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 offer systematic support for incrementally building intentional, structural and operational models of the software and its environment. They also provide various techniques for early analysis, notably, 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 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 [22, 24]. *Goals* are prescriptive statements of intent whose satisfaction requires 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 [9, 26, 27];
- they allow one to trace low-level details back to high-level concerns [8, 29, 1];
- strategic goal dependencies among agents can be analyzed for responsibility assignment [38];
- goals provide a criterion for requirements completeness and pertinence [39];
- positive and negative interactions among goals can be captured and managed appropriately [20, 6, 33];
- exceptional conditions in the environment that may prevent critical goals from being achieved can be pointed out [32] or even generated and then resolved to produce more robust requirements [23].

Tables have long been recognized to be a convenient format for presenting operational specifications in a compact, readable form amenable to various kinds of exhaustiveness or redundancy checks [31]. In the context of embedded control software, such tables may capture input-output functions and software behavior on a firm, precise mathematical basis [17, 36, 14]. As a consequence a wide range of analysis techniques can be defined and automated including:

- dedicated consistency/completeness checks [13, 14],

- model simulation [15],
- model checking [3, 16],
- test data generation [11],
- invariant generation [19]
- or theorem proving [2, 34].

The integration of goal-oriented RE methods such as KAOS [24] and tabular specification techniques such as SCR [14] thus provide the following complementary benefits.

- Complex domain and requirements models can be captured in terms of a richer ontology - goals, agents, requirements, expectations, conflicts, objects (including entities, associations, explicit events, monitored/controlled attributes), operational services, scenarios, etc. The specification can now be structured and documented using goal refinement and abstraction as a basic structuring mechanism. It can be built incrementally using a constructive *method* [22]. Requirements can be analyzed at earlier stages of the RE process using available techniques for goal refinement and operationalization, goal mining from scenarios, responsibility assignment, 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 of specification units. The rich arsenal of techniques and tools can then be deployed on such tables for later analysis to point out inadequacies, inconsistencies or incompleteness 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 [8, 27], 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 presents and illustrates the various steps of the transformation procedure together with the KAOS/SCR semantic, structural or syntactic difference resolved by each step. Section 4 provides some evaluation based on the use of the SMV model checker; the SCR specification derived by our technique is compared with other published specifications of our running example.

2. Background

Our presentation will rely on the safety injection system for a nuclear power plant introduced in [7]. The reader may refer to [25] for a full KAOS elaboration of the goal, object, agent and operation models, and to [7, 14] for SCR specifications of this system .

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. The derivation proceeds according to the following steps [24].

- *Goal modeling:* A goal refinement graph is elaborated first by identifying relevant goals from input material (such as interview transcripts and available documents) – typically, by looking for intentional keywords in natural language statements and by asking *why* and *how* questions about such statements.
- *Object modeling:* UML classes, attributes and associations are derived systematically from the goal specifications.
- *Agent modeling:* Agents are identified together with their potential monitoring/control capabilities; alternative assignments of goals to agents are explored.
- *Operationalization:* Operations and their domain pre- and postconditions are identified from the goal specifications; strengthened pre-, post- and trigger conditions are derived so as to ensure the corresponding goals.

The above steps are ordered by data dependencies and, of course, intertwined. Each step may be guided by the use of heuristics and derivation patterns associated with specific tactics [9, 26, 27]. Additional parallel steps of the method handle goal mining from scenarios [21], the management of conflicts between goals [20] and the management of obstacles to goal satisfaction [23], respectively.

As introduced before, a *goal* is a prescriptive statement of intent to be satisfied through cooperation of various *agents* making the system – humans such as operators in 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 [8]. *AND-refinement* links relate a goal to a set of subgoals (called refinement) and domain properties; 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 graph whose edges capture refinement/abstraction links. 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 here for 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 portion of the goal model 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 [26].

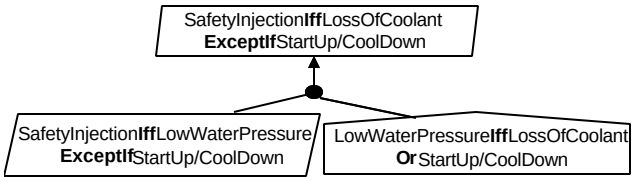

Figure 1 – Goal refinement towards monitorability

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 [40].

