Deriving Operational Software Specifications from System Goals

Emmanuel Letier and Axel van Lamsweerde

Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)

{eletier,avl}@info.ucl.ac.be

ABSTRACT

Goal orientation is an increasingly recognized paradigm for eliciting, modeling, specifying and analyzing software requirements. Goals are statements of intent organized in AND/OR refinement structures; they range from high-level, strategic concerns to low-level, technical requirements on the software-to-be and assumptions on its environment. The operationalization of system goals into specifications of software services is a core aspect of the requirements elaboration process for which little systematic and constructive support is available. In particular, most formal methods assume such operational specifications to be given and focus on their a posteriori analysis.

The paper considers a formal, constructive approach in which operational software specifications are built incrementally from higher-level goal formulations in a way that guarantees their correctness by construction. The operationalization process is based on formal derivation rules that map goal specifications to specifications of software operations; more specifically, these rules map real-time temporal logic specifications to sets of pre-, post- and trigger conditions. The rules define operationalization patterns that may be used for guiding and documenting the operationalization process while hiding all formal reasoning details; the patterns are formally proved correct once and for all. The catalog of operationalization patterns is structured according to a rich taxonomy of goal specification patterns.

Our constructive approach to requirements elaboration requires a multiparadigm specification language that supports incremental reasoning about partial models. The paper also provides a formal semantics for goal operationalization and discusses several semantic features of our language that allow for such incremental reasoning.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification - *methodologies, languages*. D.2.4 [**Software Engineering**]: Software Verification - *formal methods, correctness proofs*.

General Terms

Design, Verification, Languages, Documentation.

Keywords

Goal-oriented requirements engineering, operationalization, patterns, incremental specification, reasoning about partial models, frame problem, KAOS.

1. INTRODUCTION

Research on formal specification has produced an impressive amount of helpful techniques and tools for specification analysis (such as algorithmic model checking or deductive verification), specification animation, specification-based testing, specification reuse and specification refinement [30]. All these techniques presuppose that a formal specification is available. Building formal specifications for complex software is not an easy task though. The problem is not merely one of translating natural language statements into some formal language. Specification-in-the-large in general requires complex models to be elaborated, structured, interrelated and negotiated [29].

Goal-oriented requirements engineering refers to the use of goals for requirements elicitation, elaboration, organization, specification, analysis, negotiation, assignment, documentation and evolution [31]. *Goals* are objectives the system under consideration must achieve. The word "system" here refers to the software-to-be together with its environment [11]. Goals are formulated in terms of optative statements [41] which may refer to functional or non-functional properties [5] and range from high-level concerns (such as "safe transportation" for a flight control system) to lower-level ones (such as "reverse thrust enabled when wheels pulse on").

There are multiple reasons why goals must be made explicit in the requirements engineering process [31]. Goals drive the elaboration of requirements to support them; they provide a criterion for requirements completeness and pertinence; they induce rich specification structuring mechanisms such as AND-decomposition/composition for refinement/abstraction and OR-decomposition for reasoning about alternatives; they thereby provide a rationale for requirements and allow one to trace low-level details back to high-level concerns. The higher-level a goal is, the more stable it is likely to be; goals are thus essential elements for managing requirements evolution. Last but not least, goals have been recognized to be the roots at which conflicts should be detected and resolved.

Achieving goals in general requires the cooperation of multiple *agents* such as humans, devices and software. For example, the high-level goal of "safe transportation" might require the cooperation of the pilot, the autopilot software, the on-board TCAS software, the on-ground tracking system, etc. The essence of goal refinement is to decompose a goal into subgoals so that each subgoal requires the cooperation of fewer agents; the refinement process stops when goals can be assigned as responsibility of single agents [6]. Terminal goals assigned to agents in the software-to-be become *requirements*; terminal goals assigned to agents in the environment become *assumptions* (or normative policies); the latter cannot be enforced by the software-to-be. In general, alternative responsibility assignments are to be explored; for example, the goal "FlightPathAngle mode engaged until aircraft near desired altitude" might be assigned to the Pilot or to the Autopilot agent. Different alternatives for goal refinement and assignment yield alternative system proposals in which more or less features are automated.

Functional goals assigned to software agents need to be operationalized into specifications of services the agents should provide to meet them [Myl92, 6]; in general, non-functional goals about quality of service are used to select goal refinement/assignment alternatives that meet them best [5]. Formally speaking, *operationalization* is a process that maps declarative property specifications to operational specifications satisfying them.

How to produce "correct" operationalizations is the topic explored by this paper. Suppose, for example, that the goal "reverse thrust enabled when wheels pulse on" has been assigned to the Autopilot agent; we would like to infer, from a formal specification of that goal, the need for two Autopilot operations, namely, EnableReverseThrust and DisableReverseThrust, together with a formal specification for these operations that guarantees goal achievement.

The paper presents a pattern-based technique for operationalizing goals, specified in real-time linear temporal logic (RT-LTL), into operations specified by pre-, post- and trigger conditions (PPTC). The technique is

- *constructive:* the operationalizations derived are guaranteed correct by construction;
- *incremental:* the operational specifications can be built gradually from partial models;
- *lightweight:* formal correctness proofs remain hidden.

Roughly speaking, an operationalization pattern captures a rule for inferring generic PPTC specifications from a generic RT-LTL specification. Our patterns are proved correct once and for all (using the STeP verification system [36]); "correct" means here that the "conjunction" of the PPTC specification patterns entails the RT-LTL specification pattern (this notion of correctness will be made fully precise in the paper). The derivation of goal operationalizations through pattern instantiation then gives instantiated correctness proofs for free.

Our operationalization patterns were identified from a taxonomy of frequently used goal patterns; this taxonomy is partially adapted from the catalog of specification patterns in [9] for which some empirical evidence has been reported on breadth of coverage.

There is a considerable body of work on the problem of refining *software specifications* into *programs*, e.g., [8, 1, 38]). In contrast, this paper is concerned with the much less explored, upstream problem of refining *declarative goals* into operational *software specifications* at requirements engineering time. From a strictly mathematical point of view, the two problems may appear as variants of a similar problem, that is, one of refining some higher-level model into some lower-level one. There are three important differences however:

- our specification language has features important for requirements engineering that are not present in specification formalisms used in the later phase of program derivation --such as the distinction between domain properties and requirements [16], the distinction between pre- and trigger conditions [6], the traceability of software specifications to system goals underlying them [31];
- our approach supports incremental derivations of partial specifications from incomplete goal models (in contrast with program derivation techniques which generally require a complete formal specification to start from);
- our derivation rules aim at supporting a lightweight, pattern-based refinement process for frequent temporal logic specifications (in contrast with the lower-level rules used in refinement calculi).

There is also a considerable body of work on formal techniques that combine declarative and operational specifications (e.g., [35, 24]). In contrast with the constructive approach proposed here, such techniques focus on the *a posteriori* verification of operational specifications with respect to declarative ones.

The principle of using derivation rules for guiding goal operationalization was already suggested in [6]. The rules there are more limited in coverage, do not rely on a fully formal basis and lack some desirable semantic properties we discuss below. The rules in this paper complement a catalog of formal refinement patterns we have developed for the upstream phase of goal refinement [7]; the patterns there involve the goal specification language only.

