Indirect Control Path Analysis and Goal Coverage Strategies for Elaborating System Safety Goals in Composite Systems

Jennifer Black
Carnegie Mellon University
Pittsburgh, PA USA
jenmorris@cmu.edu

Philip Koopman
Carnegie Mellon University
Pittsburgh, PA USA
koopman@cmu.edu

Abstract

Correctly specifying requirements for composite systems is essential to system safety, particularly in a distributed development environment. Goal-oriented requirements engineering can be used to formally specify system goals and decompose them into realizable subgoals for system components. However, an additional aim of safety goal elaboration is to meet a goal coverage strategy. In this paper we propose new tactics for elaborating system safety goals across a composite system. First, Indirect Control Path Analysis (ICPA) is used to identify safety-related components and their relationships to the parent goals. Then, goal coverage strategies guide goal elaboration along indirect control paths identified by the ICPA. We demonstrate applicability in real safety critical embedded systems with two case studies: a distributed elevator and a semiautonomous automotive system.

1. Introduction

Research has shown that system safety, defined by Leveson as "freedom from accidents or losses" [13], is closely linked to requirements. One case study of safety-critical software isolation revealed some interfaces between safety-critical and non-safety-critical components were missed during safety analysis [1]. Other research has linked safety-related software errors and operational anomalies to latent requirements, misunderstood requirements, and misunderstood interfaces between the physical system and the software [15, 16].

System safety is difficult to define and ensure in a distributed development environment. In the study of software errors in spacecraft [15], miscommunication between development teams was the primary cause of safety-related interface faults. In the automotive industry, vehicle subsystems are often developed by different internal departments or external suppliers, often without access to vehicle-level requirements or requirements for other subsystems. In order to manage safety at the subsystem level prior to system integration, system safety requirements must be clearly defined for subsystems.

Unfortunately, decomposing non-functional requirements, also known as goals [6] or quality attributes [9, 2], is not straightforward. Some quantitative goals, such as cost or performance, may be decomposed by allocating a fixed limit on each component in the functional decomposition [17]. However, other goals may be qualitative or not easily represented as a sum of parts. For example, an automotive safety goal might be "the vehicle shall experience zero collisions." Unlike performance goals, where the concept of "time" is the same for systems and subsystems, the concept of "collision" in system safety does not have the same meaning at lower levels of the system hierarchy.

This paper addresses the problem of elaborating system safety goals across a composite system. Related work in goal-oriented requirements engineering is described in Section 2. The main contributions are: the ICPA technique for identifying indirect control sources of goal variables (Section 3), a definition of *goal coverage strategy* and classifications of *goal assignment* and *goal scope* (Section 4). Safety goals from a distributed elevator system are used to demonstrate ICPA and the goal coverage strategies. In Section 5, the full technique is applied to a safety goal for a semiautonomous automotive system. Section 6 contains a discussion of the limitations of the approach, and future work. Finally, conclusions are presented in Section 7.

2. Related work

Formal reasoning about specification of composite systems was introduced in [8], which proposed separate specifications for describing decomposition of a system into components and for describing composite system behavior. Specifications of component behaviors were derived by pruning (eliminating behaviors which vio-

P	current state	$\neg$**P**	false
$\bigcirc$**P**	next state	$\bullet$**P**	previous state
$\Diamond$**P**	current or some future state	$\blacklozenge$**P**	some previous state
$\square$**P**	current and all future states	$\blacksquare$**P**	all previous states
@**P**	$\bullet\neg$P $\wedge$ P; true in current state, but not in previous state		
P $\rightarrow$ **Q**	P implies Q	**P** $\Leftrightarrow$ **Q**	P iff Q
P $\Rightarrow$ **Q**	$\square$(P $\rightarrow$ Q); P implies Q in all states		
$\bullet\blacksquare_{<T}$**P**	true for duration T in previous state		
$\bullet\blacklozenge_{<T}$**P**	true at least once in duration T in previous state		

Figure 1. Temporal logic operators

late some constraint) and decomposing constraints to assign to agents.

A framework for specifying non-functional requirements in composite systems was proposed in [17]. It presented a process-oriented approach that defined non-functional requirements in terms of goals, links between goals, methods for goal refinement, correlation rules, and a labeling structure for linking goals to design decisions. Similar to our approach to safety goal decomposition, accuracy goals were decomposed by assigning a separate goal to each component of the information (e.g. report accuracy was divided by report types, report sections, and function and input parameters).

Other approaches to specifying non-functional requirements include intent specifications [14] and safety patterns [3, 4]. Intent specifications provide hierarchical system abstraction that relies on means-to-ends representation, rather than part-to-whole. Safety patterns attempt to make formal specification more accessible to non-formalists by mapping natural language representations of safety requirements to formal temporal logic patterns. These approaches provide frameworks for expressing system safety goals, but do not offer tactics for safety goal decomposition.

2.1. Goal-oriented requirements engineering

The approach proposed in this paper builds primarily upon goal-oriented requirements engineering as defined by the KAOS framework [6]. In goal-oriented requirements elicitation, goals are constructed in temporal logic expressions [10]. Figure 1 lists the temporal operators of KAOS used in this paper.