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 [30]. 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). Objects are characterized by attributes and domain invariants; as opposed to goals, the latter are *descriptive* statements. 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 [26, 24]. Although not necessarily required for simple control systems, 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* into 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**LowWaterPressure**AndNot**Overridden:
WaterPressure ≥ 'Low' ∨ Overridden

In this specification, the required trigger condition prescribes 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.

Our mapping from KAOS to SCR assumes that correct and complete operationalizations have been derived from goal specifications using the techniques described in [27].

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 [30]. 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

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

which means

$$C \wedge \neg v \wedge v'$$

where *C* and *v* are evaluated in the current state and *v'* is evaluated in the next state. For example,

$$@T (Block = On) \text{ WHEN } Reset = Off$$

amounts to

$$\neg\, Block{=}On \wedge Block'{=}On \wedge Reset{=}Off.$$

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; it therefore prescribes deterministic machine behaviors [14]. 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 row 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 rows 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* [14]

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*. The last 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* [14]

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 some underlying goals and obstacles. For example, our derivation in [25] 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 such 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 [37]. We argue that such mechanism is not really needed when goal refinement is used upstream as a mechanism for structuring and documenting the specification.

3. Deriving SCR Tables from KAOS models

Our mapping procedure assumes that the goal model has been transformed into a KAOS operation model according to the techniques described in [27]. 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 – a semantic difference (e.g., allowed form of non-determinism, synchrony hypothesis), a structural difference (e.g., grouping of expressions) or a syntactic difference. The various steps are now reviewed successively, namely,

- moving 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; every behavioral change is allowed except the ones explicitly forbidden by the specification [27]. On the other hand, SCR like other state machine formalisms has a *generative* semantics; 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 those 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 in the SCR specification (i.e., a term or a mode class).

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 one of 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* are 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 a lazy scheme still guarantees that no permission is violated because of the consistency meta-rule of the KAOS language imposing any required trigger condition on an operation to imply all its required preconditions:

$$(\bigvee_k \text{ReqTrig}_k) \wedge \text{DomPre} \Rightarrow \bigwedge_i \text{ReqPre}_i$$

Similarly, an eager scheme still guarantees that any obligation is fulfilled.

3.4. Grouping transitions by output

Operational specification units 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 units are grouped by transition class (the latter being grouped by agent and by goal the operation contributes to). SCR specification units are grouped by output.

A standard way of resolving structure clashes is to introduce an intermediate data structure [18]. An *output table* is therefore 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.) The last row of this table is derived from the StopSafetyInjection operation specification given in Section 2.1 by transferring its DomPre, ReqPre/Trig and DomPost conditions to the DomPre, Trigger and TargetValue cells, respectively. The previous row is similarly obtained from the specification of the StartSafetyInjection operation.

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

Table 5 shows the output table for the controlled variable Overridden, similarly derived from the specification of the operations OverrideSafetyInjection and EnableSafetyInjection.

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 (in the latter case the conjunction of required preconditions is satisfied because of the KAOS consistency meta-rule recalled in Section 3.3);
- the new value taken by the target controlled variable 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 [27], that is,

$$R.DomPre \;\grave{U}\; R.Trigger \;\acute{E}\; \bigcirc (x = R.TargetValue)$$

where "$\bigcirc P$" means "P holds in the next state".

Overridden				
Operation	**DomPre**	**Trigger**	**Target value**	**Goals**
Override-Safety-Injection	¬ Overridden	@ (Block = On) ∧ WaterPressure < 'Permit'	**true**	…
Enable-Safety-Injection	Overridden	@ (Reset = On) ∨ WaterPressure ≥ 'Permit'	**false**	…

Table 5: Output table for Overridden

3.5. Translating output table expressions into SCR expressions

The next step extends output tables with an extra column composed of SCR translations of the corresponding *DomPre* and *Trigger* expressions. The objective is to translate the DomPre and Trigger conditions of output tables into disjunctions of SCR events, each taking the following form:

SafetyInjectionSignal			
Operation	**SCR expression**	**Target value**	**Goals**
StartSafetyInjection	@T (WaterPressure < 'Low') WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden) OR @T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off' AND WaterPressure < 'Low')	'On'	SILW
StopSafetyInjection	@T (WaterPressure ≥ 'Low') WHEN (SafetyInjectionSignal = 'On') OR @T(Overridden) WHEN (SafetyInjectionSignal = 'On')	'Off'	SILW

Table 6. Output table for SafetyInjectionSignal **with SCR expressions**

Overridden			
Operation	**SCR expression**	**Target value**	**Goals**
OverrideSafetyInjection	@T(Block = On) WHEN (WaterPressure < 'Permit' AND NOT Overridden)	**true**	…
EnableSafetyInjection	@T(Reset = On) WHEN Overridden OR @T (WaterPressure ≥ 'Permit') WHEN Overridden	**false**	…

Table 7. Output table for Overridden **with SCR expressions**

@T(c) WHEN d

where *c* is a simple condition and *d* is a simple condition or a conjunction of simple conditions. A *simple condition* is an expression of the form r # v where r is a SCR variable, v a value and # is a binary relation in {=, ≠ <, >, ≥, ≤} [14].

The main problem here 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 our conversion procedure at the beginning of Section 3. This shift may be visualized as follows:

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

Correspondingly, a pair (DomPre, Trigger) in an output table row will be mapped to a SCR expression according to the following definition of the mapping function *M*.

- If the trigger is a *simple condition* Trig or an event @Trig on such a simple condition:

  M (DomPre, Trig) = @T (Trig) WHEN DomPre

  M (DomPre, @Trig) = @T (Trig) WHEN DomPre

- If the trigger is a *conjunction* Trig1 ∧ Trig2 or @Trig1 ∧ Trig2 where Trig1 and Trig2 are simple conditions:

  M (DomPre, Trig1 ∧ Trig2) =
  @T(Trig1) WHEN (DomPre AND Trig2)
  OR @T(Trig2) WHEN (DomPre AND Trig1)

  M (DomPre, @Trig1 ∧ Trig2) =
  @T (Trig1) WHEN (DomPre and Trig2)

- If the trigger is a *disjunction* $Trig_1 \vee \ldots \vee Trig_n$ where $Trig_1$, …, $Trig_n$ are not necessarily simple conditions:

  M (DomPre, $Trig_1 \vee \ldots \vee Trig_n$) =
  M (DomPre, $Trig_1$) OR … OR M (DomPre, $Trig_n$)

The result of applying this translation step to our running example is shown in Tables 6 and 7.

Strictly speaking, our translation rules do not fully preserve the semantics of our tables. Full semantics preservation is impossible because of the restricted form allowed for SCR expressions and the incompatibility between SCR's synchrony hypothesis and the KAOS semantics. The above rules have been defined so as to keep the semantics "as close as possible" when a one-state shift is tolerated between the KAOS and SCR expressions.

The rule for conjunctive triggers *Trig1 Ù Trig2* is motivated by the following logical property of the "@T" operator:

@T (c1 AND c2) WHEN d
≡ @T(c1) WHEN (d AND c2) OR @T(c2) WHEN (d AND c1)
under the condition that it never happens that simultaneously c1 becomes true and c2 becomes false (or vice-versa).

The rule for disjunctive triggers $Trig_1 Ú \ldots Ú Trig_n$ is necessary to avoid that the SCR model has less transitions than the KAOS model. For example, the disjunctive

trigger *WaterPressure ³ 'Low' Ú Overridden* on the transition to *SafetyInjectionSignal = 'Off'* in Table 4 is translated according to that rule into the following SCR expression:

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

If we had used the translation rule for simple conditions instead, we would have obtained the expression

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

(this would then have been rewritten into a valid SCR expression using logical properties of the "@T" operator). The latter event requires that (i) *WaterPressure ³ 'Low'* and *Overridden both* be false in the current state, and (ii) at least one of them becomes true in the next state. Requiring both conditions to be false in the current state is too restrictive and not required by the semantics of the KAOS model. In contrast, the SCR expression generated by our translation rule for disjunctive triggers does not require both *WaterPressure ³ 'Low'* and *Overridden* to be false in the current state for the transition to take place.

