

Transforming Timing Diagrams into KAOS

Tossaporn Joochim and Michael R. Poppleton
Dependable Systems and Software Engineering group
School of Electronics and Computer Science
Southampton University, United Kingdom
{tj04r, mrp} @ecs.soton.ac.uk

Abstract

Requirements engineering is an important part of developing programs. It is an essential stage of the software development process that defines what a product or system should to achieve. The UML Timing diagram and Knowledge Acquisition in Automated Specification (KAOS) model are requirements engineering techniques. KAOS is a goal-oriented requirements approach while the Timing diagram is a graphical notation used for explaining software timing requirements. KAOS uses linear temporal logic (LTL) to describe time constraints in goal and operation models. Similarly, the Timing diagram can describe some temporal operators such as X (next), U (until) and R (release) over some period of time. Thus, our aim is to use the Timing diagram to generate parts of a KAOS model. In this paper we demonstrate techniques for creating a KAOS goal model from a Timing diagram. The Timing diagram which is used in this paper is adapted from the UML 2.0 Timing diagram and includes features to support translation into KAOS. We use a case study of a Lift system as an example to explain the translation processes described here.

Keywords: Requirements engineering, UML, Timing diagram, KAOS

1. Introduction

In this paper, we present work in progress on the modeling of time constraints in a Timing diagram, and their translation to a KAOS [6, 7, 8, 9] *Goal model* using patterns. The UML Timing diagram is a new artifact introduced in UML 2.0 [12] while KAOS is a goal-oriented requirements approach. Those models

use temporal operators to describe system behaviour over some period of time. KAOS describes LTL [11] features in *Goal* and *Operational models* while the Timing diagram illustrates them by graphical notations. Thus, if one prefers to describe KAOS *Goal/Operational models*, one needs to understand LTL operators and conditions using in KAOS; for example *DomPre*, *DomPost* and *ReqTrig* [8]. Our intention is use the UML Timing diagram graphical notation, a well-known technique widely used among software developers, to create KAOS models as a contribution to software development. We believe that the demonstration of requirements in graphical form helps software developers to define specifications in an easier way. The intended scope of this paper is to transform the Timing diagram into a KAOS *Goal model*. We have amended UML 2.0 Timing diagram to have features to support translation. A lift position display based on Jackson's work [4] is used as a sample case study with one lift in the system. The case study is modified and some requirements added, such as time constraints and some objects: floor sensor and door, to make it suitable for modeling the Timing diagram.

2. Background

2.1 KAOS

The KAOS Framework was designed by the informatics department at Louvain-La-Neuve (UCL) in the early 1990s [7]. It is a goal-oriented specification language that specifies system requirements and their constraints via the *Goal Model*. KAOS is made up of 5 submodels: *Goal model*, *Object model*, *Agent responsibility model*, *Agent Interface model* and *Operation model* as illustrated in figure 1 [8]. In this paper, we focus on the *Goal model* since it is a primary model to create the others. A goal defines an objective the composite system should meet, usually through the

cooperation of multiple agents. The agents are active objects that are capable of performing operations. They can be software, hardware devices or humans. KAOS uses the *Goal model* to declare the system requirements. Goals can be categorized into 4 groups in the following.

- *Achieve* and *Cease* goals are used to identify system behaviors that require some target properties to be eventually satisfied or denied respectively, in some future state. This goal category is the one in which required liveness properties are specified. Examples of temporal operators represent my *Achieve/Cease* goals are $P \Rightarrow \diamond Q$ and $P \Rightarrow \circ T$, where the “ \diamond ” and “ \circ ” are temporal operators mean “some time in the future” and “in the next state”, respectively [8].

- *Maintain* and *Avoid* goals are used to identify system behaviors that require some target properties to be permanently satisfied or denied respectively, in every future state. This goal category is the one in which required safety properties are specified. Examples of temporal operators represent *Maintain/Avoid* goals are $P \Rightarrow Q$, $P \Rightarrow \square Q$ and $P \Rightarrow \bullet Q$, where “ \square ” and “ \bullet ” are “always in the future” and “in the previous state”, respectively [8, 10].

