Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models

Renaud De Landtsheer, Emmanuel Letier and Axel van Lamsweerde
Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
{rdl, eletier, avl}@info.ucl.ac.be

Abstract *Goal-oriented methods are increasingly popular for elaborating software requirements. They provide systematic support for incrementally building intentional, structural and operational models of the software and its environment together with various techniques for early analysis, e.g., to manage conflicting goals or anticipate abnormal environment behaviors that prevent goals from being achieved. On the other hand, tabular event-based methods are well-established for specifying operational requirements for control software. They provide sophisticated techniques and tools for late analysis of software behavior models through, e.g., simulation, model checking or table exhaustiveness checks.*

The paper proposes to take the best out of these two worlds to engineer requirements for control software. It presents a technique for deriving event-based specifications, written in the SCR tabular language, from operational specifications built according to the KAOS goal-oriented method. The technique consists in a series of transformation steps each of which resolves semantic, structural or syntactic differences between the KAOS source language and the SCR target language. Some of these steps need human intervention and illustrate the kind of semantic subtleties that need to be taken into account when integrating multiple formalisms.

As a result of our technique SCR specifiers may use upstream goal-based processes à la KAOS for the incremental elaboration, early analysis, organization and documentation of their tables while KAOS modelers may use downstream tables à la SCR for later analysis of the behavior models derived from goal specifications.

1. Introduction

Goal orientation is an increasingly recognized paradigm for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting and modifying software requirements [19, 21]. *Goals* are prescriptive statements of intent whose satisfaction may require the cooperation of agents (or active components) in the software and its environment. Goals are organized in AND/OR refinement structures; they may refer to functional or non-functional concerns and range from high-level, strategic concerns (such as "safe coolant system for nuclear power plant") to low-level, technical prescriptions; the latter can be *requirements* on the software-to-be (such as "safety injection overridden when block switch is on and pressure is less than 'Permit'") or *expectations* on its environment (such as "block switch is on when plant enters normal cooldown phase'").

Goal-based modeling and reasoning has many advantages. Goals may be specified precisely in a declarative fashion and refined incrementally into operational software specifications that provably assure the higher-level goals [8, 23, 24]; they allow one to trace low-level details back to high-level concerns [7, 26, 1]; strategic goal dependencies among agents can be analyzed for responsibility assignment [34]. Goals provide a criterion for requirements completeness and pertinence [35]; positive and negative interactions among goals can be captured and managed appropriately [18, 5]; exceptional conditions in the environment that may prevent critical goals from being achieved can be pointed out [29] or even generated and then resolved to produce more robust requirements [20].

Tables have long been recognized to be an appropriate format for presenting operational specifications in a compact, readable form amenable to various kinds of exhaustiveness or redundancy checks [28]. In the context of embedded control software, such tables may capture input-output functions and software behavior on a firm, precise mathematical basis [15, 32, 12]. As a consequence a wide range of analysis techniques can be defined and automated including dedicated consistency/completeness checks [11, 12], model simulation [13], model checking [3, 14], test data generation [10], invariant generation [17] or theorem proving [2, 30].

The integration of goal-oriented RE methods such as KAOS [21] and tabular specification techniques such as

In *Proceedings of RE'03, 11th IEEE Joint International Requirements Engineering Conference*, Monterey (CA), Sept. 2003, 200-210 (Best Research Paper Award).

SCR [12] thus provide the following complementary benefits.

- Domain and requirements models can be captured in terms of a rich ontology --goals, agents, requirements, expectations, objects (including entities, associations, explicit events, monitored/controlled attributes), operational services, conflicts, etc. Such models can be organized and documented using goal refinement/abstraction as a basic *structuring* mechanism. They can be built incrementally using a constructive *method* [19]. They can be analyzed at the early stages of the RE process using available techniques for goal refinement and operationalization, responsibility assignment, scenario abstraction, conflict management and obstacle anticipation.
- Once converted into SCR tables the goal operationalizations are presented in a more readable format due to tabular display and output-driven structuring. The rich arsenal of analysis techniques and tools can then be deployed on such tables to point out inadequacies, inconsistencies or incompletenesses in the operational software specification or in the underlying goals this specification operationalizes.

Our work is motivated by this complementarity. The objective of this paper is to discuss and illustrate our procedure for transforming KAOS specifications of operational services, derived from goals according to techniques described in [7, 24], into SCR tables.

The paper is organized as follows. Section 2 introduces some required background on KAOS and SCR together with our running example. Section 3 discusses the various steps of the transformation procedure and the KAOS/SCR semantic, structural or syntactic difference resolved by each step. Section 4 provides some evaluation by comparing the SCR specification we derive for our running example with others, by use of the SMV model checker.

2. Background

Our presentation will rely on the safety injection system for a nuclear power plant introduced in [6]. The reader may refer to [22] for a full KAOS elaboration of the goal, object, agent and operation models, and to [6, 12] for the original SCR specification.

2.1. Goal-Oriented Modeling with KAOS

Operational software requirements are derived gradually from the underlying system goals. The word "system" here refers to the software-to-be together with its environment. A goal refinement graph is elaborated first by identifying relevant goals from input material, typically, by looking for intentional keywords in natural language statements and by asking *why* and *how* questions about such statements (*goal elaboration step*); UML classes, attributes and associations are derived from the goal specifications (*object modeling step*); agents are identified together with their potential monitoring/control capabilities, and alternative assignments of goals to agents are explored (*agent modeling step*); operations and their domain pre- and postconditions are identified from the goal specifications, and strengthened pre-, post- and trigger conditions are derived so as to ensure the corresponding goals (*operationalization step*). Parallel steps of the method handle goal mining from scenarios, the management of conflicts between goals and the management of obstacles to goal satisfaction, respectively.