One advantage of using formal specifications is that goal structure can guide elaboration. High-level goals are refined into sub-goals by *AND/OR* reductions [18], by applying formal refinement patterns for logical expressions [7], and by defining subgoals that are realizable by particular agents [11]. Goal patterns also guide operationalization into constraints and triggers for actions performed by agents in the system [12]. In addition, there are several techniques for identifying and resolving conflicts between goals [20].

Additional tactics are needed for safety because existing tactics define subgoals that exactly meet the parent goal without being more restrictive or redundant [11]. Safety goals may require redundancy or restriction beyond exactly meeting the goal in order to take into account component reliability.

3. Indirect Control Path Analysis (ICPA)

The aim of goal elaboration is to define subgoals that meet the parent goal and are realizable by agents in the system (i.e. can be operationalized) [11]. An additional aim of *safety* goal elaboration is to employ a goal coverage strategy. To do this, it is first necessary to identify all potential agents that may influence the safety goal.

This section presents a new technique for identifying indirect control sources of goal variables called Indirect Control Path Analysis (ICPA). ICPA is a top-down search, similar to Fault Tree Analysis (FTA) [13]. Whereas FTA traces a top-level hazard to its lower-level causal events, ICPA traces a top-level state variable through the design to all components that influence it. ICPA uses a table structure similar to Failure Modes and Effects Analysis (FMEA) [13] to record these indirect control relationships. Agents along the trace path belong to the indirect control path and require further analysis of their relationships to the root variable.

3.1. Identifying indirect control sources

In the KAOS framework, goal realizability is driven by monitorability and controllability of system state variables [11]. A goal relation can be expressed as $G_{(M,C)}$, where G is the goal to be realized in the system, M is the set of variables in the goal to be monitored, and C is the set of variables in the goal to be controlled. A goal is realizable by an agent if M is a subset of the variables monitored by agent ag and C is a subset of the variables controlled by agent ag, ($M \subseteq Mon(ag)$ and $C \subseteq Ctrl(ag)$).

In KAOS, only one agent may directly change the value of a given state variable [11]. However, there may be other agents in the system that influence how those variables are controlled. Consider the following goal that restricts movement in an overweight elevator:

Goal: Achieve[DriveStoppedWhenOverweight]
InformalDef: *If the elevator weight exceeds the weight threshold, then the drive shall be commanded to stop.*
FormalDef: $\forall$ e: Elevator, dr: Drive, wt: WeightThreshold
$\bullet$ (e.weight $>$ wt) $\Rightarrow$ dr.Command = 'STOP'

The drive controller directly changes dr.Command, but it may do so based on input from an elevator scheduling agent. The goal coverage strategy may require additional subgoals for these types of influential agents. Furthermore, the notion of direct control in this paper does not require strict controllability; more than one agent may directly control a system state variable. In the same elevator system, a hall button controller on each floor may generate a hall call message on the network. It is

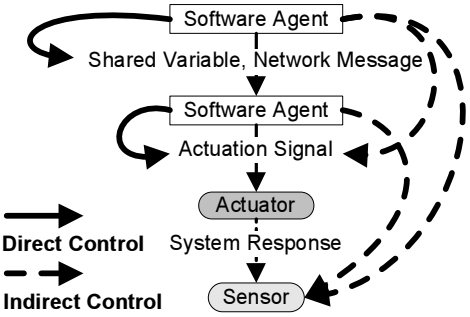

Figure 2. Indirect control paths

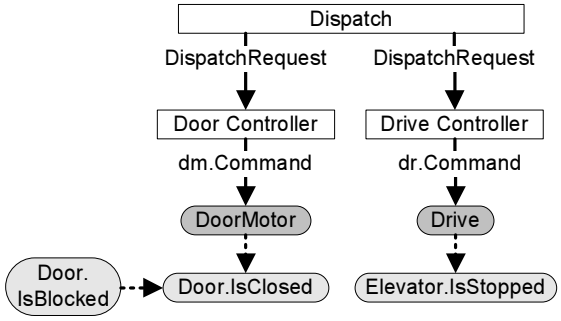

Figure 3. Elevator indirect control paths

necessary to apply goals that constrain these messages to all agents that produce them.

The terms *direct control* and *indirect control* are introduced to distinguish between the ability to change and the ability to influence change in variables. Figure 2 shows the direct and indirect control relationships for an embedded system with sensing and actuation.

3.2. Defining indirect control relationships

Once indirect control sources have been identified, their relationship to the original variable must be defined in such a way that the general agent-based elaboration tactic *Introduce Actuation Goal* from [11] can be applied. If these relationships can be defined in the form $(o = c)$ or $(Q \Rightarrow P)$, then the goal G can be defined as $G(o|c)$ or $G(P|Q)$. In other words, functions that relate the variable in the parent goal to the indirect control variables must be defined.