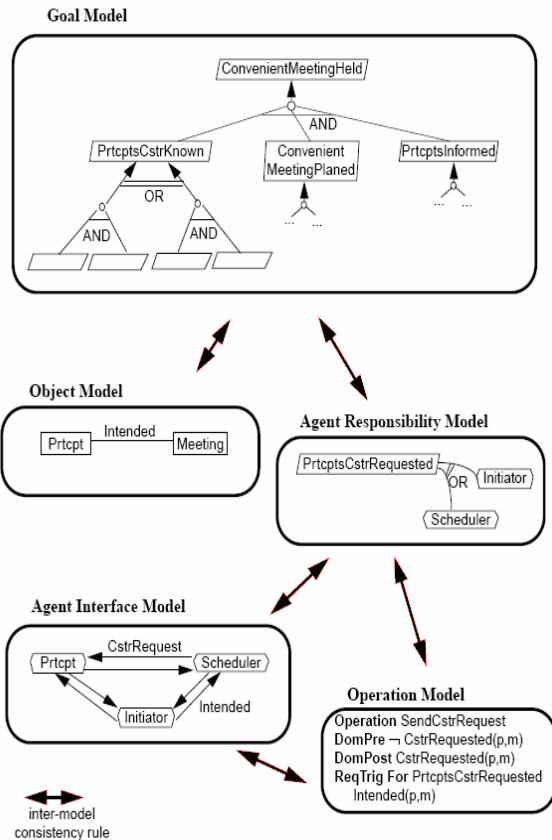


Figure 1. KAOS models [8]

A goal is declared by textual syntax which is composed of goal name, definition and formal definition, where the latter is written in temporal logic statements. An example of a goal is illustrated in figure 2.

An *Object model* is used to describe objects that are involved in the system. Objects can be entities, events, agents or relationship. For example, as illustrated in figure 1, *Prtcpt* and *Meeting* are objects while *Intended* is relationship used to associate between those objects. An *Agent Responsibility model* is used to declare responsibilities of goals to agents where a choice of agent is linked by *OR-relationship*. This relationship is used to specify which alternative agents will be assigned to the goals. As shown in figure 1, *Initiator* and *Scheduler* are agents in the *Agent Responsibility Model*. An *Agent Interface model* is used to declare *Monitoring* and *Control* links between agents and attributes of objects. *CstrRequest* and *Intended* are *Monitoring/Control* links as illustrated in figure 1. In monitoring, the agent monitors values of object attributes while in control, the agent controls values of object attributes. An *Operational model* defines state transitions of a goal. Operations are characterized by *pre*-, *post*-, and *trigger* conditions [8].

Achieve[LiftStopAtFloor]

Definition : Whenever the floor sensor at floor f is on and there is a request for the floor f, lift will stop at the floor f within a time interval of 5 – 10 seconds.

FormalDef: $\forall f : \text{Floor}$

$\text{reqFl}(f) = \text{True} \wedge$
 $\text{FloorSensorState}(f) = \text{On}$

\Rightarrow

$\diamond_{(5,10)} \text{LiftState} = \text{StopAtfloor}$

Figure 2. Goal Achieve[LiftStopAtFloor]

According to figure 2, goal *Achieve*[LiftStopAtFloor] is used to describe condition for lift to stop at a floor. Since the goal is identified by an *Achieve*, we use “ \diamond ” notation (which represents the temporal logic notation of eventually) to identify shape of the goal. Notation (5,10) is used to explain time constraints. Expressions before “ \Rightarrow ” notation define states in which the goal is obliged; they are eventually guards. Expressions after “ \Rightarrow ” notation are used to define the state that the goal specifies.

A goal can be refined into several alternative combinations of subgoals; this is called goal refinement [10]. Subgoals must be generated by preserving a pattern of goal refinement as described in [2, 8]. Using logic to decompose and express the goal one can refine

goal into alternative combinations of subgoals. There are two kinds of goal logic combinations: “AND” and “OR” refinement. Using AND-refinement means the goal G can be refined detail into goal g1 and g2. In order to achieve goal G, both subgoals g1 and g2 must be selected. OR-refinement is an alternative goal refinement. That means more than one alternative subgoals can be selected e.g. to achieve goal G, we may select either g1 or g2. Goal refinement will be finished whenever each goal is defined to single agent. An example of goal refinement is illustrated in figure 1, *Goal model*. Normally, *Goal models* are illustrated as tree structure. A root node represents general goal for the whole system while leaf nodes represent individual goals in detail.

