A Functional Solution for Goal-oriented Policy Refinement

Javier Rubio-Loyola[1], Joan Serrat[1], Marinos Charalambides[2], Paris Flegkas[2], George Pavlou[2]
[1]*Universitat Politècnica de Catalunya,* [2]*University of Surrey*
[1]*{jrloyola, serrat}@tsc.upc.edu,* [2]*{M.Charalambides, P.Flegkas, G.Pavlou}@eim.surrey.ac.uk*

Abstract

Policy refinement is a key but still unsolved area of policy based management. Goal oriented requirements engineering methodologies have been suggested as a prominent alternative to address policy refinement. Practical approaches that capture the administrative requirements and enable systematic policy refinement are still missing although such integrated solutions are rather convenient to make policy-based management systems really useful. In this paper we present a functional solution for goal oriented policy refinement grounded in linear temporal logic and reactive systems analysis techniques. We describe the technical foundations and demonstrate how these were used to develop an integrated solution for policy refinement, focusing on the details of the implemented prototype. Our policy analysis techniques that enable systematic policy refinement are demonstrated through a scenario applied to the domain of QoS Management for Differentiated Services (DiffServ) networks.

1. Introduction

Policy refinement is meant to derive lower-level policies from higher level ones so these more specific policies are better suited for use in specific environments. The goal-oriented policy refinement framework proposed in [1] is a promising methodology to address the policy refinement paradigm. This methodology has opened a window for research into different analysis techniques in favor of policy refinement. The authors propose to elaborate goal graphs by decomposing high-level goals into refined ones from which additional processes may be applied to abstract the necessary policy information aimed at fulfilling the refined goals.

Current solutions in the area of requirements engineering provide support for both, formalizing large-scale requirements documentation, and goal elaboration support [2]. The goal elaboration method proposed in [1] is the KAOS methodology [3]. While KAOS provides support to document and elaborate goal graphs, it does not provide support to relate managed objects' behavior to goal fulfillment finding. Moreover, in order to systematically refine policies from goal-oriented high-level directives, these limitations must be addressed efficiently.

In our previous work [4], we demonstrated how linear temporal logic and model checking verification could be used as the analysis techniques for the goal-based policy refinement methodology proposed in [1]. We described a generic procedure to obtain system trace executions aimed at fulfilling lower-level goals. From system trace executions, the managed entities, conditions and actions (i.e. policy fields) necessary to fulfill administrative guidelines may be abstracted. In this paper we move one step ahead towards the materialization of the policy refinement paradigm following this approach.

This paper discusses the critical nature of providing a functional solution for generating policies, from abstract requirements, reflecting system behavior. We present a functional solution that provides support to document system behaviour, to elaborate goal-graphs, and to refine policies in a goal-oriented fashion. This paper provides several novel contributions with respect to our previous work [4]. We describe the nature of the novel technical foundations of our solution and demonstrate how these were used in our implementation. We establish the formal procedures and the necessary mechanisms to realize a goal-oriented policy refinement solution. In addition, we present the execution of a refinement scenario applied to the domain of DiffServ QoS Management.

This work was carried out in the context of the FP6 IST EMANICS Network of Excellence (IST-026854)

Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)
0-7695-2598-9/06 $20.00 © 2006 IEEE

The rest of the paper is organized as follows: Section 2 describes the technical foundations. Section 3 details the goal-oriented framework in which our solution relies. Section 4 describes our prototype implementation. Section 5 describes a scenario and Section 6 its execution. Relevant issues and future work are discussed in Section 7. Related work and Conclusions are provided in Sections 8 and 9.

2. Technical Foundations

In this section we describe the technical foundations, highlighting the relevant issues with respect to the processes incorporated in our solution.