The rest of the paper is organized as follows. Section 2 introduces some background material on goal-oriented elaboration of requirements that will provide some context, terminology and examples used in the sequel. Section 3 discusses some semantic features of our operational language needed for incremental reasoning about partial models; a formal characterization of operationalization is then provided in this semantic framework. Section 4 presents our pattern-based approach, illustrates the use of a few typical operationalization patterns and discusses the structure and coverage of our pattern catalog.

2. GOAL-ORIENTED ELABORATION OF REQUIREMENTS

Our operationalization techniques were developed in the KAOS framework for goal-oriented requirements engineering [6, 29]. We briefly introduce some background which the paper relies on.

2.1 Modeling and specifying requirements

An application model is composed of four submodels: a *goal model* in which the goals to be achieved by the system are described together with their alternative refinement links and their conflict links; an *object model* in which the application objects involved are described together with their relationships and attributes; an *agent model* in which the agents in the system are described together with their interfaces and responsibilities with respect to the goals; and an *operation model* in which the services operationalizing the goals assigned to software agents are described. Each model has a separate semantics and is related to the others through inter-model consistency rules. The paper concentrates on the derivation of the operation model from the goal model; some features of the operation model will therefore be presented in greater detail as they provide the basis for our operationalization process.

The specification language is a multiparadigm language with a two-layer structure: an outer graphical layer for *modeling* concepts (such as goals, objects or agents) and an inner assertion layer for *specifying* such concepts formally. The assertion layer is optional and used for formal reasoning.

2.1.1 The goal model

The various objectives the system should meet are defined in this model and interrelated through AND/OR refinement links. For example, the following safety goal might be considered for a mine pump control system [23, 21]:

Goal Maintain [PumpOnWhenHighWater]
  InformalDef *The pump shall be on when the water level is too high*
  FormalDef $\forall$ m: Mine, p: Pump
    m.WaterLevel $\geq$ 'High' $\wedge$ HasPump (m, p)
    $\Rightarrow \circ$ p.Motor = 'On'
  Refines Avoid [MineOverflowed]
  RefinedTo HighWaterDetected,
    PumpSwitchOnWhenHighWaterDetected,
    PumpOnWhenPumpSwitchOn

This specification fragment introduces a concept of type "goal" named PumpOnWhenHighWater, corresponding to some property that should always hold in future states ("Maintain" verb), defined by some informal statement, refining a parent goal Avoid[MineOverflowed], and refined into three subgoals. (For lack of space the graphical layer is represented in textual form here.)

A goal defines a set of admissible histories in the composite system under consideration. A history is a temporal sequence of global system states; a state is a mapping that assigns a value to each object attribute and relationship from the object model. A real-time linear temporal logic is therefore natural for specifying goals [35, 22, 6]. The following temporal operators are used in this paper:

$\circ$	(in the next state)	$\bullet$	(in the previous state)
$\Diamond$	(some time in the future)	$\blacklozenge$	(some time in the past)
$\Box$	(always in the future)	$\blacksquare$	(always in the past)
$\boldsymbol{W}$	(always in the future *unless*)	$\boldsymbol{U}$	(always in the future *until*)
$\Diamond_{\leq d}$	(some time in the future within deadline *d*)		
$\Box_{\leq d}$	(always in the future up to deadline *d*)		
$P\ \boldsymbol{B}\ Q$	(P has remained true *back to the last time* Q was true, if any)		
$P\ \boldsymbol{S}_{=d}\ Q$	(P has remained true *since* Q was true *d* time units ago)		

The following standard logical connectives are also used: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (strongly implies), $\Leftrightarrow$ (strongly equivalent), with

$$P \Rightarrow Q \ \text{ iff } \ \Box\,(P \rightarrow Q) \qquad P \Leftrightarrow Q \ \text{ iff } \ \Box\,(P \leftrightarrow Q)$$

Goal definition patterns are used for lightweight specification of goals at the modeling layer, e.g.,

Achieve:	$C \Rightarrow \Diamond\, T$, $C \Rightarrow \Diamond_{\leq d}\, T$
Cease:	$C \Rightarrow \Diamond \neg T$, $C \Rightarrow \Diamond_{\leq d} \neg T$
Maintain:	$C \Rightarrow T$, $C \Rightarrow T\, \boldsymbol{W}\, N$
Avoid:	$C \Rightarrow \neg T$, $C \Rightarrow \neg T\, \boldsymbol{W}\, N$

where C, T, and N denote some current, target, and new condition, respectively. Section 4 will introduce further specializations of these patterns as a basis for identifying relevant goal operationalization patterns.

In the above specification of the goal PumpOnWhenHighWater, the conjunction of the assertions formalizing the subgoals HighWaterDetected, PumpSwitchOnWhenHighWaterDetected and PumpOnWhenPumpSwitchOn must entail the formal assertion of the parent goal PumpOnWhenHighWater they refine together. Every formal goal refinement generates a corresponding proof obligation [7].

2.1.2 The object model

This model defines the domain entities, relationships and attributes that are relevant to goal formulations.

For example, in the assertion formalizing the goal PumpOnWhenHighWater above, WaterLevel is an attribute of the Mine object; the predicate HasPump (m, p) captures a relationship between Mine and Pump objects. The corresponding declarations are found in the object model, e.g.,

Entity Mine
  Has WaterLevel: *WaterLevelUnit*

Relationship HasPump
  Links Mine {**card** 0:1}, Pump {**card** 0:1}

Objects are further specified informally and/or formally by means of domain invariants that capture their indicative properties in the domain [41].

2.1.3 The agent model

This model defines the responsibilities and interfaces of the various agents forming the composite system (humans, devices or software). Goal refinement ends when all subgoals can be assigned to single agents. For example, consider the goal PumpSwitchOnWhenHighWaterDetected that was declared to be subgoal of the goal PumpOnWhenHighWater; it might be specified as follows:

$\forall$ c: PumpController
c.HighWaterFlag = 'On' $\Rightarrow \circ$ c.PumpSwitch = 'On'

This goal might be assigned to the PumpController agent provided the latter has sufficient monitoring and control capabilities to *realize* the goal [39, 16, 33]; the agent must be able to monitor the quantity denoted by c.HighWaterFlag and to control the quantity denoted by c.PumpSwitch. Such monitoring and control links define the agent's interface in the agent model:

Agent PumpController
  Monitors PumpController.HighWaterFlag, ...
  Controls PumpController.PumpSwitch, ...
  ResponsibleFor PumpSwitchOnWhenHighWaterDetected, ...
  Has PumpSwitch, HighWaterFlag: *{On, Off}*

Agents are in fact objects; they may be further characterized by domain-specific attributes (e.g., PumpSwitch), relationships and invariants.

2.1.4 The operation model

This model defines the various services to be provided by software agents. An operation is an input-output relation over components of the object model; operation applications define state transitions along the histories prescribed by the goal model. Operations are declared by signatures over objects and specified by pre-, post- and trigger conditions. An important distinction is made between *domain* pre-/postconditions, which capture the elementary state transitions defined by operation applications in the domain, and *required* pre-/postconditions, which capture additional strengthenings to ensure that the goals are met. The former conditions are *descriptive* whereas the latter conditions are *prescriptive* [16, 41]. Traceability between operations and their underlying goals is thereby supported.