2.2 UML Timing diagram

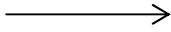
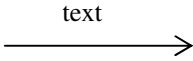

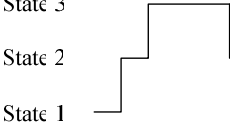
The UML is a standard language used for explaining software requirements in a form of graphical notations. It is independent of particular programming languages and development processes. Currently, an official version of the UML is UML 2.0 [12]. UML Timing diagram are one of the new artifacts added to UML 2.0. It is a specific type of an Interaction diagram and used to explore the behaviors of one or more objects throughout a given period of time [5, 12]. The Robust Timing diagram [14] is a variant of the Timing diagram. It shows states of each object on the left-hand side of the diagrams (Y-axis) while timing constraints are on X-axis. Arrowed lines are messages which define a specific kind of communication between the objects.

2.3 Our Timing diagram

The UML Timing diagram is composed of many notations specifying properties that we do not deal with in this work. We prefer to select a subset of notations for transforming. We define a state predicate in the Timing diagram to invoke changing object’s states. We wish to specify precisely duration constraints notations which use the well-known mathematical notations, for example [...], (...), \geq and \leq . These notations are used instead of using {...} as in the UML Timing diagram. We create our Timing diagram by selecting some notations from the UML Timing diagram and adding new features into our Timing diagram. We call an individual participant in the interaction of the Timing diagram an *object*. The Timing diagram which uses in this research is based on the Robust Timing diagrams notation. The notations for graphic nodes and paths to

be included in the Timing diagram are described as in table 1.

Table 1. Timing diagram notations

Node Type and Notation	Reference
Cause / Effect 	An arrowed line indicates cause and effect between objects. The beginning of line represents the cause while the end of the line (with arrow) represents the effect.
Condition 	Conditions are additional circumstances that cause a state changing. A condition is a state predicate. They are optional and are described by plain text. The conditions are presented above the arrowed line.
Duration constraint $[t1 .. t2]$ or $(t1..t2)$ or $[t1 .. t2)$ or $(t1..t2]$ or $\geq t1$ to $\leq t2$	Duration indicates time constraints and is used to describe how long a state or value must be in effect. Time unit in the duration constraint can be second or minute. The duration constraints can be identified by using symbols i.e. $[t1..t2]$ indicates the time constraint starts from t1 to t2. Sign $(t1..t2)$ indicates the time constraint in between t1 to t2. Sign $>$, $<$, \geq and \leq indicates grater than, less than, greater or equal than, and less or equal than, respectively. “to” indicates time interval.
Synchronization 	Synchronization is represented by dash-line and is used to synchronize duration constraints between objects.
State 	State notation is used to indicate all possible states of an object.

Considering the UML Timing diagram notations' definition [12], *Message* is a specific kind of communication. Thus, our *Cause/Effect* notation is a particular kind of *Message* which specifies a requested state change. As same as the *Condition* notation, it is a kind of *Message* which is used for identifying predicates. The *Condition* is used as extra information apart from whatever mentions in the *Cause/Effect*. The *Duration constraint* and *Synchronization* notations are new artifacts we introduce here. The *Synchronization* is not only used to synchronize duration constraints between objects, but also used to identify which objects change state simultaneously to other objects. The *State* notations are as same as states in the UML 2.0 Timing diagram. An example of TD is illustrated in figure 3.

3. A case study: Lift System

We select the lift system specification based on Jackson's work [5]. The specification is modified to have features suitable for modeling with the Timing diagram. Parts of the specification are described in the following.

- The lift will be stopped at the requested floor within a time interval of 5 – 10 seconds after floor sensor is set on.
- Whenever the lift starts moving up/down, current floor sensor will be off within a time interval of 5 – 10 seconds after the lift starts moving.
- Whenever the lift stops at the requested floor, the lift door will be opened within a time interval of 1 – 5 seconds.
- While the lift is moving, the lift's door must be closed.
- If the lift is stationary at floor f and there are new requests for other floors, then the lift door must be closed within a time interval of 1 – 5 seconds.
- If the lift is stationary, the Up and Down arrows lamp must be unlit.
- While the lift is moving up, the Up arrow lamp must be lit and the Down arrow lamp must be unlit. While the lift is moving down, the Down arrow lamp must be lit and the Up arrow lamp must be unlit.