2.1. KAOS goal elaboration methodology

KAOS [3] is a formal approach for elaborating goals grounded in Temporal Logic [5]. It represents goals as temporal logic rules and uses refinement patterns to decompose high-level goals into sets of sub-goals that logically entail the original ones. KAOS considers goal management based on the Temporal Prescription (TP) of goals, e.g. *Achieve, Cease, Maintain* and *Avoid*. While *Achieve* and *Cease* goals obey to system behaviors that require some target property to be *eventually* satisfied or denied respectively, *Maintain* and *Avoid* goals restrict behaviors in that they require some target property to be *permanently* satisfied or denied respectively. In addition to managing goals with respect to their TPs, KAOS provides support to guide their refinements by re-using domain-independent refinement patterns (RP) [3]. These patterns have been included in a set of libraries that have been previously proved to be correct. The libraries are grouped by the Temporal Prescription of the high-level goals.

Table 1 presents two KAOS Refinement Patterns (RPs) that represent different ways to decompose the high-level *Achieve* goal P→◊R, into their respective sub goals. In addition to the classical logical operators, in this paper we use the classical temporal operators; ◊ eventually in the future, □ always in the future, *U* always in the future until and *W* always in the future unless, ○ next state. The high-level goal P→◊R must be interpreted as *"If P then eventually R in the future"*.

Table 1. Two refinement patterns for P→◊R *Achieve* goal

RP	Subgoals
RP_1	$P \rightarrow \Diamond Q$ $\quad Q \rightarrow \Diamond R$
RP_2	$P \wedge P1 \rightarrow \Diamond R1 \quad P \wedge P2 \rightarrow \Diamond R2 \quad \Box(P1 \vee P2)$ $\quad R1 \vee R2 \rightarrow R$

The refinement pattern RP_1 in Table 1 for instance, defines a *milestone-driven* tactic where an intermediate state satisfying *Q* must first be reached, from which a final state satisfying *R* must be reached. The refinement pattern RP_2 proposes decomposition by cases meaning that either satisfying *R1* or *R2* suffices to satisfy *R*.

Relevant to our analysis is the temporal prescription of goals and their relationship. For example, Figure 1 shows the decomposition of an *Achieve* goal G_1. The left part of Fig. 1 shows the syntactical representation of the decomposition while the right part shows the formal temporal representation (see Formal Expression pointers). The high-level goal G_1 is refined into sub goals G_{11} and G_{12} with the refinement pattern RP_1 (see Refinement Pattern instantiation in Fig.1). For this refinement, KAOS methodology considers the temporal prescription of the high-level goal G_1 (TP_1) and that of its refinements (TP_{11} and TP_{12}). For instance, the temporal prescription of G_1, formally expressed as P→◊R, identifies that *"if property P holds at some point, then property R would eventually hold in the future"*. Similar expressions may be expressed for the temporal prescriptions TP_{11} and TP_{12} of G_{11} and G_{12} respectively.

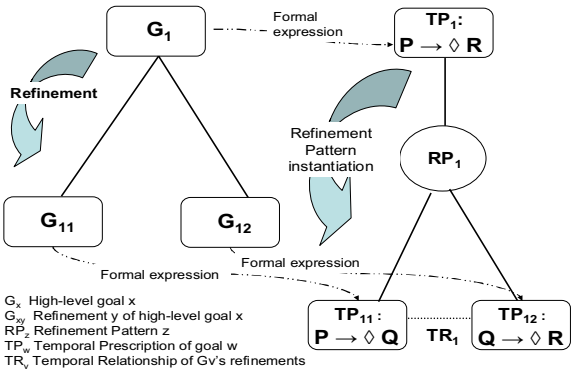

Figure 1: KAOS temporal considerations amongst goals

Central to our study is the intrinsic Temporal Relationship (TR_1) between the sub goals G_{11} and G_{12}. The intrinsic TR_1 states that *"if property P holds at some point, an intermediate property Q must hold in advance from which a final state satisfying property R must eventually be reached"*. Hence, satisfying G_{11} and G_{12}, committing to TR_1, suffices to satisfy G_1.

