου παρουσιάζονται οι μέθοδοι ελέγχου μοντέλου και τυπικής επαλήθευσης ενώ στη συνέχεια παρατίθεται το εργαλείο μοντελοποίησης UPPAAL[12,24] που χρησιμοποιήσαμε για την παρούσα διπλωματική. Στο υποκεφάλαιο αυτό περιγράφεται συνοπτικά ο γενικευμένος τρόπος λειτουργίας του ενώ παρουσιάζονται κάποια από τα κύρια μέρη του, η γλώσσα περιγραφής , τον προσομοιωτή και τον έλεγχο μοντέλου. Έπειτα εισαγάγουμε τις έννοιες των χρονισμένων αυτομάτων στη λειτουργία του UPPAAL, καθώς η μοντελοποίηση πραγματοποιείται μέσω αυτών, ενώ στη συνέχεια παραθέτουμε τα συστατικά και τα βασικά χαρακτηριστικά του εργαλείου που χρησιμοποιήσαμε για την μοντελοποίηση των προβλημάτων μας. Τέλος ο τρόπος που το εργαλείο ελέγχει και επαληθεύει τα μοντέλα που έχουν δημιουργηθεί.

Στο τέταρτο κεφάλαιο παρουσιάζεται η συλλογιστική πορεία της μοντελοποίησης δηλαδή η περιγραφή του κάθε μοντέλου που υλοποιήθηκε και εφαρμόστηκε. Αρχικά αναλύεται μια πρωταρχική απλοϊκή ιδέα μοντελοποίησης της διεργασίας και των πόρων που αποτέλεσε βάση για τον σχεδιασμό και την επέκταση τους. Εφόσον παραθέσουμε τις δομές δεδομένων που δημιουργήσαμε, τις εφαρμόζουμε σαν εργαλείο για τη δημιουργία των μοντέλων της διεργασίας, του γράφου εξαρτήσεων μεταξύ των διεργασιών αλλά και των

πόρων. Εκτός των παραπάνω μοντέλων που αποτελούν την βάση για το κάθε πρόβλημα που θα υλοποιήσουμε, έχουμε δημιουργήσει και μοντέλα $3^{ων}$ αλγορίθμων χρονοδρομολόγησης, του FIFO, του FPS, και του EDF, τα οποία περιγράφουμε αναλυτικά τόσο ως προς την κατασκευή αλλα και ως προς τη λειτουργία τους. Τα δύο τελευταία υποκεφάλαια αφορούν τον τρόπο της πειραματικής εγκατάστασης, δηλαδή την εισαγωγή επιθυμητών τιμών στα μοντέλα μέσω των δομών που δημιουργήσαμε, αλλά και την διαδικασία ελέγχου του αποτελέσματος των πειραμάτων με τη βοήθεια του εργαλείου μοντελοποίησης UPPAAL.

Το πέμπτο κεφάλαιο περιλαμβάνει την επεξήγηση όλων των πειραμάτων που εκτελέσαμε μέσω των μοντέλων που δημιουργήθηκαν. Το πρώτο σκέλος αφορά παραδείγματα που περιεγράφηκαν στο κεφάλαιο 1 και το δεύτερο αφορά πιο σύνθετα παραδείγματα.

Τέλος το έκτο κεφάλαιο παρουσιάζει ολοκληρωμένα τα συμπεράσματα της διπλωματικής αλλά και τις προοπτικές επέκτασης που θα μπορούσαν να πραγματοποιηθούν.

1 ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗΣ

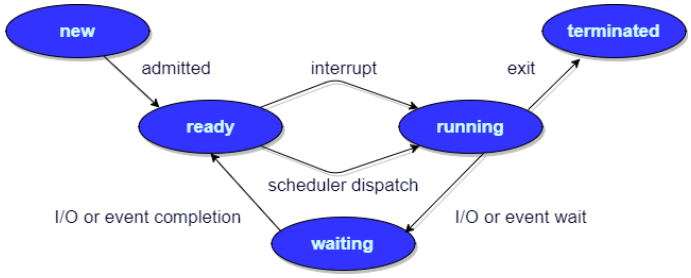
1.1 Εισαγωγή

Η έννοια της χρονοδρομολόγησης είναι άρρηκτα συνδεδεμένη με τις έννοιες πόροι και διεργασίες (resource and tasks). Σε γενικότερο πλαίσιο η χρονοδρομολόγηση, αποτελεί την μοντελοποίηση του Διαμοιρασμού χρόνου μέσω του χρονοδρομολογητή, ενώ παράλληλα οι διεργασίες έχουν την (ψευδ-)αίσθηση ότι χρησιμοποιούν αποκλειστικά τον επεξεργαστή, δηλαδή καταναλώνουν πόρους-resources, για συγκεκριμένο χρονικό διάστημα. Ο χρονοδρομολογητής αναλαμβάνει την επιλογή της διεργασίας που θα χρησιμοποιήσει τον επεξεργαστή αλλά και την αλλαγή της (διεργασίας) ώστε να τον χρησιμοποιήσει κάποια άλλη (context switch). Στόχος του χρονοδρομολογητή λοιπόν αποτελεί η ανάθεση των διεργασιών στον επεξεργαστή ώστε να μην χάνεται χρόνος εν αναμονή για την ολοκλήρωση λειτουργειών αλλά και ο διαμοιρασμός του επεξεργαστή στις διεργασίες ώστε η απόκριση του συστήματος να είναι επιθυμητή. Η προσθήκη περαιτέρω περιορισμών αυξάνοντας την περιπλοκότητα του προβλήματος ορίζει ένα «πρόβλημα χρονοδρομολόγησης».

Προκείμενου να έχουμε βέλτιστη παραγωγικότητα και αποδοτικότητα σε ένα σύμπλεγμα εργασιών κρίνεται αναγκαία η επιλογή ενός αλγορίθμου χρονοδρομολόγησης έτσι ώστε να (χρονο-)προγραμματίσει την κάθε εργασία (job) στην ουρά εργασιών. Το παρακάτω τμήμα της μελέτης παρέχει τον εύρη ορισμό της διεργασίας (job/process),την θεμελίωση κάποιων βασικών εννοιών για την χρονοδρομολόγηση, την διαφορά μεταξύ των αλγορίθμων χρονοδρομολόγησης, την αξιολόγηση του κάθε αλγορίθμου αλλά και κάποια γνωστά θετικά και αρνητικά του κάθε αλγορίθμου.

1.2 Ορισμός Διεργασίας

Σύμφωνα με το βιβλίο Operating System Concepts with Java μία διεργασία ορίζεται ως ένα πρόγραμμα υπό εκτέλεση (από τον επεξεργαστή) ή πιο συγκεκριμένα όταν ένα εκτελέσιμο πρόγραμμα φορτώνεται στη μνήμη, τότε αποτελεί μια διεργασία. Η διεργασία περιέχει πολλές πληροφορίες για την κατάσταση που βρίσκεται ένα πρόγραμμα υπό εκτέλεση, όπως τον program counter ή άλλες μεταβλητές όπως οι δείκτες του σωρού η της στοίβας (heap, stack)και τα τμήματα δεδομένων ή τις μεταβλητές (data segment, variables) [28]. Κατά τη διάρκεια εκτέλεσης μια διεργασία μπορεί να βρίσκεται σε πολλές καταστάσεις όπως φαίνεται παρακάτω:



Καταστάσεις Διεργασίας

Κάθε κατάσταση της κάθε διεργασίας είναι ξεχωριστή. Όταν μία διεργασία γεννιέται τότε περνά στην κατάσταση **“new”** καθώς παράλληλα σηματοδοτεί πως είναι έτοιμη να περάσει στην ουρά έτοιμων διεργασιών, ώστε στην συνέχεια να της παραχωρηθούν πόροι (resources) για να μπορέσει να εκτελέσει τις εντολές της. Μία διεργασία βρίσκεται στην κατάσταση **“running”** ακριβώς τη στιγμή που ξεκινά να εκτελείται στη CPU (Central Processing Unit). Μπορεί να βρεθεί σε κατάσταση **“waiting”** μόνο όταν συμβεί κάποιο interrupt, και θα μείνει σε αυτή την κατάσταση περιμένοντας να ολοκληρωθεί κάποιο συμβαν ,όπως Input/Output completion, καθώς εκείνη τη στιγμή δεν είναι υποψήφια για εκτέλεση στη CPU. Μία διεργασία βρίσκεται σε κατάσταση **“ready”** όταν είναι έτοιμη προς εκτέλεση ή είναι έτοιμη να γίνει resource allocation στη CPU. Τέλος η κατάσταση **“terminated”** υποδηλώνει όταν η εκτέλεσή της έχει ολοκληρωθεί [19,28].

1.3 Χρονοδρομολόγηση

Ένα πρόβλημα χρονοδρομολόγησης αποτελείται πάντα από πεπερασμένο αριθμό από tasks το οποίο θα ορίσουμε ως $T = t_1, t_2, \ldots, t_n$. Κάθε τέτοιο task, για την παρούσα εργασία, έχει τα παρακάτω χαρακτηριστικά [28]:

- INITIAL_OFFSET: $T \rightarrow \mathbb{N}$
  Αριθμός μονάδων χρόνου πριν την πρώτη αποστολή του task

- BCET: $T \rightarrow \mathbb{N}_{\geq 0}$
  Βέλτιστος χρόνος εκτέλεσης ενός task

- WCET: $T \rightarrow \mathbb{N}_{\geq 0}$
  Χείριστος χρόνος εκτέλεσης ενός task

- MIN_PERIOD: $T \rightarrow \mathbb{N}$
  Ελάχιστος χρόνος μεταξύ αποστολής του ιδίου task (μόνο σε periodic tasks)

- MAX_PERIOD: $T \rightarrow \mathbb{N}$
  Μέγιστος χρόνος μεταξύ αποστολής του ίδιου task (μόνο σε periodic tasks)

- OFFSET: Αριθμός μονάδων χρόνου σε κάθε περίοδο πριν το task αποσταλεί

- DEADLINE: $T \rightarrow N_{\geq 0}$
  Αριθμός μονάδων χρόνου στις οποίες το κάθε task θα επιβάλλεται να έχει τελειώσει το execution του

- PRIORITY: Αριθμός προτεραιότητας του task (σε σύγκριση με άλλα)

Για τα παραπάνω χαρακτηρίστηκα είναι προφανές ότι ισχύει η εξής ανισότητα:

$BCET(t) \leq WCET(t) \leq DEADLINE(t) \leq MIN\_PERIOD(t) \leq MAX\_PERIOD(t)$

Που τα MIN_PERIOD και MAX_PERIOD αγνοούνται σε περίπτωση μη περιοδικών tasks.

Πιο αναλυτικά, ένα task t_i δεν μπορεί να εκτελεσθεί για τις πρώτες $OFFSET(t_i)$ μονάδες χρόνου, ενώ στην συνέχεια θα εκτελείται ακριβώς μία φόρα κάθε $PERIOD(t_i)$. Κάθε τέτοια εκτέλεση έχει εύρος τιμών χρόνου ολοκλήρωσης $[BCET(t_i),WCET(t_i)]$ (non-deterministic execution time). Η μη ντετερμινιστική συμπεριφορά του χρόνου εκτέλεσης προκύπτει λόγω της ιδιομορφίας των tasks, καθώς η εκτέλεσή τους μπορεί να εξαρτάται απο την εκτέλεση πολύπλοκων λειτουργιών όπως πχ βρόχους , conditionals.

Θα λέμε πως ένα task *t* βρίσκεται στο “ready state” και είναι έτοιμος προς εκτέλεση στο χρόνο τ αν και μόνον αν:

1. $\tau \geq INITIAL\_OFFSET(t)$

2. Το*t*δεν έχει εκτελεσθεί στην περίοδο που επιβάλλει ο χρόνος τ

3. Πληρούνται όλες οι προϋποθέσεις αλληλεξαρτήσεων για να εκτελεσθεί το t (task dependencies complete)

1.4 Αλληλεξάρτηση Διεργασιών (Task Dependencies):

Πολύ συχνά ένα task δεν έχει τη δυνατότητα να φτάσει στο “ready state” καθώς κατά την εκτέλεση του χρειάζεται δεδομένα από ένα δεύτερο task το οποίο όμως δεν έχει εκτελεσθεί. Τέτοιες εξαρτήσεις μεταξύ των στοιχείων ενός συνόλου $T = t_1 , t_2, \ldots, t_n$ μοντελοποιούνται μέσο ενός άκυκλου γραφήματος (*V , E)* που τα tasks είναι οι κόμβοι του διαγράμματος (*V = T)*και οι αλληλοεξαρτήσεις των tasks έιναι οι αποστάσεις μεταξύ των κόμβων (task graph). Με άλλα λόγια η απόσταση (t_i , t_j) € *E* από το task t_i στο task t_j, υποδηλώνει ότι το task t_j δεν μπορεί να ξεκινήσει την εκτέλεση του πριν ολοκληρωθεί η εκτέλεση του task t_i.

1.5 Πόροι (Resources):

Όπως προαναφέρθηκε οι πόροι είναι τα στοιχεία που φέρνουν εις πέρας την εκτέλεση των διεργασιών (πχ επεξεργαστής). Κάθε τέτοιος πόρος χρησιμοποιεί ένα χρονοδρομολογητή για να καθορίσει ποιο task θα εκτελεστεί και σε ποια χρονικά στιγμή. Οι πορόι ή αλλιώς resources είναι περιορισμενοι καθώς επιτρέπουν την εκτέλεση ενός και μόνο task/διεργασίας για την κάθε χρονική στιγμή. Σε ένα σύστημα πραγματικού χρόνου οι πόροι δρουν σαν επεξεργαστές, δίαυλοι επικοινωνίας (communication busses) κλπ. που σε συνδυασμό με τα άκυκλα γραφήματα διεργασιών, προσομοιάζουν με μεγάλη συνέπεια πολύπλοκα και πολυσύνθετα συστήματα με αλληλεξαρτήσεις διεργασιών. Αν δηλαδή θέλουμε να μοντελοποιήσουμε δύο διεργασίες t_iκαι t_j με αλληλεξάρτηση $t_i \rightarrow t_j$ αλλα αυτές οι δύο διεργασίες εκτελούνται σε ***διαφορετικούς*** επεξεργαστές και το t_j χρειάζεται το αποτέλεσμα εκτέλεσης του t_iτότε μέσο ενός νέου resource και μίας βοηθητικής (ιδεατής) κατάστασης που θα λειτουργήσουν σαν δίαυλος επικοινωνίας μεταξύ των διεργασιών, πραγματοποιείται η μοντελοποίηση.

1.6 Αλγόριθμοι Χρονοδρομολόγησης (Scheduling Policies):

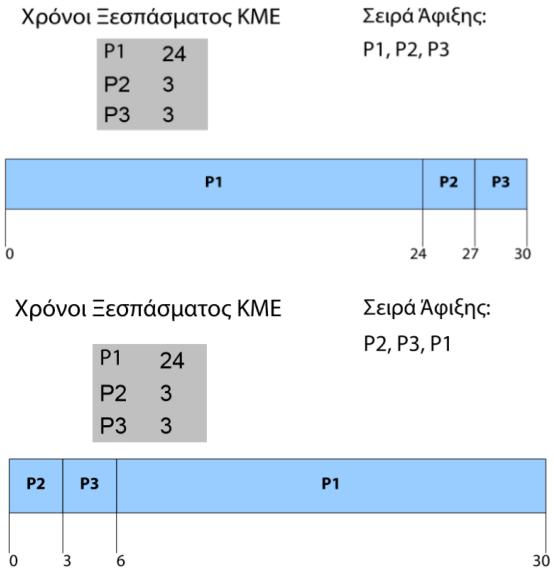
Η θεμελίωση της χρονοδρομολόγησης είναι συνυφασμένη με τους αλγόριθμους χρονοδρομολόγησης. Αυτοί οι αλγόριθμοι είναι υπέυθυνοι για τον ορθό διαμοιρασμό των πόρων στις διεργασίες οταν αυτές τους ζητούν ταυτόχρονα και ασύγχρονα [11]. Οι αλγόριθμοι χρονοδρομολόγησης χρησιμοποιούνται σε ένα μεγάλο εύρος εφαρμογών και αντικειμένων όπως routers(για packet traffic) , λειτουργικά συστήματα (CPU timesharing για threads και processes), ενσωματωμένα συστήματα αλλά και εκτυπωτές. Βασικό τους μέλημα αποτελεί η αφενός η ελαχιστοποίηση του φαινομένου της λιμοκτονίας των διεργασιών αλλά και αφετέρου η εξασφάλιση “δικαιοσύνης” στην χρήση των πόρων [28].

Οι αλγόριθμοι χρονοδρομολόγησης έχουν την δυνατότητα να είναι είτε διακοπτόμενοι είτε μη-διακοπτόμενοι. Όταν μία κατάσταση μεταβεί σε διαδικασία αναμονής (ή τερματιστεί) τότε έχουμε μη-διακοπτόμενο αλγόριθμο (non-preemptive). Στους διακοπτόμενους αλγορίθμους χρονοδρομολόγηση μπορεί να συμβεί ακόμα όταν μία διεργασία μεταβαίνει απο το “running”στο “ready” state μέσο μίας διακοπής (interrupt) και όταν μία διεργασία μεταβαίνει από το “waiting” στο “ready” state. Υπάρχει ένας πολύ μεγάλος αριθμός απο αλγόριθμους χρονοδρομολόγησης με ποικιλία στην πολυπλοκότητά τους, ανάλογα με τις απαιτήσεις και τους περιορισμούς του κάθε προβλήματος χρονοδρομολόγησης που καλούμαστε να λύσουμε.

Στην παρούσα διπλωματική εργασία εξετάζουμε και μελετούμε συγκεκριμένους αλγορίθμους που παρατίθενται παρακάτω. Ο κάθε αλγόριθμος που μελετήθηκε, μοντελοποιήθηκε στο εργαλείο UPPAAL (4.15) [κεφ 3].

1.6.1 First-In First-Out (FIFO) αλλιώς First Come, First Serve (FCFS) :

Ο αλγόριθμος FIFO (ή αλλιώς FCFS) είναι μία από τίς απλότερες υλοποιήσεις χρονοδρομολόγισης που υφίστανται. Αποτελεί έναν μη-διακοπτόμενο αλγόριθμο το οποίο, όπως προαναφέρθηκε, σημαίνει ότι οι διεργασίες δεν μπορούν να διακοπούν από τη στιγμή που τους έχει εκχωρηθεί κάποιος πόρος. Με απλά λόγια, τα tasks στο “ready”state προστίθενται σε μία ουρά , με ακριβώς τη σειρά που έγιναν ready. Έτσι το πρώτο ready task θα εισέλθει πρώτο , θα του εκχωρηθούν resource και μόλις τελειώσει η εκτέλεση του το δεύτερο στην ουρά task θα είναι έτοιμο προς εκτέλεση. Ο αλγόριθμος FIFO είναι «δίκαιος» και εύκολα υλοποιήσιμος, πράγμα που σημαίνει ότι δεν έχουμε φαινόμενα λιμοκτονίας στις διεργασίες. Ωστόσο παρακάτω παρατίθεται ένα παράδειγμα FIFO που μας παρουσιάζει , πέραν της ευκολίας του, και τους περιορισμούς που έχει.



Σχήμα 1.6.1 θεωρητικό παράδειγμα FIFO

Στο παράδειγμα αυτό, θεωρούμε πως το σύστημα αποτελείται από έναν επεξεργαστή (Resource) που θα εκχωρηθεί στο κάθε task όταν αυτό κρίνεται απαραίτητο. Ο χρόνος άφιξης των tasks ανήκει στο διάστημα [0,24].

Ο χρόνος αναμονής του συστήματος εξαρτάται εξολοκλήρου από την σειρά άφιξης των tasks. Εδώ η σειρά άφιξης είναι P_1, P_2, P_3, με μέσο χρόνο αναμονής των tasks: P_1 = 0 + P_2 = 24 + P_3 = 27/3 = 17 μονάδες χρόνου. Αν ωστόσο τα tasks φτάσουν με διαφορετική σειρά π.χ. P_2 ,P_3 , P_1 ο μέσος χρόνος αναμονής τους μειώνεται κατά 3.Συμπερασματικά, ο μέσος χρόνος αναμονής των tasksποικίλλει και εξαρτάται από την σειρά άφιξης στην ουρά FIFO. Ένα ακόμα εμφανές πρόβλημα του αλγορίθμου αυτού εμφανίζεται όταν ένα task που απαιτεί μεγάλη υπολογιστική ισχύ (άρα περισσότερα resources, παραπάνω ενδεχομένως από όσα διαθέτει το σύστημα), φτάνει στην κεφαλή της ουράς ενώ παράλληλα πίσω του βρίσκονται tasks με πολύ μικρότερες απαιτήσεις resources για να εκτελεσθούν. Έτσι το πρώτο (μεγάλο) task εμποδίζει τα μικρότερα από το να χρησιμοποιήσουν τα resources και κατ’ επέκταση να εκτελεσθούν, με αποτέλεσμα να μην έχουμε την βέλτιστη αποδοτικότητα στην εκτέλεση και ολοκλήρωση των tasks του προβλήματός μας. Αυτό το πρόβλημα λύνεται

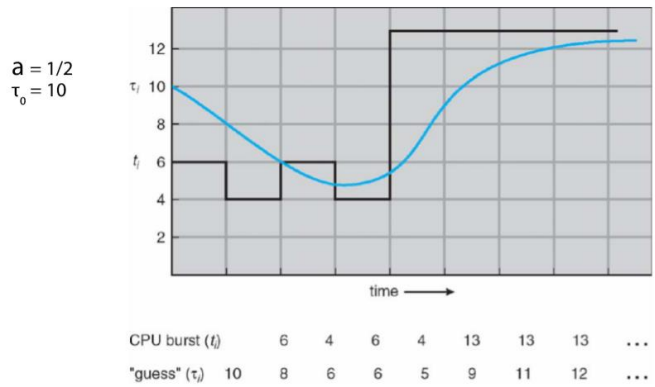
πολύ εύκολα τροποποιώντας τον αλγόριθμο FIFO ή αλλάζοντας πολιτική χρονοδρομολόγησης, έτσι ώστε κάποια tasks να έχουν μεγαλύτερη προτεραιότητα σε σύγκριση με κάποια άλλα [28]. Περισσότερα πάνω σε αυτό το θέμα θα αναλυθούν παρακάτω.

1.6.2 Shortest Job First (SJF):

Ο αλγόριθμος SJF είναι ένας εξίσου απλός αλγόριθμος κατα τον οποίο επιλέγεται το task με το μικρότερο χρόνο ξεσπάσματος στη CPU (burst time). Ο SJF καθίσταται βέλτιστος καθώς επιτυγχάνει ελάχιστο μέσο χρόνο αναμονής, ωστόσο συνήθως το μήκος του επόμενου ξεσπάσματος δεν είναι απαραίτητα γνωστό, με αποτέλεσμα η προσέγγιση να γίνεται με βάση τις προηγούμενες τιμές. Σε περίπτωση που δύο tasks έχουν το ίδιο CPU burst time τότε χρησιμοποιείται ο αλγόριθμος FIFO ή FCFS για αυτά, δηλαδή εκτελείται πρώτο αυτό που έφτασε πρώτο εκ των δύο [28].

Ένα συχνό σημαντικό πρόβλημα του αλγορίθμου ωστόσο εμφανίζεται όταν πολλά “μικρά” tasks (μικρό CPU burst time) εισέρχονται στην ουρά και επιλέγονται πρώτα έναντι ενός “μεγάλου” task, με αποτέλεσμα η εκχώρηση και εκτέλεση του να καθυστερεί αισθητά. Σε μερικές υλοποιήσεις αυτού του σεναρίου το “μεγάλο” task δεν θα φτάσει ποτέ σε σημείο να εκτελεστεί , με αποτέλεσμα να συναντούμε το φαινόμενο της “λιμοκτονίας”., το οποίο είναι συχνό φαινόμενο σε απλοϊκούς αλγορίθμους.