For example, the operation SwitchPumpOn might be partially specified by:

Operation SwitchPumpOn
  Input c: PumpController; **Output** c: PumpController/PumpSwitch
  DomPre c.PumpSwitch $\neq$ 'On'
  DomPost c.PumpSwitch = 'On'

The signature part declares an input-output relation between sets of states of the instance variable *c* of type PumpController. The Output clause restricts the scope of the operation to the PumpSwitch attribute of the PumpController object; this means that the operation is allowed to change the value of this attribute only. The above domain conditions describe what an application of the operation means in the domain without any prescription as to when the operation must be applied and when it may not be applied [34].

Operationalization refers to the process of prescribing additional pre-, trigger-, and postconditions on operations in order to achieve goal specifications. For example, the following operational requirement has to be added in order to ensure the goal PumpSwitchOnWhenHighWaterDetected introduced before:

Operation SwitchPumpOn
  ...
  ReqTrig for PumpSwitchOnWhenHighWaterDetected:
    c.HighWaterFlag = 'On'

This trigger condition captures an obligation to trigger the operation when the high water flag is 'On' provided the pump switch is not 'On' (see the **DomPre** assertion).

The domain precondition of an operation is formalized by a state predicate on the initial state of the operation; the domain and required postconditions are formalized by predicates on the operation's final state, with possible references to the previous state (that is, the operation's initial state); the required pre- and trigger conditions are formalized by predicates on the operation's initial state, with possible references to past states through past temporal operators. The semantics of required pre-, trigger-, and postconditions is the following:

- a *required precondition* captures a permission to perform the operation when the condition is true;
- a *required trigger condition* captures an obligation to perform the operation when the condition becomes true *provided* the domain precondition is true;
- a *required postcondition* captures an additional condition that must hold after any application of the operation.

Each required pre-, trigger-, and postcondition is linked to the goal it operationalizes. In general, several required conditions on different operations may be needed to operationalize the goal completely. A set of required pre-, trigger-, and postconditions on operations is said to be a *complete operationalization* of a goal if satisfying all required conditions in the set guarantees the satisfaction of the goal. Like for goal refinement, every goal operationalization generates a corresponding proof obligation. A formal semantics of what is meant for a goal operationalization to be complete is presented in Section 3.

Back to our example, a complete operationalization of the goal PumpSwitchOnWhenHighWaterDetected will be given by the above required trigger condition on the operation SwitchPumpOn *together with* the following required precondition on the operation SwitchPumpOff:

Operation SwitchPumpOff
  Input c: PumpController; **Output** c: PumpController/PumpSwitch
  DomPre c.PumpSwitch ≠ 'Off'
  DomPost c.PumpSwitch = 'Off'
  ReqPre for PumpSwitchOnWhenHighWaterDetected:
    c.HighWaterFlag ≠ 'On'

This required precondition is necessary to ensure the goal; it states that the pump switch may not be turned off when the high water flag is 'On'.

Operations capture *atomic* state transitions. In our real-time temporal logic framework, the application of an operation maps a state to a *next* state whose time distance is one time unit in the smallest unit scale. When an activity lasting over several states has to be modelled, one has to introduce an atomic operation that starts the activity, an atomic operation that ends the activity (or several operations if there are different ways in which the activity can terminate); constraints on activities are then formulated as assertions in the goal model that link events corresponding to applications of the start/end operations. Operation models that include such operations to encode activities can be derived systematically from the goal model like any other operation model. Note thus that all the structuring and decomposition/refinement in KAOS is done at the declarative, goal level.

Operations may be applied *concurrently*. A *non*-interleaving semantics is required by the semantics of trigger conditions as immediate obligations. With an interleaving semantics, an operation model would be inconsistent when the trigger conditions of two (or more) operations are true at the same time.

Goal operationalization is an *incremental* process. New goals may need to be operationalized through new operations and/or new required pre-, trigger and postconditions on operations already identified. The semantics of the operational language therefore needs to be compositional (see Section 3).

The incremental operationalization process may result in inconsistent operation models in which, for example, an operation must be applied due to one of its required trigger conditions (to achieve some goal G1) and at the same time it may not be applied due to one of its required preconditions (to achieve some other goal G2). Such situations correspond to violations of the following consistency meta-rule of the KAOS language [6]:

$$\text{ReqTrig} \land \text{DomPre} \Rightarrow \text{ReqPre}$$

Violations of this rule can actually be traced back to *conflicts* between the goals from which the operational requirements were derived (see [32] for examples of this). Such conflicts are better resolved at the goal level as discussed in [26].

2.2 Elaborating the models

A method is available for elaborating the goal, object, agent and operation models in a systematic fashion; see, e.g., [6, 25, 29] for details and illustrations. Roughly, the method consists in the following intertwined steps: (1) the goal model is elaborated first by asking HOW questions (top-down goal refinement) and WHY questions (bottom-up goal abstraction) and by using refinement patterns [7]; (2) the object model is derived by collecting the objects, attributes and relationships appearing in goal formulations and domain properties involved in goal refinements; (3) agents are identified together with their interfaces and possible responsibilities with respect to goals; (4) "best" alternatives are selected among multiple goal OR-refinements and agent OR-assignments as best trade-offs to achieve the non-functional goal offsprings from the goal graph; (5) functional goals assigned to software agents are operationalized into operations to meet them.

Variants of this general scheme are available to integrate scenario-based elicitation [30], conflict analysis [26] and obstacle analysis [28]. The next sections focus on the last step of goal operationalization.

3. SEMANTICS OF OPERATIONALIZATION

This section provides the necessary foundations for the operationalization techniques presented in the next section; in particular, the notion of "correct" operationalization is made fully precise. As mentioned before, we want operationalization to be an incremental process; the correctness of a goal operationalization should be established locally, based *only* on the specifications of the goal and operations involved in the operationalization. We first discuss how such incrementality influences the semantics to be given to the operation model. The formal semantics of operationalization is presented next.

3.1 Generative semantics vs. pruning semantics

Two alternative styles of semantics are generally considered for an operational specification language.

***Generative semantics**: Every behavioral change is forbidden, except the ones explicitly required by the specification.*

***Pruning semantics**: Every behavioral change is allowed, except the ones explicitly forbidden by the specification.*

With a generative semantics, operations are viewed as generating the set of admissible histories of the system; these are assumed to cover the only transitions that are possible. Examples of languages defined through such a semantics include languages based on transition systems (e.g., the state machines in [35], Statecharts [14] or SCR tables [15]) and state-based languages such as VDM [20]. One advantage of this style of semantics is its built-in assumption that nothing changes except if an operation explicitly requires it; the specifier is relieved from explicitly specifying what does not change --in other words, a generative semantics avoids the frame problem [2]. There is a price to pay though; the built-in frame assumption is incompatible with the idea of partial models or views [17]. A generative semantics makes it extremely difficult to support incremental reasoning about partial models.