2.2. Design patterns for finite-state verification

In terms of finite-state verification, a specification pattern is a generalized description of a commonly occurring requirement on state/event sequences in a system execution [6]. Pivotal in our analysis is the representation of verification properties to characterize goal fulfillment. While the refinement patterns used to elaborate goals describe the "requirements" for a system (e.g. both G_{11} and G_{12} must be fulfilled so that G_1 is fulfilled in Fig. 1), the patterns described in this section deal with the translation of "particular" aspects

of such requirements (e.g. G_{11} is achieved *"before/after"* G_{12}) into formal specifications suitable for finite state verification tools (e.g., model checking).

In the upper part of Figure 2 we show the classification of the design patterns for finite-state verification in two major groups: Occurrence and Order.

On the one hand, the Occurrence patterns are used to represent the following situations: states/events to occur or not to occur (Existence and Absence pattern), states/events to occur throughout a scope (Universality pattern), or state/event occurs *k* times within a scope (Bounded Existence pattern).

On the other hand, Order patters are applied to represent constraints in the order of states/events (Response pattern), or to specify that a given state/event *P* to be always preceded by a state/event *Q* within a scope (e.g. the Precedence pattern).

All these patterns for finite-state verification have a scope – the extent of system execution over which the pattern must hold. The lower part of Fig. 2 shows a classification of five scope types: Global (the entire system execution), Before (the execution up to a given state/event), After (the execution after a given state/event), Between (any part of the execution from one given state/event to another state/event) and After-until (like between but the designated part of the execution continues even if the second state/event does not occur).

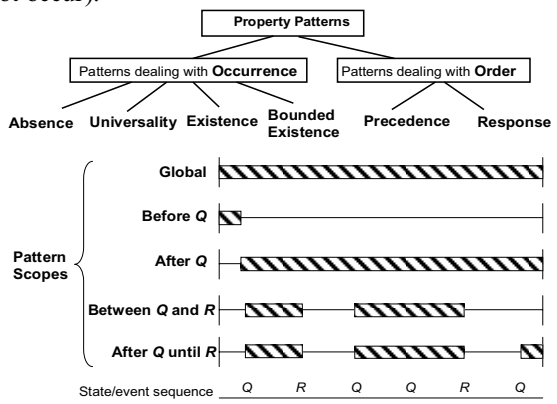

Figure 2: Property pattern mappings and pattern scopes

Broadly speaking, we can say that any reactive system requirement can be mapped to their respective pattern/scope. Consider the requirements R_1 and R_2:

R_1 - "Goal G_{12} is fulfilled after Goal G_{11}"
R_2 - "Goal G_{12} is not fulfilled after Goal G_{11}"
(R_2 is the opposite requirement of R_1)

The requirement R_1 corresponds to an Existence-based property pattern since the initial condition demands such condition to occur (i.e. *Goal G_{12} is fulfilled*). Additionally, R_1 clearly prescribes the After scope (i.e. the initial condition must hold *after* G_{11}). The requirement R_2 instead corresponds to an *Absence*-based pattern since the initial condition demands such condition not to occur (i.e. *Goal G_{12} is not fulfilled*). R_2 also prescribes the After scope.

The requirements for reactive systems and their representation into different temporal logics have been the subject of research for some time. It is possible to classify them in databases [6] as to find the logic formula representation of any requirement systematically. Consider the database shown in Table 2 including two entries of patterns/scope and their Linear Temporal Logic (LTL) representation.

Table 2. Two property pattern/scope and LTL representation

Id	Pattern	Scope	LTL representation
PP1	Existence	After	$\Box(\neg$ Antecedent$) \mid \Diamond$(Antecedent$\&\Diamond$Consequence$))$
PP2	Absence	After	$\Box($ Consequence $\rightarrow \Box(\neg$ Antecedent$))$

Instantiating PP1 and PP2 of Table 2 with the information of our requirements R_1 and R_2, we may find their LTL formula representation. Considering G_{11} and G_{12} as the Antecedent and Consequence conditions, the formula "$\Box(\neg G_{11}) \mid \Diamond(G_{11}\&\Diamond G_{12}))$" may represent the requirement "*G_{12} is fulfilled after G_{11}*" and the formula "$\Box(G_{12} \rightarrow \Box(\neg G_{11}))$" may represent the requirement "*G_{12} is **not** be fulfilled after G_{11}*".