Γενικά , όπως προαναφέρθηκε, δεν είναι γνωστή η διάρκεια του επόμενου ξεσπάσματος κάθε task, με αποτέλεσμα ο αλγόριθμος να εκτελείται προσεγγιστικά. Η προσέγγιση της διάρκειας του επόμενου ξεσπάσματος επιτυγχάνεται με τον εκθετικό μέσο όρο $\tau_{n+1} = \alpha \times t_n + (1-\alpha) \times \tau_n$, όπου τ είναι η πρόβλεψη , t η πραγματική τιμή και α Ε [0,1] το σχετικό βάρος των t και τ , με $\alpha=0$ οι μετρήσεις των πραγματικών τιμών καθυστέρησης δεν επηρεάζουν και με $\alpha=1$ μόνο η τελευταία πραγματική καθυστέρηση μετράει. Παρακάτω παρουσιάζεται ένα διάγραμμα (ως παράδειγμα) για προσέγγιση της διάρκειας του επόμενου ξεσπάσματος μίας διεργασίας με τιμές $\alpha = 1/2$ και τ = 10:

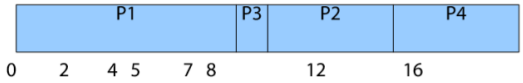


Σχήμα 1.6.2a Διάγραμμα CPU burst

Ο αλγόριθμος SJF είναι συνήθως μη-διακοπτός (non-preemptive), δηλαδή όταν η CPU παραχωρείται σε ένα task πρέπει αυτό να ολοκληρώση τη χρήση της CPU έως ότου

παρθεί νέα απόφαση χρονοδρομολόγησης. Υπάρχουν βέβαια περιπτώσεις που ο αλγόριθμος αυτός υλοποιείται και ως διακοπτός (preemptive), δηλαδή αν εισέλθει στην ουρά έτοιμων διεργασιών ένα νέο task με μικρότερο αναμενόμενο χρόνο εκτέλεσης από τον εναπομείναντα του τρέχων, το τρέχων task θα αντικατασταθεί. Η τελευταία αυτή περίπτωση ονομάζεται SRTF: Shortest Remaining Time First, και είναι μία παραλλαγή του SJF [28,29]. Έστω λοιπόν 4 tasks με τα CPU burst times που φαίνονται στο παρακάτω διάγραμμα τα οποία εισέρχονται στην ουρά μέσα στο χρονικό πλαίσιο [0,24] και χρησιμοποιούν μία CPU για την εκτέλεσή τους. Όπως είναι φανερό τα tasks με τη μικρότερη διάρκεια CPU burst time εκχωρούνται πρώτα από το σύστημα(και επομένως είναι πρώτα στην ουρά) ενώ ο μέσος χρόνος αναμονής του αλγορίθμου SJF είναι αισθητά μικρότερος από αυτόν του FIFO. Ο αλγόριθμος SJF γίνεται ακόμα πιο αποδοτικός αν στο πρόβλημά μας προσθέσουμε επιπλέον συστήματα για τον προσδιορισμό της διάρκειας του κάθε task, καθώς αυτό έχει ως αποτέλεσμα την μείωση του CPU burst time και ως συνέπεια αυτού, την συντομότερη εκχώρηση των tasks στα resources. Παρακάτω παρατίθενται δύο παραδείγματα, ένα με SJF και ένα με SRTF χρονοδρομολόγηση:

Διεργασία	Άφιξη	Διάρκεια
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Μέσος χρόνος αναμονής: (0 + (8-2) + (7-4) + (12-5)) / 4 = 4

Σχήμα 1.6.2b Παράδειγμα 1

Διεργασία	Άφιξη	Διάρκεια
P1	0	7
P2	2	4
P3	4	1
P4	5	4

t	Γεγονός	Διεργασίες	Επιλογή
0-2	Άφιξη P1	(P1,7)	P1
2-4	Άφιξη P2	(P1,5) (P2,4)	P2
4-5	Άφιξη P3	(P1,5) (P2,2) (P3,1)	P3
5-7	Άφιξη P4 / Ολοκλήρωση P3	(P1,5) (P2,2) (P4,4)	P2
7-11	Ολοκλήρωση P2	(P1,5) (P4,4)	P4
11-16	Ολοκλήρωση P4	(P1,5)	P1

Μέσος Χρόνος αναμονής: ((11-2) + (5-4) + (4-4) + (7-5))/4 = 3

Σχήμα 1.6.2c Παράδειγμα 2

1.6.3 Fixed Priority Scheduling (FPS):

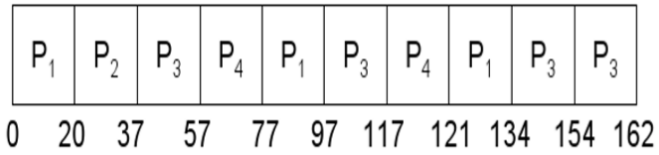
Ο αλγόριθμος αυτός είναι ευρέως χρησιμοποιούμενος σε πολλά συστήματα πραγματικού χρόνου. Στις πιο συνήθεις περιπτώσεις αποτελεί διακοπτόμενο αλγόριθμο (preemptive) ενώ τα tasks αποκτούν ένα περαιτέρω χαρακτηριστικό, το PRIORITY (έγινε αναφορά στο κεφάλαιο ...). Το PRIORITYείναι ένας ακέραιος θετικός αριθμός που διαθέτουν τα task στη δομή τους , με βάση τον οποίο ο χρονοδρομολογητής καλείται να επιλέξει το κατάλληλο task προς εκτέλεση, δηλαδή αυτό με το υψηλότερο PRIORITY. Όσο μεγαλύτερος είναι αυτός ο αριθμός τόσο πιο “αναγκαίο” είναι να εκτελεσθεί το taskμε τον αντίστοιχο αριθμό. Ο αλγόριθμος SJF αποτελεί μια ειδική περίπτωση χρονοδρομολόγησης με προτεραιότητα, όπου ο ακέραιος PRIORITY σε αυτή την περίπτωση είναι ίσος με το μέγεθος του CPU burst time. Ένα πρόβλημα αυτού του αλγορίθμου είναι τα φαινόμενα λιμοκτονίας καθώς tasks με αρκετά χαμηλή προτεραιότητα μπορεί να μην εκτελεσθούν ποτέ, όπως είδαμε και στο παράδειγμα με τον SJF αλγόριθμο που ένα task με μεγάλο CPU burst time μπορεί να μην εκτελεσθεί ποτέ. Ωστόσο τα προβλήματα λιμοκτονίας είναι αντιμετωπίσιμα μέσω μίας διαδικασίας που ονομάζουμε “γήρανση”, κατά την οποία η προτεραιότητα του κάθε task αυξάνεται με την πάροδο του waiting time (και όχι execution time) [10].

1.6.4 Round Robin (RR):

Ο αλγόριθμος χρονοδρομολόγησης Round Robin είναι περίπου ίδιος με τον αλγόριθμο FCFS. Ο χρονοδρομολογητής και σε αυτή την περίπτωση διαθέτει μια ουρά FIFO για τις έτοιμες προς εκτέλεση διεργασίες (ready queue), οι οποίες όμως εδώ εκτελούνται για ένα συγκεκριμένο παράθυρο χρόνου ή αλλιώς time slice. Ουσιαστικά λοιπόν ο αλγόριθμος RR δεν είναι παρά ο αλγόριθμος FCFS αλλά διακοπτόμενος (preemptive). Πιο αναλυτικά, τα tasks λαμβάνουν ένα κλάσμα χρόνου κατά το οποίο η κάθε διεργασία θα μπορεί να εκτελεσθεί καθώς μόνο για αυτό το χρονικό διάστημα εκχωρούνται resources. Με το πέρας αυτού του κβάντου χρόνου η διεργασία (παρόλο που βρίσκεται στην αρχή της ουράς) διακόπτεται και τοποθετείται στο τέλος της ουράς. Η επόμενη διεργασία που επιλέγεται θα είναι αυτή που είναι πλέον πρώτη στην ουρά μετά το τέλος της προηγούμενης διαδικασίας. Αυτό επαναλαμβάνεται αέναα μέχρι να εκτελεσθούν (και να τερματιστούν) όλες οι διεργασίες στην ουρά.

Το παρακάτω παράδειγμα διαθέτει 4 διεργασίες με κβάντο χρόνου 20 για την κάθε μία:

Διεργασία	Ξέσπασμα ΚΜΕ
P1	53
P2	17
P3	68
P4	24



Σχήμα 1.6.4 Παράδειγμα RR

Το παράδειγμα δείχνει πως πρώτα εκτελείται η διεργασία P_1 για 20 μονάδες χρόνου και στη συνέχεια ο χρονοδρομολογητής την διακόπτει έτσι ώστε να εκτελεσθεί η επόμενη, με αποτέλεσμα να μην προλάβει να τελειώσει (εκτελέσθηκε για 20 μονάδες χρόνου ενώ το CPU burst time είναι 53, επομένως χρειάζεται ακόμα 33 μονάδες χρόνου για να τερματιστεί). Η επόμενη διεργασία είναι η P_2 η οποία μάλιστα προλαβαίνει να εκτελεσθεί και να τερματιστεί καθώς το κβάντο χρόνου είναι 20 και αυτή χρειάζεται μόνο 17 μονάδες χρόνου. Έτσι οι διεργασίες εκτελούνται κυκλικά, για το κβάντο χρόνου που τους έχει ορίστεί, μέχρι να τις διακόψει ο χρονοδρομολογητής ή μέχρι να τερματιστούν. Στό τέλος παρατηρούμε πως η μόνη διεργασία που έχει μείνει είναι η P_3 και η εκτέλεση της διακόπτεται και ξαναξεκινά. Αυτό οφείλεται στο ότι δεν υπάρχει καμία διεργασία στην ουρά πέραν της P_3 και επομένως ξαναξεκινά να εκτελείται. (Το μοτίβο Εκτέλεση- Διακοπή-Εκτέλεση στην τελευταία διεργασία θα συνέβαινε μέχρι αυτή να ολοκληρωθεί, καθώς με το που εκπνέει το κβάντο χρόνου ο χρονοδρομολογητής διακόπτει την διεργασία από την εκτέλεση της. Ο ίδιος δεν αναγνωρίζει πως έχει μείνει μία διεργασία στην ουρά)

Ο Αλγόριθμος Round Robin θεωρείται δίκαιος αλγόριθμος καθώς όλες οι διεργασίες(tasks) δαπανούν τον ίδιο χρόνο στο σύστημα. Γενικότερα η κάθε διεργασία έχει μεγάλο waiting time αλλά αυτό εξαρτάται από το κβάντο χρόνου που θα επιλεχθεί (συνήθως 10-100ms). Έτσι για μεγάλα κβάντα χρόνου προσεγγίζουμε τον αλγόριθμο FCFS ενώ για μικρά κβάντα το σύστημα αναλώνεται σε πολλές εναλλαγές.

1.6.5 Earliest Deadline First (EDF):

Ο αλγόριθμος EDF η αλλιώς αλγόριθμος συντομότερης χρονικής προθεσμίας αποτελεί αλγόριθμο χρονοπρογραμματισμού δυναμικής προτεραιότητας που χρησιμοποιείται σε συστήματα πραγματικού χρόνου τοποθετώντας διεργασίες σε μία ουρα προτεραιοτήτων. Σύμφωνα με αυτόν η εκτέλεση και δρομολόγηση των εργασιών αποδίδονται με βάση τη χρονική τους προθεσμία. Έτσι οι εργασίες με την μικρότερη χρονική προθεσμία έχουν μεγαλύτερη προτεραιότητα. Όταν μία διεργασία βρίσκεται σε κατάσταση Ready έτοιμη προς εκτέλεση γίνεται απόπειρα τοποθέτησής της στην ουρά EDF. Σε αυτό το σημείο ο χρονοδρομολογητής επαναϋπολογίζει τους χρόνους προθεσμίας όλων των

διεργασιών στην ουρά (έτοιμες προς εκτέλεση) και τους συγκρίνει με τη χρονική προθεσμία της νέας διεργασίας που εισέρχεται, αναδιοργανώνοντας τη σειρά επεξεργασίας των διεργασιών. Με αυτό τον τρόπο αν η νέα διεργασία έχει υψηλότερη προτεραιότητα συγκριτικά με την υπό εκτέλεση διεργασία τότε γίνεται διακοπή της εκτέλεσής της και εκτελείται η νέα διεργασία, ενώ η εργασία που διακόπηκε θα συνεχιστεί αργότερα. Σε διαφορετική περίπτωση, αν η προτεραιότητα της νέας άφιξης είναι μικρότερη τοποθετείται σε κατάλληλη θέση στην ουρά [19].

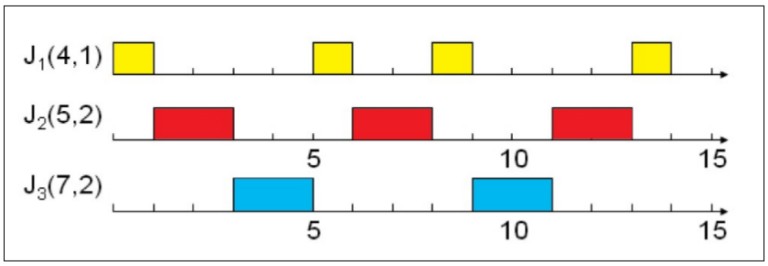
Ένα σύνολο n περιοδικών διεργασιών με χρονικές προθεσμίες ίσες με την περίοδο είναι χρονοπρογραμματίσιμες μέσω του αλγορίθμου EDF αν και μόνο αν η χρησιμοποίηση του επεξεργαστή (Utilization) είναι μικρότερη ή ίση της μονάδας:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Σχήμα 1.6.5a τύπος processor utilization

που C_i είναι ο χείριστος χρόνος εκτέλεσης των nδιεργασιών και T_i οι αντίστοιχες περίοδοι άφιξης. Τα δύο σημαντικότερα πλεονεκτήματα του αλγορίθμου EDF είναι η επίτευξη εως και 100% χρησιμοποίησης του επεξεργαστή αλλά και η υποστήριξη ουράς δυναμικών προτεραιοτήτων. Ένα ακόμα μικρό κέρδος του αλγορίθμου είναι πως δεν προαπαιτεί οι διεργασίες που χρονοδρομολογούνται από αυτόν να είναι περιοδικές και άρα χρησιμοποιείται τόσο σε περιοδικές όσο και απεριοδικές συναρτήσεις. Ωστόσο ο αριθμός του συνόλου των διεργασιών που θα χάσουν τον χρόνο προθεσμίας τους όταν το σύστημα υπερφορτώνεται είναι σε μεγάλο βαθμό απρόβλεπτος, ενώ παράλληλα ο αλγόριθμος είναι δύσκολα εφαρμόσιμος σε hardware καθώς είναι δύσκολη η αναπαράσταση των χρονικών προθεσμιών σε διαφορετικές σειρές. Για αυτό το λόγο υπάρχουν μελέτες που επεκτείνουν τον αρχικό αλγόριθμο όπως για παράδειγμα ο αλγόριθμος πρόβλεψης χρονικής προθεσμίας (Predictive Deadline Algorithm) ή ο αλγόριθμος εξυπηρετητή ολικού εύρους ζώνης (Total Bandwidth Server). Βέβαια η μελέτη μας εκτείνετε μόνο μέχρι τον αλγόριθμο EDF και όχι στις επεκτάσεις του.

Παρακάτω παρατίθεται ένα παράδειγμα χρονοπρογραμματισμού τριών διεργασιών με τον αλγόριθμο EDF:



Σχήμα 1.6.5b Παράδειγμα EDF

Το παράδειγμα αποτελείται από τρείς διεργασίες J_1 , J_2 και J_3 με χρονικές προθεσμίες 4, 5 και 7 και χρόνο εκτέλεσης 1, 2 και 2 αντίστοιχα. Στο σχήμα αυτό δεν υπάρχουν χαμένες χρονικές προθεσμίες. Σύμφωνα με τον αλγόριθμο EDF ο χρονοδρομολογητής σωστά τη χρονική στιγμή 0 επιλέγει την διεργασία J_1 για να εκτελεσθεί καθώς έχει την μικρότερη

χρονική προθεσμία εκ των τριών (4 < 5, 7). Παρόμοια στη δεύτερη χρονική στιγμή ο χρονοδρομολογητής επιλέγει τη διεργασία J_2 καθώς η J_1 έχει ολοκληρώσει την εκτέλεση της και ο χρόνος προθεσμίας της J_2 είναι μικρότερος από αυτόν της J_3. Μία σημαντική παρατήρηση έρχεται στη χρονική στιγμή 4 που ξεκινά νέος κύκλος περιόδου για την J_1. Βλέπουμε λοιπόν πως η J_3 δεν σταματά την εκτέλεση της αλλά συνεχίζεται κανονικά καθώς σε αυτό το σημείο η προθεσμία της J_1 έχει μικρότερη προτεραιότητα από την J_3. Έτσι η J_3 προλαβαίνει να εκτελεσθεί πριν το πέρας της χρονικής προθεσμίας και αυτό επιτυγχάνεται με την δυναμική αλλαγή προτεραιότητας. Στην αρχή (χρονική στιγμή 0) η J_1 έχει μεγαλύτερη προτεραιότητα από την J_3 ενώ στην χρονική στιγμή 4 έχει μικρότερη.

2 ΕΙΣΑΓΩΓΗ ΣΤΑ ΧΡΟΝΙΣΜΕΝΑ ΑΥΤΟΜΑΤΑ

2.1 Εισαγωγή

Τα χρονισμένα αυτόματα (ΤΑ) αναπτύχθηκαν και εξελίχθηκαν από τους Y.Abdeddaim και O.Maler, σαν μία μέθοδο μοντελοποίησης συστημάτων πραγματικού χρόνου[2,21].Χρησιμοποιώντας τέτοια μοντέλα μας δίνεται η δυνατότητα να εκφράσουμε φυσικά χρονικούς περιορισμούς όπως η διάρκεια μίας συγκεκριμένης διεργασίας ή την διαφοροποίηση του χρόνου μεταξύ δύο γεγονότων, ενώ στην γραφική τους αναπαράσταση εκχωρούν πόρους σε κάθε κόμβο μέσο μεταβάσεων μεταξύ τους. Σε συνδυασμό με αυτό, έχουν την δυνατότητα να υπόκεινται σε μοντέλα αλγοριθμικής επαλήθευσης, πράγμα που συνέβη συχνά όπως στα εργαλεία Kronos και UPPAAL [21].

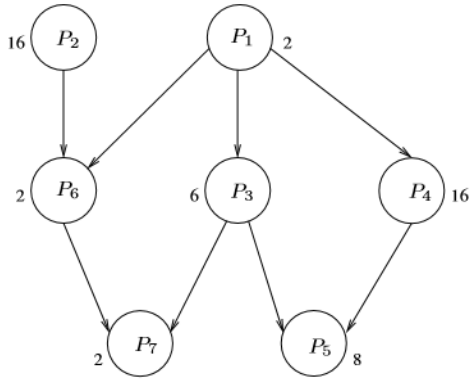
Τα χρονισμένα αυτόματα είναι αυτόματα με μεταβλητές συνεχούς χρόνου ρολογιών, του συστήματος, των οποίων οι τιμές αυξάνονται σε κάθε διαφορετική κατάσταση. Ένας βασικός λόγος για τον οποίο τα χρονισμένα αυτόματα είναι ιδανικά για την περιγραφή ταυτόχρονων χρονο-εξαρτώμενων συμπεριφορών είναι λοιπόν οι τιμές αυτών τον ρολογιών, καθώς μπορούν είτε να καθορίσουν μεταβάσεις είτε να επαναφερθούν στην αρχική τους τιμή [3]. Παρακάτω δίνεται ο πλήρης μαθηματικός ορισμός ενός χρονισμένου αυτομάτου αλλά και κάποιες βασικές έννοιες χρόνου που θα χρησιμοποιήσουμε στη συνέχεια.

2.2 Γράφος Αλληλοεξαρτήσεων – Task Graph:

Ένας γράφος αλληλεξάρτησης αποτελεί ένα από τα μεγαλύτερα εργαλεία για την επίλυση προβλημάτων χρονοδρομολόγησης. Ένας τέτοιος γράφος είναι ένα σχεδόν ταξινομημένο σύνολο από tasks, με ένα ακέραιο αριθμό (task duration) που συνδέεται με τον κάθε κόμβο. Μία διεργασία του γράφου μπορεί να ξεκινήσει να εκτελείται μόνο όταν όλοι οι προκάτοχοι του έχουν τελειώσει την εκτέλεσή τους [17]. Στη συνέχεια δίνεται ο αυστηρός μαθηματικός ορισμός ενός task graph.

Ένα διάγραμμα αλληλεξάρτησης ορίζεται ως μία τριπλέτα $G = (P , < , d)$τέτοια ώστε το $P = \{ P_1,\ldots,P_m \}$να είναι ένα σύνολο από mδιεργασίες, το < είναι μία σχέση μερικής τάξης στο Pκαι το d : $P \rightarrow$Νείναι μία συνάρτηση που αναθέτει «διάρκεια» στην κάθε διεργασία.

Παρακάτω παρατίθεται ένας τέτοιος γράφος αλληλεξάρτησης μεταξύ επτά διεργασιών. Σημειώνεται πως στα προβλήματα χρονοδρομολόγησης που αντιμετωπίζουμε δεν χρησιμοποιούμε την έννοια του γράφου αλληλεξάρτησης σχηματικά, αλλά σε μορφή Booleanπινάκων nxn που η κάθε σειρά του πίνακα αναπαριστά την κάθε διεργασία του προβλήματος αλλά και τις εξαρτήσεις τις από τις υπόλοιπες (κεφ. 4).



Σχήμα 2.2 Γράφος αλληλοεξάρτησης

2.3 ΟΡΙΣΜΟΣ(χρονισμένο αυτόματο-timed automaton):

Ως χρονισμένο αυτόματο καθίσταται ένα διάνυσμα *A = (Q,C,s,f,Δ)* , όπου *Q* είναι ένα πεπερασμένο σύνολο καταστάσεων, *C* ένα πεπερασμένο σύνολο χρονικών μεταβλητών-ρολογιών και *Δ* είναι μία σχέση μεταβάσεων που αποτελείται από στοιχεία της μορφής *(q,φ,ρ,q')* όπου *q* και *q'* είναι καταστάσεις, *ρ* υποσύνολο του *C* και *φ* (συνθήκη ελέγχου μετάβασης) είναι ένας αλγεβρικός συνδυασμός τύπων της μορφής (*C* ε *I*) για κάποιο ρολόι c και κάποιο διάστημα ακέραια περιορισμένο. Οι καταστάσεις *s* και *f* αποτελούν την αρχική και την τελική κατάσταση αντίστοιχα [17].

2.4 ΟΡΙΣΜΟΣ(εκτίμηση χρόνου-clock valuation):

Εκτίμηση χρόνου ονομάζεται μία συνάρτηση **v** : $C \rightarrow R_+ \cup \{0\}$ ή ισοδύναμα ένα διάνυσμα μήκους |*C*| ε R_+ [15]. Ορίζοντας το σύνολο των χρονικών εκτιμήσεων ως *H* προκύπτει πως μία παραμετροποίηση του αυτόματου είναι το ζεύγος *(q , **v**) ε Q × H*, που αποτελείται από μία διακεκριμένη κατάσταση και μία χρονική εκτίμηση. Κάθε *ρ* που είναι υποσύνολο του *C* εισάγει μία συνάρτηση μηδενισμού (επαναφοράς):

$Reset_\rho$:$H \rightarrow H$, ορισμένη για κάθε εκτίμηση χρόνου **v**και κάθε μεταβλητή ρολογιού *c*ε *C* ως:

$$Reset_\rho \mathbf{v}(c) = \begin{cases} 0 \text{ αν } c \in \rho \\ \mathbf{v}(c) \text{αν } c \neq \varepsilon \rho \end{cases}$$

Με άλλα λόγια η $Reset_\rho$ μηδενίζει και επαναφέρει όλα τα ρολόγια στο *ρ* αφήνοντας τα υπόλοιπα αμετάβλητα. Για δική μας ευκολία στον ορισμό, χρησιμοποιούμε **1** για το μοναδιαίο διάνυσμα (1,1,...,1) και **0** για το μηδενικό διάνυσμα.

Ένα βήμα σε ένα αυτόματο μπορεί να είναι:

- Διακριτό βήμα: $(q, \mathbf{v}) \rightarrow_0 (q', \mathbf{v'})$ όπου υπάρχει $\delta = (q,\varphi,\rho,q')$ ε Δ, τέτοιο ώστε το **v**να ικανοποιεί τα *φ* και **v'**= $Reset_\rho(\mathbf{v})$

- Χρονικό βήμα: $(q, \mathbf{v}) \rightarrow_t (q, \mathbf{v} + t \times \mathbf{1}), t \varepsilon R_+$

Μία εκτέλεση ενός αυτομάτου που ξεκινά από μία παραμετροποίηση $(q_0, \mathbf{v_0})$ αποτελεί μία πεπερασμένη ακολουθία βημάτων:

$$\xi : (q, \mathbf{v}) \rightarrow_{t1} (q_1, \mathbf{v_1}) \rightarrow_{t2} \ldots \rightarrow_{tn} (q_n, \mathbf{v_n})$$

Το λογικό μήκος μίας τέτοιας εκτέλεσης είναι n και το μετρικό μήκος είναι $t_1 + t_2 + ... + t_n$, όπου οι διακριτές μεταβάσεις δεν δαπανούν χρόνο.

Πολλές φορές μας διευκολύνει η διεύρυνση ενός αυτομάτου *A* με ένα επιπρόσθετο ρολόι *t*το οποίο είναι ενεργό σε κάθε κατάσταση του αυτομάτου και δεν επαναφέρεται ποτέ στην τιμή μηδέν. Το προκύπτων αυτόματο συμβολίζεται με *A'* και ονομάζεται διευρυμένο αυτόματο, ενώ οι εκτελέσεις του ονομάζονται διευρυμένες εκτελέσεις του αυτομάτου *A'* . Εφόσον το *t*είναι μεταβλητή απόλυτου χρόνου, το (*q*, **v**, *t*) προσεγγίζεται στο *A'* αν και μόνο αν το (*q*, **v**) προσεγγίζεται στο *A* τη χρονική στιγμή *t [17,22]*.

Σημειώνεται ακόμα πως οι μεταβάσεις "έναρξη" και "λήξη" , οι οποίες χρησιμοποιήθηκαν στις προηγούμενες παραγράφους δεν χρησιμοποιούνται στη συνέχεια καθώς στη θέση τους χρησιμοποιείται η χρονική τους διάρκεια η οποία είναι μηδέν. Παρόλο που από τον ορισμό των ρολογιών εμφανίζεται ομοιομορφία σε αυτά καθώς η παράγωγός τους είναι 1 πάντα, υπάρχουν καταστάσεις που συγκεκριμένες τιμές ρολογιών δεν είναι τόσο σημαντικές καθώς σε κάθε μονοπάτι με αφετηρία τις συγκεκριμένες καταστάσεις οι τιμές των ρολογιών δεν ελέγχονται πριν επαναφερθούν στο μηδέν (π.χ. ρολόι X_1 σε κατάσταση $(\bar{p}_1, p_2)$). Θα λέμε πως ένα ρολόι σε μία τέτοια κατάσταση είναι ανενεργό.

2.5 ΟΡΙΣΜΟΣ(χρονισμένο αυτόματο σε διεργασία-timed automaton for a task)

Ας θεωρήσουμε έστω G ένα διάγραμμα διεργασιών. Για κάθε διεργασία/task P που ανήκει σε αυτό το διάγραμμα ένα χρονισμένο αυτόματο *A = (Q,{C},s,f,Δ)* με *Q* = { p, p^-, p_-} είναι συνδεδεμένο μαζί της, με p^- την αρχική κατάσταση και p_- την τελική. Η σχέση μετάβασης Δ αποτελείται από τις εξής δύο μεταπτώσεις:

Αρχική:

$$(\overline{p}, \bigwedge_{P \in \Pi(P)} \underline{p'}, \{c\}, p)$$

και

Τελική:

$$(p, c = d(p), \emptyset, \underline{p})$$

Το καθολικό αυτόματο αντιπροσωπεύει κάθε εφικτό χρονοδιάγραμμα που μπορεί να προκύψει ως σύνθεση των επιμέρους διεργασιών, λαμβάνοντας υπόψη πως η αρχική μετάβαση δεν παραβιάζει την προτεραιότητα και τους περιορισμού των πόρων/resources [22]. Αυτό επιτυγχάνεται επιτρέποντας μια τέτοια μετάβαση σε μία καθολική κατάσταση μόνο όταν ο «φρουρός» της (σ.σ. guard, θα εξηγηθεί στο κεφάλαιο UPPAAL) ικανοποιείται και ο αριθμός των ενεργών διεργασιών στην προκύπτουσα κατάσταση δεν είναι μεγαλύτερος απο τον αριθμό των μηχανών που είναι αρμόδιες για την κάθε διεργασία [15,17].

Ωστόσο ενώ το καθολικό αυτόματο αυτό μπορεί να περιγράφει πλήρως όλες τις προσβάσιμες καθολικές καταστάσεις, διαθέτει κάποια χαρακτηριστικά που μπορεί να μετατρέψουν μία ανάλυση ενός προβλήματος σε ανέφικτη. Αυτο οφείλεται στο ότι οι καθολικές καταστάσεις περιγράφονται ως διανύσματα μεγέθους m, που το m έχει την δυνατότητα να είναι ένας πολύ μεγάλος αριθμός πράγμα που κάνει εξίσου μεγάλο και τον αριθμό των ρολογιών (έχουν και αυτά αριθμό m) [17].