According to the specification, we can identify *Floor Sensor*, *Lift*, *Door*, *UpArrow* and *DownArrow* as objects. Figure 3 illustrates an example of the Timing diagram which corresponds to the specification where f and g represent floors. Predicates $reqFl(f) = True$, $reqFl(g) = True$, $g \neq currentFl$ and $f = currentFl$ are additional conditions in plain text annotating the arrowed lines. The predicates $reqFl(f) = True$ and $reqFl(g) = True$ are used to indicate there is

a request for floor f and g , respectively. We use three colors, blue, red and green, for different *Cause/Effect* arrowed lines to separate the distinctive actions. Symbols (1), (2) and (3) are not part of the Timing diagram notations. We merely use them for explanation of examples in this paper.

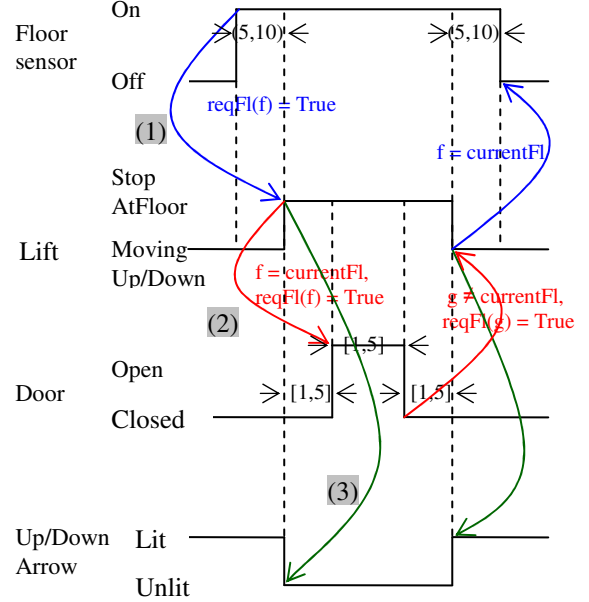


Figure 3. Timing diagram

4. Transforming Timing diagram into KAOS goal models

4.1 Patterns

This section describes patterns which are used to create an individual goal from Timing diagrams. There are two kinds of patterns: pattern to generate variables using in a goal formal definition, and pattern to generate an individual goal.

4.1.1 Patterns to generate variables using in a goal formal definition. The patterns are described in the following

-A variable name is identified by object's name following with a word "State". For example, the object *Door* is created to be a variable as *DoorState* while the object *Lift* is created as *LiftState*.

-Values of a variable are obtained from its states. For example, the object *Door* composes of *Open* and *Closed* states. The variable which is generated from the *Door* is the *DoorState*. Thus, values of this variable are *DoorState = Open* and *DoorState = Closed*.

4.1.2 Patterns to generate an individual goal. The patterns are described in the following.

- Each arrowed line in the Timing diagram is generated as a goal.
- Addition conditions on arrowed lines and cause of changing states (which is represented by states at the beginning of *Cause/Effect* arrowed lines) are expressed before “ \Rightarrow ” notation in goal’s formal definition (*FormalDef*).
- Expressions after “ \Rightarrow ” notation are obtained from resulting state changing (*Effect*) of the *Cause/Effect* arrowed lines.
- Time constraints are used to identify with temporal logic operators.

Figure 2, 4 and 5 illustrate examples of goals which are achieved from our patterns. The goal in figure 4 explains requirement specification *c* in section 3. According to figure 3, the arrowed line number (2), *f* is a global variable while expressions $f = \text{currentFl}$ and $\text{reqFl}(f) = \text{True}$ are additional conditions. An expression $\text{LiftState} = \text{'StopAtFloor'}$ is a cause of changing state which is also the cause of the arrowed line. Since this goal is declared as an *Achieve*, we use eventually temporal logic “ \Diamond ” to describe time constraint. An expression $\Diamond_{[1,5]} \text{DoorState}(f) = \text{'Open'}$ is derived from the effect of the arrowed line.

Achieve[DoorOpen]

Definition: Door will be opened within a time interval of 1-5 seconds after lift stops at a requested floor.