To illustrate the point, consider again the operationalization of the goal Maintain[PumpSwitchOnWhenHighWaterDetected] into a required trigger condition on the operation SwitchPumpOn and a required precondition on the operation SwitchPumpOff (see Section 2.1.4). Assume now that instead of this complete operationalization, we would have built an operation model without the operation SwitchPumpOff; the goal would have been operationalized only through the required trigger condition on the operation SwitchPumpOn. With a generative semantics, it would be possible to prove that this required trigger condition alone is sufficient to guarantee the satisfaction of the goal, because the semantics would rely on the assumption that there are no transitions of the pump switch from 'On' to 'Off'. The problem now is that if the operation SwitchPumpOff is added later on to the model (e.g., to operationalize another goal), the satisfaction of the original goal is no longer guaranteed by the required trigger condition on the operation SwitchPumpOn alone; the goal is violated if the operation SwitchPumpOff is applied when the high water signal is 'On'. A process in which the correctness of a goal operationalization needs not be reconsidered every time a new operation is added to the model is of course undesirable.

To avoid the problem induced by a generative semantics, languages such as Z [40], LARCH [13] or temporal logic-based formalisms have taken the dual perspective of a *pruning semantics* in which operational specifications are viewed as restrictions on the state transitions allowed; the specifications prune the set of admissible histories of the system. Incremental elaboration and reasoning through composition of partial models then becomes possible provided the frame problem is handled in an appropriate way.

We take this approach and handle the frame problem through two built-in axioms within our semantics in order to relieve the specifier from the obligation to explicitly state everything that does not change.

Frame axiom 1: *Any attribute/relationship variable not declared in the output clause of the specification of an operation is left unchanged by any application of this operation.*

Note that this built-in axiom is not incompatible with incremental elaborations from partial models because our semantics allows operations to be applied concurrently. A variable that is not among the output of an operation is still free to change through the application of other operations provided the latter are applied concurrently with that operation.

This frame axiom is enforced by requiring the **DomPost** and **ReqPost** conditions of an operation to refer only to those state variables which are explicitly declared in the output clause of the operation (in a way similar to LARCH [13]).

Frame axiom 2: *Every state transition that satisfies the domain pre- and postconditions of an operation corresponds to an application of this operation.*

This second axiom corresponds to the following meta-rule (which will be made more rigorous later on):

for any operation *op*:

$\text{DomPre (op)} \wedge \circ\, \text{DomPost (op)} \Rightarrow \text{Performed (op)}$

Note again that this axiom is not incompatible with incremental elaborations from partial specifications. It does not say that the operation is the only one that can cause a transition from a state satisfying its domain precondition to a state satisfying its domain postcondition. What it does is to force the simultaneous application of operations with overlapping domain pre- and postconditions; this allows multiple views to be combined by synchronization of their operations, as discussed in [17].

To illustrate how this second frame axiom enables one to reason locally about the correctness of goal operationalizations, let us come back to the operationalization of the goal Maintain[PumpSwitchOnWhenHighWaterDetected] into the trigger condition on the operation SwitchPumpOn and the required precondition on the operation SwitchPumpOff (see Section 2.1.4). The latter condition requires that the operation SwitchPumpOff be performed only if the high water flag is not 'On'. Our second frame axiom instantiated to this operation states that every transition from a state where the pump switch is not 'Off' to a state where it is 'Off' corresponds to an application of the SwitchPumpOff operation. This guarantees that the goal operationalization is complete even if another operation that turns the pump switch to 'Off' is added later to the model; in such a case the second frame axiom will allow the new operation to be included only if the operation SwitchPumpOff can also be included.

Our second frame axiom bears some similarities with the frame axioms in [2] and the locality axioms in [10].

3.2 A formal semantics for goal operationalization

The formal semantics of the operation model is defined by mapping every construct of the operational language into temporal logic assertions.

As mentioned before, an operation defines a relation over states; this relation is defined by the domain pre- and postconditions of the operation. For every operation *op* in the operation model with logical variables *arg1*, ..., *argn* as arguments and *res1*, ..., *resn* as results, we introduce a temporal logic predicate denoted by

$$[\![\,\text{op}\,]\!]\,(\text{arg1}, ..., \text{argn}, \text{res1}, ..., \text{resn}),$$

which expresses that the operation is currently being applied on the given arguments and results.

Definition 1 *(Semantics of operations)*. For every operation *op* in the operation model, the predicate $[\![\,\text{op}\,]\!]$ is defined as follows:

$$[\![\,\text{op}\,]\!]\,(\text{arg1}, ..., \text{argn}, \text{res1}, ..., \text{resn}) \Leftrightarrow \text{DomPre (op)} \wedge \circ\, \text{DomPost (op)}$$

where DomPre(op) and DomPost(op) are the domain precondition and postcondition of the operation, respectively.

This definition states that (i) every application of an operation implies that the operation's domain precondition is satisfied in the state before the application and the domain postcondition is satisfied in the state after the application, and (ii) every state transition that satisfies the domain pre- and postconditions of an operation corresponds to an application of this operation. Condition (ii) thus captures our second frame axiom.

For example, the operation predicate $[\![\,\text{SwitchPumpOn}\,]\!]\,(\text{c})$ associated with the SwitchPumpOn operation in Section 2.1.4 is defined as follows:

$$[\![\,\text{SwitchPumpOn}\,]\!]\,(\text{c}) \Leftrightarrow \text{c.PumpSwitch} \neq \text{'On'} \wedge \circ\, \text{c.PumpSwitch} = \text{'On'}$$

Let us denote by *ReqPre(op)*, *ReqTrig(op)* and *ReqPost(op)* the sets of required pre-, trigger- and postconditions of an operation *op* in the operation model, respectively. To define the formal semantics of these conditions, we introduce a temporal logic predicate for each required condition *R*, denoted by $[\![\,R\,]\!]$.

Definition 2 *(Semantics of required pre-, trigger- and postconditions)*. For every required condition *R* on an operation *op* in the operation model, the predicate $[\![\,\text{R}\,]\!]$ is defined as follows:

if $\text{R} \in \text{ReqPre (op)}$ **then** $[\![\,\text{R}\,]\!] =_{\text{def}} (\forall *)\, [\![\,\text{op}\,]\!] \Rightarrow \text{R}$

if $\text{R} \in \text{ReqTrig (op)}$ **then** $[\![\,\text{R}\,]\!] =_{\text{def}} (\forall *)\, \text{R} \wedge \text{DomPre (op)} \Rightarrow [\![\,\text{op}\,]\!]$

if $\text{R} \in \text{ReqPost (op)}$ **then** $[\![\,\text{R}\,]\!] =_{\text{def}} (\forall *)\, [\![\,\text{op}\,]\!] \Rightarrow \circ\, \text{R}$

In the above definition, we use the standard notation $(\forall *)\,\text{P}$ for the universal closure of P.

For example, the semantics of the required trigger condition on the SwitchPumpOn operation in Section 2.1.4 is expressed by the following temporal logic assertion:

$$\begin{aligned}&c.HighWaterFlag = \text{'On'} \land c.PumpSwitch \neq \text{'On'}\\ &\quad \Rightarrow [\| SwitchPumpOn \|]\ (c)\end{aligned}$$

We now turn to the definition of the semantics of goal operationalization. As introduced before, a set of required pre-, trigger- and postconditions operationalizes a goal if the satisfaction of the required conditions on the corresponding operations guarantees the satisfaction of the goal.