3 ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΩΝ ΧΡΟΝΙΣΜΕΝΩΝ ΑΥΤΟΜΑΤΩΝ

3.1 Έλεγχος Μοντέλου – Model Checking:

Η μοντελοποίηση είναι μια πολύ ισχυρή μέθοδος για τον εντοπισμό σφαλμάτων σε περίπλοκα συστήματα όπως στοιχεία hardware, ενσωματωμένους ελεγκτές και πρωτόκολλα επικοινωνίας. Κατά τη διαδικασία ελέγχου μοντέλου οι προδιαγραφές ενός τέτοιου συστήματος εκφράζονται σαν τύποι χρονικής λογικής ενώ παράλληλα χρησιμοποιούνται αποδοτικοί συμβολικοί αλγόριθμοι ώστε να «διασχίσουν» το μοντέλο που ορίζει το σύστημα και να ελέγξουν εάν εν τέλη ο προσδιορισμός ισχύει. Το 2007, οι E.M. Clarke , E.A. Emerson και J.Sifakis βραβεύτηκαν με το ACM Turing Award για την ανάπτυξη του ελέγχου μοντέλου σε μία εξαιρετικά αποτελεσματική τεχνολογία επαλήθευσης, που οικειοποιήθηκε η βιομηχανία.[8]

Οι ελεγκτές μοντέλου παρέχουν την δυνατότητα ανάλυσης μοντέλων μέσω της δυναμικής συμπεριφοράς των συστημάτων. Τέτοια συστήματα ποικίλλουν όπως ένα δίκτυο υπολογιστών ή εκτυπωτών, ένα παζλ ή μία αποικία μυρμηγκιών, ένα αυτόνομο ρομπότ ή ένα λογισμικό ελέγχου για τραίνα κλπ. Στην πραγματικότητα κάθε σύστημα μπορεί να αναλυθεί χρησιμοποιώντας την τεχνική του ελέγχου μοντέλου, αρκεί να διαθέτει καταστάσεις (states) και μεταβάσεις (transitions) μεταξύ των καταστάσεων.[Frits Vaandrager: A First Introduction to Uppaal, 14, 21]

3.2 Τυπική Επαλήθευση

Στο πλαίσιο των συστημάτων υλικού και λογισμικού (hardware-software) η τυπική επαλήθευση είναι η πράξη της απόδειξης ή της απόδειξης της ορθότητας των προβλεπόμενων αλγορίθμων που διέπουν ένα σύστημα σε σχέση με μία συγκεκριμένη τυπική προδιαγραφή η ιδιότητα, χρησιμοποιώντας επίσημες μαθηματικές μεθόδους [5]. Κατά συνέπεια, η τυπική επαλήθευση σε ένα σύστημα είναι καθοριστική για την ορθότητά του και πραγματοποιείται με την παροχή μίας επίσημης απόδειξης για ένα αφηρημένο μαθηματικό μοντέλο του συστήματος, που η σχέση μεταξύ αυτού του μοντέλου και της φύσης του συστήματος είναι γνωστή από την αρχική κατασκευή. Μερικά παραδείγματα μαθηματικών στοιχείων που περιγράφουν τέτοια συστήματα είναι οι μηχανές πεπερασμένων καταστάσεων, τα χαρακτηρισμένα συστήματα μετάβασης, τα συστήματα προσθήκης φορέα, τα υβριδικά αλλά και τα χρονισμένα αυτόματα.

Η τυπική επαλήθευση εξερευνεί τον πλήρη χώρο των πιθανών συμπεριφορών σχεδιασμού ενός συστήματος, σε αντιδιαστολή με τις τεχνικές δοκιμών, προσομοίωσης και εξομοίωσης, οι οποίες χρησιμοποιούνται για την εύρεση και εξάλειψη σφαλμάτων αλλά δεν έχουν την δυνατότητα να αποδείξουν την απουσία ελαττωμάτων σχεδίασης. Βέβαια, η τυπική επαλήθευση λύνει προβλήματα που έχει αποδειχθεί πως είναι πολυπλοκότητας NP-complete(ή χειρότερα), πράγμα που σημαίνει πως είναι σχεδόν αδύνατη η εύρεση ενός γενικευμένου αποτελεσματικού αλγορίθμου που θα εξασφαλίζει σε κάθε περίπτωση μία

έγκαιρη λύση. Ωστόσο, ύστερα από δεκαετίες έρευνας έχουμε οδηγηθεί στη σημερινή γενιά των εργαλείων τυπικής επαλήθευσης που είναι σε θέση να αναλύσουν υπαρκτά βιομηχανικά σχέδια σε εύλογο χρονικό διάστημα.

Η διαδικασία της τυπικής επαλήθευσης δημιουργείται από αποδοτικές δομές δεδομένων που καταγράφουν κάθε πιθανή συμπεριφορά ενός μοντέλου, συνδυαζόμενα με αποδοτικού αλγορίθμους που αναζητούν πάνω από το χώρο για να βρουν αντιπαραδείγματα σε περίπτωση που οι απαιτήσεις σχεδίασης δεν έχουν ικανοποιηθεί. Με το συνδυασμό αποτελεσματικών αλλά και ευέλικτων αναπαραστάσεων της δυαδικής λογικής με ένα έξυπνο σύνολο δομών δεδομένων και στρατηγικών αναζήτησης, τα σύγχρονα εργαλεία τυπικής επαλήθευσης μπορούν να λειτουργήσουν αποτελεσματικά σε πολλές πραγματικές εφαρμογές με εκατοντάδες ή ακόμα και χιλιάδες μεταβλητές.[13,17]

3.3 Μοντελοποίηση

Το UPPAAL αναπτύχθηκε και εξελίχθηκε σε συνεργασία μεταξύ του τμήματος Πληροφορικής του πανεπιστημίου Uppsala της Σουηδίας και του τμήματος Επιστήμης των υπολογιστών του πανεπιστημίου Aalborg της Δανίας [https://www.uppaal.org/]. Αποτελεί ένα εργαλείο προσομοίωσης και επαλήθευσης της μοντελοποίησης μέσο διαφόρων τεχνικών, σε ένα σύστημα πραγματικού χρόνου βασισμένο σε προβλήματα μεταβαλλόμενων περιορισμών. Το UPPAAL αποτελεί ιδανικό εργαλείο περιγραφής συστημάτων που μοντελοποιούνται από ένα σύνολο μη ντετερμινιστικών διεργασιών με πεπερασμένη δομή ελέγχου και ρολόγια πραγματικού χρόνου που επικοινωνούν μεταξύ τους μέσο καναλιών και διαμοιραζόμενων μεταβλητών. Σε τέτοια προβλήματα-συστήματα προς μελέτη οι έννοιες του χρόνου έχουν υψίστη σημασία, με αποτέλεσμα το εργαλείο UPPAAL μέσο των δυνατοτήτων του να καθίσταται ιδανικό, όπως μοντέλα πρωτοκόλλων επικοινωνίας και μοντέλα ελεγκτών χρόνου. [Kim G. Larsen, Paul Pettersson, and Wang Yi: UPPAAL in a Nutshell]

Το UPPAAL αποτελείται από τρία κύρια μέρη: τη γλώσσα περιγραφής, τον προσομοιωτή (simulator) και τον έλεγχο μοντέλου (model checker). Η γλώσσα περιγραφής είναι μία μη-ντετερμινιστική φυλασσόμενη γλώσσα εντολών με μεταβλητές ρολογιών πραγματικού χρόνου και απλούς τύπους δεδομένων. Χρησιμεύει ως γλώσσα μοντελοποίησης ή σχεδιασμού ώστε να περιγράψει την συμπεριφορά του υπό μελέτης συστήματος, μέσο ενός συμπλέγματος από αυτόματα επαυξημένα με ρολόι και μεταβλητές (χρονισμένα αυτόματα). Ο προσομοιωτής (simulator) είναι ένα εργαλείο επικύρωσης που επιτρέπει την εξέταση πιθανών δυναμικών εκτελέσεων ενός συστήματος στα αρχικά στάδια μοντελοποίησής του, και κατ’ επέκταση παρέχει έναν ανέξοδο έλεγχο ανίχνευσης βλαβών, πριν λάβει χώρα η επικύρωση του model checker. Ο model checker είναι αρμόδιος για τον έλεγχο των invariants[κεφ..] και των ιδιοτήτων οριοθετημένης διαβίωσης (bounded-liveness) εξερευνώντας τη συμβολική κατάσταση χώρου του συστήματος.

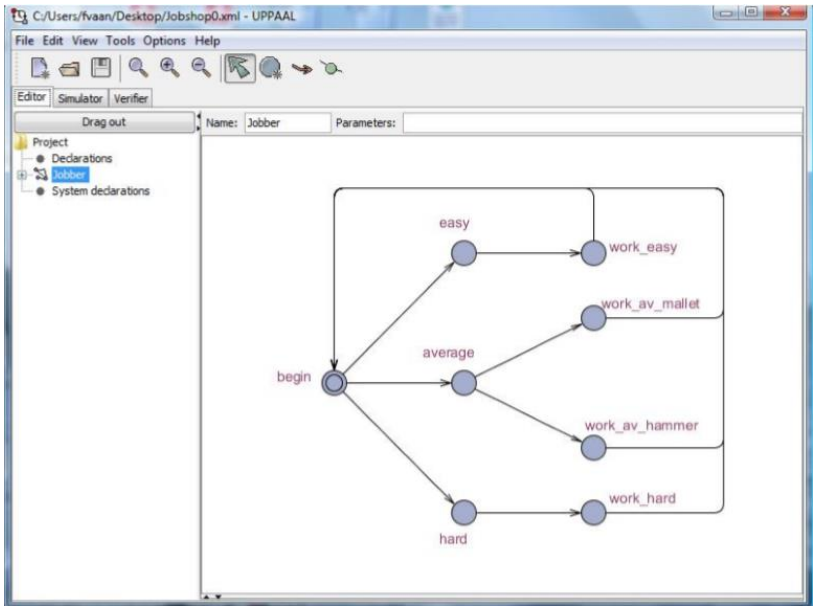
Κύριος στόχος σχεδίασης του εργαλείου αποτέλεσε η αποδοτικότητα και η ευκολία στη χρήση του. Ο περιορισμός στην ανάλυση προσεγγισημότητας υπήρξε καθοριστικός για την αποδοτικότητα του του μοντέλου ελέγχου του UPPAAL. Ένας ακόμα σημαντικός παράγοντας που επηρέασε την αποδοτικότητά του είναι η εφαρμογή της επαλήθευσης κατά τη διάρκεια της εξερεύνησης του χώρου καταστάσεων (on-the-fly verification) [4] σε συνδυασμό με την συμβολική τεχνική η οποία περιορίζει προβλήματα επαλήθευσης σε προβλήματα χειρισμού και επίλυσης απλοποιημένων περιορισμών. Για να διευκολύνουμε

την μοντελοποίηση και τον εντοπισμό σφαλμάτων, ο έλεγχος μοντέλου (model checker) του UPPAAL έχει την δυνατότητα να δημιουργήσει αυτόματα διαγνωστικό ίχνος (diagnostic trace) το οποίο απαντά στο αν μία ιδιότητα ικανοποιείται (ή όχι) από την περιγραφή του συστήματος. Τα diagnostic traces που δημιουργούνται από τον model checker του UPPAAL γίνονται ορατά μέσο του ίδιου του προσομοιωτή. Από την πρώτη έκδοση που κυκλοφόρησε το 1995 έως και σήμερα, το UPPAAL έχει εφαρμοστεί σε αμέτρητες μελέτες ενώ παράλληλα έχει επεκτείνει και εξελίξει πολλά από τα χαρακτηρίστηκα του. [Johan Bengtsson, Kim Larsen, Fredik Larsson, Paul Pettersson, Wang Yi: UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems. / Paul Pettersson – Modelling and Verification of Real-Time Systems Using Timed-Automata].

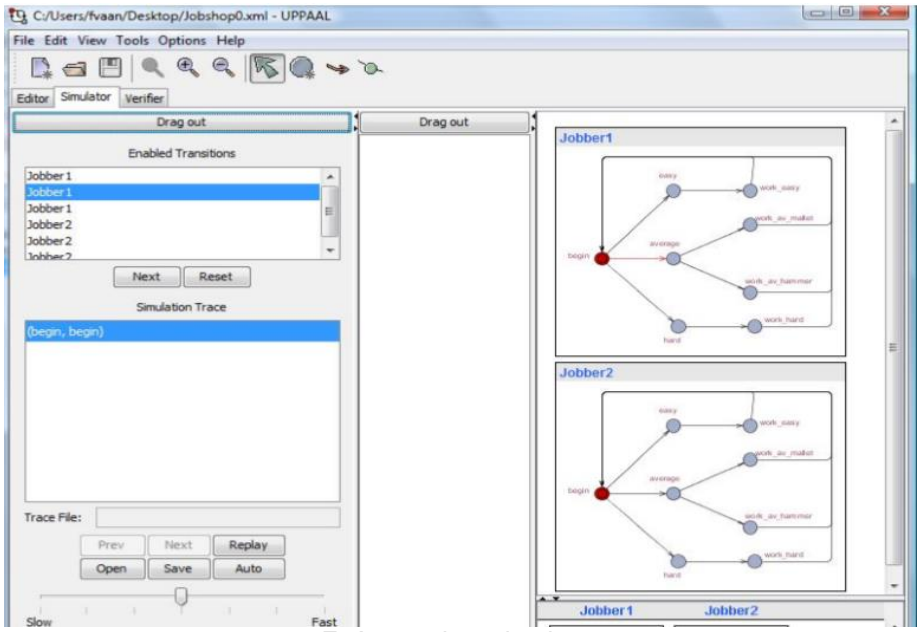
Ο Server του UPPAAL παρέχει ένα αποδοτικό υπολογισμό του συμβολικού χώρου του συστήματος κατά τη διαδικασία επαλήθευσης μιας δοσμένης ιδιότητας. Το GUI (επαν)εκκινεί το πρόγραμμα του Server κάθε φορά που ο χρήστης αποφασίζει να ανανεώσει τον προσομοιωτή ή την μονάδα επαλήθευσης με ένα νέο μοντέλο συστήματος. Το GUI και ο Server επικοινωνούν μέσω μηχανισμού σύνδεσης με TCP/IP sockets. Η σύνδεση αυτή εγκαθίσταται μετά το ξεκίνημα του Server.

Το γραφικό περιβάλλον του UPPAAL αποτελείται απο τα εξής τρία κύρια μέρη, τα οποία είναι προσβάσιμα μέσω τριών καρτελών στο ανοίγοντας το πρόγραμμα:

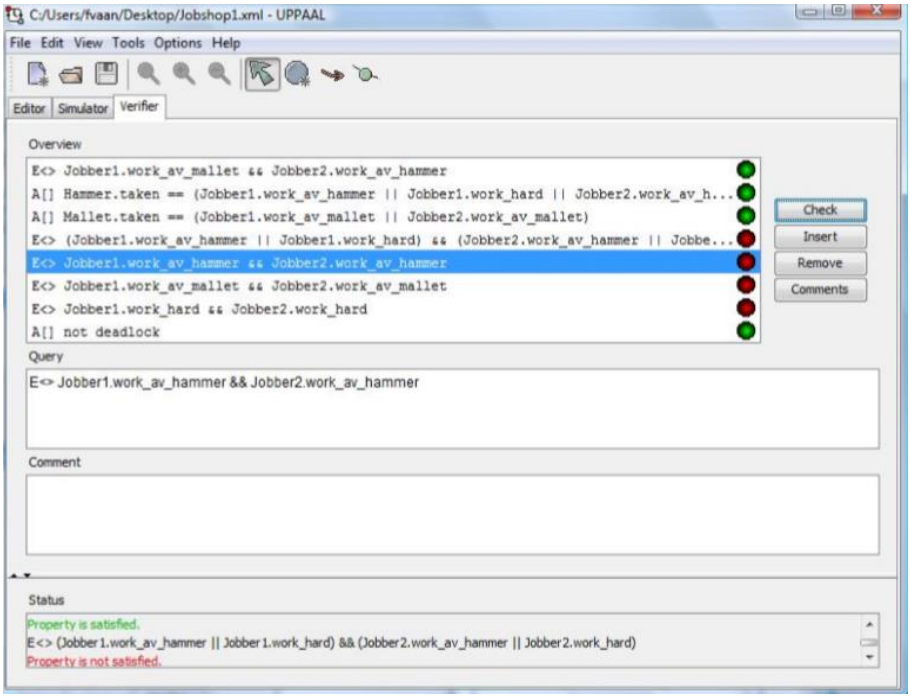
Τη **μονάδα σύνταξης του μοντέλου (System editor)** που μπορεί να χρησιμοποιηθεί για την κατασκευή μοντέλων. Το μοντέλο αποτελείται από ένα φραγμένο αριθμό από παράλληλες διαδικασίες (concurent processes), κάθε μία από τις οποίες περιγράφεται από ένα χρονισμένο αυτόματο. Κάθε διαδικασία ενδέχεται να περιλαμβάνει τοπικής εμβέλειας (local) ρολόγια και μεταβλητές, τα οποία μπορούν να χρησιμοποιηθούν μόνο μέσα σε αυτή και μπορεί να κάνει χρήση των γενικής εμβέλειας (global) ρολογιών και μεταβλητών. Οι διαδικασίες επικοινωνούν μεταξύ τους μέσω των μεταβλητών γενικής εμβέλειας και των δυαδικών συγχρονισμών.
Τον **προσομοιωτή (Simulator)**, στον οποί μπορεί ο χρήστης να απεικονίσει τη συμπεριφορά των μοντέλων και τη **μονάδα επαλήθευσης (Verifier)**, στην οποία μπορεί να αναλυθεί η συμπεριφορά των μοντέλων και δέχεται κατάλληλα μορφοποιημένες ιδιότητες για να επαληθευτούν για ένα συγκεκριμένο μοντέλο αυτομάτων και απεικονίζει το αποτέλεσμα: σωστό ή λάθος αν η ιδιότητα ικανοποιείται ή όχι αντιστοίχως. Στη συνέχεια παρουσιάζονται τρία σχήματα που απεικονίζουν την κάθε καρτέλα του UPPAAL που περιεγράφηκε(system editor, simulator , verifier αντίστοιχα).
[Frits Vaandrager, A First Introduction to Uppaal, Δίπλας-Τσακίρης 2004]



Σχημα 3.3a – System Editor



Σχήμα 3.3b – Simulator

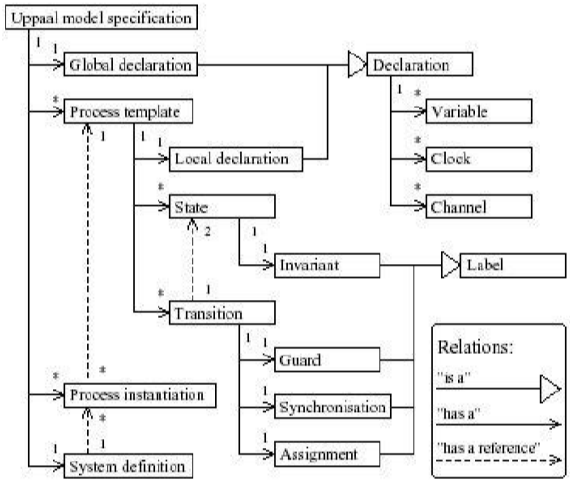


Σχήμα 3.3c –Verifier

3.4 Χρονισμένα Αυτόματα στο εργαλείο μοντελοποίησης

Όπως προαναφέραμε, η μοντελοποίηση των προβλημάτων στο εργαλείο UPPAAL πραγματοποιείται μέσο των χρονισμένων αυτομάτων, τα οποία είναι μηχανές πεπερασμένων καταστάσεων με ρολόγια. Για να παρέχει ένα περισσότερο εκφραστικό μοντέλο και να διευκολύνει το έργο της μοντελοποίησης, το UPPAAL επεκτείνει τα timed automata με γενικότερους τύπους μεταβλητών δεδομένων όπως λογικές (boolean) και ακέραιες μεταβλητές. Ο τελικός σκοπός είναι η μορφοποίηση μια γλώσσας μοντελοποίησης που να είναι όσο το δυνατό περισσότερο συναφής με τις υψηλού επιπέδου και πραγματικού χρόνου γλώσσες προγραμματισμού που περιέχουν διάφορους τύπους δεδομένων. Κάτι τέτοιο ενδεχομένως θα δημιουργούσε προβλήματα λήψης απόφασης (decidability problems) κατά τον έλεγχο του μοντέλου. [18,15,20]

Παρακάτω απεικονίζεται μια ολοκληρωμένη προδιαγραφή μοντέλου χρονισμένων αυτομάτων στο UPPAAL:



Σχήμα χ.χ (Ασπραδάκης 2013)

3.5 Συστατικά και βασικα χαρακτηριστηκα εργαλείου μοντελοποίησης:

- **Εκφράσεις (expressions):** Οι εκφράσεις στο UPPAAL μοιάζουν πολύ με τις εκφράσεις γλωσσών προγραμματισμού, όπως η Java και η C++. Είναι δυνατή η χρήση λογικών τελεστών (and, or, not), ισότητας και ανισότητας, αριθμητικών και εκχώρησης τιμής. Οι τελεστές συνοψίζονται στον εξής πίνακα:

()	Καθορίζει την σειρά αποτίμησης των εκφράσεων	<	Μικρότερο από
[]	Indexing σε πίνακα	<=	Μικρότερο ή ίσο από
.	Lookup operator για πρόσβαση στις local μεταβλητές	==	Τελεστής ισότητας
!	Λογική άρνηση	!=	Τελεστής ανισότητας
++	Αύξηση	>=	Μεγαλύτερο ή ίσο από
--	Μείωση	>	Μεγαλύτερο από
-	Integer subtraction (can also be used as unary negation)	&	Τελεστής and για bits
+	Άθροιση ακεραίων	^	Τελεστής xor για bits
*	Γινόμενο ακεραίων	|	Τελεστής or για bits
/	Διαίρεση ακεραίων	&&	Λογικό and
%	Modulo	||	Λογικό or
<<	Αριστερή μετακίνηση bit	?:	Τελεστής If-then-else
>>	Δεξιά μετακίνηση bit	not	Λογική άρνηση
<?	Ελάχιστο	and	Λογικό and
>?	Μέγιστο	or	Λογικό or
: += -= *= %= &= |= >>= ^=	Τελεστής ανάθεσης τιμής	imply	Λογική συνεπαγωγή

Σχημα 3.5.1 (Δίπλας – Τσακίρης 2004)

Πέραν ωστόσο των συνηθισμένων (κυρίως μαθηματικών) εκφράσεων του παραπάνω πίνακα, οι εκφράσεις στο UPPAAL μπορούν να επεκταθούν και σε εκφράσεις ρολογιών ή ακεραίων μεταβλητών. Μερικά σημαντικά expressions που χρησιμοποιούνται είναι τα παρακάτω [6,15]:

- **Synchronization**Μία ταμπέλα συγχρονισμού είναι είτε της μορφής Expression! και Expression? είτε μία κενή ταμπέλα. Το κομμάτι αυτό θα αναλυθεί περαιτέρω παρακάτω στην έννοια των μεταβάσεων.

- **Assignment**Μία ταμπέλα Assignment είναι ένα σύνολο από εκφράσεις με side-effects διαχωριζόμενες με ένα κόμμα. Οι εκφράσεις αυτές πρέπει να αναφέρονται αποκλειστικά σε ρολόγια ακέραιες μεταβλητές και σταθερές, ενώ έχουν την δυνατότητα να αντιστοιχίζουν μόνο ακέραιες τιμές σε ρολόγια.

- **Invariant**Είναι μία έκφραση που ικανοποιεί τις εξής προϋποθέσεις: Δεν έχει παρενέργειες (side-effects), αναφορά γίνεται μόνο στις ακέραιες μεταβλητές ρολογιών και σταθερών και τέλος να καθίσταται μία σύζευξη καταστάσεων της μορφής x<e ή x≤e που το x θα είναι μία αναφορά τιμής ρολογιού και το e «αξιολογείται» σε ένα ακέραιο.

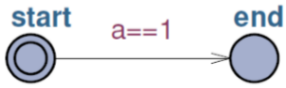
- **Τα ρολόγια(clocks)** είναι μεταβλητές που μπορούν να αξιολογήσουν σε ένα πραγματικό αριθμό και οποίες μπορούν να ορισθούν σε κάθε αυτόματο προκειμένου να μετρηθεί την πρόοδο του χρόνου [21]. Όλα τα ρολόγια εξελίσσονται με τον ίδιο ρυθμό, προκειμένου να εκπροσωπεί το παγκόσμια πρόοδο του χρόνου. Η πραγματική τιμή ενός ρολογιού μπορεί είτε να δοκιμαστεί ή επαναφερθεί (όχι να ανατεθεί). Δεδομένου ότι το εργαλείο UPPAAL είναι ειδικά σχεδιασμένο για την επαλήθευση των συστημάτων πραγματικού χρόνου, τα ρολόγια αποτελούν ένα θεμελιώδη μέσο μοντελοποίησης και επαλήθευσης. Ένα μοντέλο UPPAAL είναι χτισμένο ως ένα σύνολο ταυτόχρονων διαδικασιών, κάθε μια από τις οποίες είναι γραφικά σχεδιασμένη ως χρονισμένο αυτόματο. Κάθε χρονισμένο αυτόματο αντιπροσωπεύεται από ένα γράφημα το οποίο έχει θέσεις (locations) ως κόμβους και ακμές (edges) ως τόξα ανάμεσα στις θέσεις.

Σχήμα 3.5.2

- **Οι μεταβάσεις(transitions)**. Οι άκρες είναι ενημερωμένες με τους παράγοντες guards (φύλακες), updates (ενημερώσεις), synchronizations (συγχρονισμούς) και selections (επιλογές)[15].

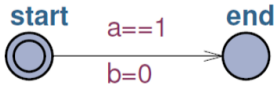
1. Ένας φύλακας(guard) είναι μια έκφραση που χρησιμοποιεί τις μεταβλητές και ρολόγια του μοντέλου, ώστε να υποδεικνύει πότε η μετάβαση είναι ενεργοποιημένη. Σημειώνεται εδώ ότι πολλές ακμές μπορούν να ενεργοποιηθούν σε μία συγκεκριμένη χρονική στιγμή,

αλλά μόνο μία από αυτές θα δράσει ώστε να οδηγήσει σε διαφορετικές δυνατότητες.
Ένας Guard (φρουρός) είναι μία έκφραση που ικανοποιεί την ακόλουθη κατάσταση: Δεν έχει «παρενέργειες» (side-effects), αξιολογείται σε δυαδική τιμή, αναφέρεται μόνο σε ρολόγια, μεταβλητές ακεραίων και σταθερές (ή πίνακες τέτοιων τύπων), ρολόγια και διαφορές μεταξύ ρολογιών είναι συγκρίσημες μόνο με ακέραιες εκφράσεις και τέλος οι Guards πάνω από τα ρολόγια με τα ίδια τα ρολόγια αποτελούν ουσιαστικά συζεύξεις[15].



Σχήμα 3.5.3 guard

2. Μια ενημέρωση(update) είναι μια έκφραση που αξιολογείται το συντομότερο καθώς το αντίστοιχο άκρο βρίσκεται σε δράση. Η αξιολόγηση αλλάζει την κατάσταση του συστήματος[15].