For paths with a single branch, the indirect control relationship is defined between each pair of agents along the path. Figure 3 shows indirect control paths for two sensed values in a distributed elevator system. A safety goal that restricts these sensed values might be:

Goal: Maintain[DoorClosedOrElevatorStopped]
InformalDef: *At all times the door shall be closed or the elevator shall be stopped.*
FormalDef: ∀ do: Door, e: Elevator
❑ (do.IsClosed ∨ e.IsStopped)

Indirect control paths for the variables in this goal are presented in Table 1. The control path of e.IsStopped includes the drive controller, via drive actuation, and dispatch, which tells the drive controller where to go.

The relationship between sensed value e.IsStopped and drive actuation indicates that when the drive is stopped the elevator will be stopped also:

$$\text{dr.Value} = \text{'STOP'} \Leftrightarrow \text{e.IsStopped} \quad (1)$$

However, a drive that is commanded to stop will do so after some delay:

$$\bullet\blacksquare_{<MaxStopDelay}\ \text{dr.Command} = \text{'STOP'} \Rightarrow \text{dr.Value} = \text{'STOP'} \quad (2)$$

$$\bullet\blacklozenge_{<MinStopDelay}\ (\neg(\text{dr.Value} = \text{'STOP'}) \wedge @(\text{dr.Command} = \text{'STOP'})) \Rightarrow \neg(\text{dr.Value} = \text{'STOP'}) \quad (3)$$

Variables with multiple branches require relationship among branches to be defined. Sometimes, the branches represent independent control paths (i.e., one path traversed at a time). In these situations each branch can be evaluated as if it were a single branch path. In others, the branches represent coordinated indirect control between agents. In Figure 3, the door controller commanding the door motor to close the doors should eventually set the do.IsClosed sensor to 'TRUE'. However, objects or passengers blocking the doors can physically prevent the door from being closed:

$$\bullet\ \text{do.IsBlocked} \Rightarrow \neg\text{do.IsClosed} \quad (4)$$

The relationship between blocking agents and the door motor is also constrained by a related safety goal that requires a door reversal if the door is blocked. As a design choice for this system, this safety goal is given priority over Maintain[DoorClosedOrElevatorStopped]. The resulting indirect control relationships are:

$$\bullet\blacksquare_{<MaxCloseDelay}\ (\neg\text{do.IsBlocked} \wedge (\text{dm.Command} = \text{'CLOSE'})) \Rightarrow \text{do.IsClosed} \quad (5)$$

$$\bullet\blacklozenge_{<MinCloseDelay}\ (\neg\text{do.IsBlocked} \wedge \neg\text{do.IsClosed} \wedge @(\text{dm.Command} = \text{'CLOSE'})) \Rightarrow \neg\text{do.IsClosed} \quad (6)$$

ICPA is a structured way to organize potential sources of indirect control and their relationships. The results of the ICPA will be applied with a goal coverage strategy to define subgoals for safety-related agents.

4. Goal coverage strategies

A *goal coverage strategy* is a plan for allocating subgoals to ensure that a high-level goal is met. Each strategy is defined by *goal assignment* and *goal scope*.

4.1. Goal assignment

Goal assignment defines which indirect control sources have subgoals and how those subgoals relate to each other. It may be driven by physical limitations of the system (e.g., actuation delays described in Section 3.2).

Table 1. ICPA for goal Maintain[DoorClosedOrElevatorStopped]

Goal: Maintain[DoorClosedOrElevatorStopped]			$\forall$ **do: Door, e: Elevator;** $\Box$ **(do.IsClosed** $\vee$ **e.IsStopped)**
Variable	**Physical**	**Software**	**Indirect Control Relationships**
do.IsClosed *Sensed*	DoorMotor (dm) *Actuated*	dm.Command	$\bullet\blacksquare_{<MaxCloseDelay}$ ($\neg$do.IsBlocked $\wedge$ (dm.Command = 'CLOSE')) $\Rightarrow$ do.IsClosed
	Passenger (p) *User*	do.IsBlocked	$\bullet\blacklozenge_{<MinCloseDelay}$ ($\neg$do.IsBlocked $\wedge$ $\neg$do.IsClosed $\wedge$ @(dm.Command = 'CLOSE')) $\Rightarrow$ $\neg$do.IsClosed $\bullet$ do.IsBlocked $\Rightarrow$ dm.Command = 'OPEN' $\bullet$ do.IsBlocked $\Rightarrow$ $\neg$do.IsClosed
e.IsStopped *Sensed*	Drive (dr) *Actuated*	dr.Command	dr.Value = 'STOP' $\Leftrightarrow$ e.IsStopped $\bullet\blacksquare_{<MaxStopDelay}$ dr.Command = 'STOP' $\Rightarrow$ dr.Value = 'STOP' $\bullet\blacklozenge_{<MinStopDelay}$ ($\neg$(dr.Value = 'STOP') $\wedge$ @(dr.Command = 'STOP')) $\Rightarrow$ $\neg$(dr.Value = 'STOP')

It may also be influenced by possible loss of monitorability and controllability by agents in the system. The three categories of goal assignment presented in this section are *single responsibility*, *redundant responsibility*, and *coordinated responsibility*.