Following this approach we may use combinations of patterns/scopes to formulate requirements that specify "particular" aspects of system executions. Moreover, this approach enables us to represent those requirements into formal specifications suitable for use with automated verification tools like model checking.

2.3. Introducing Translation Primitives

In our previous work [4], we demonstrated that finite-state verification properties could be used to characterize goal fulfillment. We also demonstrated that, by using verification properties, automatic tools may provide system trace executions that would make goals be fulfilled. Nevertheless, we are still missing the appropriate mechanisms to abstract policies from system trace executions in a systematic manner. To tackle this, we introduce the concept of Translation Primitives.

Translation Primitives are used to abstract policies from system trace executions. These primitives take advantage from the fact that the system trace executions indicate the pre- and post-conditions, and the actions taken by the involved managed objects.

The first step towards the application of the Translation Primitives is the identification of *transition plans*. A transition plan is a sub-section of a system trace execution consisting of the following elements:

- A pre-condition in a managed entity *S* (PS_i)
- A state transition $T_{Si,Si+1}$ in the managed entity *S*
- A state transition $T_{Qi,Qi+1}$ in the managed entity *Q* as a result of transition $T_{Si,Si+1}$

The *transition plan* TP demonstrated in the left part of Figure 3 is represented as $TP=[PS_{i1}, T_{Si,Si+1} => PQ_{i1}, T_{Qi,Qi+1}]$. Given that S and Q are two different managed objects, TP prescribes that on the occurrence of PS_{i1} in the managed object S, preceding the transition $T_{Si,Si+1}$, the managed object Q must enforce the transition $T_{Qi,Qi+1}$. Considering that the transition $T_{Qi,Qi+1}$ is policy-controlled (i.e. policy-enforceable), the Transition Primitives shown in the right part of Figure 3 enable us to encode the above information into Obligation policies [7] in a systematic manner.

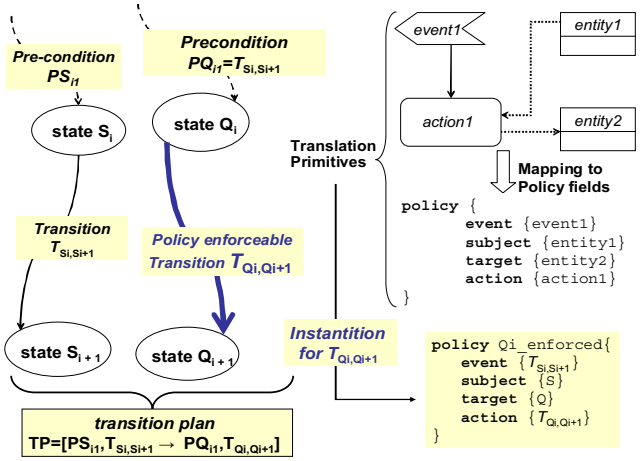

Figure 3: Transition plan and translation primitives

In the Translation Primitives, managed entities are mapped to subjects and targets. Transitions correspond to policy actions that can be anything between a single function call or entire scripts executed on a target. We consider an event to have happened, when an event notification has been received (e.g. a signal expressing an action, a change of an attribute i.e. a transition, etc.), and the conditions (if any) yield true. Here, we consider an unconditional execution of an action in response to an event. In the bottom of the right part of Figure 3, the Translation Primitives are instantiated for the policy enforcement corresponding to the enforceable transition $T_{Qi,Qi+1}$. The entities responsible for the triggering event, the target of the action and the triggering event are also specified.