3. Ο συγχρονισμός(synchronization) είναι ο βασικός μηχανισμός που χρησιμοποιείται για συντονίζει τη δράση των δύο ή περισσότερες διεργασίες. Μοντελοποιεί, για παράδειγμα, την επίδραση των μηνυμάτων. Προκαλεί δύο (ή περισσότερες) διεργασίες να λάβουν μια μετάβαση την ίδια στιγμή. Όταν ένα κανάλι c δηλώνεται, τότε μια διαδικασία θα έχει επισημασμένη την άκρη ως c! Και η άλλη (ες) διαδικασία (ες) ως c?. Υπάρχουν τρία είδη συγχρονισμών, τα οποία περιγράφονται ακολούθως [6,15]:

- **Regular channel**: δηλώνεται ως chan c. Όταν μια διαδικασία βρίσκεται σε μια θέση από την οποία υπάρχει μια μετάβαση που επισημαίνεται με το c!, ο μόνος τρόπος για τη μετάβαση να είναι ενεργοποιημένη είναι μια άλλη διαδικασία να είναι σε μια θέση από την οποία υπάρχει μετάβαση που επισημαίνεται με c? και το αντίστροφο. Εάν σε μια συγκεκριμένη στιγμή, υπάρχουν διάφοροι τρόποι για να έχουν ζεύγος c! και c?, ένας από αυτούς είναι μη- ντετερμινιστικά επιλεγμένος κατά τον έλεγχο μοντέλου. Η ενημέρωση σε μια άκρη συγχρονισμού για c! εκτελείται πριν από την ενημέρωση σε μια άκρη συγχρονισμού για c?[15].

- **Urgent channel**: δηλώνεται ως urgent chan c. Είναι παρόμοιο με το regular channel, εκτός από το γεγονός ότι δεν είναι δυνατό να καθυστερήσει στο σταθμό προέλευσης, αν είναι εφικτό να ενεργοποιήσει το συγχρονισμό πάνω από το regular channel. Αυτό σημαίνει ότι δεν μπορεί να περάσει χρόνος, αλλά μπορεί να εναλλάσσει άλλες

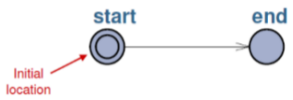
μεταβάσεις που δεν απαιτούν χρόνο για να περάσει. Σημειώνεται ότι τα ρολόγια «φύλακες» δεν επιτρέπονται στις άκρες πάνω από ένα urgent channel.

Σχήμα 3.5.4 α, β

- **Broadcast channel**: δηλώνεται ως broadcast chan c. Όταν μία διεργασία βρίσκεται σε μια θέση από την οποία υπάρχει μια μετάβαση που επισημαίνεται με c! και μία ή περισσότερες διεργασίες είναι σε θέσεις από τις οποίες υπάρχει μια μετάβαση επισημασμένη με c?, όλες οι μεταβάσεις είναι ενεργοποιημένες. Ωστόσο, εάν δεν υπάρχουν διεργασίες στις θέσεις από τις οποίες υπάρχει μια μετάβαση επισημασμένη με c?, η μετάβαση επισημασμένη με c! είναι ενεργοποιημένη ούτως ή άλλως. Παρατηρείται ότι τα ρολόγια «φύλακες» δεν επιτρέπονται στις άκρες που λαμβάνουν πάνω σε ένα broadcast channel. Η ενημέρωση σχετικά με την άκρη εκπομπής εκτελείται πρώτα. Η ενημέρωση σχετικά με τις ακμές που λαμβάνουν εκτελούνται αριστερά προς τα δεξιά, σύμφωνα με τη διάταξη που οι διεργασίες καταχωρήθηκαν κατά τον ορισμό του συστήματος[15].

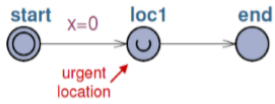
- **Οι θέσεις(locations)**. Αναφορικά με τις θέσεις (locations), μπορούν να έχουν ένα προαιρετικό όνομα, το οποίο χρησιμεύει στο να εντοπίζεται η θέση κατά τη διάρκεια του ελέγχου και της τεκμηρίωσης του μοντέλου. Οι θέσεις μπορεί να είναι τριών ειδών [6,15]:

    1. **Initial (αρχικές)**: Κάθε πρότυπο πρέπει να έχει προετοιμαστεί σωστά, πράγμα που σημαίνει ότι πρέπει να ξεκινήσει σε μια συγκεκριμένη θέση. Ως εκ τούτου, κάθε πρότυπο πρέπει να έχει ακριβώς μία θέση που επισημαίνεται ως αρχική. Οι αρχικές θέσεις που προσδιορίζονται με διπλό κύκλο [15].



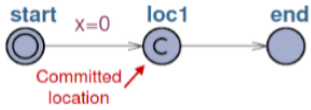
Σχήμα 3.5.5 Initial-Starting location

    2. **Urgent locations**: οι οποίες παγώνουν το χρόνο, δηλαδή δεν επιτρέπεται να παρέλθει χρονικό διάστημα όσο μια διαδικασία βρίσκεται σε εξέλιξη. Η κάθε θέση πρέπει να εγκαταλειφθεί προτού περάσει ο χρόνος. Οι υπόλοιπες μεταβάσεις μπορούν να πραγματοποιηθούν προηγουμένως εφόσον δεν απαιτούν την παρέλευση χρόνου. Στο μοντέλο, αυτού του είδους οι θέσεις επισημαίνονται με U [15].



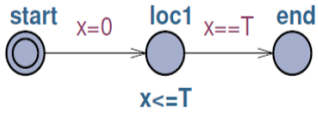
Σχήμα 3.5.6 Urgent location

3. **Committed locations**: οι οποίες επίσης παγώνουν το χρόνο. Όταν ένα μοντέλο έχει ενεργές μία ή περισσότερες τέτοιες θέσεις τότε καμία άλλη διεργασία, πέρα από αυτές που εγκαταλείπουν τις επισημασμένες θέσεις, δεν επιτρέπεται να ενεργοποιηθεί. Οι συγκεκριμένες θέσεις είναι χρήσιμες στο να δημιουργούν ατομικές ακολουθίες. Μέσα στο μοντέλο επισημαίνονται με C [15].



Σχήμα 3.5.7 Committed location

4. **Normal Locations:** όλες οι υπόλοιπες. Τόσο οι αρχικές (initial) όσο και οι normal locations μπορεί να διαθέτουν σταθερές, μέσο των οποίων να ικανοποιούνται συνθήκες όσο το αυτόματο παραμένει σε αυτή τη θέση. Οι σταθερές αυτές μπορούν να σχετίζονται είτε με τα ρολόγια είτε με τις μεταβλητές [15].



Σχήμα 3.5.8 Normal location με σταθερά (που ικανοποιείται όσο βρισκόμαστε εκεί)

3.6 Έλεγχος και επαλήθευση του υπό εξέταση συτήματος

Εφόσον έχει χρησιμοποιηθεί ο προσομοιωτής (simulator), για να εξασφαλιστεί ότι το μοντέλο λειτουργεί και συμπεριφέρεται όπως ακριβώς το προς μοντελοποίηση σύστημα (και μερικές φορές για να ανιχνευτούν τυχόν λάθη στον αρχικό σχεδιασμό), η επόμενη φάση είναι να γίνει έλεγχος κατά πόσο επαληθεύονται οι ιδιότητες του συστήματος. Αρχικά λοιπόν αυτές θα πρέπει να γίνουν γνωστές και να «επισημοποιηθούν» ενώ κατά δεύτερον να μεταφραστούν στη γλώσσα επερωτήσεων του UPPAAL (Uppaal query language). Τα είδη των ιδιοτήτων που μπορούν να ελέγχονται άμεσα χρησιμοποιώντας τα UPPAAL ερωτήματα είναι αρκετά απλά. Οι σχεδιαστές έχουν υιοθετήσει αυτή την προσέγγιση, αντί να επιτρέπουν σύνθετα ερωτήματα, προκειμένου να βελτιωθεί η απόδοση του εργαλείου. Για το λόγο αυτό, η επαλήθευση των σύνθετων ιδιοτήτων μπορεί να απαιτεί τον έλεγχο πολλών διαφορετικών ερωτημάτων και ακόμη και την προσθήκη στο μοντέλο ειδικά σχεδιασμένων "ελεγμένων αυτομάτων".

Η query γλώσσα του UPPAAL αποτελείται από τους τύπους διαδρομής (path formulae) και τους τύπους κατάστασης (state formulae). Οι τύποι κατάστασης περιγράφουν ξεχωριστές ανεξάρτητες καταστάσεις, ενώ οι τύποι διαδρομής ποσοτικοποιούνται σε μονοπάτια η ίχνη του μοντέλου. Οι τύποι διαδρομής κατηγοριοποιούνται σεreachability, safety, liveness. Παρακάτω θα επέλθει περεταίρω ανάλυσή τους ενώ θα δοθούν και αντίστοιχα διαγράμματα περιγραφής για τον κάθε τύπο.

- State formulae: Ένας τύπος κατάστασης ισοδυναμεί με μία έκφραση που έχει τη δυνατότητα να αναχθεί σε μία κατάσταση χωρίς να ελέγξουμε την συμπεριφορά του μοντέλου. Για παράδειγμα η απλή έκφραση i == 7 είναι αληθινή σε μία κατάσταση όποτε το i ισούται με 7. Η σύνταξη των τύπων κατάστασης είναι υπερσύνολο της σύνταξης των guards (φυλάκων) , δηλαδή ένας τύπος καταστάσεων είναι μία έκφραση χωρίς παρενέργειες αλλά σε αντίθεση με τους guards η χρήση διαφραγμάτων δεν «απαγορεύεται» [21].

  Στο UPPAAL[1] το Deadlock εκφράζεται χρησιμοποιώντας έναν ειδικό τύπο κατάστασης. Κατάσταση deadlock επικρατεί εφόσον είναι δυνατό το μοντέλο να εξελίσσεται σε μια διαδοχική στάση χωρίς να περιμένει κάποιο χρονικό διάστημα ή μια μετάβαση μεταξύ θέσεων. Δύο τυπικά παραδείγματα είναι:

  - E <> deadlock = “Exists deadlock”

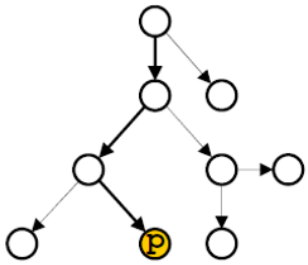
  - A [ ] not deadlock = “There is no deadlock

  Σημειώνεται εδώ ότι η λέξη «deadlock» μπορεί να χρησιμοποιηθεί στο εσωτερικό οποιασδήποτε έκφρασης επισημοποιώντας μια συγκεκριμένη ιδιότητα. Όταν χρησιμοποιείται έλεγχος μοντέλου για την επαλήθευση ενός συστήματος, έχει συνήθως μελετηθεί αν υπάρχει αδιέξοδο στην μοντέλο ή όχι. Ακόμη και αξιοποιώντας την αφαιρετικότητα ο χώρος κατάστασης μπορεί να «εκραγεί». Υπάρχουν επιλογές ελέγχου που μπορούν να βοηθήσουν. Αν είναι ενεργοποιημένες κάποιες επιλογές, η έξοδος του ελεγκτή μπορεί να το γεγονός ότι η ιδιότητα ίσως είναι ικανοποιητική. Ο ελεγκτής δεν μπορεί να προσδιορίσει την τιμή αληθείας της ιδιότητας λόγω των χρησιμοποιούμενων προσεγγίσεων.

- **Reachability Properties**Τα reachability properties καθίστανται οι πιο απλές όλων των ιδιοτήτων. Ερωτούν αν ένας δοθέν τύπος κατάστασης ,έστω *φ*, μπορεί πιθανώς να ικανοποιηθεί από οποιαδήποτε προσεγγίσιμη κατάσταση. Ένας διαφορετικός τρόπος να ορίσουμε το παραπάνω είναι μέσω της εξής ερώτησης: «Υπάρχει διαδρομή που ξεκινά από την αρχική κατάσταση-initial location , τέτοια ώστε ο τύπος κατάστασης *φ* ικανοποιείται μέσα σε αυτή τη διαδρομή?»

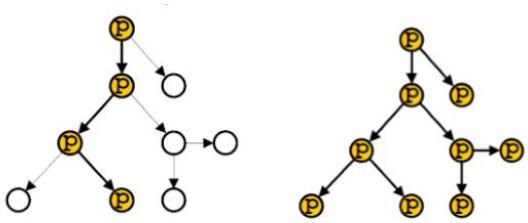
  Η συνηθέστερη χρήση των reachability properties αποτελεί ο έλεγχος λογικής για ένα μοντέλο. Για παράδειγμα σε ένα μοντέλο ενός πρωτοκόλλου επικοινωνίας που περιλαμβάνει ένα αποστολέα και ένα παραδότη, ο πιο λογικός έλεγχος που μπορεί να πραγματοποιηθεί είναι αν ο αποστολέας έχει την δυνατότητα να στείλει μηνύματα ή αν ένα μήνυμα μπορεί να παραληφθεί στο μοντέλο αυτό. Παρόλο που αυτές οι ιδιότητες από μόνες τους δεν εγγυώνται την ορθότητα του πρωτοκόλλου, επικυρώνουν την ορθή λειτουργία της βασικής συμπεριφοράς του μοντέλου. Με τον

τύπο E <>*φ* “Exists eventually *φ*”, που σημαίνει ότι υπάρχει μια διαδρομή εκτέλεσης όπου το *φ* τελικά παραμένει [21].



- **Safety Properties**Τα Safety properties εξασφαλίζουν πως «δεν θα συμβεί ποτέ κάτι κακό». Για παράδειγμα σε ένα μοντέλο πυρηνικού σταθμού ένα safety property μπορούσε να είναι η οριοθέτηση ώστε η θερμοκρασία λειτουργίας να μην υπερβαίνει ποτέ (invariantly) ένα συγκεκριμένο όριο, ώστε να μην συμβεί ποτέ έκρηξη. Μία παραλλαγή της παραπάνω φράσης είναι η «κάτι δεν θα συμβεί ποτέ». Για παράδειγμα, όταν παίζουμε ένα παιχνίδι ένα safe state είναι μία κατάσταση που μπορούμε να κερδίσουμε το παιχνίδι και επομένως να μην χάσουμε [21].

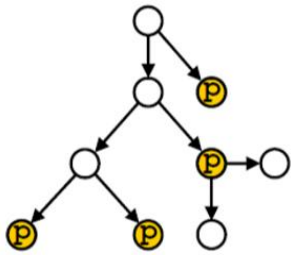
  Στο UPPAAL αυτές οι ιδιότητες διατυπώνονται θετικά (π.χ. something good is invariantly true). Έστω *φ* ένας τύπος κατάστασης. Εκφράζεται πως το *φ* πρέπει να ικανοποιείται σε όλες τις προσβάσιμες καταστάσεις με τον τύπο διαδρομής A<>φ^2 , ενώ το E<>*φ* υποδηλώνει πως πρέπει να υπάρχει μέγιστη διαδρομή κατά την οποία το *φ* ικανοποιείται πάντα[1]. Στο UPPAAL γράφουμε:
  - E [ ] p “Exists globally p”, που σημαίνει ότι υπάρχει ένα μονοπάτι εκτέλεσης κατά το οποίο το p παραμένει σε όλους τους σταθμούς του μονοπατιού (α)
  - A [ ] p “Always globally p”, όπου για κάθε μονοπάτι εκτέλεσης το p διατηρείται σε όλο το μονοπάτι (β)



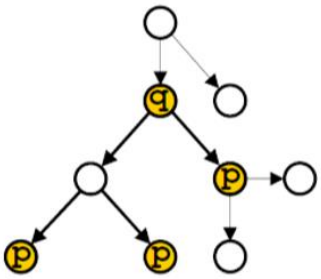
α, β

- **Liveness Properties**Οι Liveness Properties εξασφαλίζουν πως «κάτι θα συμβεί εν τέλη» π.χ. αν πατήσουμε το on button του τηλεκοντρόλ της τηλεόρασης κάποια στιγμή τελικά η τηλεόραση θα ανοίξει ή σε ένα μοντέλο πρωτοκόλλου επικοινωνίας κάθε μήνυμα που έχει σταλεί εν τέλη θα παραδοθεί.

Σε απλή μορφή το Liveness εκφράζεται με τον τύπο διαδρομής A<>*φ* θεωρώντας πως το *φ* τελικά θα ικανοποιηθεί. Ακόμα πιο χρήσιμη μορφή του Liveness γράφεται ως *φ→ψ* , και διαβάζεται ως «όταν ικανοποιηθεί το *φ* τελικά ικανοποιείται το *ψ*» [21].



A<> p “Always eventually *φ*” όπου για κάθε μονοπάτι εκτέλεσης το *φ* συγκρατείται μόνο για ένα σταθμό του μονοπατιού.

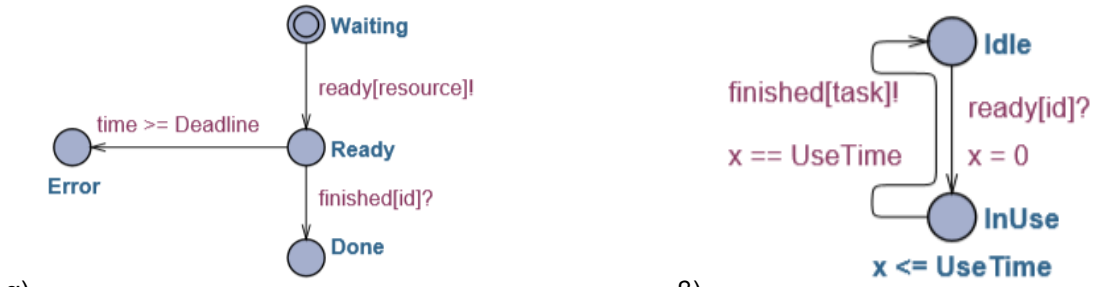


φ→ψ “*φ* always leads to *ψ*”, όπου κάθε μονοπάτι που ξεκινά από ένα σταθμό που παραμένει το *φ* καταλήγει πάντα σε ένα σταθμό που παραμένει το *ψ*.

4 ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗΣ

4.1 Ιδέα της μοντελοποίησης:

Προκειμένου να εξηγηθεί με βέλτιστο τρόπο τα μοντέλα της εργασίας παρατίθεται ένα πιο αφηρημένο μοντέλο χρονοδρομολόγησης, που θα αποτελέσει τη βάση των τελικών μοντέλων[7]. Το αφηρημένο αυτό μοντέλο βασίζεται στις βασικές αρχές χρονοδρομολόγησης και στον ορισμό της διεργασίας (εξηγήθηκε στο κεφ 1).



α) β)

Σχήμα 4.1 Αφηρημένο μοντέλο taskκαι resource

Το παραπάνω μοντέλο που παρουσιάζεται στο σχήμα 4.1 διαιρεί το πρόβλημα χρονοδρομολόγησης στα tasks και το resource (όπως έχει προαναφερθεί το resource θα μπορούσε να είναι ένας επεξεργαστής). Το κάθε task και resource που δημιουργούνται μέσο αυτών των template models έχει ένα μοναδικό αναγνωριστικό αριθμό (id). Αρχικά τα tasks βρίσκονται σε κατάσταση Waiting και όταν ένα task είναι έτοιμο να εκτελεστεί, σηματοδοτεί το resource με το κατάλληλο resource id στο οποίο το ίδιο το task έχει ανατεθεί, χρησιμοποιώντας το ready channel (η μεταβλητή resource αποτελεί το resource id). Αφού ολοκληρωθεί η παραπάνω διαδικασία το task μεταβαίνει στην κατάσταση Ready που παραμένει εκεί είτε μέχρι να τελειώσει η προθεσμία χρόνου του task , που σε αυτή την περίπτωση μεταβαίνουμε στην κατάσταση Error, είτε σηματοδοτείται από το resource ότι η εκτέλεσή του ολοκληρώθηκε, μέσω του καναλιού finished (με μεταβλητή τιμή task id), και στη συνέχεια μεταβαίνει στην κατάσταση Done.

Τα Resources έχουν δύο θέσεις-locations, Idle και InUse , που περιγράφουν την κατάσταση του resource την κάθε χρονική στιγμή. Τα resources αλλάζουν κατάσταση από Idle σε InUse όταν τους σηματοδοτηθεί ένα ready signal από ένα task, και επιστρέφουν στο Idle (από InUse) σε περίπτωση που έχει περάσει ο απαραίτητος χρόνος εκτέλεσης των tasks.

Με αυτό το μοντέλο η έννοιες της χρονοδρομολόγησης πληρούνται και εγγυροποιούνται μέσω της παρακάτω ερώτησης στο verifier του UPPAAL:

A[] forall(i : task_id) not Task(i).Error

Που σημαίνει, Θα ισχύει πάντα πως σε όλες τα σενάρια εκτέλεσης κανένα task δεν θα βρίσκεται στη θέση Error?

Το μοντέλο που περιεγράφηκε παραπάνω αποτέλεσε τη βάση της τελικής μοντελοποίησης που υλοποιήσαμε. Η περαιτέρω επέκταση αυτού του μοντέλου ήταν αναγκαία για τον κατάλληλο χειρισμό περιπτώσεων διακοπτής/μη διακοπτής χορνοδρομολόγησης, περιοδικότητας στα tasks αλλά και εφαρμογής διαφορετικών αλγορίθμων χρονοδρομολόγησης. Πριν την παρουσίαση του τελικού μοντέλου θα αναλύσουμε κάποιες βασικές δομές που χρησιμοποιούνται στον κώδικα.

4.2 Data Structures

Για κάθε πρόβλημα χρονοδρομολόγησης με $T = t_0,...,t_n$ tasks-διεργασίες και $R = r_0,...,r_k$ resources-πόρους ορίζουμε τα εξής στοιχεία τύπων δεδομένων:

t_id: Αναγνωριστικός αριθμός των διεργασιών με έκταση 0 έως n

r_id: Αναγνωριστικός αριθμός των πόρων με έκταση 0 έως k

time_t: Ακέραια τιμή (χρόνου) που εκτείνεται απο 0 έως τον αριθμό μέγιστης περιόδου των διεργασιών.

Εφόσον έχουμε ορίσει καλώς τους παραπάνω βασικούς τύπους δεδομένων μπορούμε στην συνέχεια να παρουσιάσουμε πιό σύνθετους τύπους (δομές δεδομένων), όπως την υλοποίηση της δομής μίας διεργασίας, η οποία ονομάζεται **task_t** και παρουσιάζεται στο παρακάτω σχήμα. Η παρακάτω δομή απαρτίζεται από μεταβλητές που έχουμε ήδη ορίσει οι οποίες είναι οι initial_offset, min_period, max_period, offset, deadline, bcet, wcet που είναι τύποι δεδομένων **time_t,**resource που είναι τύπος δεδομένων **r_id** και pri που είναι μία απλή ακέραια τιμή. Σημειώνουμε εδώ πως η λέξη priority ήταν δεσμευμένη στο UPPAALγια τη μεταβλητή prio και για αυτό δεν την χρησιμοποιήσαμε.

Για να επιτύχουμε την προσθήκη μεγάλου αριθμού διεργασιών στα προβλήματά μας, δημιουργήσαμε ένα καθολικό πίνακα (global array) που ονομάζεται **task** που είναι τύπου **task_t**, με κάθε είσοδό του να αντιστοιχεί σε ένα task του προβλήματός μας. Ο δείκτης του αριθμού στοιχείου του πίνακα αντιστοιχεί στον αναγνωριστικό του αριθμό.

```
typedef struct {
        time_t   initial_offset ;
        time_t   min_period;
        time_t   max_period;
        time_t   offset;
        time_t   deadline;
        time_t   bcet;
        time_t   wcet;
        r_id     resource;
        int      pri;
} task_t;
```

Κώδικας 4.2.1 Δομή ενός task

Η τελευταία δομή δεδομένων είναι η **buffer_t** που αποτελεί την κεντρική δομή που περιγράφει τους πόρους του εκάστοτε προβλήματος. Ο κάθε πόρος του προβλήματος διαθέτει ένα buffer το οποίο κράτα όλες τις πληροφορίες για τον χρόνο εκτέλεσης των διεργασιών αλλά και για την ταξινόμηση που θα χρειαστεί να γίνει στις διεργασίες, ώστε να εκτελεσθούν σωστά , σύμφωνα με τον κατάλληλο αλγόριθμο χρονοδρομολόγησης που έχουμε εφαρμόσει. Το στοιχείο buffer αποτελείται από μία ακέραια μεταβλητή length που έχει σύνολο τιμών από 0 έως τον αριθμό των tasks που υπάρχουν στο πρόβλημα αλλά και ένα πίνακα που ονομάζεται element και είναι length θέσεων με τύπο δεδομένων t_id. Το buffer παρατίθεται στο παρακάτω σχήμα:

```
typedef struct {
  int [0,Tasks] length;
 t_id element [Tasks];
 } buffer_t;
```

Κώδικας 4.2.2 Buffer

4.3 Task Template

Η μοντελοποίηση διεργασίας στο UPPAAL ικανοποιεί περιπτώσεις τόσο περιοδικών όσο και μη περιοδικών tasks. Αυτό προσδιορίζεται μέσω μίας καθολικής δυαδικής-Boolean μεταβλητής η οποία ονομάζουμε **Periodic.** Αυτή η μεταβλητή εξετάζεται στο μοντέλο του task για να εξασφαλίσει αν εν τέλη η υπό εξέταση διεργασία έχει περιοδική η μη περιοδική συμπεριφορά. Παρακάτω παρατίθεται το τελικό πρότυπο των tasks (βασισμένη στην αρχική ιδέα) που δέχεται σαν είσοδο μία μονή παράμετρο, τον αναγνωριστικό αριθμό του κάθε task, που χρησιμοποιείται για δείκτης στον πίνακα που τα στοιχεία του είναι tasks. Σημειώνεται επίσης πως λόγω ευκολίας έχουμε δημιουργήσει νέες συναρτήσεις στα local declarations του task που μας διευκολύνουν να πάρουμε άμεσα τις τιμές των πεδίων του task[id] (πχ η InitialOffset() μας επιστρέφει την τιμή initial_offset του task με αναγνωριστικό αριθμό id). Στη συνέχεια θα εξηγήσουμε τα σημαντικά locations της τοπολογίας αλλά και το κάθε κομμάτι κώδικα της υλοποίησης.

- **Initial (αρχική):** Το task περιμένει να περάσει initial offset χρονική διάρκεια για να αλλάξει κατάσταση. Παρατηρούμε ότι στα global declarations του προβλήματος έχει οριστεί πίνακας ρολογιών που η κάθε του είσοδος αποτελεί το ρολόι του κάθε task.Έτσι αυτός ο πίνακας παίρνει σαν όρισμα την μονή παράμετρο id τύπου t_id. Η συνθήκη για το location ορίζεται μέσο invariant και το guard στην ακμή ανανεώνει το ρολόι της διεργασίας σε initial offset χρονικές μονάδες.