FormalDef : $\forall f : \text{Floor}$

$f = \text{currentFl} \wedge \text{reqFl}(f) = \text{True}$

$\text{LiftState} = \text{'StopAtFloor'}$

\Rightarrow

$\Diamond_{[1,5]} \text{DoorState}(f) = \text{'Open'}$

Figure 4. Goal Achieve[DoorOpen]

Figure 5 illustrates goal model which expresses the requirement specification *f* and Timing diagram arrowed line number (3). An expression $\text{LiftState} = \text{'StopAtFloor'}$ is obtained from cause of the *Cause/Effect* arrowed line. An expression $\text{UpArrowState} = \text{'Unlit'} \vee \text{DownArrowState} = \text{'Unlit'}$ illustrates effect of the *Cause/Effect* arrowed line. Since the type of goal is *Maintain*, the shape of temporal logic operator which suitable with the goal is $P \Rightarrow Q$.

Maintain[ShowLiftStopAtFloor]

Definition : While the lift is stationary, the Up and Down arrow lamps must be unlit.

FormalDef : $\text{LiftState} = \text{'StopAtFloor'}$

\Rightarrow

$\text{UpArrowState} = \text{'Unlit'} \wedge$

$\text{DownArrowState} = \text{'Unlit'}$

Figure 5. Goal Maintain[ShowLiftStopAtFloor]

As mention in section 2.1, the *Goal model* is represented as a tree structure. Usually, we first create the root node which expresses a general goal of the system. Next, to create the *Goal model*, subgoals are generated by broking down the root node corresponding to KAOS goal refinement patterns [8]. In contrast, in this paper, we start by identifying leaf node goals. Each leaf node goal is an individual goal derived from applying our patterns to Timing diagrams. Combination of the leaf node goals forms a parent goal. That is, our *Goal model* is created by applying KAOS goal refinement pattern to those individual goals. We call creating the *Goal model* in this way a “bottom-up” technique, because we know exactly the whole detail of the system specification since from the Timing diagram. We found that, it is easier to obtain the *Goal model*, following the pattern of goal refinement, by using the “bottom-up” technique than using a “top-down” practice. The top-down technique starts from identifying a parent goal then refining it into a combination of subgoals. Even though the top-down technique is easy to declare the top level of goal tree, we found it is later hard to identify the parent goal into subgoals with a pattern.

4.2 Identify a parent goal from a pattern of the Timing diagram

Even though we start from identifying subgoals, then combining of subgoals to form a parent goal, the relationship between the subgoals and their corresponding parent goal should be defined by a pattern as well. This section introduces two Timing diagram patterns which are found in the Lift System case study. Those patterns regularly appear in the Timing diagram and are often used as a *Cause/Effect* relationship of changing states.

4.2.1 Timing diagram pattern 1. Arrowed lines in the Timing diagram have a pattern as a *transitive relationship*.



Figure 6. Timing diagram for transitive relationship

Figure 6 illustrates an example of a *transitive relationship* which is usually found in the Timing diagram; where P , Q , and R are objects' states in the Timing diagram. The transitive relationship in the Timing diagram corresponds to a KAOS goal model pattern: *Split lack of Monitorability/Control with Milestone* [6, 10], as illustrated in figure 7; where P , Q , and R are predicates in KAOS goal formal definitions. This KAOS goal pattern is used to resolve the lack of monitorability/control for variables in the antecedent of an *Achieve* goal. It introduces some intermediate milestones in between the first and the last goals. According to figure 7, Q is an intermediate milestone in this pattern.

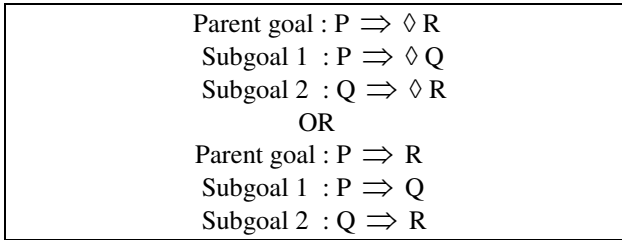


Figure 7. Split lack of Monitorability/Control with Milestone

The *Goal model* which is created by this pattern illustrated in figure 8.