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. An algorithm implementing this heuristics is described in [35] and applied to the systematic derivation of mode classes for the Autopilot case study [5].
- *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 (see Tables 6 and 7). The following range partition is thereby derived using the *variable abstraction* heuristics to define modes of a mode class named *PressureState*:

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

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 example, the event

@T (WaterPressure < 'Low')
WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden)

in Table 6 is rewritten as

@T (PressureState = 'TooLow')
WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden)

Mode-based expressions may involve disjunctions; the following logical properties of the "@T" operator may therefore need to be applied to get valid SCR expressions using events on simple conditions.

@T(c) WHEN (d1 OR d2) ≡
@T(c) WHEN d1
OR @T(c) WHEN d2

@T (c1 OR c2) WHEN d ≡
@T(c1) WHEN (d AND NOTc2)
OR @T(c2) WHEN (d AND NOT c1)

For example, the mode-based expression for the event

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

in Table 6 is given by

@T (PressureState = 'Permitted' **or** PressureState = 'TooHigh')
WHEN SafetyInjectionSignal = 'On'

which is then expanded using the second rule above into:

@T (PressureState = 'Permitted') WHEN
SafetyInjectionSignal = 'On' ∧ PressureState ≠ 'TooHigh'
OR @T (PressureState = 'TooHigh') WHEN
SafetyInjectionSignal = 'On' ∧ PressureState ≠ 'Permitted'

The rewriting of SCR expressions in Table 6 and 7 into mode-based expressions yields Table 8 and 9, respectively.

According to the "*finite output variable promotion*"

SafetyInjectionSignal			
Operation	**SCR expression**	**Target value**	**Goals**
StartSafetyInjection	@T (PressureState = 'TooLow') WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden) OR @T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off' AND PressureState = 'TooLow')	'On'	SILW
StopSafetyInjection	@T (PressureState = 'Permitted' WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'TooHigh') OR @T (PressureState = 'TooHigh') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'Permitted') OR @T (Overridden) WHEN (SafetyInjectionSignal = 'On')	'Off'	SILW

Table 8. Output table for SafetyInjectionSignal **with *mode-based* SCR expressions**

Overridden			
Operation	**SCR expression**	**Target value**	**Goals**
OverrideSafetyInjection	@T (Block = On) WHEN (PressureState = 'Permitted' AND NOT Overridden) OR @T (Block = On) WHEN (PressureState = 'TooLow' AND NOT Overridden)	**true**	…
EnableSafetyInjection	@T (Reset = On) WHEN Overridden OR @T (PressureState = 'TooHigh') WHEN Overridden	**false**	…

Table 9. Output table for Overridden **with *mode-based* SCR expressions**

heuristics, *Overridden* could be considered as a mode class as it is an auxiliary variable with finite range.

3.7. Generating SCR tables

The next step generates mode transition tables and event tables from the output tables in which 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. The table collects the transitions from each mode to its adjacent modes; the event triggering a transition is obtained by prefixing by "@T" the simple condition that appears in the target mode definition and whose negation appears in the source mode definition.

In our running example, the generated mode transition table for mode class PressureState is the one shown in Table 1. Based on the range partition for PressureState in Section 3.6, the event triggering the transition from 'Permitted' to 'TooHigh' is @T(WaterPressure ≥ 'Permit') because WaterPressure ≥ 'Permit' is among the conditions for the target state TooHigh and its negation is in the definition of the source state Permitted.

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 [19].