The approach of abstracting policy fields from system executions is similar to the *translation patterns* presented in [8]. The authors generate policies automatically from process specifications. Here, we specialise these patterns to system trace executions.

3. Goal-oriented Policy Refinement Framework

Our solution follows the goal-based approach to policy refinement initially proposed in [1], making use of linear temporal logic and model checking verification as policy analysis techniques [4]. In this section we describe the overall policy refinement framework of our solution (Figure 4) based on two functionalities: Goal Management tasks and Policy Refinement mechanisms.

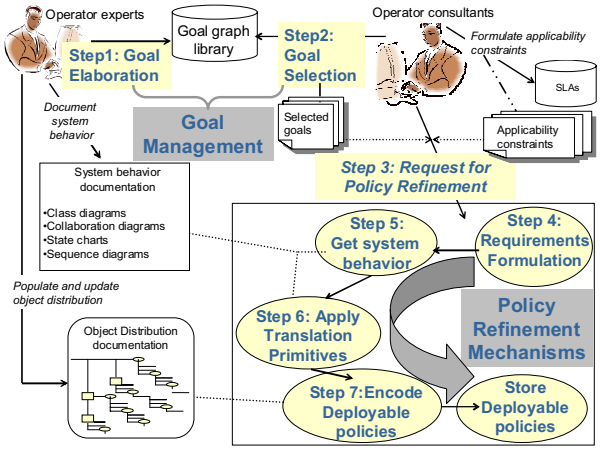

Figure 4: Overall policy refinement framework

3.1 Goal management tasks

The goal management tasks are the ***Goal Elaboration*** and ***Goal Selection*** (see Steps 1 and 2 in Fig. 4). These tasks are operator-directed. This is by no means a drawback of the framework considering that policy refinement is in general an off-line process. In order to better understand this section, the reader may have in mind the KAOS goal elaboration process introduced in Section 2.1.

During the ***Goal Elaboration*** process (see step 1 in Fig.4), the operator expert documents and classifies KAOS goal-graph structures and stores them into a goal-graph library. The expert makes use of refinement patterns (like RP_1 in Fig 1) to decompose goals into sub-goals that logically entail the original ones.

For example in Figure 1 we showed a goal graph with a single hierarchy (G_1 is refined into G_{11} and G_{12}), nevertheless, typical/practical goal graphs are multi-hierarchy structured (as we will see later). Since the refinement patterns have been proved to be correct, structures following the KAOS methodology are considered logically consistent.

In the ***Goal Selection*** (see step 2 in Fig.4) the operator consultant chooses from the goal-graph library, the goal strategies that better commit with his directives.

The gap between Goal Management and Policy Refinement Mechanisms is filled with the ***Request for Policy Refinement*** (Step 3 in Fig 4). It is activated by the consultant after goal selection.

3.2 Policy refinement mechanisms

The policy refinement mechanisms are the ***Requirements Formulation, Get System Behavior, Apply Translation Primitives*** and ***Encode/store Deployable Policies*** (see Steps 4~7 in Fig. 4). All these are carried out in an automatic manner after a ***Request for Policy Refinement*** is submitted. To better understand this section, in addition to the temporal foundations of KAOS (Section 2.1), the reader may have in mind the patterns for finite-state verification and the concept of Translation Primitives introduced in in Sections 2.2 and 2.3 respectively.

The ***Requirements Formulation*** mechanism produces the LTL formulae that characterize goal fulfillment. For this it initially abstracts the requirements of the goal selection, and then applies patterns for finite-state verification to produce LTL formulae characterizing these requirements.

In order to abstract the requirements, this mechanism must consider the temporal prescription, the refinement patterns and the temporal relationships amongst the goal selection. For example, having the consultant chosen to satisfy the goal G_1 shown in Figure 1, this mechanism may abstract the Temporal Relationship requirement: `TR_G1 = "G12 must be achieved after G11"`.