4.1.1. Single responsibility. In the base case for general goal elaboration, the safety goal is met by assigning one or more subgoals to a single agent. For system safety, a single responsibility goal assignment facilitates isolation of safety-critical behaviors from other non-critical components. It also allows more rigorous (and expensive) development processes to be applied to only those isolated, fewer agents. The agent responsible for meeting the goal may be responsible for other, non-critical functionality. Alternately, an agent's behaviors may be limited to the safety goal. Consider the following safety goal that restricts elevator position relative to the end of the hoistway:

Goal: Maintain[ElevatorBelowHoistwayUpperLimit]
InformalDef: *The top of the elevator shall not exceed the upper limit of the hoistway.*
FormalDef: $\forall$ e: Elevator, h: Hoistway
$\Box$ (e.Top $\leq$ h.UpperLimit)

This goal could be met by requiring the drive controller to stop the elevator before the hoistway limit is reached:

Goal: Achieve[StopBeforeHoistwayUpperLimit]
InformalDef: *If the elevator nears the upper hoistway limit, then the drive shall be stopped.*
FormalDef: $\forall$ e: Elevator, dr: Drive
$\bullet$(e.Top $\geq$ dr.UpperStoppingPoint)
$\Rightarrow$ dr.Command = 'STOP')

Another solution is to have an emergency brake, triggered by a physical switch or software monitor, stop the elevator before the end of the hoistway:

Goal: Achieve[EmergencyStopBeforeHoistwayUpperLimit]
InformalDef: *If the elevator nears the upper hoistway limit, then the emergencybrake shall be applied.*
FormalDef: $\forall$ e: Elevator, eb: Emergency Brake
$\bullet$(e.Top $\geq$ eb.UpperTriggerPoint)
$\Rightarrow$ eb.Command = 'APPLY')

In this example, the drive controller is responsible for passenger delivery and safety, whereas the emergency brake is responsible only for passenger safety.

4.1.2. Redundant responsibility. Functional redundancy is a common strategy for fault tolerance in which different agents perform the same set of required functions [13]. This redundant functionality may be identical, such as duplicate networks for tolerating dropped messages, or different, such as a backup that provides a minimal set of functions when the primary fails.

Goal redundancy is achieved by assigning primary responsibility for a goal to one agent; secondary, to one or more others. If at least one of the agents meets its subgoal, the parent goal will be met. If the subgoals vary in restriction, the agent with primary responsibility for the goal has the most restrictive subgoals and agents with secondary responsibility have less restrictive subgoals (i.e., normal behavior has a greater safety margin than emergency backup behavior). Goal scope is discussed in more detail in Section 4.2.

In the elevator system, the motion controller reliability may be too low or too unmeasurable to ensure the safety goal is met, particularly for complex software control. Physical component reliability, such as a physical emergency brake trigger, is better known. However, relying on the emergency brake alone to meet the safety goal is also undesirable because of equipment wear and harm to passengers with sudden stops. By assigning primary responsibility to the elevator drive controller and secondary responsibility to the emergency brake, the safety goal may be reliably met while largely avoiding application of the physical emergency brake.

4.1.3. Shared responsibility. Sometimes a safety goal may require coordination among agents, or physical system dynamics may limit the extent to which variables can be controlled. In shared responsibility, two or more agents are assigned subgoals that must all be met in order to meet the parent goal.

The goal Maintain[DoorClosedOrElevatorStopped] defined in Section 3.2 cannot be assigned to the drive controller or door controller alone because of physical actuation delays. Suppose the door controller alone is given responsibility for the goal with the subgoal:

$$\bullet\blacklozenge_{<MaxCloseDelay}(\neg \text{e.IsStopped} \wedge \neg\ \text{do.IsBlocked}) \Rightarrow \text{dm.Command = 'CLOSE'} \quad (7)$$

Operationalization of this goal prohibits opening the door while the elevator is in motion and prescribes closing the door if the elevator moves. If the drive controller activates the drive motor *while* the doors are already open, the safety goal will be violated while the door controller is closing the doors. Another subgoal is required to prevent the drive controller from moving the elevator while the doors are open:

$$\bullet\blacklozenge_{<MaxStopDelay}\ \neg\text{do.IsClosed} \Rightarrow \text{dr.Command = 'STOP'}) \quad (8)$$

Even though the behavior of both controllers is restricted, the goal may be violated when the elevator is stopped and the doors are closed, if the door controller attempts to open the doors at the same time as the drive controller attempts to move the elevator. By including both the sensed value of the monitored variable and its indirect control source, the two controllers may be able to avoid violating the parent goal. The new subgoals for the door controller and drive controller are:

$$\bullet\blacklozenge_{<MaxCloseDelay}(\neg \text{e.IsStopped} \wedge \neg\ \text{do.IsBlocked} \vee \neg(\text{dr.Command = 'STOP'})) \Rightarrow \text{dm.Command = 'CLOSE'} \quad (9)$$

$$\bullet\blacklozenge_{<MaxStopDelay}\ (\neg\text{do.IsClosed} \vee \neg(\text{dm.Command = 'CLOSE'})) \Rightarrow \text{dr.Command = 'STOP'} \quad (10)$$

The door controller monitors both elevator motion and drive commands. The drive controller monitors both the door closed sensor and the drive actuator commands. If the physical actuation delays are much smaller than the network message delays and there is atomic broadcast of system state in the composite system, each controller will be able to cancel its own actuation command when it observes the command of the other, before they have actually begun to open the doors or move the drive.