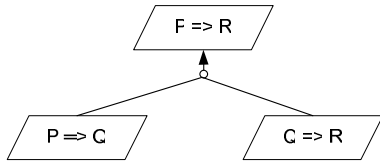


Figure 8. Goal model pattern for transitive relationship

As shown in figure 3, there are two transitive relationships: $(1) \rightarrow (2)$ and $(1) \rightarrow (3)$ in the Timing diagram. At this point, we select the first relationship to demonstrate an example; where (1) represents the goal *Achieve[LiftStopAtFloor]* while (2) represents the goal *Achieve[DoorOpen]*. Thus, P is identified from predicates which appear before “ \Rightarrow ” notation of the goal *Achieve[LiftStopAtFloor]*. Since, Q is an intermediate milestone, Q is defined from both predicates which appear after and before “ \Rightarrow ” notation of the goal *Achieve[LiftStopAtFloor]* and the goal *Achieve[DoorOpen]*, respectively. P , Q and R are defined in the following.

$P : \forall f : \text{Floor}$
 $\text{reqFl}(f) = \text{True} \wedge$
 $\text{FloorSensorState}(f) = \text{'On'}$

$Q : f = \text{currentFl} \wedge \text{reqFl}(f) = \text{True} \wedge$
 $\text{LiftState} = \text{'StopAtFloor'}$

$R : \text{DoorState}(f) = \text{'Open'}$

Note that, we ignore time constraints while applying the pattern for generating a parent goal. That is because those time constraints are previously declared in leaf node goals as a detail design. A parent goal which is created from this pattern is illustrate in figure 9.

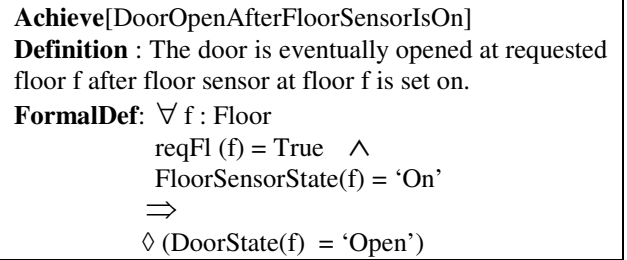


Figure 9. Goal
Achieve[DoorOpenAfterFloorSensorIsOn]

4.2.2 Timing diagram pattern 2. Arrowed lines have a pattern as a *case-driven relationship*. Figure 10 illustrates this pattern where P , Q and R are objects' states in the Timing diagram. The case-driven pattern in The Timing diagram corresponds to KAOS goal model pattern *Case-driven Split consequent* as illustrated in figure 11; where P , Q , and R are predicates in KAOS goal formal definition. This pattern is used for splitting lack of monitorability by cases.

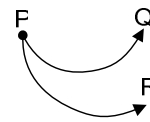


Figure 10. Timing diagram for case-driven relationship

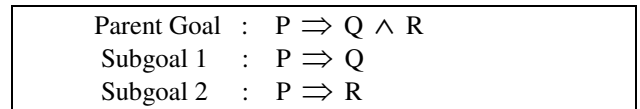


Figure 11. KAOS goal model pattern:
Case-driven Split consequent

There are two case-driven relationships in the Timing diagram. At this point, we select only one case-driven relationship for demonstrating an example. That is, we select effect of the arrowed line (1), *LiftState* = "StopAtFloor" as *P*, where *P* is later a cause of *Q* and *R*. *Q* equals to $\Diamond_{[1,5]}$ *DoorState*(*f*) = *Open* which is derived from effect of the arrowed line (2). *UpArrowState* = "Unlit" and *DownArrowState* = "Unlit" represent *R* which they are effect of the arrowed line (3). *P*, *Q* and *R* are used for creating a parent goal described in the following.

- P*: $\forall f : \text{Floor}$
 $f = \text{currentFl} \wedge \text{reqFl}(f) = \text{True} \wedge$
 $\text{LiftState} = \text{'StopAtFloor'}$
- Q*: $\Diamond_{[1,5]} (\text{DoorState}(f) = \text{'Open'})$
- R*: $(\text{UpArrowState} = \text{'Unlit'} \wedge$
 $\text{DownArrowState} = \text{'Unlit'})$

Thus, the parent goal which is created by KAOS Case-driven pattern is illustrated in figure 12.