For this requirement, this mechanism may apply the finite-state verification pattern PP1 (see Table 2) to abstract the LTL representation. The application of PP2 may represent the opposite to `TR_G1`. Hence, instantiating PP1 and PP2 by plugging in the information relevant to `TR_G1`, this process may abstract the following LTL formulae f_1 and f_1'.

PP1[TR_G1] = $\Box(\neg G_{11}) \mid \Diamond(G_{11} \& \Diamond G_{12})$ …….. f_1
PP2[TR_G1] = $\Box(G_{12} \rightarrow \Box(\neg G_{11}))$ …….. f_1'

f_1 represents the requirement "*G_{12} must be achieved after G_{11}*" and f_1' the requirement "*G_{12} must **not** be achieved after G_{11}*".

The ***Get System Behavior*** mechanism coordinates the system trace executions acquisition. For this, we rely on the automated support provided by the SPIN model checker [9], specifically in its ability to produce system trace executions reports. With this regard, in our previous work [4], we demonstrated that by providing opposite LTL requirements, model checking engines would generate system trace executions committing to the original LTL requirement. Given that the Requirements Formulation mechanism (earlier explained) produces LTL requirements formulae characterizing goal fulfillment (e.g. f_1), this mechanism should make use of the opposite characterization of goal fulfillment (e.g. f_1') as to produce system trace executions fulfilling G_{11} and G_{12}, and consequently the higher-level goal G_1.

The ***Apply Translation Primitives*** mechanism (step 6 in Fig. 4) abstracts the policy fields from the system trace executions produced by the *Get System Behavior* mechanism. Typical reports of system trace executions may include more than one *transition plan*. Hence, this mechanism should first abstract all of these transition plans and then apply Transition Primitives to them.

Finally, the ***Encode Deployable Policies*** step takes as input the information described above and the object distribution in order to encode deployable policies. In this paper we have considered the Ponder specification language [7].

4. Goal-oriented Policy Refinement Implementation

This section describes our overall implementation and is divided in two sub-sections: Goal Management and Policy Refinement Mechanisms components. Our functional implementation is demonstrated in Figures 5a and 5b. They show the relevant components interactions and the class diagram of our solution. This implementation prototype logically entails the framework described in the last section.

4.1 Goal management components

The Goal Management components in our implementation are the Objectiver package and the Goal Manager. These two components materialize the Goal Management tasks described in Section 3.1.

The **Objectiver package** integrates Objectiver [2], a Requirements Engineering tool that successfully manages large-scale goals, supporting the KAOS methodology. In our solution, Objectiver enables the operator expert to elaborate and document goal graphs. In this sense, it is the support for the *Goal Elaboration* step of the refinement process (`elaborateGoals` interaction in Fig. 5a). Objectiver provides visual support to manage the goals and we use this capability not only to elaborate goals but also to guide the operator consultant throughout the *Goal Selection* step of the policy refinement process (`selectGoals` interaction in Fig 5a). Relevant to our implementation

are the Objectiver APIs that have enabled us to use it as a server within our framework.