Definition 3 *(Correctness of goal operationalization).* A set $\{R1, ..., Rn\}$ of required conditions on operations in the operation model correctly operationalize a goal G in the goal model iff the following conditions hold:

1. $[\| R1 \|], ..., [\| Rn \|] \models G$ *(completeness)*
2. $[\| R1 \|], ..., [\| Rn \|] \not\models \textbf{false}$ *(consistency)*
3. $G \models [\| R1 \|], ..., [\| Rn \|]$ *(minimality)*

The completeness condition for goal operationalization has similarities with the corresponding condition for goal refinement [7]. A first important difference however is that the semantics of goal operationalization does not rely on domain properties to guarantee the satisfaction of the goal. This is due to the fact that the agent responsible for the goal may not rely on domain properties to realize the goal [33]. A second important difference is in the definition of the minimality condition. For operationalization, this condition requires that the requirements operationalizing the goal be not stronger than required by the goal. This is related to the fact that an agent responsible for a goal must have the capability of satisfying the goal without being more restrictive than required by the goal [33].

The formal specification of goals and operations allows the completeness, consistency and minimality of operationalizations to be formally verified. For example, the completeness of the operationalization of the goal Maintain[PumpSwitchOnWhenHighWaterDetected] in Section 2.1.4 is established by verifying the following proof obligation (e.g., using STeP [36]):

$$\begin{aligned}&c.HighWaterFlag = \text{'On'} \land c.PumpSwitch \neq \text{'On'}\\ &\quad \Rightarrow [\| SwitchPumpOn \|]\ (c)\\ &\qquad \{\textbf{ReqTrig}\ \text{of SwitchPumpOn}\}\\ &\land [\| SwitchPumpOff \|]\ (c) \Rightarrow \neg\ c.HighWaterFlag = \text{'On'}\\ &\qquad \{\textbf{ReqPre}\ \text{of SwitchPumpOff}\}\\ &\models\\ &c.HighWaterFlag = \text{'On'} \Rightarrow \circ\ c.PumpSwitch = \text{'On'}\\ &\qquad \{\textbf{Goal}\ \text{PumpSwitchOnWhenHighWaterDetected}\}\end{aligned}$$

where the predicates [|SwitchPumpOn|](c) and [|SwitchPumpOff|](c) are defined as in Definition 1:

$$\begin{aligned}&[\| SwitchPumpOn \|]\ (c) \Leftrightarrow\\ &\qquad c.PumpSwitch \neq \text{'On'} \land \circ\ c.PumpSwitch = \text{'On'},\\ &[\| SwitchPumpOff \|]\ (c) \Leftrightarrow\\ &\qquad c.PumpSwitch \neq \text{'Off'} \land \circ\ c.PumpSwitch = \text{'Off'}.\end{aligned}$$

The minimality condition is obtained by inverting the antecedent and the consequent in the above formula, and can be verified similarly.

4. OPERATIONALIZATION PATTERNS

Our concern now is to define inference rules based on this semantics that allow a complete, consistent and minimal operationalization to be derived from a goal specification in real-time linear temporal logic (RT-LTL). The operationalization takes the form of a set of operations specified by domain and required pre, post- and trigger conditions.

The derivation rules correspond to operationalization patterns that are defined for frequent goal specification patterns. An *operationalization pattern* is an abstract AND-operationalization link between a goal specification pattern in RT-LTL and a set of required pre-, trigger and postcondition specification patterns that operationalize the root correctly (in the sense of Definition 3).

The proof of correctness with respect to the above semantics of goal operationalization is done once and for all; when using such patterns the specifier is thus relieved from tedious proofs of completeness, consistency and minimality of operationalizations. More constructively, patterns guide specifiers in the process of deriving operational specifications from system goals.

A sample of patterns is first presented together with various examples of use in order to highlight the benefits of the approach; the process of building the pattern catalog is discussed next.

4.1 A sample of patterns

Figure 1 shows a first pattern for operationalizing goals of the form $C \Rightarrow \circ T$, where T is a meta-variable for a target state formula and C is a meta-variable for a formula on the current state (and possibly on past states as well).

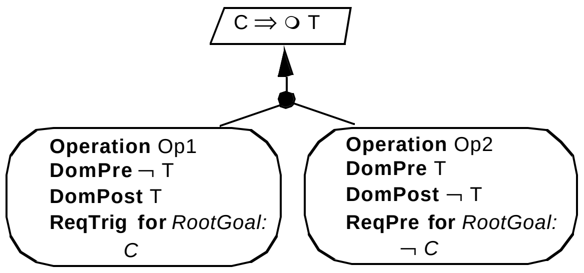


Figure 1 - The *Immediate Achieve* pattern

The operation model fragment in that pattern states that in order to operationalize a goal $C \Rightarrow \circ T$, two operations have to be introduced (with appropriate domain-specific renaming):

- an operation *Op1* with domain pre- and postcondition given by the pair $[\neg T, T]$ and with a required trigger condition stating that the operation *must* be applied when C holds provided $\neg T$ holds;
- an operation *Op2* with domain pre- and postcondition given by the pair $[T, \neg T]$ and with a required precondition stating that the operation *may* be applied only when C does not hold.

The correctness of this pattern is established by proving conditions 1-3 in Definition 3. For example, completeness is easily proved by verifying the following assertion with a temporal verifier such as STeP:

$$\begin{aligned}&(C \land \neg T) \Rightarrow \circ T\\ &\land\\ &(T \land \circ\ \neg T) \Rightarrow \neg C\\ \models&\\ &C \Rightarrow \circ T\end{aligned}$$

The operationalization of the goal Maintain[PumpSwitchOnWhenHighWaterDetected] through the operations SwitchPumpOn and SwitchPumpOff in Section 2.1.4 was derived by instantiation of the *Immediate Achieve* pattern.

Patterns can of course be instantiated to completely different situations. For example, consider a flight control system and the goal "reverse thrust enabled when wheels pulse on" mentioned in the Introduction section:

$$WheelsIsPulseOn \Rightarrow \circ\ ReverseThrustEnabled$$

This goal specification matches the root of the *Immediate Achieve* pattern; the derived instantiations are

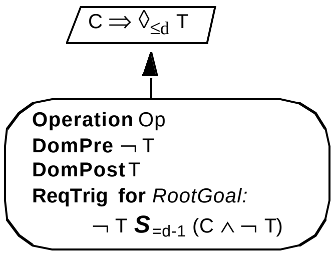
C : WheelsIsPulseOn　　T: ReverseThrustEnabled

The following operational specifications are thereby derived:

Operation EnableReverseThrust
DomPre ¬ ReverseThrustEnabled
DomPost ReverseThrustEnabled
ReqTrig for ReverseThrustEnabledWhenWheelsPulseOn:
WheelsPulseOn

Operation DisableReverseThrust
DomPre ReverseThrustEnabled
DomPost ¬ ReverseThrustEnabled
ReqPre for ReverseThrustEnabledWhenWheelsPulseOn:
¬ WheelsPulseOn

Figure 2 shows a frequently used pattern for operationalizing bounded *Achieve* goals.



$C \Rightarrow \Diamond_{\leq d} T$
Operation Op
DomPre ¬ T
DomPost T
ReqTrig for *RootGoal:*
$\neg T \, \boldsymbol{S}_{=d-1} \, (C \wedge \neg T)$

Figure 2 - The *Bounded Achieve* pattern

Applicability: The goal has the form $C \Rightarrow \Diamond_{\leq d} T$, where T is a target state condition to be reached from a condition C on the current (and possibly past) state(s).