A *goal* is a statement of intent to be satisfied through cooperation of various *agents* making the system – humans such as operators of the nuclear power plant, devices such as sensors and actuators, and software such as the safety injection controller. Goals capture sets of intended behaviors; they can be formalized in a real-time temporal logic [7]. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition in the domain for satisfying the goal. The core of the *goal model* for a given system thus amounts to an AND/OR refinement/abstraction graph. An *obstacle* to some goal is a condition whose satisfaction may prevent the goal from being achieved.

For example, the goal named EffectiveCoolantSystem is a basic one in any nuclear power plant system. This goal can be obstructed by an obstacle such as LossOfCoolant. The goal SafetyInjection**Iff**LossOfCoolant is introduced to mitigate that obstacle. This goal is seen to be *conflicting* with another goal elicited from the source document, namely, the goal **No**SafetyInjection**When**StartUp/CoolDown. (Formal developments are skipped for obvious space reasons.) The conflict is resolved by weakening the first goal which yields a new goal textually specified as follows:

Goal Maintain [SafetyInjection**Iff**LossOfCoolant**ExceptIf-**StartUp/CoolDown]

Def *The safety injection signal should be 'On' whenever there is a loss of coolant except during normal start-up or cool down.*

FormalSpec SafetyInjectionSignal = 'On' $\Leftrightarrow$ LossOfCoolant $\wedge \neg$ (StartUp $\vee$ CoolDown)

Fig. 1 shows a goal model portion in which this weakened goal is refined by application of the "Introduce accuracy goal" tactics; the latter is frequently used to make phenomena referenced in goal formulations monitorable or controllable by software agents [23].

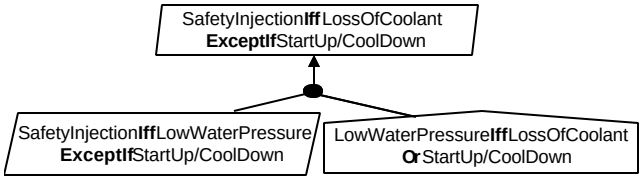

Figure 1 – Goal refinement towards monitorability

The textual specifications annotating the two child nodes in Fig. 1 are the following:

Goal Maintain [SafetyInjection **Iff** LowWaterPressure **ExceptIf-** StartUp/CoolDown]

Def *The safety injection signal should be 'On' whenever the water pressure is below the 'Low' set point except during normal start-up or cool down.*

FormalSpec SafetyInjectionSignal = 'On' $\Leftrightarrow$
WaterPressure < 'Low' $\wedge \neg$ (StartUp $\vee$ CoolDown)

DomProp LowWaterPressure **Iff** LossOfCoolant-**Or** StartUp/CoolDown

Def *The water pressure is below the 'Low' set point if and only if there is a loss of coolant or the plant is in normal start-up or cool down mode.*

FormalSpec WaterPressure < 'Low' $\Leftrightarrow$
LossOfCoolant $\vee$ StartUp $\vee$ CoolDown

(Note that goals are prescriptive whereas domain properties are descriptive.)

Goal refinement ends up when every subgoal is realizable by some candidate individual agent, that is, expressible in terms of objects that are monitorable and controllable by the agent. Goal refinement is thus partially driven by the target of reaching a 2-variable model for each agent [27]. A *requirement* is a terminal goal assigned to an agent in the software-to-be; an *expectation* is a terminal goal assigned to an agent in the environment.

The state of the system is defined by aggregation of the states of its objects. An *object* can be an entity, a relationship, an event or an agent (active object); it is characterized by attributes and domain properties (invariants). The *object model* is represented as a UML class diagram; it is derived systematically from the goal model by highlighting the attributes and relationships referenced in goal formulations.

The *agent model* captures responsibility links between agents and goals together with monitoring/control links between agents and object attributes. The object attributes monitored and controlled by an agent define its interface to other agents; these monitored/controlled variables are derived systematically, see [23, 21]. Modeling objects and agents may be essential for multi-component applications involving complex, inter-related objects.

A goal assigned to some agent in the software-to-be is *operationalized* in functional services, called operations, to be performed by the agent. The *operation model* collects all such operations together with their operationalization links to the goal model, performance links to the agent model and input/output links to the object model. An *operation* is an input-output relation over objects; operation applications define state transitions. When specifying an operation, a distinction is made between domain pre/postconditions and additional pre-, post- and trigger conditions required for achieving some underlying goal.

- A pair *(domain precondition, domain postcondition)* captures the elementary state transitions defined by operation applications in the domain.
- A *required precondition* for some goal captures a permission to perform the operation when the condition is true.
- A *required trigger condition* for some goal captures an obligation to perform the operation when the condition becomes true provided the domain precondition is true.
- A *required postcondition* defines some additional condition that any application of the operation must establish in order to achieve the corresponding goal.

To produce consistent operation models, a required trigger condition on an operation must imply the conjunction of its required preconditions.