Achieve[DoorOpenAndUp&DownArrowUnlitWhenLiftStopAtFloor]

Definition : Door is opened, up and down arrows are unlit when lift stops at floor.

FormalDef: $\forall f : \text{Floor}$
 $f = \text{currentFl} \wedge \text{reqFl}(f) = \text{True} \wedge$
 $\text{LiftState} = \text{'StopAtFloor'}$
 \Rightarrow
 $\Diamond_{[1,5]} (\text{DoorState}(f) = \text{'Open'}) \wedge$
 $(\text{UpArrowState} = \text{'Unlit'} \wedge$
 $\text{DownArrowState} = \text{'Unlit'})$

Figure 12. Goal

Achieve[DoorOpenAndUp&DownArrowUnlitWhenLiftStopAtFloor]

Applying the *Case-driven Split consequent* pattern to generate the parent goal *Achieve*[DoorOpenAndUp&DownArrowUnlitWhenLiftStopAtFloor] causes the goal *Maintain*[ShowLiftStopAtFloor] to be modified. This is, the parent goal has the expression $f = \text{currentFl}$ and $\text{reqFl}(f) = \text{True}$ which has never appeared in the subgoal *Maintain*[ShowLiftStopAtFloor]. Thus to make our goal refinement process accurate, we have to add the expression $f = \text{currentFl}$ and $\text{reqFl}(f) = \text{True}$ to the goal *Maintain*[ShowLiftStopAtFloor]. Adding this expression does not demolish the goal. Figure 13 illustrates the adjusted goal

Maintain[ShowLiftStopAtFloor] while *Goal model* is shown in figure 14.

Maintain[ShowLiftStopAtFloor]

Definition : While the lift is stationary, the up and down arrows must be unlit.

FormalDef : $\forall f : \text{Floor}$
 $f = \text{currentFl} \wedge \text{reqFl}(f) = \text{True} \wedge$
 $\text{LiftState} = \text{'StopAtFloor'}$
 \Rightarrow
 $\text{UpArrowState} = \text{'Unlit'} \wedge$
 $\text{DownArrowState} = \text{'Unlit'}$

Figure 13. Goal Maintain[ShowLiftStopAtFloor] (modified)

Each subgoal is combined to a parent goal by using "AND" relationship. As shown in figure 14, to accomplish the goal *Archeive*[DoorOpenAfterFloorSensorIsOn], the goal *Achieve*[LiftStopAtFloor] and the goal *Achieve*[DoorOpen] must be selected.

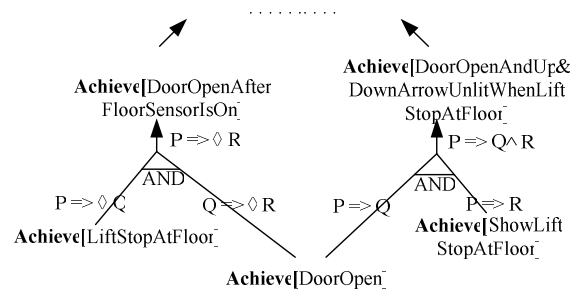


Figure 14. Goal model

5. Conclusion

The point of our work is to establish a pattern to support transformation from the Timing diagram into KAOS goal model where the Lift System is used as a case study. We introduce our Timing diagram by modifying the original UML 2.0 Timing diagram. Some graphical notations are added and colors are used for the purpose of clearly identifying each object. We define patterns to create variables and individual goals in the KAOS *Goal model* from the Timing Diagram. To create a *Goal model*, we use the bottom-up technique where a combination of subgoals creates a parent goal. After many attempts of transforming Timing diagrams into KAOS *Goal models*, we found some patterns of the Timing diagram which relate to patterns of KAOS *Goal model*. These patterns often found in the Timing diagram as cause-effect

relationship. At present, we found two kinds of patterns: transitive and case-driven relationship patterns; those patterns correspond with KAOS goal refinement patterns: *Split lack of Monitorability/Control with Milestone* and *Case-driven Split consequence*, respectively. There are some cases where creation of a parent goal is affected by its subgoals. To preserve the goal refinement concept, we have to modify those subgoals' formal definitions so that the meaning of the goal does not change. Our work actively provides an option for software developers to identify parts of LTL specifications in an easy way by using Timing diagram's graphical notations.