3.7.2. Generating event tables: initialization

Initial event tables for controlled variables 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, (c) populating the last row with the various target values from the output table, and (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 10: Initialized event table for SafetyInjectionSignal

Table 10 shows the initialized event table for the 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. The

Mode	Events	
TooLow	@T (PressureState = 'TooLow') WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden) OR @T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off' AND PressureState = 'TooLow')	@T (PressureState = 'Permitted') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'TooHigh') OR @T (PressureState = 'TooHigh') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'Permitted) OR @T (Overridden) WHEN SafetyInjectionSignal = 'On'
Permitted	@T (PressureState = 'TooLow') WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden) OR @T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off' AND PressureState = 'TooLow')	@T (PressureState = 'Permitted') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'TooHigh') OR @T (PressureState = 'TooHigh') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'Permitted') OR @T (Overridden) WHEN SafetyInjectionSignal = 'On'
TooHigh	@T (PressureState = 'TooLow') WHEN (SafetyInjectionSignal = 'Off' AND NOT Overridden) OR @T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off' AND PressureState = 'TooLow')	@T (PressureState = 'Permitted') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'TooHigh') OR @T (PressureState = 'TooHigh') WHEN (SafetyInjectionSignal = 'On' AND NOT PressureState = 'Permitted') OR @T (Overridden) WHEN SafetyInjectionSignal = 'On'
SafetyInjection-Signal	'On'	'Off'

Table 11. Intermediate Event Table for SafetyInjectionSignal **(before simplification)**

Mode	Events	
TooLow	@T (Block = 'On') WHEN (PressureState = 'Permitted' AND NOT Overridden) OR @T (Block = 'On') WHEN (PressureState = 'TooLow' AND NOT Overridden)	@T (Reset = 'On') WHEN Overridden OR @T (PressureState = 'TooHigh') WHEN Overridden
Permitted	@T (Block = 'On') WHEN (PressureState = 'Permitted' AND NOT Overridden) OR @T (Block = 'On') WHEN (PressureState = 'TooLow' AND NOT Overridden)	@T (Reset = 'On') WHEN Overridden OR @T (PressureState = 'TooHigh') WHEN Overridden
TooHigh	@T (Block = 'On') WHEN (PressureState = 'Permitted' AND NOT Overridden) OR @T (Block = 'On') WHEN (PressureState = 'TooLow' AND NOT Overridden)	@T (Reset = 'On ') WHEN Overridden OR @T (PressureState = 'TooHigh') WHEN Overridden
Overridden	True	False

Table 12. Intermediate Event Table for Overriden **(before simplification)**

initialized event table for Overridden is similar.

3.7.3. Generating event tables: filling in tables with SCR expressions

The SCR event tables are then derived by (a) brute force filling of columns of initialized event tables with SCR expressions from the corresponding output tables, without considering the associated mode values in each row, and then (b) simplifying the SCR expressions by propagation of mode properties (see Section 3.7.4 hereafter).

Brute force filling of event table columns is done by filling each cell in the column associated with some target value with the SCR expression associated with that target value in the corresponding output table. (Each cell in a column is therefore filled with the same SCR expression.) The SCR table thereby obtained is trivially equivalent to the original output table. Tables 11 and 12 show the event tables obtained for variables SafetyInjectionSignal and Overridden, respectively.

3.7.4. Simplifying event tables

Some cells in the event tables obtained thus far may be simplified in some specific cases according to the SCR semantics of event tables [14]. We capture this by a set of simplification rules.

Mode	Events	
TooLow	@T (NOT Overridden)	@T (PressureState = 'Permitted') OR @T (Overridden)
Permitted	@T (PressureState = 'TooLow') WHEN NOT Overridden	@T (Overridden)
TooHigh	False	@T (Overridden)
SafetyInjection-Signal	'On'	'Off'

Table 13. Simplified Event Table for SafetyInjectionSignal

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

Table 14. Simplified Event Table for Overridden

Rule 1: Eliminate entry in self-mode

For a cell in a row associated with mode *m*, an event *@T(AMC = **m**) WHEN d* may be removed from the cell, since the event then amounts to 'False' (if the current mode is already *m*, it is impossible to enter into *m*).

Rule 2: Eliminate event with WHEN clause inconsistent with current mode

For a cell in a row associated with mode *m*, an event *@T(c) WHEN d* may be removed when one of the conjuncts in *d* is inconsistent with mode *m* (in particular, when one of the conjuncts has the form *AMC* ¹ *m* or *AMC* = *n* with n ≠ m), since the event then amounts to 'False'.

Rule 3: Simplify WHEN clause in self mode

For a cell in a row associated with mode *m*, the condition *AMC* = *m* may be removed from the conjuncts of a WHEN clause, since the condition then amounts to 'True'.