Explanation: The operation model fragment in the *Bounded Achieve* pattern states that in order to operationalize a goal taking the form $C \Rightarrow \Diamond_{\leq d} T$, an operation *Op* has to be considered with domain pre- and postcondition given by the pair $[\neg T, T]$ and with a required trigger condition stating that the operation *must* be applied when T has remained false since C was true *d-1* time units ago without T being true.

Example of use: We come back to the mine pump case study and consider the goal Achieve[AlarmWhenCriticalMethaneLevel]. The formal specification of this goal is

∀ c: PumpController
c.MethaneMeasure ≥ 'Critical' $\Rightarrow \Diamond_{\leq d}$ c.Alarm = 'On'

The goal matches the root of the *Bounded Achieve* pattern; the operational specification derived after instantiation of the matching meta-variables is the following:

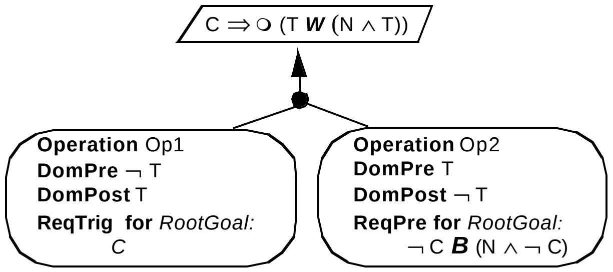
Operation RaiseAlarm
DomPre c.Alarm ≠ 'On' ; **DomPost** c.Alarm **=** 'On'
ReqTrig for AlarmForCriticalMethaneMeasure:
c.Alarm ≠ 'On' $\boldsymbol{S}_{=d-1}$ (c.MethaneMeasure ≥ 'Critical'
∧ c.Alarm ≠ 'On')

The derived trigger condition captures an obligation to perform the operation if the methane measure has been above critical level *d-1* time units ago with the alarm remaining off since then.

Figure 3 shows a frequently used pattern for operationalizing *Maintain/Avoid* goals.

Applicability: the goal has the general form $C \Rightarrow T \, \boldsymbol{W} \, N$, where T is a state formula to be maintained under some condition C on the current (and possibly past) state(s) unless some new condition N becomes true with T still being true; T is not instantly enforceable in every state C holds (for operational specification languages equipped with a synchrony hypothesis such as SCR [15] the latter condition does not hold and the root goal in Figure 3 simplifies to $C \Rightarrow T \, \boldsymbol{W} \, N$).

Explanation: The operation model fragment in the *"InBetween" Invariance* pattern states that in order to operationalize a goal taking the form $C \Rightarrow \circ \, (T \, \boldsymbol{W} \, (N \wedge T))$, two operations *Op1* and *Op2* have to be considered: the first one with domain pre- and postcondition given by the pair $[\neg T, T]$ and with a required trigger condition stating that the operation *must* be applied when C holds provided $\neg T$ holds; and the second one with domain pre- and postcondition given by the pair $[T, \neg T]$ and with a required precondition stating that the operation *may* be applied only if C has not been true *back to* (and including) *the last time* N was true.



$C \Rightarrow \circ \, (T \, \boldsymbol{W} \, (N \wedge T))$
Operation Op1
DomPre ¬ T
DomPost T
ReqTrig for *RootGoal:*
C
Operation Op2
DomPre T
DomPost ¬ T
ReqPre for *RootGoal:*
$\neg C \, \boldsymbol{B} \, (N \wedge \neg C)$

Figure 3 - The *"InBetween" Invariance* pattern

Example of use: Consider the following goal for a simplified light control system [3].

Goal Maintain [LightOnWhenRoomOccupied]
InformalDef *A room light must remain 'On' between the time a first person is entering the room and the time a last person is leaving it.*
FormalDef ∀ r: Room
r.FirstEntry ⇒ ❍ (r.Light = 'On' ***W*** (r.LastExit ∧ r.Light = 'On'))

The goal specification matches the root of the *"InBetween" Invariance* pattern; the operational specification derived after instantiation of the matching meta-variables is the following:

Operation TurnLightOn
Input r: Room;**Output** r: Room/Light
DomPre r.Light ≠ 'On'; **DomPost** r.Ligth = 'On'
ReqTrig for LightOnWhenRoomOccupied:
r.FirstEntry

Operation TurnLightOff
Input r: Room; **Output** r: Room/Light
DomPre r.Ligth ≠ 'Off'; **DomPost** r.Ligth = 'Off'
ReqPre for LightOnWhenRoomOccupied:
¬ r.FirstEntry ***B*** (r.LastExit ∧ r.Light = 'On')

This required precondition captures that the light may be turned off only if no first entry in the room has been detected back to the last time a last exit from the room was detected with the lights being on. Note that the derived precondition and the original goal as it is formulated do not require the light to be turned off as soon as a last exit occurs. Also note that the ❍-operator is introduced in the goal specification to ensure that the goal is *realizable* by the *LightController* agent [33]; the latter cannot both detect a first entry and switch the light on within the same state.

4.2 Benefits of operationalization patterns

We now argue that operationalization patterns are helpful for a variety of reasons. Although we have limited experience with operationalization patterns so far, our arguments are grounded on extensive experience we had in using patterns for the upstream phase of goal refinement [7] (over 15 industrial projects undertaken by a consulting company).

4.2.1 Abstraction from formal details

Checking the completeness, consistency and minimality of an operationalization is in practice a tedious, complex and error-prone. As mentioned before, each pattern is proved once before inclusion in the catalog. Patterns involving propositional qualitative temporal logic were proved correct using the SteP verification tool [36]

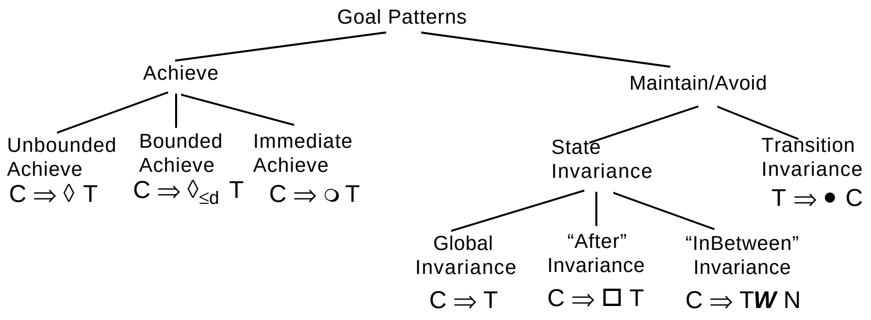

Goal Patterns
Achieve
Maintain/Avoid
Unbounded Achieve $C \Rightarrow \Diamond T$
Bounded Achieve $C \Rightarrow \Diamond_{\leq d} T$
Immediate Achieve $C \Rightarrow \circ T$
State Invariance
Transition Invariance $T \Rightarrow \bullet C$
Global Invariance $C \Rightarrow T$
"After" Invariance $C \Rightarrow \Box T$
"InBetween" Invariance $C \Rightarrow T \mathcal{W} N$

Figure 4 - A taxonomy of goal patterns

whereas the ones with real-time constructs were proved by hand. The user of a pattern then gets an instantiated, hidden correctness proof for free.