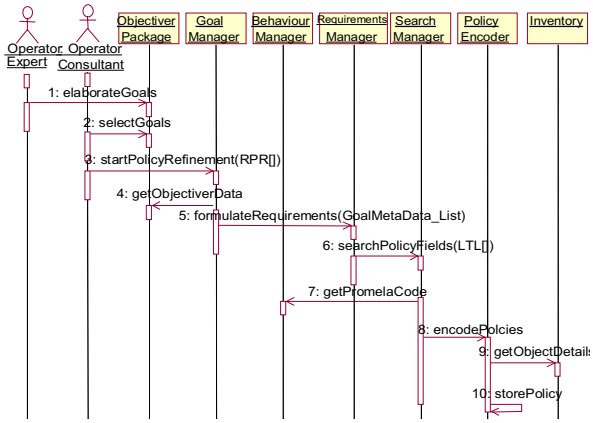

Figure 5a: Relevant interactions of our solution

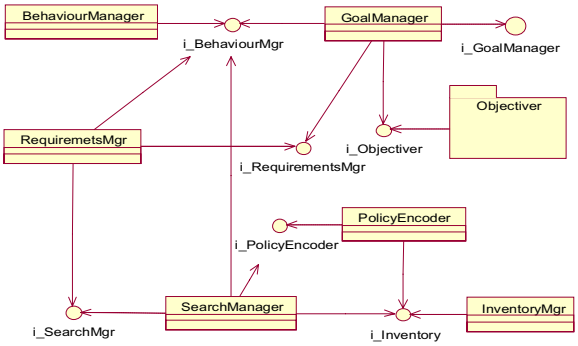

Figure 5b: Class diagram of our solution

Goal Manager: This component handles with the Requests for Policy Refinement (RPRs). When a RPR is submitted, the Goal Manager gets from the Objectiver package, the goal entities (`Goal`) that are influenced by the goal selection. For each `Goal` entity, this component deduces the following information:

- Temporal prescription of the goal, eg. Achieve, etc
- Refined goal sons (not for lowest-level goals)
- Refinement pattern used for the lower level goals (not for lowest-level goals)

The Goal Manager implements Objectiver APIs to allow iterative queries to it. In addition, our current implementation has been designed to process many RPRs for large-scale refinements.

4.2 Policy refinement mechanisms components

Here, we describe the implementation of the Policy Refinement Mechanisms described earlier. These are implemented by the Requirements Manager, Search Manager and Policy Encoder. Additional support components, are described as well.

Requirements Manager: This component implements the Requirements Formulation mechanism. It processes the goal-related information abstracted by the Goal Manager and produces the LTL formulae that characterize goal fulfillment. It internally implements a database of patterns for finite-state verification. This approach allows producing LTL formulae from specific requirements systematically. The overall procedure followed ***for every RPR*** is as follows:

Input: `GoalMetaData_List` object $=[Goal_1,..,Goal_u]$

Procedure:

- Find the highest-level goal "$\mathbf{Goal_{HG}}$" from $[Goal_1,...,Goal_u]$
- Find the lowest-level goals **`LLGoal`** that make $Goal_{HG}$ be fulfilled: $\mathbf{LLGoal} = [Goal_{L1},...,Goal_{Lv}]$
- Find a set of temporal relationships **`TR`** amongst the lowest-level goals: $\mathbf{TR}[LLGoal] = [TR_1,...,TR_w]$
- Formalise a set of requirements **`R`** (i.e. temporal requirements) with the temporal relationships $TR[LLGoal]$ and Temporal Prescription (TP) of the lowest-level goals $LLGoal[]$ $\mathbf{R}[TR[LLGoal],TP[LLGoal[]]=[R_1,...,R_x]$
- Deduce a set of property patterns **`PP`**: for each requirement, identify a pattern/scope property pattern that matches the requirement (the amount of PP-property patterns does not necessarily have to match with the amount of requirements) $\mathbf{PP}[R_1,...,R_x]=[PP_{R1},...,PP_{Ry}]$
- For the set of property patterns **`PP`**, identify the opposite pattern **`PP'`** $\mathbf{PP'}[PP]=[PP'_{R1},...,PP'_{Ry}]$
- Instantiate each opposite requirement with the requirement values and get a set of **`LTL`** formulae: $\mathbf{LTL}[PP'] = [ltl_1,...,ltl_y]$
- Merge all `ltl` formulae into a single LTL formula making use of logic connectors

Output: A LTL formulae characterising goal fulfilment.

Search Manager: This component implements the *Get system behavior* and *Apply Translation Primitives* mechanisms. It implements a SPIN model checker to acquire the system trace executions from the behavior specification documentation. After this, the Search Manager applies Translation Primitives to abstract the different policy fields systematically. The overall procedure followed ***for every RPR*** is as follows:

Input: A LTL formula characterizing goal fulfillment.