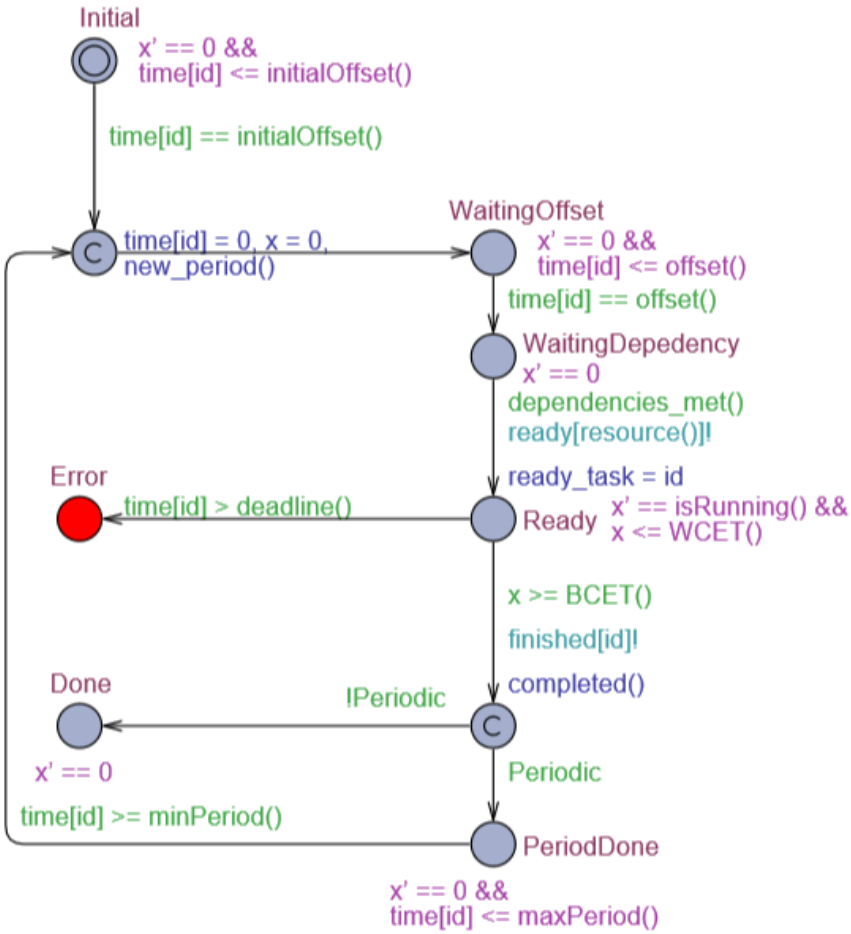
- **Waiting:** Αυτή η τοποθεσία χωρίζεται σε δύο υποπεριπτώσεις. Η πρώτη είναι το WaitingOffset που η διεργασία περιμένει το offset περιόδου να περάσει και η δεύτερη είναι το WaitingDependencies που η διεργασία ελέγχει τις αλληλεξαρτήσεις που έχει με άλλες διεργασίες (Θα επιστρέψουμε στο κομμάτι των αλληλεξαρτήσεων

παρακάτω). Αντίστοιχα με την Initial και εδώ σε κάθε τοποθεσία έχουμε Invariants και η μεταφορά από ένα location σε ένα άλλο πραγματοποιείται μέσω του φρουρού (guard).

- **Ready:** Σε αυτό το σημείο το task είτε εκτελείται είτε περιμένει , έτοιμο προς εκτέλεση, στον αρμόδιο πόρο (resource). Παρατηρούμε πως Για να εισέλθουμε στην κατάσταση Ready Έχει δωθει σήμα readyμέσο urgent channel , παίρνοντας παράμετρο το r_id (αναγνωριστικό αριθμό του αντίστοιχου αρμόδιου resource για το task). Μέσο αυτού του synchronization channel το task επικοινωνεί με το resource ώστε να ξεκινήσει να έρχεται σε λειτουργία.

- **Error:** Μια διεργασία φτάνει σε αυτή την τοποθεσία μόνο σε περίπτωση που είναι περιοδική και δεν κατάφερε να τελειώσει την εκτέλεσή της πριν το χρόνο προθεσμίας της.

- **Done:** Αντίστοιχα και με την περίπτωση του Waiting , και εδώ η τοποθεσία χωρίζεται σε δύο υποπεριπτώσεις. Όταν μία διεργασία καταφέρει να φτάσει στο Done location σημαίνει πως πρόλαβε να εκτελεσθεί πριν τον χρόνο προθεσμίας της. Αν το task δεν είναι περιοδικό τότε η διεργασία φτάνει στο Done και τελειώνει ενώ αν όχι η διεργασία μεταβαίνει στο PeriodicDone που σηματοδοτεί την ορθή εκτέλεση και λήξη της περιόδου του task. Εν συνεχεία θα μεταβεί στο committed location και θα εκκινήσει νέα περίοδος.



Σχήμα 4.3 Task model

Το κάθε task που δημιουργείται χρησιμοποιεί δύο μεταβλητές ρολογιού που ονομάζονται **time** και **x,** που το πρώτο αντιπροσωπεύει το χρόνο που έχει περάσει από την αρχή της τρέχουσας περιόδου ενώ το δεύτερο αντιπροσωπεύει το χρόνο εκτέλεσης της διεργασίας στην τρέχουσα περίοδο. Το **x** αποτελεί τοπική μεταβλητή ενώ, όπως αναφέρθηκε, το **time** μια καθολική, εξού και ο λόγος που χρησιμοποιείται πίνακας για τις μεταβλητές **time.** Λόγω του ότι είχαμε την δυνατότητα να εργαστούμε στο UPPAAL version 4.1.15 μας δίνεται η δυνατότητα να χρησιμοποιήσουμε χρονισμένα αυτόματα με χρονόμετρο (μεταβλητές x' στα invariants των τοποθεσιών) [9]. Έτσι το **x**έχει τιμή μηδέν σε όλες τις τοποθεσίες εκτός από την Ready, καθώς μόνο εκεί υπάρχει χρόνος εκτέλεσης της διεργασίας άρα μόνο εκεί έχει νόημα το ρολόι **x**. Ο έλεγχος για το άν εν τέλη εκτελείται η διεργασία συμβαίνει μέσο του λογικού ελέγχου x'== isRunning() &&x<= WCET(), που υποδηλώνει πως θέλουμε να προσμετράται χρόνος όσο αυτός είναι (το πολύ ίσος με) worst case execution time αλλά και μέσο της τοπικής διεργασίας IsRunning(). Η IsRunning() παίρνει τιμές 0 ή 1 (Boolean) και το μόνο που κάνει είναι να επιστρέφει την τιμή του στοιχείου element[r_id] του buffer. Αν αυτή η τιμή ισούται με το id της διεργασίας τότε η συνάρτηση επιστρέφει 1 (true) και άρα βρίσκεται υπό εκτέλεση. Στα παρακάτω σχήματα παρατίθεται τόσο η σχηματική απεικόνιση του μοντέλου όσο και τα αντίστοιχα κομμάτια κώδικα. Από την άλλη η μεταβλητή **time** τρέχει πάντοτε και

ανανεώνεται στην τιμή μηδέν σε κάθε περίοδο, έτσι ώστε όταν περάσει το χρόνο προθεσμίας το task μεταβαίνει στην τοποθεσία Error.

Με το που μία διεργασία εισέρχεται στην κατάσταση Ready, η διεργασία απευθείας ανανεώνει την μεταβλητή ready_task με τον αναγνωριστικό αριθμό της για να μεταδώσει μέσο σήματος στον πόρο πιο task θα δρομολογηθεί. Αργότερα θα δούμε πως το συγκεκριμένο id number χρησιμοποιείται από τον πόρο για την εκτέλεση των διεργασιών στην ουρά.

Για να συγκεκριμενοποιήσουμε το πρόβλημά μας με περαιτέρω περιορισμούς εισάγουμε τρείς τοπικές συναρτήσεις τις **new_period**, **dependencies_met** και **completed** που δίνουν Boolean αποτέλεσμα (true-false)και οι οποίες επίσης παρατίθενται παρακάτω. Η πρώτη εκτελείται στις ακμές που οδηγούν στις Waiting καταστάσεις και χρησιμοποιείται έτσι ώστε να οριστεί νέα περίοδος στις δομές δεδομένων και στους μετρητές, ενώ η τρίτη εκτελείται στις ακμές φεύγοντας από την κατάσταση Ready. Η **dependencies_met** συνάρτηση εκτελείται στο φρουρό της κατάστασης Waiting Dependencies (καθώς μεταβαίνουμε στο Ready) και δίνει τιμές true αν υπάρχει αλληλεξάρτηση μεταξύ tasks και false αν δεν υπάρχει.

```
int [0,1]  isRunning() { return (buffer [resource () ]. element[0] == id? 1:0);}
```

Κώδικας 4.3.1 Συνάρτηση IsRunning()

```
bool dependencies_met ()  {
        return forall ( j : t_id )  Depend[id] [j] imply complete[j];
}

void completed () {
        complete[id] = true;
}

void new period () {
        // int I = 0;
        // for ( I = 0; I < Tasks; i++) {
        // complete[i] = false;
        // }
        complete [id] = false;
}
```

Κώδικας 4.3.2 Boolean Συναρτήσεις new_period(),dependencies_met() και completed()

4.4 Μοντελοποίηση τον Γραφημάτων Αλληλοεξάρτησης-TaskGraph

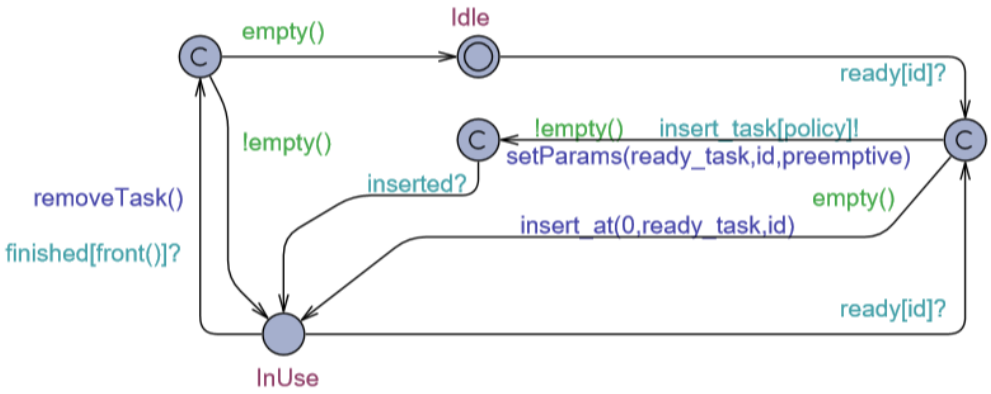
Για να μοντελοποιήσουμε τα γραφήματα αλληλεξάρτησης μεταξύ των διεργασιών χρειαζόμαστε μια καθολική δομή που θα περιέχει τις εξαρτήσεις τους. Το διάγραμμα μοντελοποείται χρησιμοποιώντας ένα τετραγωνικό Boolean πίνακα εξαρτήσεων που η κάθε είσοδος *(i,j)* υποδηλώνει αν η διεργασία *i*εξαρτάται από τη διεργασία *j*, όπως φαίνεται και στο σχήμα παρακάτω. Στο παράδειγμα έχουμε ένα πίνακα τεσσάρων διεργασιών t_0 , … , t_3 που η διεργασία 0 εξαρτάται από τη διεργασία 1 που εξαρτάται από τη διεργασία 2 που εξαρτάται από τη διεργασία 3. Επιπλέον, όπως αναφέρθηκε και πριν έχουμε ένα δυαδικό – Boolean πίνακα-μεταβλητή complete που ελέγχει αν μία διεργασία έχει τελειώσει την εκτέλεσή της. Η κάθε συνάρτηση είναι αρμόδια να επαναφέρει τη συγκεκριμένη είσοδο αυτού του πίνακα στην αρχή κάθε περιόδου, που γίνεται μέσω της συνάρτησης **new_period**. Η τιμή της μεταβλητής complete γίνεται true (καθώς είναι Boolean) όταν μία διεργασία τελειώνει την εκτέλεσή της στην συνάρτηση **completed**. Τελικά, οι διεργασίες δεν μπορούν να μεταβούν στην κατάσταση Ready από την κατάσταση WaitingDependency έως ότου η συνάρτηση **dependencies_met** πάρει την τιμή true. Αυτή η συνάρτηση επαναλαμβάνει την αντίστοιχη γραμμή του πίνακα αλληλεξαρτήσεων και βεβαιώνει πως άμα η διεργασία εξαρτάται από μία άλλη, η είσοδος του πίνακα complete για αυτή τη διεργασία πρέπει να είναι true. Σημειώνεται πως απαιτείται επιπλέον προσοχή όταν θέλουμε να προσδιορίσουμε την περίοδο των διεργασιών χρησιμοποιώντας το γράφο αλληλεξάρτησης καθώς η έννοια της εξάρτησης δεν προσδιορίζει το συγχρονισμό των περιόδων των διεργασιών. Η παραπάνω χρήση των γράφων αλληλεξάρτησης επαναφέρουν τις τιμές της μεταβλητής-πίνακα complete σε κάθε νέα περίοδο που είναι ασφαλές για διεργασίες που εξαρτώνται και έχουν ίδια περίοδο ενώ δεν είναι απαραίτητα ασφαλές για διεργασίες που δεν έχουν ίδια περίοδο. Συνεπώς ένας καλός κανόνας καθίσταται η χρήση του γραφήματος task graph μόνο σε περίπτωση που δύο διεργασίες έχουν εξάρτηση μεταξύ τους και έχουν ίδια περίοδο.

```
//Global Declaration
const bool TaskGraph[Tasks][Tasks] = {
{ 0,1,0,0 },
{ 0,0,1,0 }
{ 0,0,0,1 }
{ 0,0,0,0 }
};
bool complete [Tasks];
```

Κώδικας 4.4.1 Task Graph και μεταβλητή-πίνακας complete

4.5 Μοντελοποίηση Resource

Η μοντελοποίηση του resourceπου χρησιμοποιήσαμε βασίστηκε στην αρχική γενική ιδέα του resource και επομένως δεν έχει καμία διαφορά για χρονοδρομολογητές σε διακοπτόμενη η μη-διακοπτόμενη χρονοδρομολόγηση. Όπως είναι φανερό και από το σχήμα του resource παρακάτω, η μεγαλύτερη διαφορά μεταξύ της αρχικής ιδέας και της τελικής υλοποίησης έγκειται στο ότι το τελικό resource δεν συμπεριλαμβάνει ρολόγια καθώς η χρονική συμπεριφορά είναι καθαρά αρμοδιότητα των διεργασιών (Το μοντέλο task διαθέτει δύο ρολόγια , ένα global και ένα local για το κάθε task που δημιουργείται).



Σχήμα 4.5 απεικόνιση του resource

Το resource δέχεται σαν είσοδο τρείς παραμέτρους, τον αναγνωριστικό αριθμό του resource που θα ενεργοποιηθεί, την Boolean μεταβλητή preemt που αν η τιμή της είναι true υποδηλώνει preemptive scheduling και τέλος μία ακέραια μεταβλητή τύπου policy_t που η τιμή της προσδιορίζει τον αλγόριθμο χρονοδρομολόγησης που θα εφαρμοστεί. Το πρότυπο του resource διαθέτει δύο βασικές τοποθεσίες Idle και InUse που σε κάθε χρονική στιγμή φανερώνουν την κατάστασή του. Τα resource που βρίσκονται στην τοποθεσία Idle περιμένουν διεργασίες να τους σηματοδοτήσουν ότι είναι έτοιμες προς εκτέλεση μέσω του καναλιού συγχρονισμού ready. Όταν ένα τέτοιο σήμα φτάσει στο resource είτε από την τοποθεσία Idle είτε από την InUse, ο πόρος μας (επεξεργαστής) θα τοποθετήσει την διεργασία στην πρώτη θέση του buffer αν αυτός είναι άδειος. Το αν ο buffer είναι άδειος εξετάζεται με την τοπική συνάρτηση **empty** οποία διαβάζει το μήκος του αντίστοιχου buffer. Μία διεργασία εισέρχεται στον buffer μέσω της καθολικής συνάρτησης **insert_at** (ορίζεται παρακάτω) και δέχεται σαν ορίσματα τη θέση του buffer που θέλουμε να εισαχθεί η νέα διεργασία, τον αναγνωριστικό αριθμό της διεργασίας και τον αναγνωριστικό αριθμό του αρμόδιου πόρου. Η insert_at μεταφέρει όλες τις διεργασίες του buffer μία θέση πίσω από τη θέση εισόδου τους και στη συνέχεια εισαγάγει την νέα διεργασία στην επιθυμητή θέση ενω παράλληλα ανανεώνει την τιμή του μήκους του buffer.

Σε περίπτωση που έχουμε τουλάχιστον μία διεργασία στο buffer και θέλουμε να εισάγουμε μία νέα η αρχή χρονοδρομολόγησης που θα ακολουθήσουμε θα αποφασίσει σε ποιο σημείο του buffer θα εισαχθεί η νέα αυτή διεργασία (σύμφωνα δηλαδή με τον αλγόριθμο χρονοδρομολόγησης που χρησιμοποιούμε). Η κάθε αρχή χρονοδρομολόγησης

(και κατ' επέκταση ο κάθε αλγόριθμος χρονοδρομολόγησης) που ακολουθούμε είναι ένα ξεχωριστό μοντέλο και ενεργοποιείται μέσω του καναλιού συγχρονισμού insert_task με όρισμα policy. Προκειμένου η αρχή χρονοδρομολόγησης που χρησιμοποιούμαι να αποφασίσει σε ποιο σημείο του buffer πρέπει να τοποθετήσει τη διεργασία οφείλει να γνωρίζει τους αναγνωριστικούς αριθμούς για τη διεργασία, για τον πόρο και αν ο πόρος υποστηρίζει διακοπτόμενη ή μη-διακοπτόμενη χρονοδρομολόγηση. Οι πληροφορίες αυτές μεταφέρονται στο μοντέλο επιλογής αλγορίθμου χρονοδρομολόγησης μέσω της συνάρτησης **SetParams** η οποία τις αντιγράφει σε meta-μεταβλητές που μπορούν να διαβαστούν από το χρονοδρομολογητή.

```
void    insert_at ( int[ 0 , Tasks] pos, t_id tid, r_id rid )        {
        int  I;
        for ( I = buffer [ rid ] . length; I > pos ; I -- ) {
                buffer [ rid ] . element [ I ] = buffer [ rid ] . element [ I – 1] ;
        }
        buffer [ rid ] . element [ pos ] = tid;
        buffer [ rid ] . length++ ;
}
```

Κώδικας 4.5.1 Συνάρτηση insert_at

Το resource παραμένει στο committed location περιμένοντας το σήμα inserted , που υποδηλώνει πως η διεργασία έχει εισαχθεί στον buffer σύμφωνα με τις προϋποθέσεις της εφαρμοζόμενης αρχής χρονοδρομολόγησης. Αυτή η τοποθεσία είναι committed για να εξασφαλίσει πως όλες οι συναρτήσεις που εισάγουν διεργασίες στο buffer είναι ατομικές, δηλαδή δεν θα υπάρξουν μεταξύ τους προβλήματα όπως deadlocks η race conditions. Σημειώνεται πως ενώ τα deadlocks και τα race conditions είναι έννοιες που απασχολούν σε μεγάλο βαθμό την χρονοδρομολόγηση διεργασιών, δεν έχει νόημα μια τέτοια έννοια σε διαδικαστικές λειτουργίες όπως το task insertion). Όταν η διεργασία εισέρχεται στο buffer σύμφωνα με την αρχή χρονοδρομολόγησης που επιλέξαμε, το resource μεταφέρεται στην κατάσταση InUse. Μένει εκεί μέχρι να του δοθεί σήμα από το κανάλι finished πως η τρέχουσα διεργασία ολοκλήρωσε την εκτέλεσή της είτε όταν μία νέα διεργασία δώσει σήμα πως είναι έτοιμη. Στην πρώτη περίπτωση το resource αφαιρεί τη διεργασία από το buffer με την τοπική συνάρτηση **removeTask** (παρατίθεται σε σχήμα παρακάτω) ενώ στην τελευταία περίπτωση το resource επαναλαμβάνει όλη την διαδικασία που περιγράψαμε μέχρι στιγμής. Ανάλογα με το αν ο buffer του resource είναι άδειος το resource επιστρέφει είτε στην Idle είτε στην InUse κατάσταση. Το παραπάνω μοντέλο μπορεί να χρησιμοποιηθεί για να περιγράψει μια μεγάλη γκάμα από πόρους ενώ παράλληλα υποστηρίζει την εισαγωγή αλγορίθμων χρονοδρομολόγησης στο μοντέλο με ελάχιστη επιβάρυνση.

```
void    removeTask() {
                int I = 0;
                buffer [ id ] . length -- ;
        do {
        buffer [ id ] . element [I ] = buffer [ id] . element [I +1];
                i++;
```

```
        } while ( I < buffer [ id ] . length );
        buffer [ id ] . element [ buffer [ id ] . length ] = 0;
}
```

Κώδικας 4.5.2 removeTask()

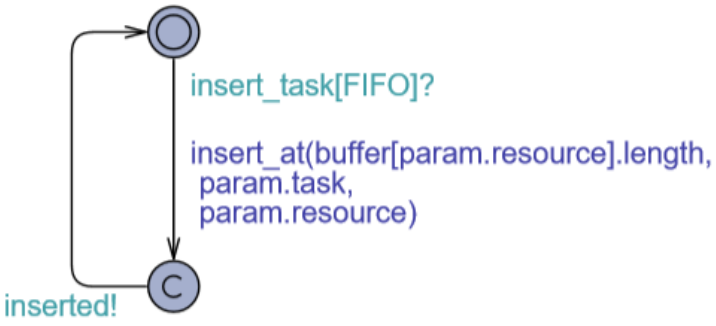
4.6 Ανάπτυξη Μοντέλων Αλγορίθμων Χρονοδρομολόγησης

Σε αυτή την ενότητα γίνεται περιγραφή μοντέλων τριών αλγορίθμων χρονοδρομολόγησης: του FIFO, του FPS και του EDF. Το κοινό όλων των αυτών των αλγορίθμων στα μοντέλα μας είναι πως καλούνται μόνο όταν μία διεργασία βρίσκεται ήδη στο resource buffer ενώ μία νέα θέλει να εισέλθει. Αυτή η συνθήκη ικανοποιείται στο resource όταν αυτό καλεί τη συνάρτηση που εφαρμόζει αλγόριθμο χρονοδρομολόγησης μόνο σε αυτή την περίπτωση.

Οι παράμετροι που απαιτούνται για να καθορίσουν την κατάλληλη αρχή χρονοδρομολόγησης που θα χρησιμοποιηθεί μεταφέρονται στα νέα μοντέλα (αλγορίθμων) μέσω της συνάρτησης **setParams** και αποθηκεύονται στην meta-μεταβλητή **param**. Με αυτόν τον τρόπο οι παράμετροι που απαιτούνται είναι προσβάσιμοι στα scheduling policies χωρίς να διογκώνουμε τον χώρο καταστάσεων, καθώς οι meta-μεταβλητές αγνοούνται για συγκρίσεις καταστάσεων. Οι παράμετροι που μεταφέρονται στο policy είναι οι **param.task** **param.prempt** και **param.resource**.

4.6.1 FIFO

Το πιο απλοϊκό scheduling policy σε υλοποίηση αποτέλεσε το FIFO καθώς οι διεργασίες εισέρχονται στο buffer ακριβώς με τη σειρά που φτάνουν. Αυτό υποδηλώνει πως το FIFO scheduling policy αγνοεί αν ο πόρος μας είναι διακοπτόμενος η μη διακοπτόμενος. Στο ακόλουθο στχήμα παρατίθεται η μοντελοποίηση του FIFO κατά την οποία η είσοδος του task στον buffer πραγματοποιείται μέσω της συνάρτησης insert_task. Το resource μέσω του synchronization channel insert_task ενεργοποιεί το παρακάτω μοντέλο το οποίο κατά την πρώτη του μετάβαση χρησιμοποιεί την συνάρτηση insert_at με ορίσματα τις μεταβλητές του setParams. Έτσι το task εισέρχεται στον buffer και το παρακάτω μοντέλο επαναρχικοποιείται.



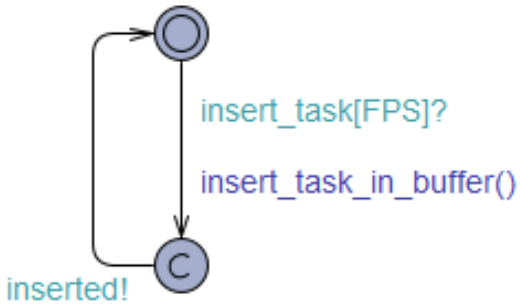
Σχήμα 4.6.a FIFO υλοποίηση στο UPPAAL

4.6.2 Fixed Priority

Το μοντέλο FPS είναι παρόμοιο με αυτό του FIFO (δηλαδή και σε αυτό η τοποθεσία του task στον buffer λαμβάνει χώρα στην ακμή των δύο τοποθεσιών του) με τη διαφορά πως η εισαγωγή του task στον buffer πραγματοποιείται μέσω της συνάρτησης insert_task_in_buffer αντί της insert_task. Η συνάρτηση insert_task_in_buffer κάνει επαναλήψεις (γραμμές 6-8) συγκρίνοντας την προτεραιότητα του ερχόμενου task με την προτεραιότητα του task στον buffer και θέτει μία μεταβλητή place τέτοια ώστε το ερχόμενο task να έχει μικρότερη προτεραιότητα συγκριτικά με τα tasks μπροστά του και μεγαλύτερη προτεραιότητα συγκριτικά με τα tasks πίσω του. Σε περίπτωση που το task που εισέρχεται στον buffer έχει την μικρότερη προτεραιότητα σε σχέση με όλα τα υπόλοιπα tasks τότε το while-loop σταματά και το task τοποθετείται στο τέλος του buffer. Προφανώς προκειμένου να λειτουργήσει αυτή η υλοποίηση απαιτείται ο buffer να είναι ταξινομημένος. Ωστόσο αυτό δεν αποτελεί πρόβλημα καθώς η μέθοδος αυτή χρησιμοποιείται για κάθε εισαγωγή νέου task (εκτός από το πρώτο) άρα ο buffer είναι μονίμως ταξινομημένος[10]. Στη γραμμή 9 της συνάρτησης η θέση τοποθέτησης του task έχει υπολογιστεί και έτσι καλείται η γνωστή insert_at για να εισάγει το task στην επιθυμητή αυτή θέση. Σημειώνεται πως στη γραμμή 4 της συνάρτησης γίνεται έλεγχος για το αν διαθέτουμε διακοπτόμενης ή μη διακοπτόμενης χρονοδρομολόγησης. Σε περίπτωση διακοπτόμενης το εισερχόμενο task τοποθετείται στην πρώτη καταχώρηση του buffer ή αλλιώς σε περίπτωση μη διακοπτόμενης, στην θέση με τον δεύτερο δείκτη πίνακα.

```
void insert_task_in_buffer () {
        t_id     t = param.task;
        r_id     r = param.resource;
        int      place = ( param.preempt ? 0 : 1);
        int      i ;
        while (place < buffer [r] . length &&
                task [ buffer [ r ] . element [place ]] . pri >= task[t] . pri {
        place++;
        }
          Insert_at ( place , t , r );
}
```

Κώδικας 4.6.α συνάρτηση insert_task_in_buffer()



Σχήμα 4.6.b υλοποίηση FPS

4.6.3 Earliest Deadline First-EDF

Θεωρητικά στα δύο προηγούμενα παραδείγματα FIFO και FPSδεν χρειάζεται η υλοποίηση δύο διαφορετικών μοντέλων καθώς η εισαγωγή του task στον buffer πραγματοποιείται μέσω μίας απλής κλήσης συνάρτησης. Ωστόσο καθώς η γλώσσα μοντελοποίησης του UPPAAL έχει πολλούς περιορισμούς δεν μας επιτρέπει την υλοποίηση του EDF με τη χρήση μιας απλής κλήσης συνάρτησης, αλλά επιβάλλει την υλοποίηση νέου μοντέλου. Το πρόβλημα έγκειται στον καθορισμό του πόσο μακριά βρίσκεται ένα task, έστω i, από το χρόνο προθεσμίας του, δηλαδή πρέπει να εξετάσουμε την διαφορά μεταξύ task[i].deadline και time[i]. Το UPPAAL δεν επιτρέπει μία τέτοια σύγκριση σε κλήσεις συναρτήσεων καθώς χρησιμοποιούμε στοιχεία από τη δομή δεδομένων ρολογιών. Βέβαια οποιαδήποτε τέτοια σύγκριση είναι δυνατόν να πραγματοποιηθεί σε έκφραση φρουρού (guard expression) και έτσι προκειμένου να βρούμε την θέση που πρέπει να τοποθετηθεί το νέο task στο resource buffer,δημιουργούμε βρόχους επανάληψης μεταξύ των τοποθεσιών του μοντέλου έτσι ώστε να πραγματοποιηθεί ο έλεγχος.