Consider the goal SafetyInjection **Iff** LowWaterPressure **And-Not** Overridden, assigned to the Engineered Safety Feature Actuation agent (ESFAS). This goal is operationalized partly by the following operation:

Operation StopSafetyInjection
Input WaterPressure, Overridden;
Output SafetyInjectionSignal

DomPre SafetyInjectionSignal = 'On'
DomPost SafetyInjectionSignal = 'Off'

ReqPre/Trig For
SafetyInjection **Iff** LowWater Pressure **AndNot** Overridden:
WaterPressure $\geq$ 'Low' $\vee$ Overridden

In this specification, the required trigger condition requires that the safety injection signal *must* be set to 'Off' *as soon as* the water pressure is higher than 'Low' or the safety injection signal is overridden. In this case the condition is a required precondition as well; the safety injection *may* be set to 'Off' *only when* the water pressure is higher than 'Low' or the safety injection signal is overridden.

This paper assumes that correct and complete operationalizations have been derived from goal specifications using the techniques described in [24].

2.2. Operational Specification with SCR

SCR is built on the original 4-variable model that defines requirements as a relation between monitored and controlled variables, and software specifications as a relation between software input and output variables [27]. The system globally consists of two agents: the *machine* consisting of the software-to-be together with its associated input/output devices, and the *environment*. The former defines values for the controlled variables whereas the latter defines values for the monitored variables.

Two basic constructs in SCR are modes and terms. A *mode class* is an auxiliary variable whose behavior is defined by a state machine on monitored variables. The states are called *modes*; a mode name is thus a shorthand for some implicit logical expression on monitored variables. Mode transitions are triggered by events. A complex machine may be defined by several mode classes operating in parallel. A *term* is an auxiliary variable defined by a function on monitored variables, mode classes or other terms; using term names instead of repeating their definition helps making the specification more concise.

A SCR specification defines the machine through a set of tables together with associated information such as variable declarations, type definitions, initial state definitions, assumptions, etc. Each table specifies the behavior of a variable as a mathematical function. A table may be a mode transition table, a condition table or an event table.

A *mode transition table* specifies a mode class by defining its various modes as a function of the previous corresponding mode and events. A *condition table* defines the various values of a controlled variable or of a term as a function of a corresponding mode from the associated mode class (AMC) and conditions. An *event table* defines the various values of a controlled variable or of a term as a function of a corresponding AMC mode and events.

A *condition* is a predicate defined on one or more monitored, controlled or internal variables at some point in time. Conditions in a condition table are expected to be disjoint (for the table to be a function) and covering the entire state space (for the function to be total). An *event* occurs when a variable changes its value. In particular, an *input event* occurs when a monitored variable changes its value. A *conditioned event* occurs if an event occurs when some specified condition is true. Unlike in KAOS, events are implicit in SCR; they are manipulated through notations such as

$$@T(v)\ \mathrm{WHEN}\ C$$

which means

$$C \wedge \emptyset\, v \wedge v'$$

where C and v are evaluated in the current state and v' is evaluated in the next state. For example: *@T (Block = On) WHEN Reset = Off* amounts to $\emptyset$*Block=On* $\wedge$ *Block'=On* $\wedge$ *Reset=Off*. This event occurs when both *Block* and *Reset* are *'Off'* in the old state and *Block* becomes *'On'* in the new one. Note thus that a condition refers to one single state whereas an event refers to a pair of consecutive states.

Unlike KAOS, SCR is built upon the *synchrony hypothesis*, that is, the machine is assumed to react infinitely fast to changes in its environment [4]; it handles one input event completely before the next one is processed. In the 4-variable framework, this means that the value of a controlled variable in the next state may depend on the values of monitored variables in the current *and next* state. The synchrony hypothesis justifies that (a) a mode transition table specifies the next value of the mode class in terms of the current and next values of monitored variables (and the current value of the mode class), (b) an event table specifies the next value of the target variable in terms of the current and next values of other variables (and the current value of this variable), and (c) a condition table defines the next value of the target variable in terms of the next value of other variables.

The formal model of SCR is defined in terms of functions and therefore prescribes deterministic machine behaviors [12] (the behavior of the environment is of course non-deterministic).

Old Mode	Event	New Mode
TooLow	@T (WaterPressure ≥ 'Low')	Permitted
Permitted	@T (WaterPressure < 'Low')	TooLow
Permitted	@T (WaterPressure > 'Permit')	TooHigh
TooHigh	@T (WaterPressure ≤ 'Permit')	Permitted

Table 1: Mode transition table for mode class PressureState

Let us come back to our running example. The SCR specification for the ESFAS software agent includes a mode class PressureState with modes TooLow, Permitted and High, a term Overridden and a controlled variable SafetyInjectionSignal; the monitored variables are Block, Reset and WaterPressure. Table 1 illustrates a mode transition table for the PressureState mode class. The first line of this table states that if PressureState is TooLow and the event @T (WaterPressure ≥'Low') occurs then the PressureState switches to 'Permitted'. If none of the lines applies to the current state, PressureState does not change.

Mode	Events	
TooLow, Permitted	@T (Block = 'On') WHEN Reset = 'Off'	@T(Reset='On') OR @T(Inmode)
TooHigh	False	@T(Inmode)
Overridden	True	False

Table 2: Event table for term Overridden [12]

Table 2 illustrates an event table defining the term Overridden to capture situations in which safety injection is blocked. This term is defined as a function of the AMC PressureState and variables Block and Reset. An entry *False* in an event table means that no event may cause the variable defined by the table to take the value in the same column as the entry; e.g., when PressureState is TooHigh no event may cause Overridden to become true. The notation @T(Inmode) in a row captures a transition into one of the modes specified in that row; e.g., the last column of the first row in Table 2 states that "if PressureState becomes TooLow or Permitted (or if Reset becomes On) then Overridden must become false."