Sometimes physical actuation and network delays are insufficient for ensuring the goal is met. An *interlock* is a common solution for enforcing sequencing in coordinated actions [13]. Suppose a safety goal coordinating two actions takes the form $\square(A \vee B)$, where A is indirectly controlled by agent agA and B, by agB. The basic patterns of the primary subgoals are:

$$\bullet\neg\text{B} \Rightarrow \text{A} \quad (11)$$

$$\bullet\neg\text{A} \Rightarrow \text{B} \quad (12)$$

Before negating A, agA must set its own interlock variable LA and check that agB's interlock variable LB is not set. Basic patterns of interlock subgoals are:

$$\bullet(\neg\text{LA} \vee \text{LB}) \Rightarrow \text{A} \quad (13)$$

$$\bullet(\neg\text{LB} \vee \text{LA}) \Rightarrow \text{B} \quad (14)$$

Now suppose A and B have actuation delays, where $A1$ causes A to be set after some delay, and $A2$ causes A to be unset after some delay. The indirect control relationships for setting and unsetting A are defined as:

$$\bullet\blacksquare_{<MaxDelay_A}\ \text{A1} \Rightarrow \text{A} \quad (15)$$

$$\bullet\blacklozenge_{<MinDelay_A}\ (@\text{A1} \wedge \neg\text{A}) \Rightarrow \neg\text{A} \quad (16)$$

$$\bullet\blacksquare_{<MaxDelay_{\neg A}}\ \text{A2} \Rightarrow \neg\text{A} \quad (17)$$

$$\bullet\blacklozenge_{<MinDelay_{\neg A}}\ (@\text{A2} \wedge \text{A}) \Rightarrow \text{A} \quad (18)$$

$$\square\neg(\text{A1} \wedge \text{A2}) \quad (19)$$

If all variables shared between agents also have communication delays, the new subgoals for agA are:

$$\bullet\blacklozenge_{<MinComDelay}(\neg\text{B} \vee \text{B2}) \Rightarrow \text{A1} \wedge \neg\text{A2} \quad (20)$$

$$\bullet\blacklozenge_{<MinComDelay}\ (\neg\text{LA} \vee \text{LB}) \Rightarrow \text{A1} \wedge \neg\text{A2} \quad (21)$$

$$\square(MinComDelay < MaxDelay_{\neg A}) \quad (22)$$

The subgoals for agB are analogous.

A *lockout* coordinates enforcement of safety goals by prohibiting an action from occurring [13]. For example, a bus guardian is used to prevent faulty nodes on a network from interfering with communication by others. In time-triggered networks a bus guardian will enable transmission access only during the node's allotted time slot [19]. Suppose a safety goal takes the form $\bullet\blacklozenge_{<T}\ D \Rightarrow \neg C$, where C is indirectly controlled by agent agA and D is observed by agent agA. The control relationship of C by agA is defined by:

$$\bullet\text{A} \Rightarrow \text{C} \quad (23)$$

$$\bullet\neg\text{A} \Rightarrow \neg\text{C} \quad (24)$$

and the safety goal for agent agA is:

$$\bullet\blacklozenge_{<T}\text{D} \Rightarrow \neg\text{A} \quad (25)$$

If a lockout agent agB is added to the system to prevent agent agA from violating the safety goal, the new shared indirect control relationship would be:

$$\bullet(\text{A} \wedge \text{B}) \Rightarrow \text{C} \quad (26)$$

$$\bullet(\neg\text{A} \vee \neg\text{B}) \Rightarrow \neg\text{C} \quad (27)$$

The safety goal for agents agA and agB would be:

$$\bullet\blacklozenge_{<T}\ \text{D} \Rightarrow \neg\text{A} \quad (28)$$

$$\bullet\blacklozenge_{<T}\ \text{D} \Rightarrow \neg\text{B} \quad (29)$$

4.2. Goal scope

Goal scope defines how closely the safety subgoals meet the system safety goal. It may be possible for agents to meet the original safety goal without restriction. However, it may sometimes be necessary or desirable to assign subgoals that are more restrictive than the original safety goal.