Για να αποφανθούμε στο αν ένα εισερχόμενο task i έχει μικρότερο χρόνο προθεσμίας από ένα task j στο buffer πρέπει να ελέγξουμε το εξής:

$$\text{task[i].deadline - time[i]} < \text{task[j].deadline – time[j]} \leftrightarrow$$
$$\text{time[i] – task[i].deadline} > \text{time[j] – task[j].deadline}$$

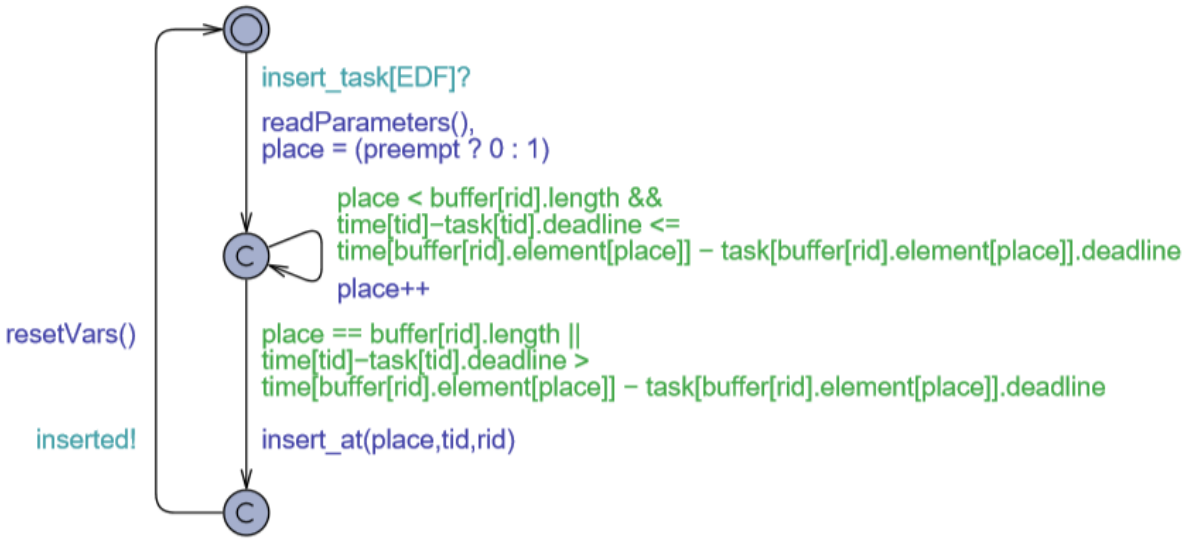
Εδώ οφείλουμε να σημειώσουμε πως το UPPAAL δεν επιτρέπει αφαίρεση τιμών ρολογιού από σταθερές εξού και η αλλαγή της ανισότητας στη δεύτερη γραμμή.

Στο παρακάτω σχήμα παρουσιάζονται μερικές χρήσιμες τοπικές μεταβλητές και συναρτήσεις που βοήθησαν στην υλοποίηση του μοντέλου EDF. Στις μεταβλητές αποθηκεύονται οι παράμετροι που μεταφέρονται από το resource, η συνάρτηση readParameters είναι αρμόδια για να ορίσει τις τιμές των τοπικών μεταβλητών και η συνάρτηση resetVars χρησιμοποιείται επαναρχικοποιεί τις τιμές των μεταβλητών μετά την είσοδο του task ώστε να ελαχιστοποιήσουμε τον χώρο αναζήτησης. Αρχικά το μοντέλο EDF διαβάζει τις παραμέτρους του resource (μέσω της readParameters) και θέτει κατάλληλα την τιμή της μεταβλητής place ανάλογα με το αν το καλούμενο resource είναι διακοπτόμενο ή μη

διακοπτόμενο (ακριβώς ίδια διαδικασία με αυτή του FPS). Τώρα ο χρόνος προθεσμίας του εισερχόμενου task συγκρίνεται με τον χρόνο προθεσμίας του task που βρίσκεται στη θέση place στον buffer. Αν το place έχει περάσει το τελευταίο στοιχείο του buffer ή το εισερχόμενο task έχει μικρότερο χρόνο προθεσμίας απότι το task στη θέση place, τότε το νέο task εισέρχεται σε αυτή τη θέση του buffer με την χρήση της καθολικής συνάρτησης insert_at. Διαφορετικά, η μεταβλητή place αυξάνεται κατά ένα και η σύγκριση των προθεσμιών επαναλαμβάνεται. Εδώ σημειώνεται πως το συγκεκριμένο μοντέλο δεν αποτελεί τίποτα διαφορετικό από την υλοποίηση ενός απλού βρόχου επανάληψης while (όπως στο FPS) με συνθήκες φρουρών.

```
Int [0, Tasks] place;
t_id     tid;
r_id     rid;
bool     preempt;
void readParameters () {
        tid = param.task; rid = param.resource; preempt = param.preempt;
}
void resetVars () { place = tid = rid = 0; }
```

Κώδικας 4.6.3 local μεταβλητές στο μοντέλο EDF



Σχήμα 4.6.c EDF Μοντελοποίηση

4.7 Εγκατάσταση πειραματικού μέρους

Σε αυτό το σημείο γίνεται η επεξήγηση και δημιουργία του πειραματικού μέρους. Προκειμένου να δημιουργήσουμε ένα πρόβλημα χρονοδρομολόγησης πρέπει πρώτα να καταχωρήσουμε τιμές και δεδομένα στα Global Declarations και τα System Definitions του

UPPAAL simulator. Πρώτο μας μέλημα αποτελεί το αν το πρόβλημα θα είναι περιοδικό, πόσα tasks και resource διαθέτει, ο προσδιορισμός της μέγιστης σταθεράς χρόνου όλων των tasks αλλά και τα χαρακτηρίστηκα των tasks. Όταν ορίζουμε το σύστημα οφείλουμε να καθορίσουμε τις ιδιότητες των διαφορετικών resources δηλαδή αν είναι διακοπτόμενες ή μη διακοπτόμενες αλλά και ποιά αρχή χρονοδρομολόγησης θα ακολουθήσουν. Στο παρακάτω σχήμα παρουσιάζεται ένα παράδειγμα καθορισμού συστήματος με μία ποικιλία από resources (P1,P2,…). Όταν δηλώνουμε το σύστημα στα system declarations με την δεσμευμένη λέξη **system** τότε όλα τα ορισμένα resources θα συμπεριληφθούν στο μοντέλο με το αντίστοιχο scheduling policy που τους έχει δοθεί. Όλα τα tasks συμπεριλαμβάνονται με την δημιουργία μίας απλής μεταβλητής Task που υποδηλώνει των αριθμό τους. Αυτό οφείλετε στο γεγονός πως ο χώρος παραμέτρων των task είναι τύπος t_id άρα άμα δεν δοθεί συγκεκριμένη παράμετρος, το UPPAAL δημιουργεί αντίγραφα του task για κάθε δυνατό task id.

```
const bool      Periodic = .. ;  // Periodic Scheduling?
const int       Tasks = .. ;     //Number of Tasks
const int       Procs = .. ;     //Number of Resources
const int       MaxTime = .. ; //Maximum time constant
const task_t    task[Tasks] = { … } ;
```

Κώδικας 4.7.1 Δηλώσεις παραμέτρων

```
P1 = Resource( 0 , true/false , EDF/FPS/FIFO );
P2 = Resource( 1 , true/false , EDF/FPS/FIFO );
…
System Task P1, P2 , … , Policy_EDF, Policy_FPS, Policy_FIFO ;
```

Κώδικας 4.7.2 Παράδειγμα System Declarations

4.8 SchedulingQueries

Η σημαντικότερη ερώτηση που τίθεται σε ένα πρόβλημα χρονοδρομολόγησης είναι το αν οι διεργασίες πληρούν πάντα τους αντίστοιχους χρόνους προθεσμίας τους. Αυτό μπορεί πολύ εύκολα να ερωτηθεί μέσω του εργαλείου Verifier μέσω της φράσης:

A[] forall (i : t_id) not Task(i).Error

Με άλλα λόγια «Ισχύει πως κανένα task δεν βρίσκεται στο Error location σε όλα τα πιθανά/δυνατά σενάρια εκτέλεσης?». Με την προσθήκη χρονομέτρων το UPPAAL υλοποιεί μία υπερ ακριβή προσέγγιση του χώρου καταστάσεων που σημαίνει πως αν η παραπάνω πρόταση είναι αληθής τότε εξασφαλίζετε πως το σύστημα που δημιουργήθηκε είναι χρονοδρομολογήσιμο σε κάθε περίπτωση. Αν όμως η ερώτηση δεν ικανοποιείται σημαίνει πως βρέθηκε ένα αντι-παράδειγμα στην προσέγγιση αυτή. Το σύστημα μπορεί ακόμα να

είναι χρονοδρομολογήσιμο καθώς το αντι-παράδειγμα δεν είναι απαραίτητα μία εφικτή εκτέλεση του αρχικού συστήματος.

5 ΕΦΑΡΜΟΓΗ ΠΡΟΤΕΙΝΟΜΕΝΗΣ ΠΡΟΣΕΓΓΙΣΗΣ ΣΕ ΠΡΟΒΛΗΜΑΤΑ ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗΣ

Σε αυτή την ενότητα της εργασίας θα ασχοληθούμε με την εφαρμογή του γενικού μοντέλου του Κεφαλαίου 4 σε συγκεκριμένα παραδείγματα διαφορετικών περιπτώσεων. Όπως έχουμε προαναφέρει το γενικό μοντέλο αποτελείται από μία πληθώρα διεργασιών (task template) , από ένα έως τρία resource templates που θα είναι αρμόδια για την χρονοδρομολόγηση των διεργασιών και από αντίστοιχο αριθμό αλγορίθμων χρονοδρομολόγησης που θα εφαρμόζονται στο κάθε resource. Με αυτό τον τρόπο έχουμε τη δυνατότητα να εκτελέσουμε κάθε διαθέσιμο αλγόριθμο χρονοδρομολόγησης, είτε αυτό σημαίνει διακοπτόμενος (ή μη) είτε σημαίνει περιοδικός (ή μη), εξάγοντας σημαντικά συμπεράσματα σε κάθε περίπτωση. Αυτό επιτυγχάνεται είτε με την τροποποίηση των δομών δεδομένων που αφορούν τα χαρακτηρίστηκα των διεργασιών είτε με μικρή τροποποίηση του κώδικα των μοντέλων. Έτσι στις επόμενες υποενότητες θα αντιμετωπίσουμε ένα πρόβλημα χρονοδρομολόγησης FIFO, ένα πρόβλημα χρονοδρομολόγησης περιοδικών διεργασιών με EDF, ένα πρόβλημα διακοπτόμενης χρονοδρομολόγησης SRTF και τέλος ένα μεικτό-σύνθετο πρόβλημα που αποτελεί συνδυασμό ενός FPS και δύο FIFO αλγορίθμων. Με αυτό τον τρόπο καλύπτουμε ένα μεγάλο εύρος διαφορετικών περιπτώσεων χρονοδρομολόγησης αποδεικνύοντας την γενικότητα, προσαρμοστικότητα και ευελιξία του μοντέλου.

5.1 ΠΡΟΒΛΗΜΑ 1 (FIFO)

Ως πρώτο μοντέλο χρησιμοποιήσαμε ένα απλό παράδειγμα χρονοδρομολόγησης με policy First in – First out πέντε διεργασιών, με διαφορετικούς χρόνους εκτέλεσης, χωρίς στοιχεία περιοδικότητας ή προτεραιοτήτων αλλά και κοινούς χρόνους προθεσμίας. Σημειώνεται οτι οι διεργασίες που χρησιμοποιήσαμε είχαν διαφορετικό χρόνο άφιξης και δρομολόγησης η κάθε μία, έτσι σύμφωνα με τον αλγόριθμο FIFO η περάτωσή τους αναμένουμε να επέλθει με τη σειρά που δρομολογήθηκαν απο το resource στοιχείο του μοντέλου (δηλαδή με τη σειρά άφιξης). Οι χρόνοι άφιξης των διεργασιών task[0] – task[4] είναι (απο την πρώτη έως την πέμπτη διεργασία με τη σειρά) 1, 5, 10, 20, 15 ενώ οι χρόνοι εκτέλεσης είναι αντίστοιχα 6, 6, 10, 4, 5. Σύμφωνα με τις παραπάνω τιμές αναμένουμε την δρομολόγηση και περάτωση των διεργασιών με τη σειρά task[0] , task[1] , task [2] , task[4] , task[3]. Παρακάτω παρατίθεται τόσο ο κώδικας που ήταν αρμόδιος για τον ορισμό των τιμών των στοιχείων των διεργασιών και ττοπολογιών όσο και στιγμιότυπα της προσομοίωσης σε χρονικές στιγμές υψηλής σημασίας. Σημειώνεται πως το συγκεκριμένο παράδειγμα δεν έχει σκοπό να ελέγξει αλληλοεξαρτήσης μεταξύ διεργασιών.

Κώδικας

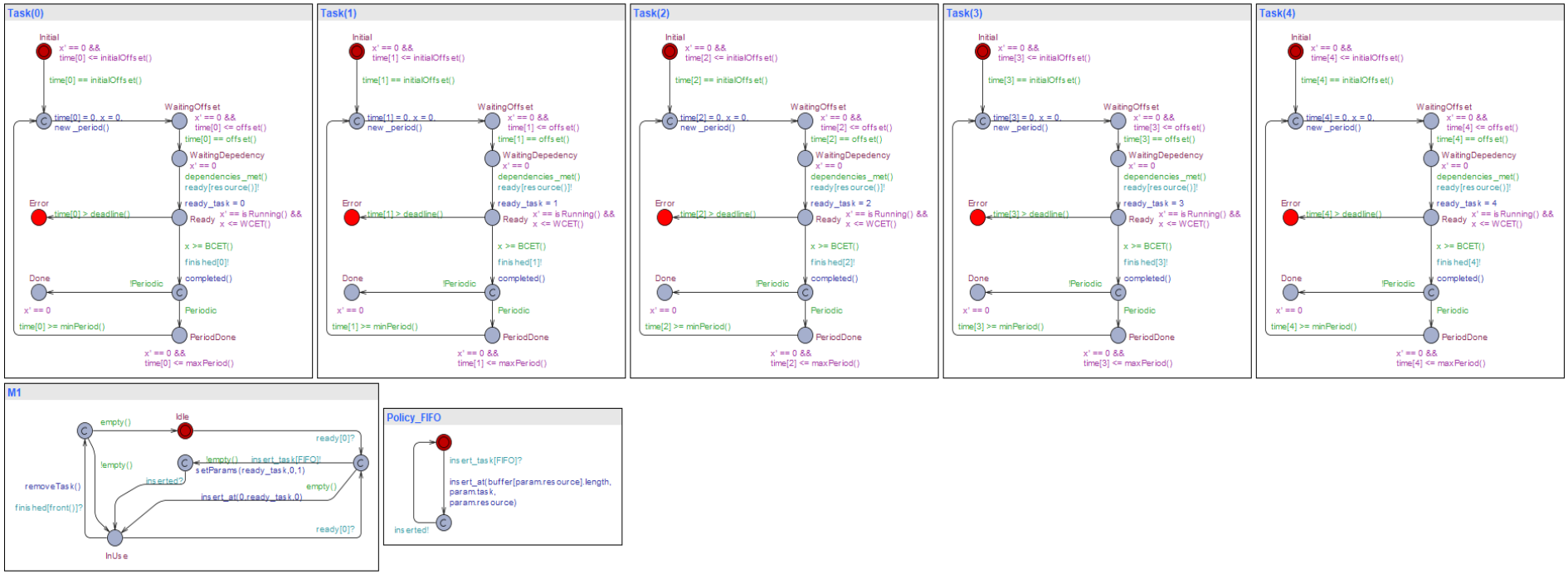
Declarations

```
const int Tasks =5; //number of Tasks
const taskt_t task[Tasks] = {
        { 1,  0 , 0 , 0 , 20 , 6 ,6 , 0 ,1 }
        { 5,  0 , 0 , 0 , 20 , 6 ,6 , 0 ,1 }
        { 10,  0 , 0 , 0 , 20 , 10 ,10 , 0 ,1 }
        { 20,  0 , 0 , 0 , 20 , 4 ,4 , 0 ,1 }
        { 15,  0 , 0 , 0 , 20 , 5 ,5 , 0 ,1 }
} ;
```

System Declarations

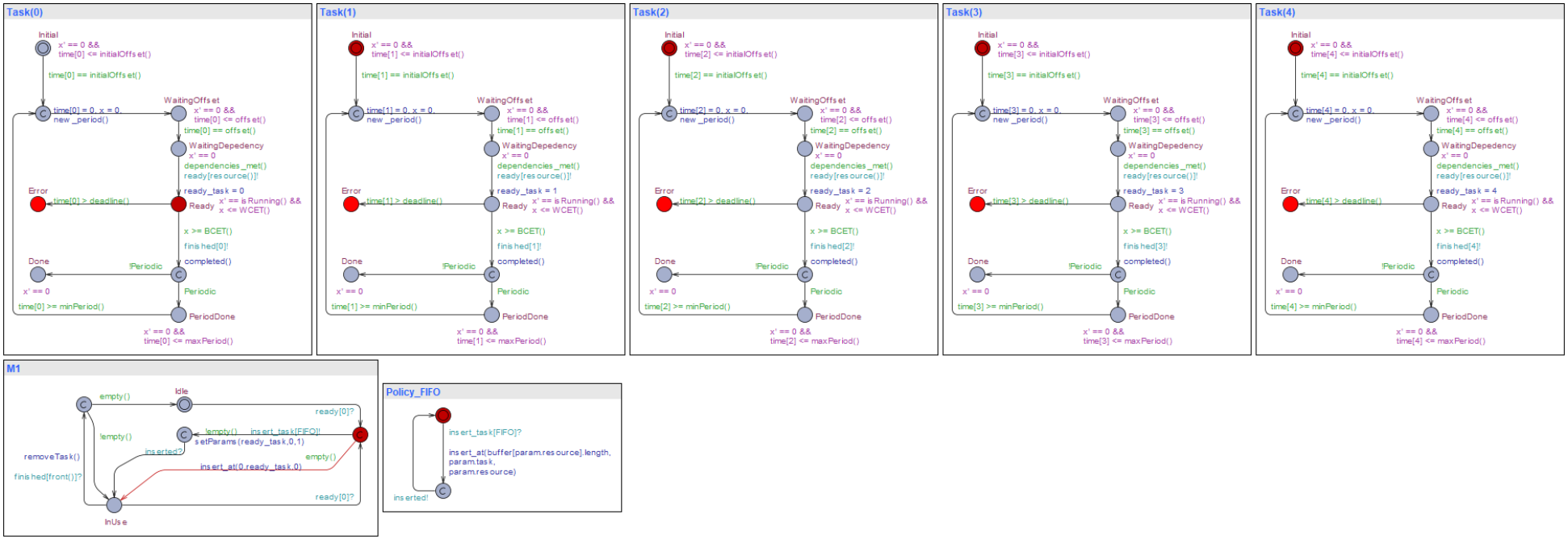
```
M1 = Resource(0,true,FIFO);
System  Task, M1, Policy_FIFO
```

Σημαντικά χρονικά στιγμιότυπα:

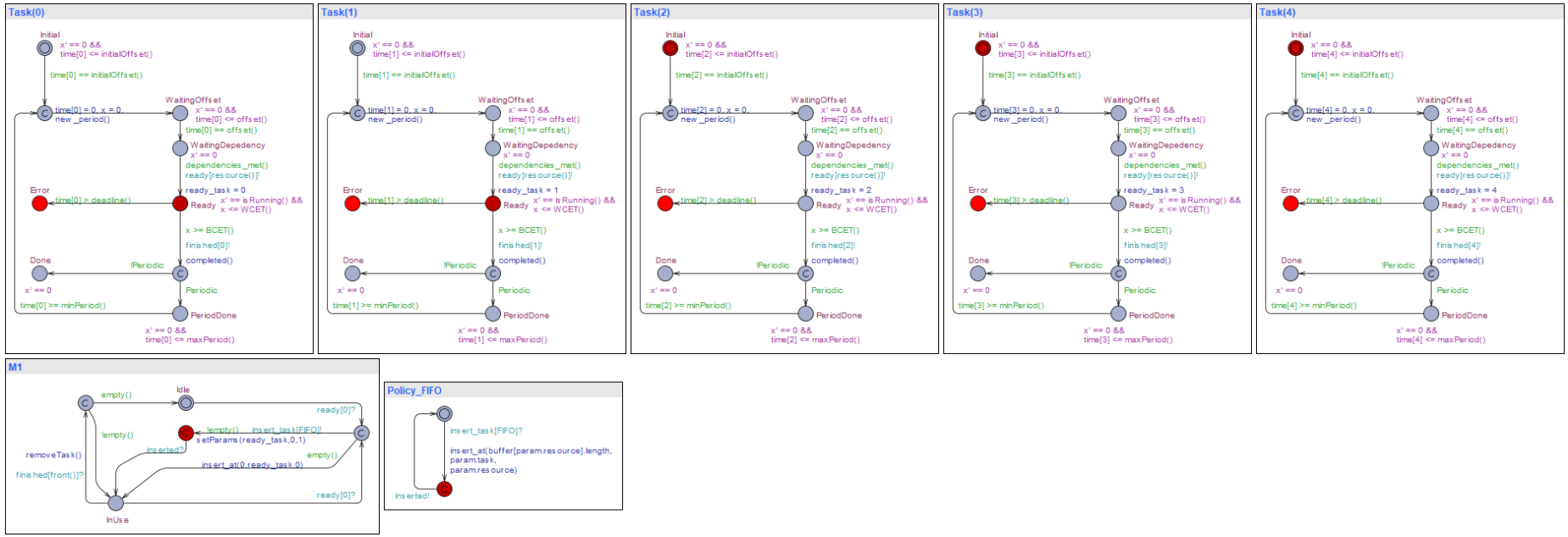


Στιγμιότυπο 5.1.1

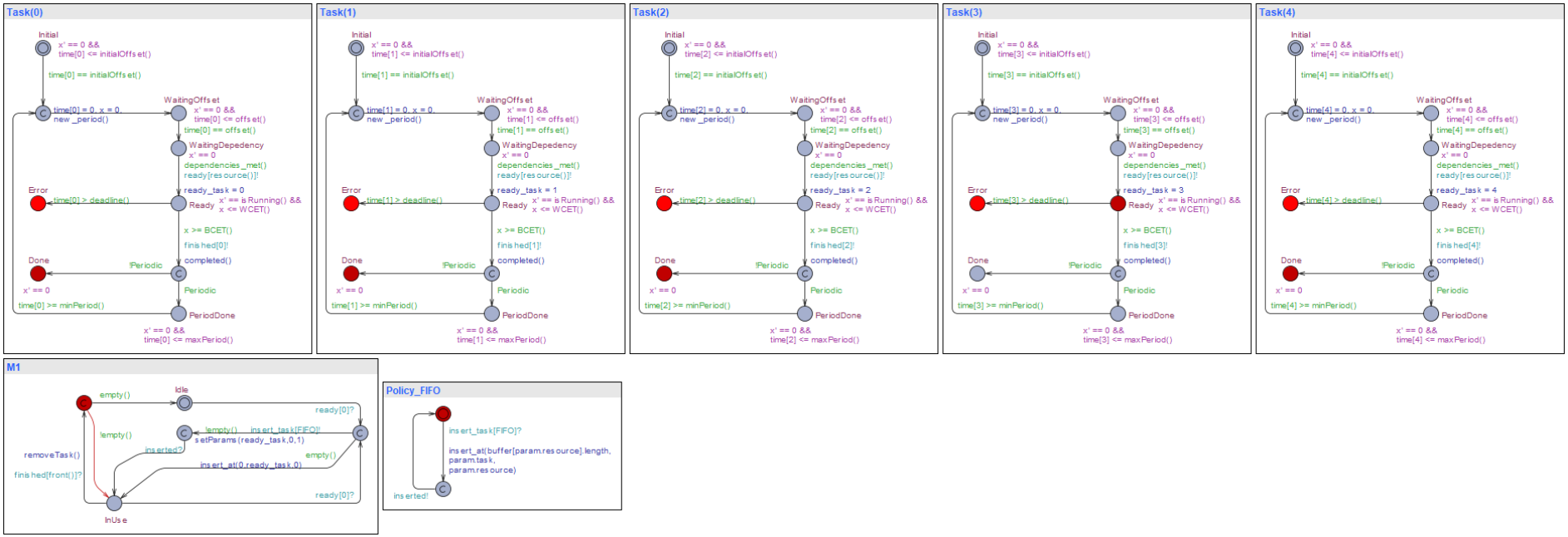
Σε αυτό το στιγμιότυπο πραγματοποιείται η εκκίνηση του πειράματος. Τα 5 tasks βρίσκονται στο initial location και το resource είναι idle.



Στιγμιότυπο 5.1.2



Στιγμιότυπο 5.1.3



Στιγμιότυπο 5.1.4

Στιγμιότυπο 5.1.2:

Όπως ήταν αναμενόμενο το task[0] (δηλαδή το πρώτο σε σειρά) είναι έτοιμο προς εκτέλεση (Ready location). Παράλληλα βλέπουμε πως το resource M1 εισάγει το task στον buffer ώστε να εκτελεσθεί με αποτέλεσμα το M1 να βρίσκεται στην κατάσταση InUse. Προσοχή ακόμα το policy δεν ενεργοποιείται καθώς δεν έχουμε scheduling conflict για να παρθεί απόφαση με βάση το FIFO.