Rule 4: Eliminate event inconsistent with mode transition table

For a row associated with mode *m*, an event of form *@T(AMC = **n**) WHEN d* may be removed when there is no transition from mode *n* to mode *m* in the mode transition table of the AMC, since the event then amounts to 'False'.

This rule can be used in Tables 11 and 12 to remove the events with *@T(PressureState = 'TooHigh')* in the row associated with mode TooLow, and in Table 11 to remove the event with *@T(PressureState = 'TooLow')* in the row associated with mode TooHigh.

A further rule allows one to simplify WHEN clauses that refer to the target value of the variable being defined.

Rule 5: Simplify WHEN clauses on target variables

For a cell in a column associated with some target value *v* for the variable *var* being defined, conditions *var* ¹ *v* in WHEN clauses may be removed since they are trivially satisfied.

Similarly, conditions *var* = *w*, with *w* ≠ *v* and *v*, *w* being the only two values for *var*, may be removed from WHEN clauses.

Back to Table 11, let us consider the cell associated with mode TooLow and target value 'On'. Rule 1 allows us to eliminate the first event in the cell; Rule 3 allows us to simplify the second event of the cell into

@T (NOT Overridden) WHEN (SafetyInjectionSignal = 'Off')

Rule 5 allows us to simplify this event further into

@T (NOT Overridden)

Let us now consider the cell associated with mode 'Permitted' and target value 'On' in Table 11. Rule 5 allows us to simplify the first event into

@T (PressureState = 'TooLow') WHEN NOT Overridden

Rule 2 allows us to remove the second event.

Tables 13 and 14 give the complete result of applying our simplification rules to Tables 11 and 12, respectively.

3.7.5. Introducing @T(InMode) expressions

The SCR language has a special notation that can be used in a row of an event table to describe system entry into the group of modes in that row. For example, the sub-expression "@T(Inmode)" in the first row of Table 2 means "if the system enters into TooLow or Permitted, then Overridden becomes false."

The use of the *@T(InMode)* notation is considered to be error-prone and is often discouraged. A table using the

Mode	Events	
TooLow	@T (NOT Overridden) OR @ T(Inmode) WHEN (NOT Overridden AND PressureState = 'Permitted')	@T (Overridden)
Permitted	False	@T (Overridden) OR @T (Inmode) WHEN PressureState = 'TooLow'
TooHigh	False	@T(Overridden)
SafetyInjection-Signal	'On'	'Off'

Table 15. Event Table for SafetyInjectionSignal **with** *InMode* **constructs (before further simplification)**

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

Table 16. Event Table for Overridden **with** *InMode* **constructs (before further simplification)**

@T(InMode) notation can be converted into an equivalent table without that notation, and vice-versa. Jeffords and Heitmeyer show how the event table for *Overridden* in Table 2 with *@T(InMode)* constructs can be transformed into an equivalent *InMode*-free event table [19]. The opposite transformation to introduce *InMode* constructs in event tables can be performed systematically by application of the following rule.

Rule 6: @T(InMode)-event introduction

Let AMC be the associated mode class of an event table. If an event having the form

@T(AMC = m) WHEN d

appears as one of the disjuncts in a table cell associated with mode *n* and target value *v*, with $n \neq m$, then this event may be removed from the disjuncts in this cell whereas the event

@T(InMode) WHEN (d AND AMC = n)

is added as a new disjunct in the cell associated with mode value *m* and target value *v*. (The original event has thereby been moved from the row associated with mode *n* to the row associated with mode *m*).

Let us illustrate this rule on Table 13. The event leading to target value 'On' from mode *Permitted* is moved to the row associated with mode *TooLow* where it is rewritten as

@T (Inmode)
WHEN (NOT Overridden AND PressureState = 'Permitted')

Tables 15 and 16 show the result of applying Rule 6 on Tables 13 and 14, respectively.

Tables generated through @T(InMode)-event introduction (Rule 6) may sometimes be simplified further thanks to the following simplification rules.