Table 3 illustrates the use of a condition table to specify the controlled variable SafetyInjectionSignal as a function of the AMC PressureState and term Overridden; e.g., the first column of the first row states that if PressureState is TooLow and Overridden is true then SafetyInjectionSignal must be 'Off'. Note that there is always one output value whose corresponding condition is true.

Mode	Conditions	
TooLow	NOT Overridden	Overridden
Permitted, TooHigh	False	True
SafetyInjection-Signal	'On'	'Off'

Table 3: Condition table for controlled variable SafetyInjectionSignal [12]

Also note that the rationale for some rows in those tables may not be obvious; a KAOS model upstream to such tables may explain them by tracing specification decisions back to underlying goals and obstacles. For example, [22] shows that the unexplained condition @T(Inmode) in the second row of Table 2 resolves an obstacle we generated, namely, the obstacle of an operator forgetting to push the "reset" button at the end of a normal start-up phase; without that resolution we might end up with no safety injection being actuated because of Overridden being still true while the plant is no longer in normal start-up phase.

Note finally that SCR is a "flat" language in that it provides no structuring mechanism for table refinement/composition and incremental specification [33]; our argument is that such mechanism is not needed when goal refinement is used upstream as a mechanism for structuring and documenting the specification.

3. Deriving SCR Tables from KAOS models

Our procedure assumes that the goal model has been transformed into a KAOS operation model according to the techniques described in [24]. The procedure converts such a model into a "semantically close" set of SCR tables. By "semantically close" we mean that the following relation must hold between the source KAOS operation model *KOM* and the target set of tables *SST*:

$$SST \vdash KOM^*$$

where *KOM** denotes the source operation model *KOM* in which a *one-state shift* is performed to squeeze the model into SCR's synchrony hypothesis --that is, required pre- and trigger conditions are evaluated in the next state with respect to the state in which the corresponding domain precondition is evaluated (see below). In other words, the set of SCR behaviors is included in the set of KAOS behaviors modulo such a one-state shift required by the synchrony hypothesis.

Every step of the procedure resolves some difference between KAOS and SCR –semantic difference (e.g., allowed form of non-determinism, synchrony hypothesis), structural difference (e.g., grouping of expressions) or syntactic difference. The various steps are now reviewed successively, namely,

- shifting from a pruning semantics to a generative one,
- mapping a multi-agent model into a bi-agent one,
- getting rid of non-determinism,
- grouping transition classes by output,
- translating transition predicates into SCR expressions,
- identifying mode classes and deriving mode conditions,
- generating SCR event tables and mode transition tables,
- simplifying some tables into condition tables.

3.1. From a pruning semantics to a generative one

The KAOS specification language has a *pruning* semantics, that is, every behavioral change is allowed except the ones explicitly forbidden by the specification. On the other hand, SCR like other state machine formalisms has a *generative* semantics, that is, every behavioral change is forbidden except the ones explicitly required by the specification.

This semantic difference is resolved by making a closure assumption; the source operation model is assumed to capture all acceptable behaviors and only those. In other words, every behavioral change not captured in the source operation model is forbidden (this corresponds to a "nothing else changes" frame assumption). In case a new operation is added to the source model, the conversion

procedure has thus to be reapplied to this model to produce a new set of tables.

3.2. From a 2N-variable model to a 4-variable one

The KAOS agent model captures multiple cooperating agents: software agents from legacy software or in the software-to-be, human agents, devices such as sensors or actuators, etc. On the other hand, the SCR specification considers two agents only: the machine and its environment (see Section 2.2). In KAOS, the controlled variables of an agent may be any object attribute whose values may be modified by the agent; a controlled variable can therefore be an interface variable or an internal variable. In SCR, the machine's controlled variables are restricted to variables at the interface with the environment.

To resolve these differences, the analyst is first asked which agent aggregate she wants to consider as the machine to be specified by SCR tables. All other agents will be aggregated to make the SCR environment. The monitored and controlled variables of the SCR specification are then derived according to the following rules:

- every KAOS variable monitored by a machine agent and controlled by an environment agent becomes a monitored variable in the SCR specification,
- every KAOS variable controlled by a machine agent and monitored by at least one agent in the environment becomes a controlled variable in the SCR specification,
- every KAOS variable controlled by a machine agent but *not* monitored by an environment agent becomes an auxiliary variable (i.e., a term or a mode class) in the SCR specification.

For our running example, the machine will be an aggregation of the ESFAS software agent and corresponding sensors and actuators; the environment will include the coolant system, the plant operator and the actual safety injection mechanisms. The variable SafetyInjectionSignal becomes a controlled variable in the SCR model since it is among the variables controlled by the ESFAS agent and monitored by the environment. The variables WaterPressure, Block and Reset become monitored variables in the SCR model since they are among the variables monitored by the ESFAS agent and controlled by the environment. On the other hand, the variable Overridden becomes an auxiliary variable (e.g., a term) because it is controlled by the ESFAS agent but it is not monitored by any agent in the environment.

3.3. Getting rid of non-determinism

KAOS agents are non-deterministic. While it is obliged to perform an operation when the operation's trigger condition becomes true an agent may have the freedom to perform an operation or not when its required preconditions are all true. On the other hand, a SCR environment is non-deterministic but a SCR machine is deterministic.

To remove KAOS non-determinism when necessary, the analyst is asked to choose between an eager or lazy behavior scheme for each operation performed by the machine agent. In the *eager* behavior scheme the agent performs the operation as soon as it can, that is, as soon as *all* required *preconditions* become true; in the *lazy* behavior scheme the agent performs the operation when it is really obliged to do so, that is, when *one* of its required *trigger conditions* becomes true.