4.2.2 Completeness assurance

First-sketch goal operationalizations produced just by intuition tend to be incomplete. Matching an incomplete operationalization against a corresponding pattern allows the missing operational specifications to be pointed out.

In our running example, a first sketch operationalization of the goal Maintain[PumpSwitchOnWhenHighWaterDetected] might typically have produced the operation SwitchPumpOn together with its required trigger condition c.HighWaterFlag = 'On'. Matching this operationalization against the *Immediate Achieve* pattern would then have revealed a missing required precondition c.HighWaterFlag ≠ 'On' on the SwitchPumpOff operation.

Our experience with a similar use of patterns for completing goal refinements [7] suggests that this particular use of patterns may be extremely helpful in practice.

4.2.3 Guidance in writing operational specifications

As the various examples of pattern instantiation suggested, the patterns can be used constructively and incrementally to identify all operations relevant to a goal and derive the requirements on these operations that are needed to achieve the goal. This contrasts with traditional formal methods where a complete model must be provided before a posteriori analysis can take place.

4.2.4 Goal mining from operational specifications

Operationalization patterns can also be used *bottom-up* to elicit the goals underlying some available operational specification. Such pattern usage is important as initial formulations found in preliminary material provided to requirements engineers tend in practice to be very operational. Goal mining from operational specifications allows for various goal-level analysis, e.g., checking the operational formulation for completeness with respect to goals left implicit; identifying and resolving obstacles to goal achievement in order to produce more robust requirements [28, 29]; identifying and resolving conflicts at the goal level [26]; and exploring alternative system proposals [33].

To illustrate pattern-based goal mining from operational specifications, consider the following excerpt from the initial problem statement for a simple autopilot [4]: *"If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the alt_eng button, the altitude mode will not directly engage. Instead, the altitude engage mode will change to "armed" and the flight-path angle (FPA) select mode is engaged"*.

This informal formulation refers to operations such as EngageALTmode, ArmALTmode, and EngageFPAmode together with required conditions on their applications. For instance, the operation EngageFPAmode may be specified as follows:

Operation EngageFPAmode
  Input a: AutoPilot, altEng: ALTengagedEvent
  Output a: Autopilot/FPAmode
  DomPre a.FPAmode ≠ 'on'
  DomPost a.FPAmode = 'on'
  ReqTrig For <*unknown goal*>:
    Occurs (altEng) $\wedge$ ALTtarget - ALTactual > 1200

The *Immediate Achieve* pattern matches the domain pre-, post- and required trigger condition; the following root goal specification is obtained as rationale for the trigger condition:

Goal Achieve [FPAModeEngagedWhenHighTargetAltitutde]
FormalDef ∀ a: AutoPilot, altEng: ALTengagedEvent
    Occurs (altEng) $\wedge$ ALTtarget - ALTactual > 1200
    $\wedge$ a.FPAmode ≠ 'on'
     $\Rightarrow \circ$ a.FPAmode = 'on'

Higher-level goals can then be elicited further by asking WHY questions. The resulting goal graph will provide the rationale for the very operational problem statement provided; goal-level analysis can then be performed on the inferred goal structure.

The inductive goal inference procedure discussed in [27] and the bottom-up use of operationalization patterns here are different; the former starts from concrete scenarios of interaction between agents whereas the latter starts from operational specifications.

4.3 A catalog of operationalization patterns

We finally discuss how relevant operationalization patterns can be identified and organized for retrievability.

4.3.1 Identifying patterns

One way of identifying operationalization patterns is to abstract them from concrete examples of goal operationalizations. Unfortunately, there is no large body of specifications available from which such patterns could be inferred. The constructive elaboration of operational requirements from goal specifications is not widely adopted yet; previous derivations of operational requirements from goals were done in an ad hoc fashion with no fully precise semantics for operationalization.

Therefore we decided to explore the space of operationalization patterns on the basis of patterns for the goal to be operationalized. In order to get a rich set of goal patterns we extended and specialized the high-level Achieve/Maintain patterns of the KAOS language with specification patterns adapted from [9]. Some of those goal patterns are not operationalizable due to unrealizability problems such as the impossibility to evaluate a condition that refers to the future or to evaluate a condition and react upon it within the same state [32, 33]. For the goal patterns in this taxonomy that are operationalizable, we identified and proved corresponding operationalization patterns.

Figure 4 shows our current taxonomy of goal patterns. Each name in the hierarchy may have several pattern *variants*; for example, the pattern in Fig. 3 is a variant of the *"InBetween" Invariance* pattern in Fig. 4. The current catalog of operationalization patterns can be downloaded from [32].

4.3.2 Coverage of the catalog

By construction, *the coverage of the catalog of operationalization patterns is relative to the coverage of the taxonomy of goal patterns*. The effectiveness of our approach is based on the assumption that most properties that occur in practice can be specified using a small set of specification patterns. This assumption is partly supported by empirical evidence. Dwyer et al report that 92% of 555 examples of property specifications found in the literature matched one of their patterns [9]. In our more limited experience, we found that in every but very rare cases, the goals that we write match one of the general patterns in Figure 4. Our taxonomy is certainly not complete however, and it could be enriched with additional goal patterns that we haven't used in our specifications yet. Further extensions of the taxonomy of goal specification patterns will trigger the identification of further corresponding operationalization patterns in our catalog.

The *granularity* of the taxonomy of patterns is probably a more important issue than its coverage. Our taxonomy of goal specification patterns and the corresponding catalog of operationalization patterns are composed of propositional patterns mainly. Although goals to be operationalized do in general match one of the propositional patterns in Figure 4, a strict application of the corresponding operationalization pattern may produce operational specifications in which the pre-, trigger and post- conditions are too coarse-grained due to the propositional nature of the pattern involved. As we did for goal refinement patterns [7], we could enrich the catalog with first-order patterns. However, the number of variants at a finer-grained level might then become too large for useful coverage, that is, the hypothesis that most properties occurring in practice can be specified using a small set of patterns might no longer hold when we consider finer-grained, first-order patterns.

As we experienced it with goal refinement patterns, operationalization patterns may prove to be helpful even when they do not match perfectly. In such cases they produce a first-sketch specification to be adapted in a second phase. The adapted specification might then be formally "model-checked" with respect to the originating goal. The specification environment we are currently building interacts with tools such as Alcoa [18] and NuSMV for that purpose.

5. CONCLUSION

There is a growing consensus that complex software requires abstract models for analysis and validation in the early phases of software development. Formal methods most often take it for granted that such models are available; they do not consider the process of building the model, focussing on the resulting product. Usually the model is fairly operational and needs to be complete before analysis can take place. The analysis then amounts to some form of model debugging.

The paper described an effort to complement such analysis with techniques for building operational models from higher-level abstractions such as the system objectives requirements engineers have to reason about when they elicit requirements. We gave a formal semantics for this notion of operationalization and discussed some features that our language provides in order to support an incremental operationalization process and reasoning techniques that are applicable to partial models.

Our operationalization patterns amount to high-level inference rules for deriving pieces of operational specifications compositionally and constructively. A pattern is proved to produce a complete, consistent and minimal operationalization once and for all; it hides formal details and may be used in fairly different situations as the various examples in the paper aimed at suggesting. Patterns may be used for guiding the specification building process, for pointing out incomplete operationalizations and for mining goal structures from operational specifications.