Στιγμιότυπο 5.1.3:

Στη συνέχεια αφού είναι έτοιμο προς εκτέλεση το task[1] και δρομολογείται, βλέπουμε πως τοποθετείται στο buffer σύμφωνα με τις εντολές της τοπολογίας FIFO καθώς σε αυτόν παρευρίσκεται ήδη το task[0]. Έτσι τοποθετείται και αυτό στον buffer και δρομολογείται το επόμενο task. Όταν ο χρόνος εκτέλεσης που έχει οριστεί για κάθε task περατωθεί τότε το εκάστοτε task μεταβαίνει στην κατάσταση Done. Παρομοίως πραγματοποιείται η χρονοδρομολόγηση και εκτέλεση των υπόλοιπων tasks.

Στιγμιότυπο 5.1.4

Σε αυτό το στιγμιότυπο τα tasks [0] , [1] , [2] και [4] έχουν δρομολογηθεί τοποθετηθεί στο buffer και έχει τελειώσει η εκτέλεσή τους άρα μεταβαίνουν στην κατάσταση Done. Τελευταίο task που απέμεινε είναι το task[3] όπως και περιμέναμε. Εδώ το M1 αφού μόλις διέγραψε και το τελευταίο ολοκληρωμένο task από τον buffer παρατηρούμε πως ξαναπηγαίνει σε κατάσταση InUse για τελευταία φορά προς χάριν του task[3]. Όταν τελειώσει ο χρόνος εκτέλεσης του task[3] τότε ο M1 εκτελεί την removeTask() και ξαναμεταβαίνει σε κατάσταση Idle.

5.2 ΠΡΟΒΛΗΜΑ 2 (PERIODIC EDF)

Το δεύτερο πειραματικό μοντέλο που υλοποιούμε αφορά τον αλγόριθμο Earliest Deadline First ή αλλιως Αλγόριθμος συντομότερης χρονικής προθεσμίας. Σε αυτό το μοντέλο θα χρησιμοποιήσουμε πέντε περιοδικές διεργασίες, όπως και πριν, οι οποίες ανάλογα με τη σειρά άφιξής τους θα τοποθετούνται στην κατάλληλη θέση του buffer. Διεργασίες με υψηλές τιμές χρόνων προθεσμίας θα έχουν μικρότερη προτεραιότητα για να δρομολογηθούν σε εκτέλεση ενώ διεργασίες με μικρότερη χρονική προθεσμία θα έχουν υψηλότερη προτεραιότητα. Βασική θεώρηση για την πραγματοποίηση αυτού του πειραματικού σκέλους αποτέλεσε τόσο η σταθερή τιμή περιόδου για κάθε διεργασία όσο και η ισότητά της με τον χρόνο προθεσμίας της αντίστοιχης διεργασίας (έτσι έχουμε period[i] = deadline[i]). Αν δεν είχαμε υϊοθετήση αυτή τη θεώρηση το πρόβλημα θα «επιβαρυνόταν» με επιπλέον υπολογισμούς που δεν ανήκει στην σκοπιά της ανάλυσής μας. Τέλος εξετάσαμε την περίπτωση που ο χρόνος εκτέλεσης κάθε διεργασίας δεν είναι πάντα βελτιστος αλλά τυχαίος (μεταξύ ενός εύρους τιμών), και παρατηρήσαμε πως σε μερικές εκτελέσεις δημιουργείται Deadlock καθώς για μεγάλους χρόνους εκτέλεσης το Schedulability EDF Test (Κεφάλαιο 1, σχήμα 1.6.5α) αποτυγχάνει άρα δεν είναι δυνατή η δρομολόγηση με EDF. Το Deadlock δημιουργείται όταν οι διεργασίες μεταβούν σε κατάσταση error, όταν δηλαδή δεν έχουν δρομολογηθεί πριν εκπνεύσει ο αντίστοιχος χρόνος προθεσμίας τους.

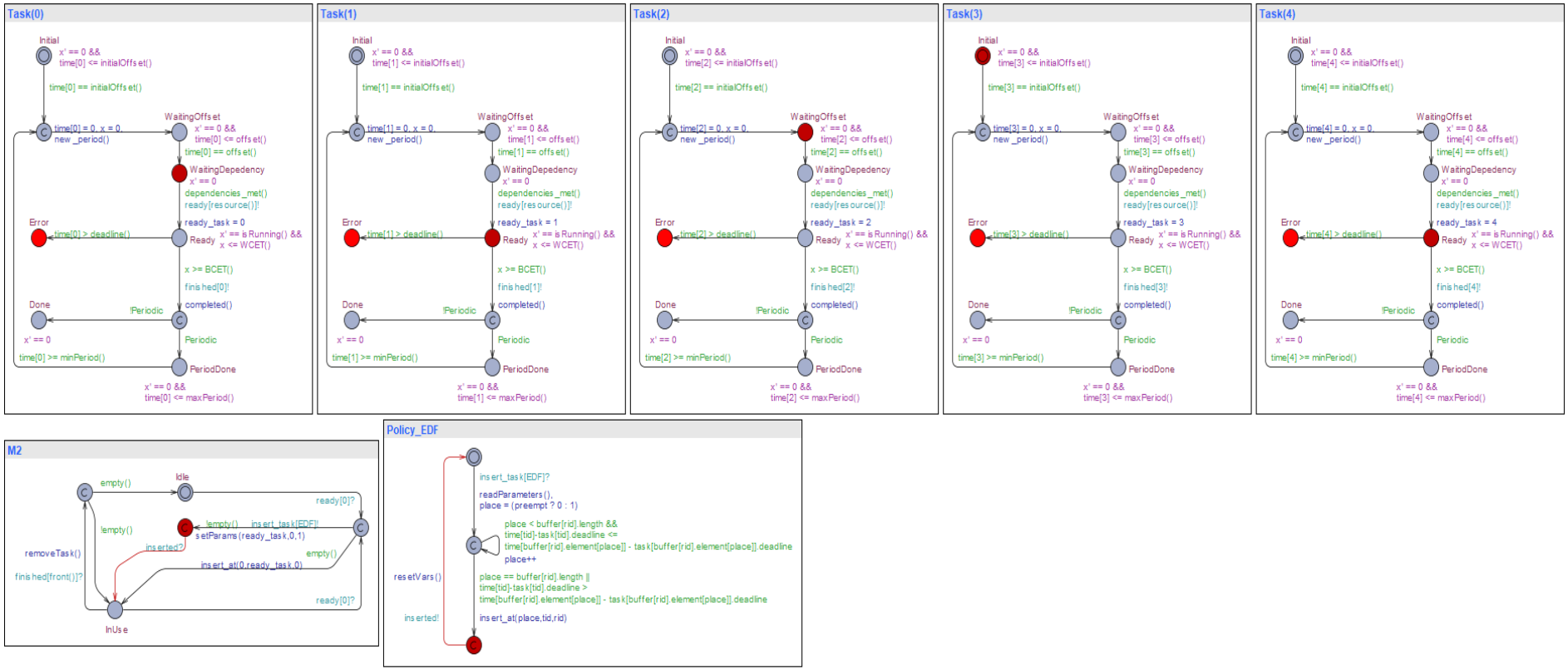
Παρακάτω παρουσιάζουμε τα χαρακτηρίστηκα που ήταν αναγκαία για την υλοποίηση του παρακάτω προβλήματος:

```
PART_A:
//System Declarations

M2 = Resource(0 , true , EDF);

System Task, M2, Policy_EDF;

//Declarations + Configuration Settings

   const int Tasks = 5; //number of Tasks
const task_t      task[Tasks] = {                    //init_offset,
                  min_period,max_period, offset, //deadline, bcet, wcet,resource, priority
          {0,18,18,0,18,1,1,0,1},
          {0,5,5,0,5,2,2,0,1},
          {0,10,10,0,10,2,2,0,1},
          {0,17,17,0,17,1,1,0,1},
          {0,19,19,0,19,2,2,0,1}
   };
```

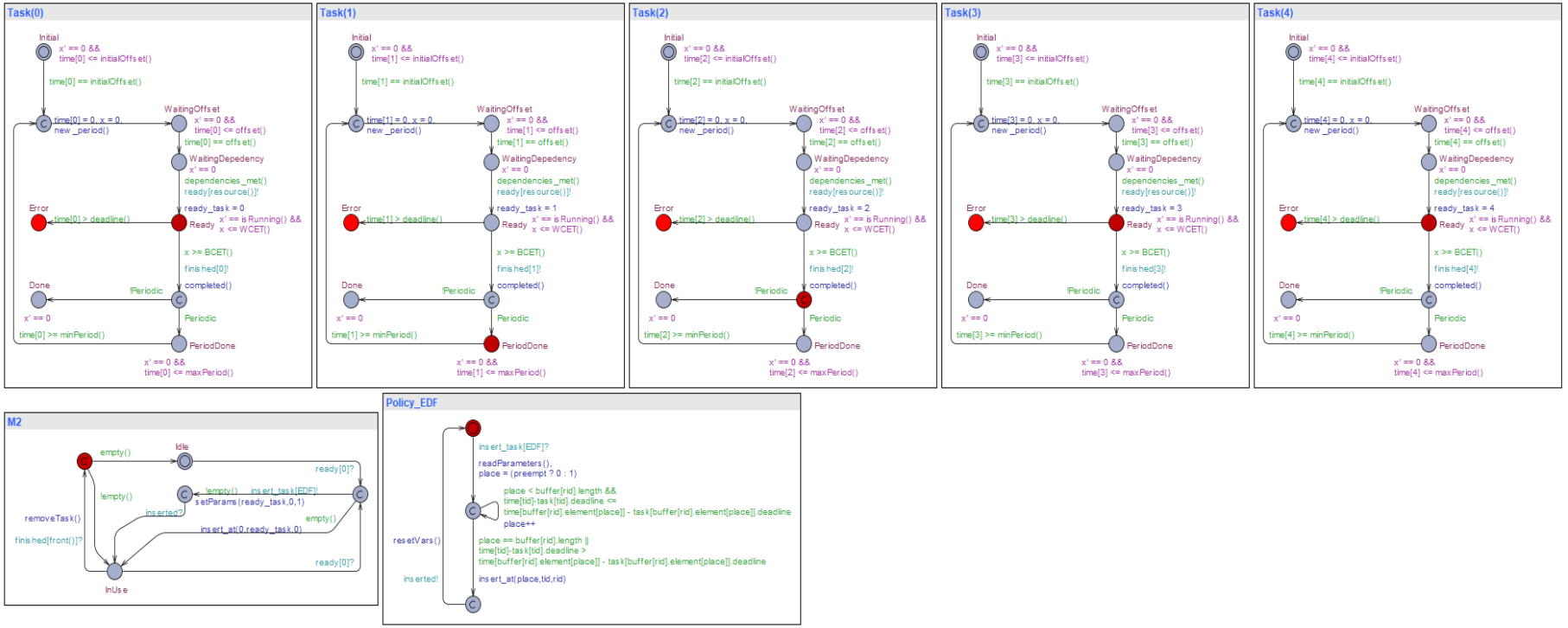
PART_B(Deadlock):

```
//System Declarations

M2 = Resource(0 , true , EDF);

System Task, M2, Policy_EDF;



//Declarations + Configuration Settings

    const int Tasks = 5; //number of Tasks
const task_t        task[Tasks] = {                         //init_offset,
                    min_period,max_period, offset, //deadline, bcet, wcet,resource, priority
            {0,18,18,0,18,1,2,0,1},
            {0,5,5,0,5,2,4,0,1},
            {0,10,10,0,10,2,4,0,1},
            {0,17,17,0,17,1,5,0,1},
            {0,19,19,0,19,2,3,0,1}
    };
```

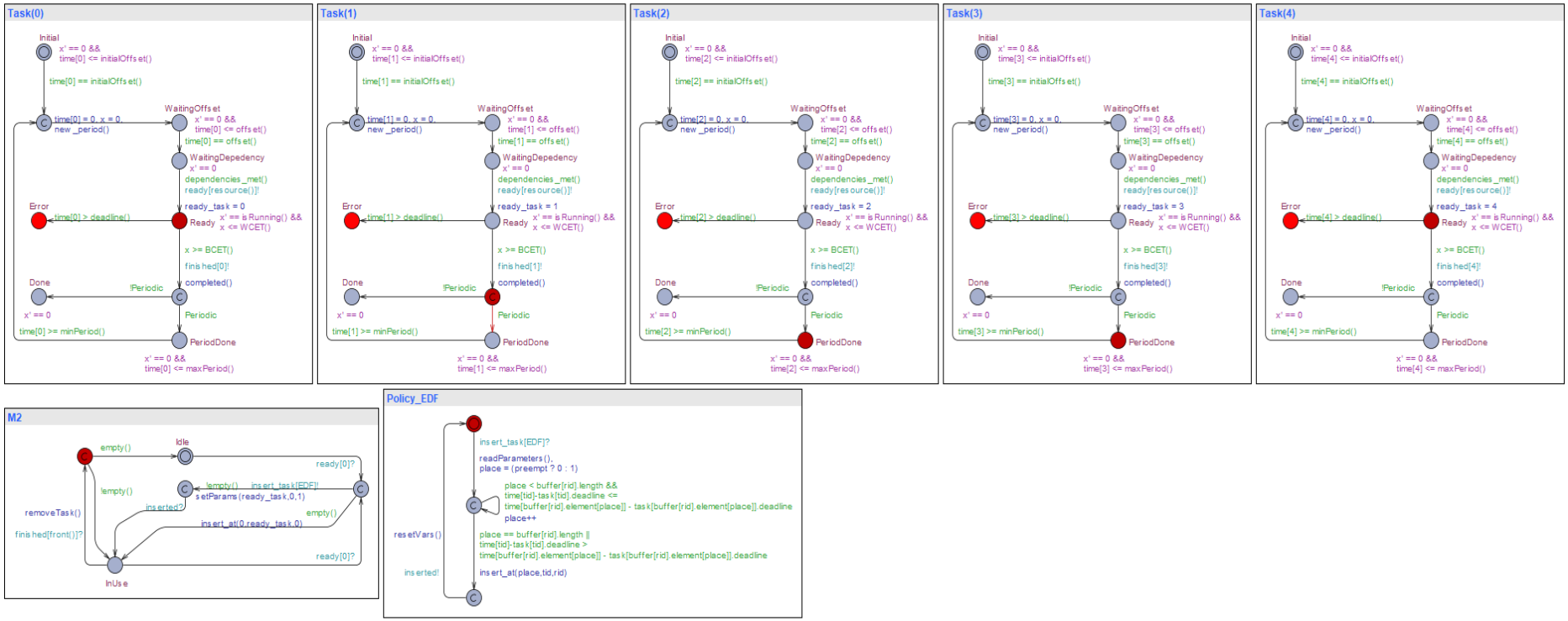
Σημαντικά Στιγμιότυπα Α:



Στιγμιότυπο 5.2.1

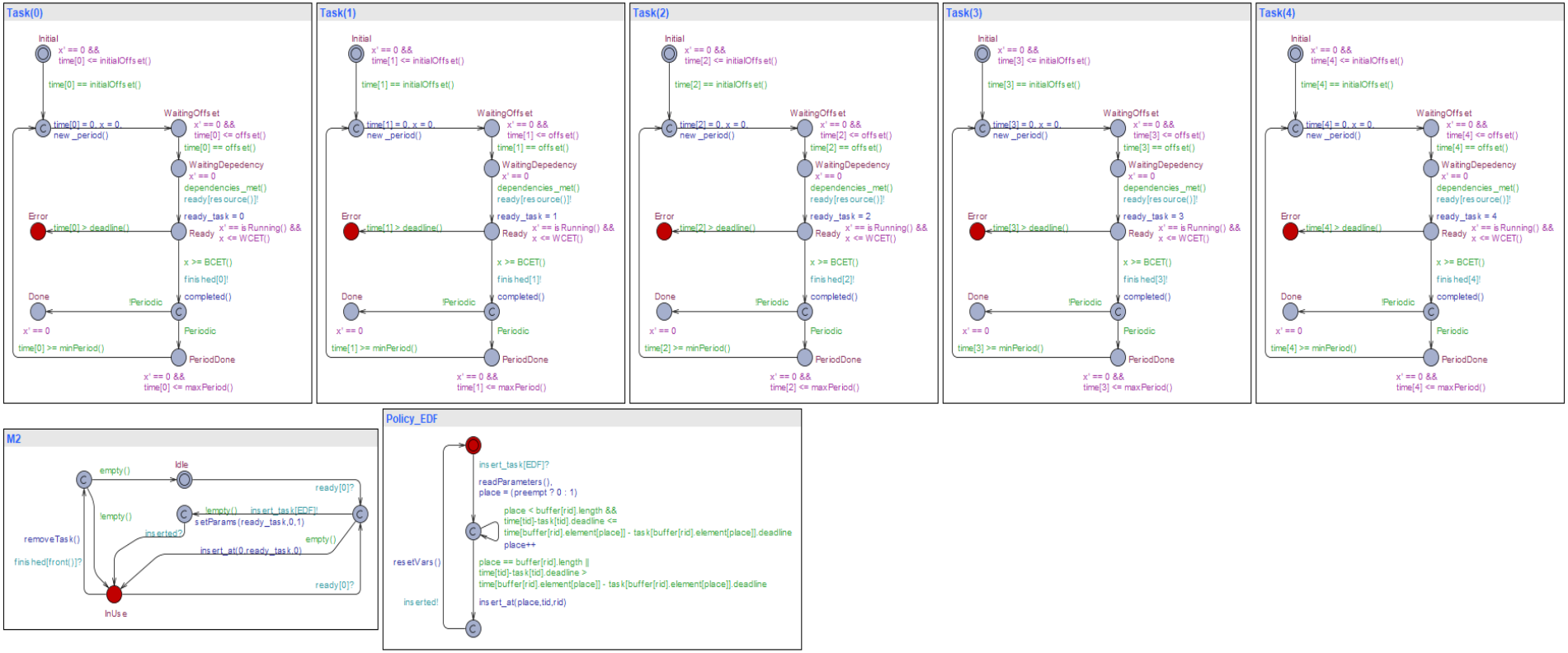


Στιγμιότυπο 5.2.2



Στιγμιότυπο 5.2.3

Σημαντικά Στιγμιότυπα Β:



Στιγμιότυπο 5.2.4

Αρχικά (πριν εκτελέσουμε την περίπτωση του bcet =/= wcet) ελέγχουμε, σύμφωνα με τον τύπο 1.6.5α του Κεφαλαίου 1 αν είναι δυνατόν να πραγματοποιηθεί χρονοδρομολόγηση μέσω του αλγορίθμου EDF. Έτσι το utilization (του επεξεργαστή) είναι (1/18 + 2/5 + 2/10 + 1/17 + 2/19) ~ = 81% , άρα εφόσον το utilization δεν καθίσταται μεγαλύτερο του 100% μπορούμε να προχωρήσουμε με πολιτική χρονοδρομολόγησης EDF. Παρακάτω παρατίθενται μερικά σημαντικά χρονικά στιγμιότυπα της προσομοίωσης.

Στιγμιότυπο 5.2.1:

Σε αυτό το χρονικό στιγμιότυπο τα task [1] και task [4] βρίσκονται στην κατάσταση Ready , είναι έτοιμα προς εκτέλεση και έχουν τοποθετηθεί στον buffer μέσω του resource M2 και με τη χρήση του Policy_EDF. Το M2 εφόσον ολοκλήρωσε την τοποθέτηση (στο buffer) του προηγούμενου task μεταβαίνει σε κατάσταση InUse και είναι έτοιμο να υποδεχθεί το επόμενο (εδώ το task [0]). Τα task [3] και task [2] δεν είναι ακόμα έτοιμα προς εκτέλεση.

Στιγμιότυπο 5.2.2:

Όπως ήταν αναμενόμενο η πρώτη διεργασία που ολοκληρώνει την περίοδό της (άρα και την εκτέλεση) είναι η task[1] και αμέσως επόμενη η task [2] (έχει ολοκληρώσει την εκτέλεση στην τοποθεσία completed και στην τοποθεσία PeriodDone ολοκληρώνει την περίοδο).

Στιγμιότυπο 5.2.3:

Εδώ παρατηρούμε την «ιδιομορφία» του αλγορίθμου EDF συγκριτικά με άλλους απλοικότερους αλγορίθμους. Βρισκόμαστε σε μία χρονική στιγμή κατά την οποία τα task [1] task [2] και task[3] έχουν δρομολογηθεί, τοποθετηθεί στον buffer εκτελεσθεί και εξέλθει (με αυτή τη ακριβώς τη σειρά), όπως άλλωστε περιμέναμε. Θεωρητικά ως επόμενο task που θα δρομολογούνταν θα περιμέναμε το task [0] αφού deadline[0] > deadline[4] ,ωστόσο εκείνη τη στιγμή μεγαλύτερο priority αποκτά ξανά το task[1] (παρόλο που είχε εκτελεσθεί ήδη μια φόρα, τώρα εκτελείται δεύτερη στη δεύτερη περίοδό του) και έτσι δρομολογείται και τοποθετείται στον buffer σε θέση μπροστά του task[0], ακριβώς όπως το παράδειγμα που παρατέθηκε στο Κεφάλαιο 1 στην ενότητα 1.6.5. Στη συνέχεια ακολουθεί το δεύτερο μέρος του παραδείγματος με best case execution time διαφορετικό του worst case execution time.

Στιγμιότυπο 5.2.4:

Στην παραπάνω περίπτωση μόνη διαφορά αποτελεί ο χρόνος εκτέλεσης, δηλαδή δεν έχουμε ένα σταθερό χρονικό αριθμό αλλά ένα εύρος τιμής από τον βέλτιστο στο χείριστο. Στην περίπτωση χείριστου χρόνου τα tasks παραμένουν στην τοποθεσία Ready πέραν της χρονικής τους προθεσμίας και κατ' επέκταση μεταβαίνουν στη θέση error και σταματά η εκτέλεσή τους. Αυτό που συμβαίνει σε στην περίπτωση χείριστου χρόνου είναι πρακτικά η αύξηση του ποσοστιαίου utilization σε βαθμό μεγαλύτερο του 100%, με αποτέλεσμα να μην καθίσταται δυνατή η δρομολόγηση μέσω EDF (μιλώντας πάντα για το σενάριο με χείριστους χρόνους. Συμπερασματικά, για να μπορέσουμε να αξιοποιήσουμε τον EDF αλγόριθμο

πρέπει να είμαστε προσεκτικοί τόσο στους χρόνους εκτέλεσης όσο και στους χρόνους προθεσμίας, ώστε το άθροισμα των λόγων τους να μην ξεπερνά το 100%.

5.3 ΠΡΟΒΛΗΜΑ 3 (PREEMPTIVE SRTF)

Σε αυτό το μοντέλο μας απασχόλησε ο διακοπτόμενος αλγόριθμος SRTF, καθώς ο SJF λειτουργεί παρόμοια με τον αλγόριθμο FIFO. Όπως έχει προαναφερθεί στο Κεφάλαιο 1 ενότητα 1.6.2, στον αλγόριθμο SRTF αν εισέλθει στην ουρά έτοιμων διεργασιών ένα νέο task με μικρότερο αναμενόμενο χρόνο εκτέλεσης από τον εναπομείναντα του τρέχων, το τρέχων task θα αντικατασταθεί. Αυτό πρακτικά σημαίνει πως αλλάζει η προτεραιότητα για την εκτέλεση ,και ολοκλήρωση, των διεργασιών ανάλογα με την τιμή της μεταβλητης του χρόνου εκτέλεσης. Έτσι αν πρώτα εισέλθει στο buffer μία διεργασία Α με execution time 3 μονάδες χρόνου και αμέσως μετά εισέλθει διεργασία Β με execution time 2 μονάδες χρόνου τότε η εκτέλεση της Α σταματά , η προτεραιότητα περνά στη Β και η Α συνεχίζει την εκτέλεση αφού ολοκληρωθεί η Β. Ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση που πολλές διεργασίες μικρού χρόνου εκτέλεσης εισέλθουν πρωτες στο buffer και τελευταία εισέλθει μία διεργασία με αισθητά μεγαλύτερο χρόνο εκτέλεσης απο όλες τις υπόλοιπες. Σε αυτή την περίπτωση αναμένεται να υπάρξει το φαινόμενο της λιμοκτονίας για την τελευταία καθώς ο χρόνος προθεσμίας της θα λήξει πριν αυτή προλάβει να εκτελεστεί. Προκειμένου να υλοποιήσουμε το μοντέλο Policy_SRTF επιτελέσαμε μερικές αλλαγές στον κώδικα της Policy_FPS. Παρακάτω παρουσιάζονται τόσο το σημαντικό τμήμα αλλαγής του κώδικα όσο και οι δηλώσεις και ορισμοί των χρονικών τιμών των tasks:

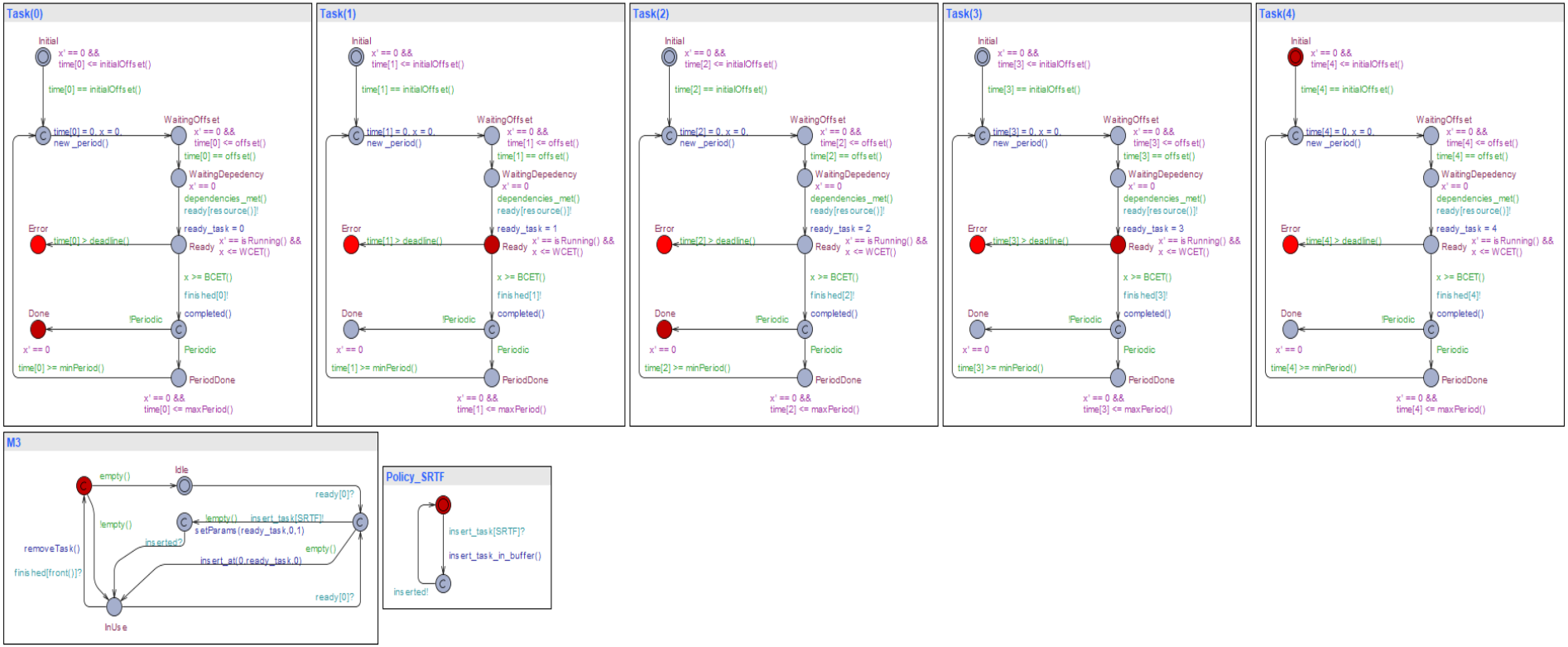
```
//System Declarations

M3 = Resource(0 , true , SRTF);

system Task, M3, Policy_SRTF;

//Declarations + Configuration Settings
const bool Periodic = false;
const int Tasks = 5; //number of Tasks
const task_t task[Tasks] = { //init_offset, min_period,max_period, offset, //deadline, bcet, wcet,resource, priority
{0,0,0,0,20,1,1,0,1},
{2,0,0,0,20,3,3,0,1},
{4,0,0,0,20,2,2,0,1},
{5,0,0,0,20,2,2,0,1},
{7,0,0,0,20,21,21,0,1}
};
```