Note that an eager scheme still guarantees that any obligation is fulfilled because of the KAOS consistency meta-constraint imposing any required trigger condition on an operation to imply all its required preconditions. Similarly, a lazy scheme still guarantees that no permission is violated because of the same KAOS consistency meta-constraint.

3.4. Grouping transitions by output

Operational specification items can be organized in several ways, e.g., grouping by input, grouping by output or grouping by transition class. In case of grouping by transition class, the focus of a specification unit is the set of state transitions that meet some pre-, trigger- and postconditions. In case of grouping by output, the focus of a specification unit is the set of state transitions that affect the value of some output.

In a KAOS operation model, specification items are grouped by transition class (the latter being grouped by agent and by goal the operation contributes to). SCR specification items are grouped by output.

A standard way of resolving structure clashes is to introduce an intermediate data structure [16]. An *output table* is associated with every controlled or term variable to collect the various transition classes for this variable. Each row in an output table is associated with a KAOS operation that declares the variable associated with the table in its **Output** clause. Table 4 shows the output table for the controlled variable SafetyInjectionSignal. (SILW is an abbreviation for the goal SafetyInjection**Iff**LowWater-Pressure **AndNot**Overridden.) This table is derived from the corresponding operation specifications in Section 2.1.

SafetyInjectionSignal				
Operation	**DomPre**	**Trigger**	**Target value**	**Goals**
StartSafety-Injection	SafetyInject-ionSignal = 'Off'	WaterPressure < 'Low' ∧ ¬ Overridden	'On'	SILW
StopSafety-Injection	SafetyInject-ionSignal = 'On'	WaterPressure ≥ 'Low' ∨ Overridden	'Off'	SILW

Table 4: Output table for SafetyInjectionSignal

A row in an output table defines a transition class through the following information:

- the name of the KAOS operation defining the transition class and the name of the goals operationalized by it (this information is used to keep track of the original specification at each step of the transformation);
- the domain precondition of this operation (in general it is a condition on the variable defined by the table);
- the condition triggering transitions in that class; this condition is the operation's conjunction of required preconditions, in case an eager behavior scheme has been selected for that operation at the previous step, or the operation's disjunction of required trigger conditions in case a lazy behavior scheme has been selected;
- the new value taken by the target controlled or term variable when a transition in that class is enabled (this format assumes that domain postconditions are specified equationally; the case of implicit, non-constructive postconditions leading to another form of non-determinism is not considered in this paper).

The semantics of a transition class captured by row *R* of an output table associated with some controlled or term variable *x* is the KAOS semantics for the corresponding operation [24], that is,

$$R.\text{DomPre} \wedge R.\text{Trigger} \supset \bigcirc (x = R.\text{TargetValue})$$

where "○ P" means "*P* holds in the next state".

3.5. Translating output table expressions into SCR expressions

This step enriches output tables with an extra column composed of SCR translations of the corresponding DomPre and Trigger expressions.

The problem is that sub-expressions of the form @T(Trigger) in the SCR translation refer to the current *and next* states whereas conditions Trigger in the output tables produced at the previous step refer to the current state but *not* to the next state. This is the point where we have to introduce a *one-state shift* in order to squeeze output table expressions into SCR's synchrony hypothesis (see the impact of the synchrony hypothesis on the evaluation of SCR expressions in Section 2.2 and the specification of the conversion procedure at the beginning of Section 3). This shift may be defined visually as follows:

	current state	**next state**
KAOS transition	DomPre, Trigger	x = TargetValue
SCR transition	DomPre, *Ø Trigger*	*Trigger*, x = TargetValue

Correspondlingly, a pair (DomPre, Trigger) in an output table row will be translated into a SCR expression as follows:

- if the trigger is non-disjunctive:

  (DomPre, Trigger) ⟶ @T (Trigger) WHEN DomPre

- if the trigger is disjunctive:

  (DomPre, Trig1 ∨ Trig2) ⟶
  @T (Trig1) WHEN DomPre ∨ @T (Trig2) WHEN DomPre

(Using the first translation rule only, with Trigger being replaced by Trig1 ∨ Trig2 in case of disjunctive trigger, would enable less transitions in the SCR specification; the second rule is aimed at keeping all transitions from the KAOS operation model in the SCR specification.)

KAOS constructs may involve temporal operators such as, e.g., the real-time operator "$\blacksquare_{\leq d}$" or the "@" operator. A comprehensive set of translation rules for mapping additional KAOS constructs into corresponding SCR ones is given in [9].

Using the second translation rule above, we obtain the enriched output table portion sketched in Table 5 for the transition class associated with operation StopSafety-Injection in Table 4. *StopCond* in Table 5 is a shorthand for the following SCR expression:

@T (WaterPressure ≥ 'Low') WHEN SafetyInjectionSignal = 'On'
∨ @T (Overridden) WHEN SafetyInjectionSignal = 'On'

Operation	**DomPre**	**Trigger**	**SCR expression**	**Target value**	**Goals**
...	...	...	...	...	...
StopSafety-Injection	...	...	*StopCond*	'Off'	...

Table 5: Enriched output table for SafetyInjectionSignal

This step might suggest that a SCR event table will be created for every controlled or term variable; this will in fact not necessarily be the case after simplifications made at subsequent steps.

3.6. Identifying mode classes

The next step consists in determining from the enriched output tables which auxiliary variables will be used as

mode classes. The choice of mode classes determines the structure of the SCR specification and has a strong impact on its readability and modifiability. Various heuristics are available to support the analyst in this task.