A few derivation rules were already proposed in [6] for goal operationalization. These rules are limited in coverage and do not rely on the formal semantics of operations defined in this paper; they do not ensure the completeness, consistency and minimality of goal operationalizations.

Our operationalization patterns are domain-independent; in a sense they are orthogonal to domain-specific [12, 37] or task-specific [19] model patterns.

Provision of tool support is under way in the context of a wider ongoing project (FAUST, Formal Analysis Using Specification Tools). The pattern management tool in the environment we are building will provide facilities for retrieving matching patterns and instantiate them, create new patterns or new variants of existing ones, find near matches and adapt the instantiations produced. The tool is designed to be generic so that it can manage refinement [7, 33], obstruction [28], conflict [26] and operationalization patterns. The tool is also expected to cooperate with the FAUST animator which generates parallel finite state machines from our operational specifications to animate them; this will allow specification derivation and adequacy checking to proceed hand-in-hand.

Compared with our extensive experience with goal refinement patterns in industrial projects, we have limited experience to date with our operationalization patterns - it mostly consists in handling a wide variety of case studies from the literature on specification, including non-trivial ones. This preliminary experience so far confirms what we have observed with goal refinement patterns, namely, (a) the patterns are helpful even when they are used *informally* to guide first specification drafts, and (b) when used *formally* the patterns provide helpful guidance even when the formal instantiations produced need to be subsequently adapted to the specifics of the problem at hand.

6. ACKNOWLEDGEMENT

The work reported herein was partially supported by the "Communauté Française de Belgique" (FRISCO project, Actions de Recherche Concertées Nr. 95/00-187) and the Belgian "Fond National de la Recherche Scientifique" (FNRS). We are grateful to the FSE reviewers for their help in improving some aspects of the presentation.

7. REFERENCES

[1] R.J. Back, "A calculus of refinements for program derivation", *Acta Informatica*, Vol. 25, 1988, 593-624.

[2] A. Borgida, J. Mylopoulos and R. Reiter, "And Nothing Else Changes: The Frame Problem in Procedure Specifications", *Proc. ICSE'93 - 15th International Conference on Software Engineering,* Baltimore, May 1993.

[3] E. Borger and R. Gotzhein, "The Light Control Case Study: Problem Description", *Journal of Universal Computer Science,* Vol. 6 No. 7, 2000.

[4] R. W. Butler. *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot*. NASA Technical Report 110255. NASA Langley Research Center, May 1996.

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

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

[7] R. Darimont and A. van Lamsweerde, “Formal Refinement Patterns for Goal-Driven Requirements Elaboration”, *Proc. FSE’4 - Fourth ACM Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[8] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[9] M.B. Dwyer, G. S. Avrunin and J.C. Corbett, “Patterns in Property Specifications for Finite-State Verification”, *Proc. ICSE’99 - 21st Intl. Conference on Software Engineering*, Los Angeles, May 1999.

[10] J. Fiadeiro and T. Maibaum, “Temporal theories as modularisation units for concurrent system specification”, *Formal Aspects of Computing*, Vol. 4 No.3, 1992, pp. 239--272.

[11] S. Fickas and R. Helm, “Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, pp. 470-482.

[12] M. Fowler, *Analysis Patterns - Reusable Object Models.* Addison-Wesley, 1997.

[13] J. V. Guttag, J. J. Horning, K.D. Jones, S.J. Garland, A. Modet and J.M. Wing. *Larch: Languages and tools for formal specification*. Springer-Verlag, 1993.

[14] D. Harel and A. Naamad, “The STATEMATE Semantics of Statecharts”, *ACM Trans. Software Eng. and Methodology*, Vol. 5 No.4, October 1996, pp. 293-333.

[15] C. Heitmeyer, R. Jeffords and B. Labaw, “Automated Consistency Checking of Requirements Specifications”, *ACM Trans. on Software Engineering and Methodology*, Vol. 5 No. 3, July 1996, pp. 231-261.

[16] M. Jackson, *Software Requirements & Specifications*. ACM Press, Addison-Wesley, 1995.

[17] D. Jackson, “Structuring Z specifications with views”, *ACM Transactions on Software Engineering and Methodology*, Vol. 4 No. 4, October 1995, 365-389.

[18] D. Jackson, “Automating First-Order Relational Logic”, Proc. FSE’2000 - ACM SIGSOFT Conf. on Foundations of Software Engineering,. San Diego, November 2000.

[19] M. Jackson, *Problem Frames - Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[20] C.B. Jones. *Systematic Software Development Using VDM.* 2nd ed., Prentice Hall, 1990.

[21] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996.

[22] R. Koymans, *Specifying message passing and time-critical systems with temporal logic*, LNCS 651, Springer-Verlag, 1992.

[23] J. Kramer, J. Magee, M. Sloman, et al, “CONIC: an Integrated Approach to Distributed Computer Control Systems”, IEE Proceedings, Part E 130, 1, January 1983, pp. 1-10.

[24] L. Lamport, “The Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems*, Vol. 16 No. 3, May 1994, 872-923.

[25] A. van Lamsweerde, R. Darimont and P. Massonet, “Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned”, *Proc. RE’95 - 2nd Int. Symp. on Requirements Engineering*, York, IEEE, 1995.

[26] A. van Lamsweerde, R. Darimont and E. Letier, “Managing Conflicts in Goal-driven Requirements Engineering”, *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, Nov. 1998, pp.908-926.

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

[28] 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.

[29] A. van Lamsweerde, “Requirements Engineering in the Year 00: A Research Perspective”. Invited Keynote Paper, *Proc. ICSE’2000: 22nd International Conference on Software Engineering*, ACM Press, 2000, pp. 5-19.

[30] A. van Lamsweerde, “Formal Specification: a Roadmap”. In *The Future of Software Engineering*, A. Finkelstein (ed.), ACM Press, 2000, pp. 147-160.

[31] A. van Lamsweerde, “Goal-Oriented Requirements Engineering: A Guided Tour”. Invited minitutorial, *Proc. RE’01 - International Joint Conference on Requirements Engineering*, Toronto, IEEE, August 2001, pp.249-263.

[32] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Ph. D. Thesis, University of Louvain, May 2001; *http://www.info.ucl.ac.be/people/eletier/thesis.html.*

[33] 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 CS Press, May 2002.

[34] T. Maibaum, “Temporal Reasoning over Deontic Specifications,” in J.Ch. Meyer and R.J. Wieringa (Eds.), *Deontic Logic in Computer Science - Normative System Specification*, Wiley, 1993, pp. 141-202.

[35] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[36] Z. Manna and the STep Group, “STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems”, *Proc. CAV’96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, pp. 415-418.

[37] P. Massonet and A. van Lamsweerde, “Analogical Reuse of Requirements Frameworks”, Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.

[38] C.C. Morgan, *Programming from Specifications*, 2nd Edition, Prentice-Hall, 1994.

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

[40] B. Potter, J. Sinclair and D. Till. *An introduction to formal specification and Z.* Prentice Hall, 1991.

[41] P. Zave and M. Jackson, “Four dark corners of requirements engineering”, *ACM Trans. Software Eng. and Methodology*, Vol. 6 No. 1, January 1997, pp. 1-30.