Rule 7: Simplify WHEN clauses based on possible mode transitions

An event having the form

@T(InMode) WHEN d AND AMC = n

in a row associated with mode *m* may be simplified into

@T(InMode) WHEN d

provided *n* is the only source of transition to mode *m* in the mode transition table of the AMC mode.

This rule can be used in Table 15 to simplify the event

@T (InMode)
WHEN NOT Overridden AND PressureState = 'Permitted'

into mode TooLow , and in Table 16 to simplify the event

@T (InMode)
WHEN PressureState = 'Permitted'

into mode TooHigh.

Rule 8: Simplify mode classes covering @T(Inmode) events

In a row associated with mode *m*, a disjunction having the form

@T (InMode) WHEN (d AND AMC = i)
OR …
OR @T (InMode) WHEN (d AND AMC = j) ,

covering all possible modes i, j except mode m, may be replaced by the single event

@T (InMode) WHEN d

There is no application of this rule in our tables. Imagine however that in mode TooLow in Table 15 we had the following disjunction of events:

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

Table 17. Event Table for SafetyInjectionSignal **with** *InMode* **constructs (after simplification)**

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

Table 18. Event Table for Overridden **with** *InMode* **constructs (after simplification)**

@T (InMode)
WHEN (PresssureState = 'Permitted' AND NOT Overridden)
OR @T (InMode)
WHEN (PresssureState = 'TooHigh' AND NOT Overridden)

Applying Rule 8 would allow us to simplify this disjunction into the following single event:

@T (InMode) WHEN NOT Overridden

Rule 9: Remove useless events due to @T(InMode)

If a row of an event table has all cells being marked 'False' except one having the form *@T(Inmode) OR @X* then this cell may be simplified into *@T(Inmode)*, since once this mode is entered no event can cause the target value of the output variable to change to another value in that mode; events *@X* have thus no effect as they just prescribe the output variable to keep that same target value.

This rule can be used in Table 16 to remove the event @T(Reset = 'On') from the row associated with mode TooHigh.

Such simplifications should by default be applied until the simplest possible tables are reached. However, keeping redundant information in tables may sometimes improve their clarity. Simplifications might therefore not be applied sometimes for sake of clarity.

Tables 17 and 18 show the result of applying the above simplifications rules to Tables 15 and 16, respectively.

3.8. From event tables to condition tables

An event table can be transformed into a semantically equivalent condition table provided that some invariants hold on the event table.

For example, consider the event table for variable SafetyInjectionSignal; an equivalent condition table, if it exists, would have the general form shown in Table 19.

Mode	Conditions	
TooLow	?A	NOT ?A
Permitted	?B	NOT ?B
TooHigh	?C	NOT ?C
SafetyInjection-Signal	'On'	'Off'

Table 19: General form of a condition table for *SafetyInjectionSignal*

In this table, ?A, ?B, and ?C are placeholders for still unknown SCR conditions. In order to satisfy the completeness and disjointness criteria on conditions tables, the conditions in the column for target value 'Off'' are negations of those in the column for target value 'On'.

The semantics of this condition table is that the following three assertions hold in all states of the system:

PressureState = 'TooLow' $\rightarrow$
(SafetyInjectionSignal = 'On' $\leftrightarrow$?A)

PressureState = 'Permitted' $\rightarrow$
(SafetyInjectionSignal = 'On' $\leftrightarrow$?B)

PressureState = 'TooHigh' $\rightarrow$
(SafetyInjectionSignal = 'On' $\leftrightarrow$?C)

The problem of building a condition table for SafetyInjectionSignal from its event table therefore amounts to finding conditions ?A, ?B, and ?C so that the above properties are invariants for the event table.

In the general case, the problem of transforming an event table into an equivalent condition table thus amounts to finding SCR conditions to fill the condition table so that:

(i) the completeness and disjointness properties of

condition tables are met,

(ii) the assertions capturing the condition table semantics are invariants on the event table.

Event tables for which no such conditions can be found have no equivalent condition table.

One possible technique for finding those conditions is to use temporal logic query checking on SCR models [12]. Given

- a SCR event table,
- the assertions defining the condition table semantics as a query in which the conditions to be found are placeholders,

temporal logic query checking will give us the solution to the query or will tell us that no solution exists. (The technique requires that the SCR model is propositional or can be treated as a propositional model, which is the case here.)