4.2.1. Nonrestrictive. Nonrestrictive subgoals meet the parent goal with no additional limitations on functional behavior. This is the base case where the system-level goal is fully realizable. Subgoal Achieve[EmergencyStopBeforeHoistwayUpperLimit] from 4.1.1 is nonrestrictive if emergency brake dynamics ensure the elevator will stop at the very end of the hoistway when the emergency brake is triggered, allowing full use of the hoistway:

$$\text{e.Top} = \text{eb.StoppingDistance} + \text{eb.UpperTriggerPoint} \quad (30)$$

$$\text{h.UpperLimit} = \text{eb.StoppingDistance} + \text{eb.UpperTriggerPoint} \quad (31)$$

4.2.2. Restrictive. A restrictive subgoal meets the parent goal but places additional limitations on system functionality. The most common restrictive subgoal is achieved by OR reductions [18]. A goal of the form $\Box(A \vee B)$ is always met if subgoal A is met (A is always true). A is more restrictive than the parent goal because it excludes some functional behaviors that are non-hazardous: when A is false and B is true. Restrictive subgoals, which are usually less complex than the parent goal and possibly easier to implement correctly and analyze, may be necessary if variables are not controllable, or if control delays are great.

A subgoal may also be made restrictive by employing a *safety margin*, a hazard reduction technique for handling variability in failure rates of components [13]. For example, in the elevator system the emergency brake trigger point could be placed far enough away from the hoistway limit so that the elevator stops some distance before the hoistway limit, rather than at the end of the hoistway limit. The elevator would no longer have full use of the hoistway, but the safety goal could be maintained if the stopping distance of the emergency brake varies. If a safety goal has the form $\Box\ (A \leq B)$, then a subgoal with a safety margin C would be:

$$\Box\ (\text{A} \leq (\text{B - C})) \quad (32)$$

It may not always be possible to restrict subgoals if the restrictions make the final product unusable. The goal Maintain[ElevatorBelowHoistwayUpperLimit] can always be met if the elevator is always stopped, but this trivial solution prevents functionality required in an elevator.

5. Example: automotive acceleration

This section demonstrates how ICPA and goal coverage strategies are combined to generate subgoals for indirect control agents of a parent goal. Figure 4 shows indirect control paths for vehicle acceleration in a semiautonomous automotive vehicle. Two features, Adaptive Cruise Control (ACC) and Collision Avoidance (CA), indirectly control vehicle motion via brake and throttle actuation. ACC commands the vehicle to a speed

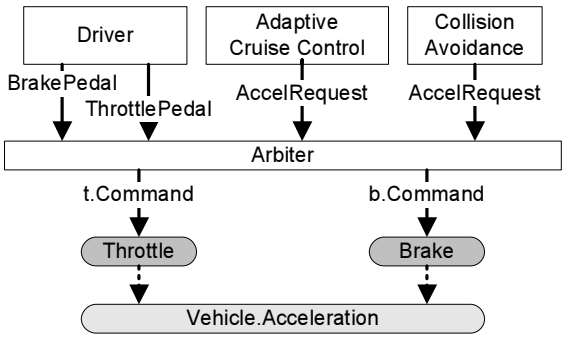

Figure 4. Indirect control of acceleration

set by the driver, or to a set following distance behind a slower lead vehicle. CA detects objects in the vehicle path and stops the vehicle to avoid them. A central arbiter chooses values for t.Command and b.Command based on the acceleration requests and the driver's brake and throttle pedal signals.

ICPA and goal coverage strategies have been applied to this system's safety goals. Due to space limitations, this section will focus on just one of these goals:

Goal: Achieve[AutoAccelBelowThreshold]
InformalDef: *Vehicle acceleration caused by autonomous vehicle control shall not exceed the threshold.*
FormalDef: $\forall$ b: Brake, t:Throttle, v: Vehicle, at:AccelThreshold
● (IsSubsystem(b.Source) $\vee$ IsSubsystem(t.Source)) $\Rightarrow$ v.Acceleration $<$ at

The intent is to prevent features from startling the driver by causing an unusually fast acceleration.

The ICPA for Achieve[AutoAccelBelowThreshold] is listed in Table 2. AccelFromThrottle() and AccelFromBrake() describe the acceleration caused by throttle and brake (brake acceleration is negative or zero). BrakeFromAccel() and ThrottleFromAccel() describe the brake and throttle components of an acceleration value.

Vehicle acceleration is indirectly controlled by coordinated brake and throttle actuation, defined by:

$$\text{v.Acceleration} = \text{AccelFromThrottle(t.Value)} + \text{AccelFromBrake(b.Value)} \quad (33)$$

Substituting (33) into the parent goal results in:

$$\bullet(\text{IsSubsystem(b.Source)} \vee \text{IsSubsystem(t.Source)}) \Rightarrow (\text{AccelFromThrottle(t.Value)} + \text{AccelFromBrake(b.Value)}) < \text{at} \quad (34)$$

This subgoal is not yet realizable because brake and throttle actuation are controlled indirectly by actuation commands. However, this vehicle employs electronic brake and electronic throttle with low actuation delays relative to communication and computation speed, such that the relationships between commands from the arbiter and actuation are defined by the simple equations:

Table 2. ICPA for goal Achieve[AutoAccelBelowThreshold]