- *Input history abstraction:* identify a mode class variable that allows one to abstract away from past histories of input events. This heuristics is used in many SCR case studies –see, e.g., the Cruise Control system.
- *Input aggregation:* build a mode class as an aggregation of several discrete monitored variables whose values in any state are constrained by exclusion and coverage rules asserted in KAOS goal specifications or domain invariants. Such rules convey in application-specific terms that a sequential state machine can at any time be in one and only one of the possible states partitioning its state space. [31] outlines an algorithm implementing this heuristics and applies it to the systematic derivation of mode classes for the Autopilot case study.
- *Continuous variable abstraction:* build a mode class that partitions the range of values for a continuous monitored variable into a discrete set of subranges. The subranges are derived from the corresponding conditions that constrain the monitored variable in the output tables where the variable appears.
- *Finite output variable promotion:* consider a variable already defined by some output table as candidate mode class if (a) it has a finite range, (b) it is an auxiliary variable (that is, neither monitored nor controlled by the machine), and (c) its promotion to mode class will improve the clarity of the specification.

For our safety injection case study, the variable WaterPressure is continuous and constrained by conditions in the output tables defining the controlled variable SafetyInjectionSignal and term Overridden. The following range partition is thereby derived using the *variable abstraction* heuristics:

TooHigh:	'Permit' < WaterPressure
Permitted:	'Low' ≤ WaterPressure ≤ 'Permit'
TooLow:	WaterPressure < 'Low'

These three ranges define modes of a mode class named PressureState.

The SCR expressions in output tables are then rewritten by replacing the conditions on continuous monitored variables by their equivalent formulation in terms of modes. For the *StopCond* SCR expression in Table 5 associated with target value 'Off' for SafetyInjectionSignal, we obtain the following mode-based expression:

@T (PressureState = Permitted ∨ PressureState = TooHigh)
WHEN SafetyInjectionSignal = 'On'
∨ @T (Overridden) WHEN SafetyInjectionSignal = 'On'

According to the "finite output variable promotion" heuristics, Overridden could be considered as a mode class as it is an auxiliary variable with finite range.

3.7. Generating SCR tables

This step generates mode transition tables and event tables from the output tables where SCR expressions are now mode-based.

3.7.1. Generating mode transition tables

For each mode class obtained through the "continuous variable abstraction" heuristics, the mode transition table is derived from its range partition: (a) all conditions on monitored variables that define the partitioning are prefixed by the "@T" operator to produce the events triggering transitions, (b) the source and target modes for each such transition are collected from the mode definitions. Table 1 shows the resulting mode table for mode class PressureState.

For each mode class obtained through the "finite output variable promotion" heuristics, the mode transition table can be obtained by first deriving an event table (see below) and then transforming that event table into a mode transition table using a technique described in [17].

3.7.2. Initializing event tables

Initial event tables for controlled or term variables are obtained by (a) determining which mode class from the corresponding output table will be the one associated with the event table, (b) populating the first column with the various AMC modes, and (c) populating the last row with the various target values from the output table, (d) filling in all other cells with "False". Selection heuristics may be used in case of multiple candidate AMC's, e.g., select the most referenced mode class in the corresponding output table.

Mode	**Events**	
TooLow	False	False
Permitted	False	False
TooHigh	False	False
SafetyInjection-Signal	'On'	'Off'

Table 6: Initialized event table for SafetyInjectionSignal

Table 6 shows the initialized event table for controlled variable SafetyInjectionSignal; in this case the mode class PressureState is the only one appearing in the SCR expressions of the output table associated with SafetyInjectionSignal. No choice is thus needed.

3.7.3. Strengthening event tables with SCR expressions

A first version of SCR event tables is then obtained from

the initialized tables by (a) strengthening, for each column, the *False* event cells into the SCR mode-based expression from the output table cell associated with the same target value, (b) simplifying the same SCR expression in each cell of a strengthened column by propagation of mode properties: for an event cell in a line associated with mode *m*, (b1) a disjunct taking the form *AMC = m* is replaced by the SCR predicate "*Inmode*", and (b2) a disjunct taking the form *AMC = n* ($n \neq m$) is deleted (i.e., replaced by *false* as the AMC cannot take two different values at the same time).

Table 7 shows the resulting intermediate event table obtained for the controlled variable SafetyInjectionSignal after processing the third column in Table 6 from the third line in Table 5 in which the mode-based expression of *StopCond* is the one obtained in Section 3.6; the AMC is PressureState.

Mode	Events	
TooLow	False	@T (Overridden) WHEN SafetyInjectionSignal = 'On'
Permitted	False	@T (Inmode) WHEN SafetyInjectionSignal = 'On' ∨ @T (Overridden) WHEN SafetyInjectionSignal = 'On'
TooHigh	False	@T (Inmode) WHEN SafetyInjectionSignal = 'On' ∨ @T (Overridden) WHEN SafetyInjectionSignal = 'On'
SafetyInjection-Signal	'On'	'Off'

Table 7: Partially strengthened event table

3.7.4. Simplifying event tables further

Domain invariants found in the upstream KAOS specification and SCR-specific rules may be used to simplify the strengthened event table further by eliminating redundancies [9]. For example,

- WHEN conditions on values of the variable being defined by the table may be trivially satisfied in the source state leading to the new target value (in view of domain restrictions on possible values);
- an event cell "@T(Inmode) ∨ @X" with all other event cells being "False" on the same line may be simplified to "@T(Inmode)" (since no other event in that mode could cause a transition to another value for that variable thereby making the disjunct "@X" of no use to get back to the target value);
- two lines with identical event cells may be merged into one single line with mode cell containing the union of the two original modes.