For the condition table defining SafetyInjectionSignal, the solution for conditions ?A, ?B, and ?C is given by:

?A = (Overridden = 'False')
?B = False
?C = False

This corresponds to the original SCR specification shown in Table 3.

It may be worth noting that the invariant generation algorithm in [19] when applied to Tables 13 or 17 generates the following invariants:

SafetyInjectionSignal = 'On' $\rightarrow$ Overridden = 'False'
$\wedge$ PressureState = 'TooLow'
SafetyInjectionSignal = 'Off' $\rightarrow$ true

These invariants are not strong enough to build the condition table for SafetyInjectionSignal.

4. Evaluation

We may now compare the three specifications for the safety injection case study, namely, the "native" SCR specification [14], the "native" KAOS specification [25] and the SCR specification 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 [14]; the only two slight differences are in the event table for term *Overridden*. The original table for this term is given in Table 2; the one derived by our procedure is shown in Table 18.

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 descriptive domain properties 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 must be downpropagated to the tables derived; we feel it much 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 [25] using the SMV model checker [28]. Basically, we translated both specifications into SMV syntax and compared the values of the controlled variables in the original KAOS model and in the derived SCR specification, respectively [10]. One would expect that the respective values of controlled variables do not match in exactly the same state in view of the one-state shift introduced to squeeze output table expressions into SCR's synchrony hypothesis (see Section 3.5).

SMV showed that these values are "almost matching" under a quiet input assumption. This means roughly that when input events are sparse the respective values are the same at any time point up to 0, 1 or 2 time unit shifts.

The reason for this is the lack of synchrony hypothesis in the KAOS model together with the length of the longest path in the variable dependency graph. For example, when the value of one of the monitored variables is changing in the KAOS model, it will take one time unit to update the value of Overridden, then another time unit to update the value of the controlled variable SafetyInjectionSignal. In the SCR model, all such changes occur synchronously.

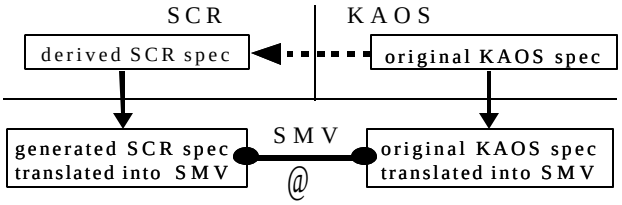

Fig. 2 – Comparing the KAOS and derived SCR specifications

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.

5. Conclusion

Thanks to our technique 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 [23] and conflicts [20]. Conversely, KAOS modelers may obtain downstream tabular specifications in a systematic way for later specification analysis through exhaustiveness checking [14], simulation [15], behavior model checking [16] and test data generation [11].

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 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 bypass 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 goal realizability [26] 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 [19]. Several insights and techniques gained from the process described in this paper would be applicable to the problem of deriving SCR tables directly from declarative goals (e.g., the techniques for extracting mode classes, for simplifying SCR tables, or the eager/lazy strategies for getting rid of non-determinism). 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 Intl. Conference on Software Enginering*, Kyoto, April 1998.

[2] M. Archer, C. Heitmeyer and S. Sims, "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* Vol. 19 No. 2, 89.

[5] R. Bharadwaj and C. Heitmeyer, "Applying the SCR Requirements Specification Method to a Simple Autopilot", *Proc. 4th NASA Langley Formal Methods Workshop*, Sept. 1997.

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

[7] 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.

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

[9] 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.

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

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

[12] A. Gurfinkel, B. Devereux and M. Chechik, "Model Exploration with Temporal Logic Query Checking", *Proc. FSE'10:* 10th *ACM Symp. Foundations of Software Engineering*, Charleston, November 2002.

[13] 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.

[14] C. Heitmeyer, R.D. Jeffords and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications", *ACM Trans. on Software Eng. and Methodology* Vol. 5 No. 3, July 1996, 231-26.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

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

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

[32] 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.

[33] W. N. Robinson, "Requirements Interaction Management", *ACM Computing Surveys*, June 2003.

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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.

[40] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, 1997, 1-30.