Goal: Achieve[AutoAccelBelowThreshold] b: Brake, t:Throttle, v: Vehicle, at:AccelThreshold

● (IsSubsystem(b.Source) ∨ IsSubsystem(t.Source)) ⇒ v.Acceleration < at

Variable	Physical	Software	Indirect Control Relationships
v.Acceleration *Sensed*	Brake (b) *Actuated*	b.Command	v.Acceleration = AccelFromThrottle(t.Value) + AccelFromBrake(b.Value) b.Value = b.Command.Value
		AccelRequest (ar)	b.Command.Value = BrakeFromAccel(ar.Value) ar.Value = AccelFromThrottle(ThrottleFromAccel(ar.Value)) + AccelFromBrake(BrakeFromAccel(ar.Value))
		BrakePedal (bp)	b.Command.Value = BrakeFromPedal(bp.Value)
	Throttle (t) *Actuated*	t.Command	v.Acceleration = AccelFromThrottle(t.Value) + AccelFromBrake(b.Value) t.Value = t.Command.Value
		AccelRequest (ar)	t.Command.Value = ThrottleFromAccel(ar.Value) ar.Value = AccelFromThrottle(ThrottleFromAccel(ar.Value)) + AccelFromBrake(BrakeFromAccel(ar.Value))
		ThrottlePedal (tp)	t.Command.Value = ThrottleFromPedal(tp.Value)
b.Source *Computed*	Brake (b) *Actuated*	b.Command	b.Source = b.Command.Source
		AccelRequest (ar)	b.Command.Source = ar.Source (b.Source = ar.Source) ∨ (t.Source = ar.Source) ⇒ b.Source = t.Source = ar.Source ∧ b.Value = BrakeFromAccel(ar.Value) ∧ t.Value = ThrottleFromAccel(ar.Value)
		BrakePedal (bp)	b.Command.Source = bp.Source
t.Source *Computed*	Throttle (t) *Actuated*	t.Command	t.Source = t.Command.Source
		AccelRequest (ar)	t.Command.Source = ar.Source (b.Source = ar.Source) ∨ (t.Source = ar.Source) ⇒ b.Source = t.Source = ar.Source ∧ b.Value = BrakeFromAccel(ar.Value) ∧ t.Value = ThrottleFromAccel(ar.Value)
		ThrottlePedal (tp)	t.Command.Source = tp.Source

$$\text{b.Value} = \text{b.Command.Value} \tag{35}$$

$$\text{b.Source} = \text{b.Command.Source} \tag{36}$$

$$\text{t.Value} = \text{t.Command.Value} \tag{37}$$

$$\text{t.Source} = \text{t.Command.Source} \tag{38}$$

Substituting relationships (35)-(38) into to subgoal (34) gives the following subgoal for the arbiter:

$$\begin{aligned}&\bullet(\text{IsSubsystem(b.Command.Source)}\\&\quad \vee \text{IsSubsystem(t.Command.Source))}\\&\quad \Rightarrow (\text{AccelFromThrottle(t.Command.Value)}\\&\quad + \text{AccelFromBrake(b.Command.Value))} < \text{at}\end{aligned} \tag{39}$$

A single responsibility goal assignment can be assigned to the arbiter alone. But, this would make the arbiter a single point of failure for the goal. The arbiter chooses brake and throttle commands based on input from the driver brake and throttle pedals, and from ACC and CA acceleration requests. Another safety goal, designed to prevent unexpected feature interactions [5], ensures one feature at a time controls both the throttle command and the brake command:

$$\begin{aligned}&(\text{b.Source} = \text{ar.Source}) \vee (\text{t.Source} = \text{ar.Source})\\&\quad \Rightarrow \text{b.Source} = \text{t.Source} = \text{ar.Source}\\&\quad \wedge \text{b.Value} = \text{BrakeFromAccel(ar.Value)}\\&\quad \wedge \text{t.Value} = \text{ThrottleFromAccel(ar.Value)}\end{aligned} \tag{40}$$

The relationships between a feature (ACC or CA) and the brake and throttle commands are defined by:

$$\text{b.Command.Value} = \text{BrakeFromAccel(ar.Value)} \tag{41}$$

$$\text{b.Command.Source} = \text{ar.Source} \tag{42}$$

$$\text{t.Command.Value} = \text{ThrottleFromAccel(ar.Value)} \tag{43}$$

$$\text{t.Command.Source} = \text{ar.Source} \tag{44}$$

$$\begin{aligned}&\text{ar. Value} =\\&\quad \text{AccelFromThrottle(ThrottleFromAccel(ar.Value))}\\&\quad + \text{AccelFromBrake(BrakeFromAccel(ar.Value))}\end{aligned} \tag{45}$$

To achieve the redundant responsibility goal assignment, relationships (40)-(45) are substituted into (39) to get the following subgoal for both ACC and CA:

$$\text{ar.Value} < \text{at} \tag{46}$$

This can be made more restrictive by using a smaller acceleration threshold than that of the original goal.