Table 8 shows the final event table obtained after processing of the second column and application of these three simplification rules.

Mode	Events	
TooLow	@T (Inmode) AND NOT Overridden	@T (Overridden)
Permitted, TooHigh	False	@T (Inmode)
SafetyInject-ionSignal	'On'	'Off'

Table 8: Final event table for SafetyInjectionSignal

3.8. From event tables to condition tables

Some of the event tables obtained may be simplified into condition tables provided the following conditions hold:

- there is no WHEN clause in the table,
- the condition table obtained meets the disjointness and coverage criteria.

The set of transformation rules includes rules such as:

@T (P) $\rightarrow$ P

@T Inmode $\rightarrow$ True

Table 9 shows the result of transforming the event table in Table 8 into a condition table.

Mode	Conditions	
TooLow	NOT Overridden	Overridden
Permitted, TooHigh	False	True
SafetyInject-ionSignal	'On'	'Off'

Table 9: Condition table for SafetyInjectionSignal

The resulting table should meet the coverage and disjointness criteria. These criteria could be checked using the SCR* toolset [12]. The decision of making this transformation is left to the analyst.

4. Evaluation

We may now compare the three specifications for the safety injection case study, namely, the "native" SCR spec [12], the "native" KAOS spec [22] and the SCR spec derived in this paper.

4.1. Derived SCR spec vs. original SCR spec

The SCR specification derived from our KAOS model is nearly identical to the original one found in [12]; the only two slight differences are in the event table for term Overridden. The original table for this term is given in Table 2; Table 10 shows the one derived by our procedure.

Mode	Events	
TooLow, Permitted	@T (Block = 'On')	@T(Reset='On')
TooHigh	False	@T(Inmode)
Overridden	True	False

Table 10: Derived event table for term Overridden

We may notice that the original specification has an extra condition WHEN Reset = 'Off' on the event @T(Block = 'On') leading to the state Overridden = True; this seems redundant in view of domain invariants about Block and Reset. The original specification also has an extra disjunct @T(Inmode) strengthening the event @T(Reset='On') that may lead to the state Overridden = False; why this disjunct is really needed is hard to understand without documentation of its rationale in some underlying goal model.

On the other hand, every expression in the SCR specification we derived may be traced back to our goal model. The latter might of course be still incomplete, e.g., because of insufficient obstacle analysis. Corrections to the goal model from further obstacle analysis can be downpropagated to the tables derived; we feel it more difficult to follow the reverse engineering path.

4.2. Derived SCR spec vs. original KAOS spec

We also compared the derived SCR specification with the original KAOS model in [22] using the SMV model checker [25]. Basically, we translated both specifications into SMV syntax and tried to prove that the controlled variables SafetyInjectionSignal in the original KAOS model and in the derived SCR specification do always have the same value. It turned out that this is not exactly the case in view of the one-state shift introduced to squeeze output table expressions into SCR's synchrony hypothesis (see Section 3.5).

However, SMV showed that the values are almost always the same under a quiet input assumption. This means roughly that input events are sparse and that the respective values are equal now, at the previous time unit or two clock cycles ago.

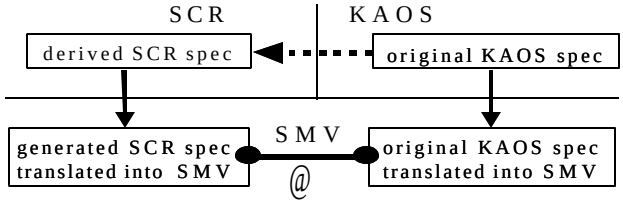

Fig. 2 – Comparing the KAOS and derived SCR specifications

Fig. 2 helps visualizing our verification experiment. Vertical arrows denote translation to SMV syntax; the horizontal dotted arrow represents our derivation process and ≅ denotes the pseudo- equivalence.

[9] also provides a formal framework to our derivation process which includes a formal representation of every step together with a formal semantics of each intermediate model.

5. Conclusion

Thanks to our method for deriving SCR specifications from KAOS models, SCR specifiers may follow upstream goal-based processes to incrementally elaborate, structure and document their tabular specification in a guided fashion, and to perform goal-level analysis for earlier detection and resolution of obstacles [20] and conflicts [18]. Conversely, KAOS modelers may obtain downstream tabular specifications in a systematic way for later specification analysis through exhaustiveness checking [12], simulation [13], model checking [14] and test data generation [10].

Our approach led us to point out complementarities and subtle differences between the two frameworks. The derivation process consists of a series of steps aimed at removing semantic, structural and syntactic differences between the KAOS and SCR frameworks. The derivation process may thereby be shown to be complete with respect to those differences.

As we discussed it, there is a price to pay for integrating such different RE frameworks. Some semantic differences due to the absence of non-determinism in SCR machine behaviors and to the synchrony hypothesis (resulting in incompatible rules for evaluation of expressions over states) make it impossible to derive a target specification whose behavior models are *exactly* the same as the source ones.

An alternative approach to combining goal-oriented specification and tabular event-based specification would be to derive SCR tables directly from goal specifications without passing through the KAOS operation model. This could make the derivation process simpler as it would shortcut the structuring by transition class to proceed directly to the output-driven restructuring required by SCR tables. Deriving SCR tables directly from goals might perhaps overcome some technical problems related to the synchrony hypothesis; this hypothesis would be injected directly in our goal model (technically, by weakening our definition of realizability [23] so as to allow machine agents to take responsibility for goals that require their synchronous reaction). The problem of deriving SCR specifications directly from declarative goal specifications is somewhat the inverse of the problem of inferring declarative invariants from SCR tables [17].