Νέο κομμάτι κώδικα που ελέγχει το bcet:

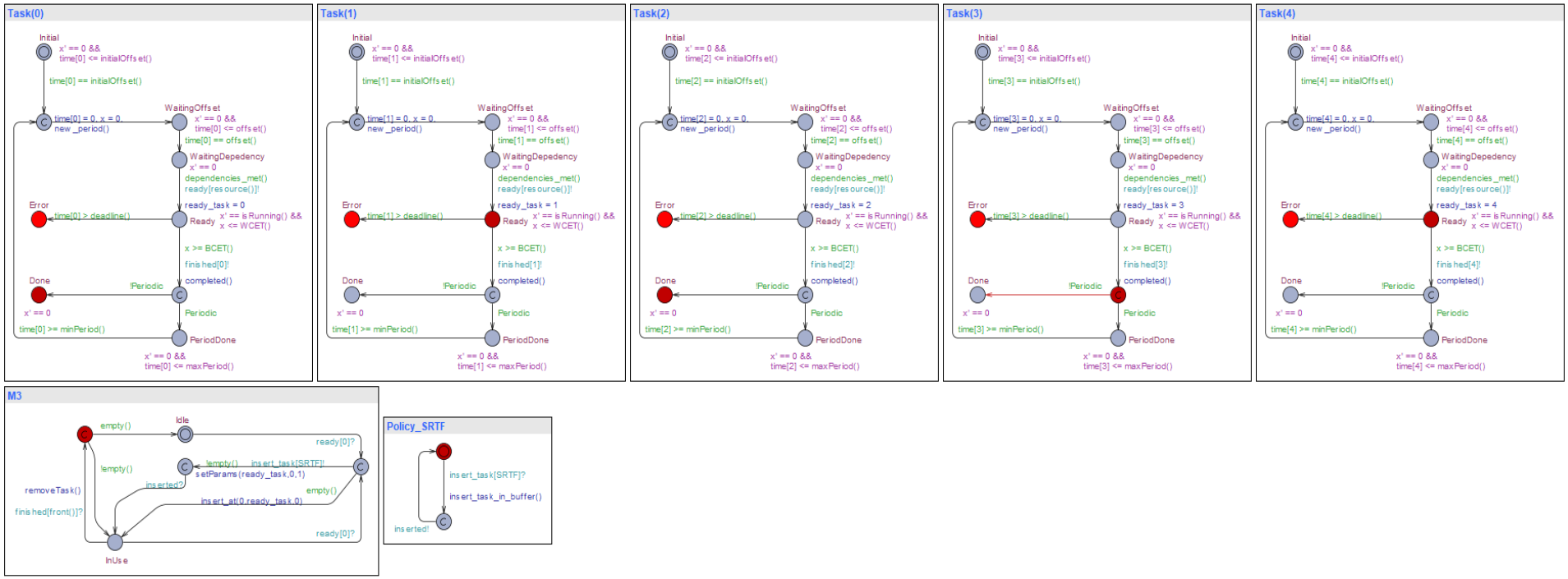
```
while ( place < buffer[r].length && task[buffer[r].element[place]].bcet <= task[t].bcet )
{
   place++;
  }
```

Απο τις τιμές που ορίσαμε στον 5x5 πίνακα task[Tasks] παρατειρείται πως τα task[0] – task [4] εκκινούν με αυτή τη σειρά. Αναμενόμενο αποτέλεσμα στη σειρά ολοκλήρωσης της εκτέλεσης των διεργασιών θα είναι task [0] -> task [2] -> task [3] -> task [1] -> λιμοκτονία task[4], ανάλογα δηλαδή με τους χρόνους εκτέλεσής τους. Δύο αξιοσημείωτες παρατηρήσεις είναι πρώτον πως το arrival time των διεργασιών είναι διαφορετικό απο το completion time , πράγμα που φαίνεται στα task [2] και task [1] (task [1] φτάνει πρώτο αλλά task [2] ολοκληρώνεται πρώτο καθώς έχει μικρότερο χρόνο εκτέλεσης) και δεύτερον πως το race condition των task [2] και task [3] επιλύεται με ένα απλό FIFO (εφόσον έχουν ίδιο χρόνο εκτέλεσης).

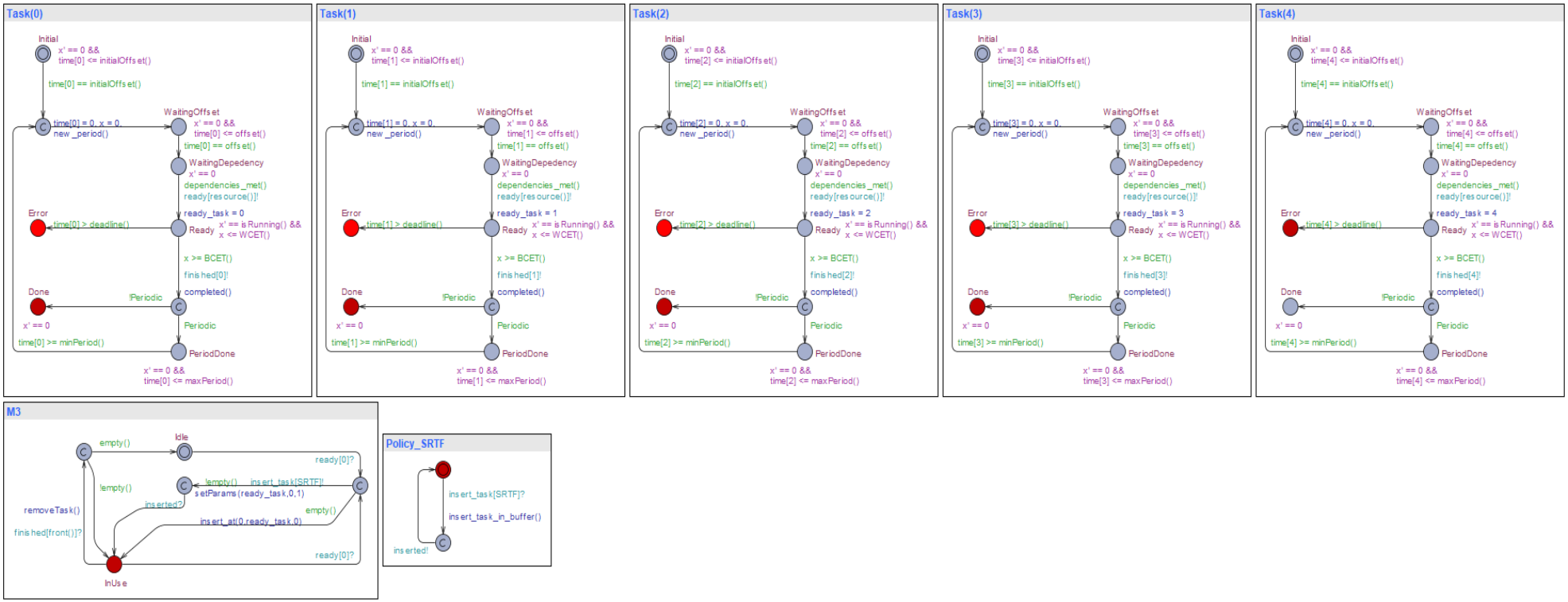
Σημαντικά χρονικά στιγμιότυπα:



Στιγμιότυπο 5.3.1



Στιγμιότυπο 5.3.2



Στιγμιότυπο 5.3.3

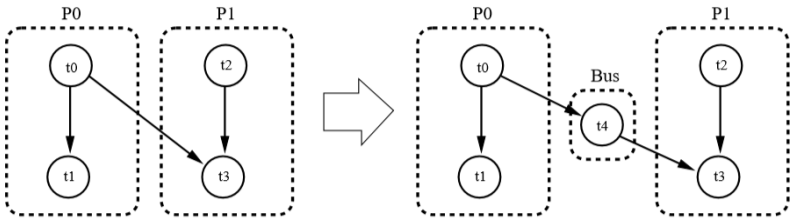
Στιγμιότυπο 5.3.1: Εδώ παρατηρούμε πως πολύ σωστά μετά την ολοκλήρωση του task [0] επέρχεται η ολοκλήρωση του task [2] έναντι του task [1] καθώς έχει μικρότερο bcet (execution time).

Στιγμιότυπο 5.3.2: Στη συνέχεια επόμενο έχει σειρά το task [3] παρόλο που το task [1] βρίσκεται σε κατάσταση Ready (έτοιμο προς εκτέλεση) από πολύ πιο πριν. Ορθώς πρώτα δρομολογήθηκε το task [2] και ύστερα το task [3] καθώς το task [2] έφτασε πρώτο και έχουν ίδιο execution time. Αν το task [3] εξακολουθούσε να ερχόταν μετά το task [2] και είχε ομώς μικρότερο execution time τότε και μόνο τότε θα έπρεπε να εκτελεσθεί πριν το task [2].

Στιγμιότυπο 5.3.3: Κατάσταση Deadlock (λιμοκτονία) για το task [4]. Όλες οι διεργασίες έχουν τελειώσει τις εκτελέσεις τους και έχουν περατωθεί , όμως η προθεσμία χρόνου του task [4] έληξε πριν αυτό ολοκληρωθεί. Αυτό αποτελεί και το μεγαλύτερο πρόβλημα του συγκεκριμένου αλγορίθμου, όταν δηλαδή πολλές «μικρές» διεργασίες προηγούνται μεγάλων προκύπτει κατάσταση λιμοκτονίας.

5.4 ΠΡΟΒΛΗΜΑ 4 (PERIODIC FPS - FIFO)

Στο παρακάτω μοντέλο παρατίθεται ένα πρόβλημα περιοδικής χρονοδρομολόγησης ενώ παράλληλα παρουσιάζεται η δημιουργία και αρχικοποίηση του. Το πρόβλημα αποτελείται από τέσσερα task T = t_0,..,t_3 και δύο resources P_0και P_1με αλληλεξαρτήσεις μεταξύ των tasks που δίνονται στο διάγραμμα αλληλεξαρτήσεων. Όπως φαίνεται και από το σχήμα τα tasks t_0 και t_1 ανατίθενται στο P_0 και τα t_2 και t_3 ανατίθενται στο P_1.



Σχήμα 5.5.1 Περιγραφή προβλήματος

Σημειώνεται πως το t_3 εξαρτάται από το t_0 αλλά τα δύο αυτά tasks έχουν ανατεθεί σε διαφορετικά resources. Αυτό αποτελεί μία συνήθης περίπτωση προβλήματος χρονοδρομολόγησης και η επίλυση της επιτυγχάνεται με την εισαγωγή ενός διαύλου επικοινωνίας δεδομένων μεταξύ των δύο resources χρησιμοποιώντας ένα μέσο μεταφοράς όπως π.χ. ένα data bus. Η επικοινωνία μεταξύ των δύο tasks χρειάζεται χρόνο αλλά δεν εμποδίζει τα resources. Όπως φαίνεται από το δεύτερο σκέλος του σχήματος προκειμένου να επιλύσουμε αυτό το ζήτημα θεωρούμε ως data bus ένα νέο εικονικό task t_4 που χρειάζεται ένα νέο resource, με όνομα έστω bus. Το t_4 καταχωρείται στο διάγραμμα αλληλεξαρτήσεων με τέτοιο τρόπο ώστε το t_4 να εξαρτάται από το t_0 και το t_3 από το t_4. Θεωρούμαι για λόγους που έχουν αναλυθεί στο θεωρητικό μέρος της εργασίας πως η κάθε διεργασία έχουν σταθερές και πανομοιότυπες περιόδους. Έτσι τα χαρακτηρίστηκα του t_4θα είναι τα εξής:

- INITIAL_OFFSET(t_4): INITIAL_OFFSET(t_0)
- BCET(t_4): Εξαρτάται από το bus
- WCET(t_4): Εξαρτάται από το bus
- MIN_PERIOD(t_4): Θεωρούμε δεδομένο ότι τα tasks έχουν πανομοιότυπες περιόδους άρα κατ' επέκταση η μέγιστη και η ελάχιστη περίοδος είναι ίσες για όλα
- MAX_PERIOD(t_4): MIN_PERIOD(t_4)
- OFFSET(t_4): 0 Το offset εδώ δεν χρειάζεται καθώς η ενεργοποίηση του t_4 εξαρτάται από την εκτέλεση του t_0
- DEADLINE(t_4): MIN_PERIOD(t_4)

- PRIORITY(t_4): 0 Εδώ δεν έχει νόημα η προτεραιότητα καθώς το bus είναι μορφής FIFO

Το bus resource θα μοντελοποιηθεί ως ένας μη διακοπτόμενος FIFO πόρος ώστε να μιμηθεί τη συμπεριφορά ενός data bus. Θεωρώντας κάποιες τιμές για βέλτιστους-χείριστους χρόνους εκτέλεσης , χρόνους περιόδου, χρόνους προθεσμιών το πρόβλημα που περιγράφουμε ορίζεται καλώς και δημιουργείται στο παρακάτω σχήμα. Αξίζει να σημειωθεί πως στο σχήμα οι τρείς διεργασίες που αλληλεξαρτώνται έχουν offset τη μονάδα. Αυτό συμβαίνει για μεγαλύτερη ευκολία καθώς το ίδιο το task έχει την αρμοδιότητα να επαναρχικοποιήσει και να επαναφέρει την τιμή της εισόδου του στο complete. Με την εκκίνηση μίας νέας περιόδου χωρίς το offset το t_1 σηματοδοτεί το αρμόδιο resource πως είναι έτοιμο προς εκτέλεση ενώ παράλληλα διαβάζει την παλιά τιμή του complete για το t_0 πριν αυτό επαναρχικοποιήση την τιμή του. Ένας άλλος τρόπος για να χειριστούμε την ισότητα στις περιόδους των tasks θα ήταν να υποχρεώναμε κάθε task να επαναφέρει τις τιμές complete σε αρχικές ακριβώς πριν ξεκινήσει την νέα του περίοδο.

```
//Global Declaration
const bool      Periodic = true;
const int       MaxTime = 20;
const int       Tasks = 5;
const int       Procs = 3;

const task_t    task[Tasks][Tasks] = {
        { 0,  20 , 20 , 0 , 20 , 4 ,7 , 0 ,1 }
        { 0,  20 , 20 , 1 , 20 , 8 ,12 , 0 ,1 }
        { 0,  20 , 20 , 0 , 20 , 10 ,12 , 1 ,1 }
        { 0,  20 , 20 , 1 , 20 , 6 ,7 , 1 ,1 }
        { 0,  20 , 20 , 1 , 20 , 5 ,5 , 2 ,1 }
} ;

const bool Depend[Tasks][Tasks] = {          //Task Graph
                { 0 , 0 , 0 , 0 , 0 }
                { 1 , 0 , 0 , 0 , 0 }
                { 0 , 0 , 0 , 0 , 0 }
                { 0 , 0 , 1 , 0 , 1 }
                { 1 , 0 , 0 , 0 , 0 }
        } ;

        //System Declaration
        P0 = Resource( 0 , true , FPS );
        P1 = Resource( 1 , true , FPS );
        Bus = Resource( 2 , false , FIFO);

        system Task, P0, P1, Bus, Policy_FPS, Policy_FIFO;
```

Κώδικας 5.5.1 Αρχικοποίησεις τιμών για εγκατάσταση του προβλήματος

6 ΣΥΜΠΕΡΑΣΜΑΤΑ & ΠΡΟΟΠΤΙΚΕΣ

Στο πλαίσιο της παρούσας διπλωματικής εργασίας μελετήθηκε η χρονοδρομολόγηση διεργασιών με τη χρήση διαφόρων γνωστών αλγορίθμων χρονοδρομολόγησης, τόσο θεωρητικά όσο και πρακτικά. Η θεωρητική προσέγγιση αφορά τόσο την θεμελίωση και τον ορισμό ορισμένων απαραίτητων εννοιών αλλά και την θεωρητική μελέτη των αλγορίθμων χρονοδρομολόγησης που εφαρμόστηκαν. Επόμενο βήμα αποτέλεσε η πρακτική προσέγγιση που αφορά την δημιουργία ενός γενικευμένου μοντέλου χρονοδρομολογητή-διεργασίας, με τη χρήση χρονισμένων αυτομάτων και με την ενσωμάτωση τεχνικών για μεγαλύτερη αποδοτικότητα.

6.1 Συμπεράσματα

Το μοντέλο που προκύπτει είναι ικανό να περιγράψει μία πληθώρα απλών περιπτώσεων χρονοδρομολόγησης με χρήση των διαφορετικών αλγορίθμων που μελετήθηκαν, το όποιο όταν εφαρμοστεί παρέχει λύσεις στις διαφορετικές αυτές περιπτώσεις. Αυτό επιτυγχάνεται τόσο μέσω της ποικιλομορφίας του στην μοντελοποίηση του χρονοδρομολογητή, όσο και στην δυνατότητα μεταβολής των παραμέτρων που αφορούν μία διεργασία. Σημειώνεται πως το μοντέλο του χρονοδρομολογητή είναι κατασκευασμένο με τέτοιο τρόπο ώστε να έχει την δυνατότητα αντιμετώπισης περιπτώσεων διακοπτής και μη-διακοπτής χρονοδρομολόγησης ενώ το μοντέλο των διεργασιών είναι κατασκευασμένο με τέτοιο τρόπο ώστε να λαμβάνονται υπόψη μη περιοδικές αλλά και περιοδικές διεργασίες.

Με τη χρήση προσομοιώσεων αρχικά επιβεβαιώσαμε την ορθότητα της σχεδίασης των μοντέλων ενώ παράλληλα καταλήξαμε πως τα χρονισμένα αυτόματα αποτελούν ένα αξιόλογο εργαλείο προς χρήση για την περιγραφή προβλημάτων χρονοδρομολόγησης, καθώς προσφέρουν μεγάλη ευελιξία στο σχεδιασμό των μοντέλων (ιδιαίτερα στο κομμάτι του χρόνου). Έτσι μέσω των προσομοιώσεων μας δίνεται η δυνατότητα επίλυσης των διαφορετικών προβλημάτων χρονοδρομολόγησης για κάθε αλγόριθμο που εξετάστηκε καθώς και μία πλήρης εικόνα του προβλήματος για κάθε χρονική στιγμή, πράγμα που μας επέτρεψε τη λήψη τιμών για τα χαρακτηρίστηκα της κάθε διεργασίας σε κάθε της κατάσταση. Επιπλέον με τη χρήση του Verifier δόθηκε η δυνατότητα επαλήθευσης των υποθέσεών μας για το κάθε πρόβλημα ξεχωριστά καθώς πριν καν εκτελεσθεί, έχουμε σαφή εικόνα για τις εκάστοτε καταστάσεις των διεργασιών. Μέσω του Verifier εξάγαμε το πολύ σημαντικό συμπέρασμα πως όλες οι διεργασίες πληρούν τους χρόνους προθεσμίας τους, δηλαδή πως σε κάθε πιθανό σενάριο εκτέλεσης καμία διεργασία δεν βρίσκεται σε κατάσταση σφάλματος.

6.2 Προοπτικές

Η μελλοντική εργασία που μπορεί να υλοποιηθεί πλέον, αφορά τόσο την μοντελοποίηση επιπρόσθετων, πιο σύνθετων αλγορίθμων χρονοδρομολόγησης όσο και την

εισαγωγή πολλαπλών επεξεργαστών και «νημάτων» στο μοντέλο, ώστε να αποκτούμε μεγαλύτερη ταχύτητα στην επίλυση της χρονοδρομολόγησης.

Η ένταξη περαιτέρω σύνθετων αλγορίθμων χρονοδρομολόγησης καθιστά δυνατή την μεγαλύτερη ευκολία στην σύγκριση και διαβάθμιση τους για το εκάστοτε πρόβλημα χρονοδρομολόγησης που αντιμετωπίζουμε. Πιο συγκεκριμένα ένα μοντέλο Cluster είναι ικανό να μας δώσει μεγάλη ακρίβεια και περισσότερα δεδομένα ώστε να συγκρίνουμε τις τιμές utilization , throughput και resource allocation του επεξεργαστή που διαφέρουν ανάλογα με την πολιτική χρονοδρομολόγησης. Μέσω αυτών των αποτελεσμάτων σε συνδυασμό με την μεταβολή των παραμέτρων των διεργασιών έχουμε μια καλύτερη εικόνα για το ποιός αλγόριθμος είναι πιο ταιριαστός σε κάθε πρόβλημα χρονοδρομολόγησης.

Όσον αφορά την εισαγωγή πολλαπλών επεξεργαστών οι δυνατότητες που υπάρχουν είναι αρκετές, καθώς ένα πολυεπεξεργαστικό μοντέλο υπόσχεται μεγαλύτερη ταχύτητα στη χρονοδρομολόγηση, απαλλαγή του φόρτου εργασίας σε ένα επεξεργαστή και άρα καλύτερη αποδοτικότητα αλλά και μεγαλύτερη ακρίβεια στα πραγματικά προβλήματα. Ένα τέτοιο σύστημα είναι πολύ φτηνό και αποδοτικό καθώς στοιχεία όπως η μνήμη, η μονάδα εισόδου-εξόδου και το busδιαμοιράζονται, με αποτέλεσμα να μπορεί να διαχειριστεί μεγάλο φόρτο δεδομένων σε μικρά χρονικά διαστήματα δουλεύοντας συμμετρικά ως προς τους επεξεργαστές.

Αυτές είναι μόνο κάποιες από τις πιθανές επεκτάσεις στη χρονοδρομολόγηση διεργασιών και αυτό αποδεικνύει τις ανεξάντλητες δυνατότητες περαιτέρω μελέτης αυτού του επιστημονικού πεδίου.

Βιβλιογραφία

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In Proc. 5th IEEE Symposium on Logic in Computer Science (LICS'90), pages 414–425. IEEE Computer Society Press, 1990.

[2] Rajeev Alur and David Dill. Automata for modeling real-time systems. In Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90), volume 443 of Lecture Notes in Computer Science, pages 322–335. Springer, 1990.

[3] Rajeev Alur and David Dill. A theory of timed automata. Theoretical Computer Science (TCS), 126(2):183–235, 1994.

[4] M. Mikucionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, ˇ J. S. Pedersen, and P. Hougaard, "Schedulability analysis using uppaal: Herschelplanck case study", in Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 175–190,

[5] Jan Madsen Aske Brekling, Michael R Hansen. Models and formal verification of multiprocessor system-on-chips. The journal of Logic and Algebraic Programming, 77(1):1–19, 2008.

[6] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell", International Journal on Software Tools for Technology Transfer, vol. 1, no. 1, pp. 134–152, 1997,

[7] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. ACM SIGMETRICS Perform. Eval. Rev., 32(4):34–40, 2005.

[8] Thomas Bœgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems, pages 106–114, New York, NY, USA, 2008. ACM.

[9] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidesi, editor, 11th International Conference on Concurrency Theory, (CONCUR'2000), number 1877 in Lecture Notes in Computer Science, pages 138–152, University Park, P.A., USA, July 2000. Springer-Verlag.

[10] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci., 354(2):301–317, 2006.

[11] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: schedulability and decidability. In In Proceedings of TACAS 2002, pages 67–82. Springer-Verlag, 2002.

[12] UPPAAL Scheduling Framework. http://www.uppaal.com/SchedulingFramework, Jan. 2009.

[13] K. Godary, I. Aug´e-Blum, and A. Mignotte. Sdl and timed petri nets versus uppaal for the validation of embedded architecture in automotive. In Forum on specification and Design Language (FDL'04), Lille, France, September 2004.

[14] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Gu Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In Software Technologies for Embedded and Ubiquitous Systems, Lecture Notes in Computer Science, pages 263–272. Springer, 2007.

[15] "A Tutorial on Uppaal" Gerd Behrman, Alexandre David, and Kimg G. Larsen. Department of Computer Science, Aalborg University, Denmark

[16] Aske Brekling Jan Madsen, Michael R. Hansen. A modelling and analysis framework for embedded systems. In this book.

[17] "Task Graph using Timed Automata" Yashmina Abdeddaïm, Abdelkarim Kerbaa and Oder Maler, VERIMAG, Centre Equation, 2 , av. De Vignate, 38610 Gieres, France

[18] Pavel Krc´al and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, TACAS, volume 2988 of Lecture Notes in Computer Science, pages 236–250. Springer, 2004.

[19] Buttazzo, Giorgio (2011), Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Third ed.), New York, NY: Springer, p. 100

[20] S. Mahadevan, M. Storgaard, J. Madsen, and K. M. Virk. Arts: A system-level framework for modeling MPSoC components and analysis of their causality. In 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE Computer Society, sep 2005.

[21] "Modeling and Verification of Real-Time Systems using Timed Automata: Theory and Practice" Paul Petterson, February 1999

[22] EECS 343 E-Handout: Adding Clocks to Automata , Prof. Branicky

[23] H. Sun. Timing constraints validation using uppaal: Schedulability analysis. In DIPES '00: Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems, pages 161–172, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[24] UPPAAL. http://www.uppaal.com, Jan. 2005.

[25] UPPAAL CORA. http://www.cs.aau.dk/~behrmann/cora/, January 2006.

[26] Δίπλας Αλέξανδρος - Τσακίρης Χρήστος, Χρονοπρογραμματισμός Παραγωγής με χρήση Αυτομάτων (Αθήνα, Φεβρουάριος 2004)

[27] Μιχαήλ Ασπραδάκης, Επίλυση προβλημάτων Χρονοπρογραμματισμού με χρήση Αυτομάτων (Αθήνα, Ιούλιος 2013)

[28] Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014), Operating Systems: Three Easy Pieces [Chapter Scheduling Introduction]

[29] Harchol-Balter, Mor; Schroeder, Bianca; Bansal, Nikhil; Agrawal, Mukesh (2003). "Size-Based Scheduling to Improve Web Performance". ACM Transactions on Computer Systems