6. Discussion and future work

Systematic analysis and record-keeping is vital to system safety. Although a set of subgoals may meet the parent goal, it is important to record why a particular set of subgoals is chosen and what process was followed to choose it. ICPA can facilitate identification of safety-critical agents and their relationships. Goal coverage strategies guide subgoal choice for these agents.

A limitation of the approach is that decompositions based on incorrect or incomplete system goals or specifications will be incorrect as well. As development progresses, the safety goals and their subgoals will have

to be reanalyzed. However, this is also true of requirements engineering and hazard analysis in general. Finally, ICPA is not intended to be an automatable synthesis approach. It provides guidance for choosing subgoals but will not decide which subgoals are "best."

A future automatable step is formal verification of the subgoals. State space explosion is a concern for verification of software systems, but the set of safety goals is much smaller than the set of all functional behaviors of the system. Future work will also include analysis of scalability of the technique, in general.

7. Conclusion

In this paper we introduce Indirect Control Path Analysis as an approach to elaborating system safety goals in composite systems based on goal-oriented requirements engineering. Critical agents are identified by tracing indirect control paths from parent goal variables. Relationships between indirect control sources and parent goal variables are cataloged to make critical assumptions about the safety subgoals explicit. Goal coverage strategies guide elaboration of subgoals and allocation to agents along the indirect control paths, and focus on defining subgoals that meet the system goal *and* achieve hazard reduction. We demonstrate that the techniques can be applied to two different safety-critical system domains: a distributed elevator and an automotive vehicle.

The importance of these contributions is twofold. First, analysis of potential subgoal agents is structured and documented. Second, the goal coverage strategy achieved by the chosen subgoals is deliberate and explicit. This not only facilitates correct implementation and analysis of safety throughout the design process, but also contributes to the documented safety case.

8. Acknowledgements

This work has been funded by the General Motors Collaborative Research Laboratory at CMU.

References

[1] E. A. Addy. A case study on isolation of safety-critical software. In *Proc. 6th Annual Conf. on Comp. Assurance (COMPASS'91)*, pages 75–83, June 1991.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1972.

[3] F. Bitsch. Classification of safety requirements for formal verification of software models for industrial automation systems. In *Proc. Intl. Conf. on SW and Sys. Engineering and their Applications (ICSSEA 2000)*, Paris, France, Dec. 2000.

[4] F. Bitsch. Safety patterns - the key to formal specification of safety requirements. In U. Voges, editor, *Proc. 20th Intl. Conf. on Comp. Safety, Reliability and Security (SAFECOMP '01)*, volume 2187 of *Lecture Notes in Computer Science*, pages 176–189, Budapest, Hungary, Sept. 2001. Springer-Verlag.

[5] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proc. 7th Intl. Conf. on SW Eng. for Telecom. Switching Sys.*, pages 59–62, Bournemouth, UK, July 1989.

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In M. Sintzoff, C. Ghezzi, and G. Roman, editors, *Science of Computer Programming*, number 1-2, pages 3–50. Elsevier Science, Amsterdam, The Netherlands, Apr. 1993.

[7] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. 4th ACM SIGSOFT Symp. on Found. of SW Eng.*, volume 21 of *Software Engineering Notes*, pages 179–190. ACM SIGSOFT, Nov. 1996.

[8] M. S. Feather. Language support for the specification and development of composite systems. *ACM Trans. on Prog. Lang., Syst.*, 9(2):198–234, Apr. 1987.

[9] S. E. Keller, L. G. Kahn, and R. B. Panara. Specifying software quality requirements with metrics. In R. H. Thayer and M. Dorfman, editors, *System and Software Requirements Engineering*, chapter 3, pages 145–163. IEEE Comp. Soc. Press, Los Alamitos, CA, 1990.

[10] R. Koymans, editor. *Specifying Message Passing and Time-Critical Systems With Temporal Logic*, volume 651 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1992.

[11] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proc. 24th Intl. Conf. on SW Eng. (ICSE'02)*, pages 83–93, Orlando, FL, May 2002.

[12] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. 10th ACM SIGSOFT Symp. on Foundations of SW Eng. (FSE-10)*, pages 119–128, Charleston, SC, Nov. 2002.

[13] N. G. Leveson. *Safeware - System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[14] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. SW Eng.*, 26(1):15–35, Jan. 2000.

[15] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proc. IEEE Intl. Symp. on Requirements Engineering*, pages 126–133, San Diego, CA, Jan. 1993.

[16] R. R. Lutz and C. Mikulski. Empirical analysis of safety-critical anomalies during operations. *IEEE Trans. SW Eng.*, 30(3):172–180, Mar. 2004.

[17] J. Mylopoulos, L. K. Chung, and B. A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. SW Eng.*, 18(6):483–497, June 1992.

[18] N. J. Nilsson. *Problem-Solving in Artificial Intelligence*. McGraw-Hill Computer Science Series. McGraw-Hill, New York, 1971.

[19] C. Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing (FTCS-28)*, pages 218–227, Munich, Germany, June 1998.

[20] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. SW Eng.*, 24(11):908–926, Nov. 1998.