Several insights and techniques gained from the process described in this paper (e.g., the techniques for extracting mode classes or the eager/lazy strategies for getting rid of non-determinism) would be applicable to the problem of deriving SCR tables directly from declarative goals. This might be a direction worth investigating in the future.

Acknowledgement. The work reported herein was partially supported by the Belgian "Fonds National de la Recherche Scientifique" (FNRS).

References

[1] A.I. Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems", *Proc. ICSE-98: 20th Int.l Conferenc e on Software Enginering*, Kyoto, April 1998.

[2] Archer, Myla M., Heitmeyer, Constance L., and Sims, Steve, "TAME: A PVS Interface to Simplify Proofs for Automata Models," *Proc. UITP '98*, July 1998.

[3] J.M. Atlee, State-Based Model Checking of Event-Driven System Requirements, *IEEE Transactions on Software Engineering* Vol. 19 No. 1, January 1993, 24-40.

[4] G.Berry, G.Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation", *Science of Computer Programming*. 19, 2, 89.

[5] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic,, 2000.

[6] P.-J. Courtois and D.L. Parnas, "Documentation for safety critical software", *Proc. ICSE'93 - 15th Intl. Conf. on Software Engineering*, 1993, pp. 315-323.

[7] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[8] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 – 4th ACM Symp. on Foundations of Software Engineering*, Oct. 1996, 179-190.

[9] R. De Landtsheer, *Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models*. Ms. Thesis, University of Louvain, June 2002.

[10] A. Gargantini and C. Heitmeyer, "Using Model Chcking to Generate Tests from Requirements Specifications", *Proc. ESEC/FSE'99*, Springer-Verlag LNCS Nr. 1687, 1999, 146-162.

[11] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements", *IEEE Transactions on Software Engineering* Vol. 22 No. 6, June 1996, 363-377.

[12] C. Heitmeyer, R.D. Jeffords and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications", *ACM Trans. on Software Eng. and Methodology* 5, 3, July 1996, 231-261.

[13] C. Heitmeyer, J. Kirby, and B. Labaw. "Tools for Formal Specification, Verification, and Validation of Requirements," *Proc. COMPASS '97*, June 1997, Gaithersburg, MD.

[14] C. Heitmeyer, J. Kirkby, B. Labaw, M. Archer and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications", *IEEE Transactions on Software Engineering* Vol. 24 No. 11, November 1998, 927-948

[15] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Application", *IEEE Transactions on Software Engineering* Vol. 6 No. 1, January 1980, 2-13.

[16] M. Jackson, *Principles of Program Design*. Academic Press, 1975.

[17] R. Jeffords and C. Heitmeyer, "Automatic Generation of State Invariants from Requirements Specifications", *Proc. FSE-6: 6th ACM Symp. Foundations of Software Engineering*, 1998, 56-69.

[18] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development*, November 1998.

[19] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective", *Keynote paper, Proc. ICSE'2000 - 22nd Intl. Conference on Software Engineering*, ACM Press, May 2000.

[20] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, October 2000.

[21] A. van Lamsweerde , "Goal-Oriented Requirements Engineering: A Guided Tour", *Invited Minitutorial, Proc. RE'01 - 5th Intl. Symp. Requirements Engineering*, Toronto, August 2001, pp. 249-263.

[22] E.Letier, "Goal-Oriented Elaboration of Requirements for a Safety Injection Control System", http://www.info.ucl.ac.be/people/eletier/safety-injection.pdf

[23] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Press, May 2002.

[24] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM Symp. Foundations of Software Engineering*, Charleston, November 2002.

[25] K.L.Mc Millan, "The SMV* system for SMV version 2.5.4", November 2000, http://www2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps

[26] J. Mylopoulos, L. Chung and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Software. Engineering*, Vol. 18 No. 6, June 1992, 483-497.

[27] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol. 25, 1995, pp. 41-61.

[28] S. Pollack and H. Hicks, *Decision Tables - Theory and Practice*. Wiley, 1971.

[29] C. Potts, "Using Schematic Scenarios to Understand User Needs", *Proc. DIS'95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, Univ. Michigan, August 1995.

[30] K.Taeho, D. Stringer-Calvert and S. Cha ,"Formal Verification of Functional Properties of an SCR-Style Software Requirements Specification Using PVS", *Proc. TACAS'2002*, Springer-Verlag, April 2002.

[31] O. Vandenbroucke, *Derivation of Tabular Specification from Goal-Oriented Specifications for a Simple Autopilot System*, M.S. Thesis, University of Louvain, June 2000.

[32] A.J. Van Schouwen, D.L. Parnas and J. Madey, "Documentation of Requirements for Computer Systems", *Proc. RE'93 – 1st Int. Symp. on Requirements Engineering*, San Diego, IEEE, 1993, 198-207.

[33] V. Wiels and S. M. Easterbrook, "Formal Modeling of Space Shuttle Software Change Requests using SCR", Proc. RE'99: 4th Intl Symp. Requirements Engineering, Limerick, IEEE, June 1999.

[34] E.S.K. Yu, "Modelling Organizations for Information Systems Requirements Engineering", *Proc. RE'93 - 1st Intl Symp. on Requirements Engineering*, IEEE, 1993, 34-41.

[35] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.