5.1 Related work

[3] introduces a technique to create a profile to allow the KAOS models to be represented in the UML. The UML is extended by introducing new stereotypes and tags which allow one to model the KAOS in the UML. However, no clear advantage is shown for this embedding of KAOS in UML. One should learn and understand how to use KAOS-UML apart from modeling only in the KAOS Framework. [1] introduces a technique and a prototype tool which describe temporal logics (TL) of a concurrent system's behaviors by using graphical notations. The researchers raise a lift system as a case study in exploring the tool. However, the graphical notations which are used to represent TL are more complicated than ours. That is, they introduce new notations apart from what those well-known in UML. Moreover, the tool does not support identifying time constraints along the system. The FAUST toolbox [13] is a set of KAOS tool including requirement checker, animator, pattern manager, obstacle generator and acceptance test generator. The tool box has features to support system analysts in specifying requirements in a graphically; we can use the tool box to verify our *Goal models*.

5.2 Further work

We aim to investigate a bigger subset of *Goal models* i.e. *Cease* and *Avoid*. Also, presently, the patterns in Timing Diagrams and LTL are of a very restricted form. We are going to extend the work in future to a richer set of Timing Diagrams/LTL patterns by looking for them from another case study: The Mine Pump Controller which is taken from [5]. Using another case study is not only for finding new patterns, but also examining for assessment and inaccuracy. Moreover, we plan to explore automatic refinement patterns. We would like to discover a technique to

automatically generate KAOS models from Timing diagrams.

References

- [1] L. K. Dillon, G. Jutty, L. E. Moset, P. M. Melliar-Smith, S. Ramakrishna, *A Graphical Interval Logic for Specifying Concurrent Systems*, ACM Transactions on Software Engineering and Methodology, Vol. 3, No 2, April 1994, pp. 131-165.
- [2] R. Darimont and A. V. Lamsweerde, *Formal Refinement Patterns for Goal-Driven Requirements Elaboration*, In Proceeding of FSE'4 – Fourth ACM SIGSOFT Symp. on the foundations of Software Engineering, SanFrancisco, 1996.
- [3] W. Heaven and A. Finkelstein, *UML profile to support requirements engineering with KAOS*, IEE Proceedings Software, Volume 151, Issue 1, February, 2004, pp. 10-27.
- [4] M. Jackson, *Problem Frames Analysis and Structuring Software development problems*, Addison-Wesley, 2001, pp. 48 - 75, 177 – 191.
- [5] M. Joseph, *Real-Time Systems Specification, Verification and Analysis*, Prentice Hall Intl., 1996.
- [6] A. V. Lamsweerde, *Goal-Oriented Requirements Engineering: A Guided Tour; Invited mini-tutorial paper*, In Proceeding of RE' 01, 5th IEEE International Symposium on Requirements Engineering, Toronto, Aug 2001, pp. 249-263.
- [7] A. V. Lamsweerde, A. Dardenne, B. Delcourt and F. Dubisy, *The KAOS Project: Knowledge acquisition in automated specifications of software*, In Proceedings of AAAI Spring Sysm-posium series, Stanford University, March 1991, pp. 59-62.
- [8] E. Letier, *Reasoning about Agents in Goal-Oriented Requirement Engineering*, Phd. Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, May 2001.
- [9] E. Letier and A. V. Lamsweerde, *Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering*, In Proceeding of SIGSOFT' 904/FSE -12, USA, Oct 31-Nov 6, 2004.
- [10] E. Letier and A. V. Lamsweerde, *Deriving Operational Software Specifications from System Goals*, In Proceeding of SIGSOFT 2002/FSE-10, Charleston, SC, USA, Nov 18 – 20, 2002.
- [11] Z. Mana and A. Pnueli, *The Temporal Logic of Reactive and Concurrent System*, Springer-Verlag, 1991.
- [12] OMG, *UML Superstructure Specification v2.0*, Available at <http://www.omg.org/cgi-bin/doc?formal/05-07-04>.
- [13] C. Ponsard, P. Massonet and J.F. Molderez, *Analyzing Fault-Tolerant Systems with FAUST*, In Proceeding of the Workshop on Rigorous Engineering of Fault-Tolerant System (REFT 2005), UK, July 19, 2005.