Procedure:

- Produce a system trace execution ***K*** from the LTL formula and the PROMELA code provided by the Behavior Manager (described later).
- Identify a set of decision-based (i.e. enforceable) transitions $\mathbf{T}[K]=[T_1,...,T_u]$.
- For `(i = 1 to i=u ):`

Find the precondition event of T_i = $event_{Ti}$
Find the managed object **MO** issuing $event_{Ti}$
MO($event_{Ti}$) = $subject_{Ti}$
Find the managed object **MO** executing the enforceable transition T_i
MO(T_i) = $target_{Ti}$.
Find the action **A** that represents the enforceable transition T_i
A(T_i) = $action_{Ti}$
Create a recipient PF_{Ti} for policy fields
PF_{Ti} = [$event_{Ti}$, $subject_{Ti}$, $target_{Ti}$, $action_{Ti}$]

Output: Instances of policy fields **PF**=[PF_{Ti} ,…, PF_{Tu}]

The **Policy Editor** implements the *Encode Deployable Policies* process of the overall policy refinement process. It follows the syntax of the Ponder specification language [7] and includes in its internal architecture, a Ponder Policy Editor and compiler (slightly modified to automate the compilation process).

Behavior Manager: This component is a support component that manages the documentation of the system behavior. Its functions are to provide information related to the system behavior specification, translation of behavior specification from UML models into PROMELA (input language of SPIN). The Behavior Manager implements the libraries provided by HUGO [10]. HUGO is a UML model translator that allows translating models with active classes, state machines, collaborations and interactions, into code for SPIN and other off-the-shelf tools.

The **Inventory Manager** provides support for the *Object Distribution documentation* supporting activity of the policy refinement process. The actual managed objects should be classified following a domain-dependent logical representation. A well-defined structure of this information is necessary to automate the policy refinement mechanisms. The Inventory component implements a database for the administrator expert to document this information and make it available for other components of our implementation, e.g. the Policy Encoder.

5. Application Scenario

5.1 Application domain

Here we present a refinement that relies on the framework developed in the EU IST TEQUILA project [11]. TEQUILA provides a policy-based functional architecture for supporting QoS in IP DiffServ networks. Due to space limitations, we limit ourselves to the resource management aspects, realized through the Dynamic Resource Management (DRsM) and the Network Dimensioning (ND) modules.

ND is a centralized off-line component responsible for mapping traffic requirements to physical network resources, and for setting provisioning directives to accommodate the predicted demand. ND directives are calculated based on estimations and are treated as "nominal" values. The dynamic resource management functions are deployed to the DRsM components.

DRsM has distributed functionality, with an instance operating in every router. It optimizes network performance in terms of resource utilization, while meeting at the same time, QoS traffic constraints. DRsM opts for dynamic functions that manage network resources (DiffServ Per-Hop-Behaviors-PHBs) following the guidelines provided by ND. ND defines minimum (ND_{min}) and maximum (ND_{max}) allocation values per-PHB. These values define the dynamic range (dynamic range=ND_{max}-ND_{min}) within which the DRsM calculates the actual allocation. While policies extend the hard-wired logic for ND and provide the administrator with strategies in performing network dimensioning, a policy-based approach for DRsM can allow for a fully flexible manner in allocating network resources. DRsM policies provide the flexibility to dynamically introduce logic and directives for tracking the utilization of a PHB and to ensure that the bandwidth allocated to the PHB is in accordance with the required one which is determined from the observed utilization. DRsM comprises two components; Monitoring and DRsM Main component. While the former is responsible for monitoring relevant links, issuing alarms upon upper or lower threshold crossings and calculating new thresholds, the DRsM Main component is responsible for the calculation of the *required* bandwidth (BW) and the handling of excess/over-provisioned BW, due to threshold crossing alarms.

5.2 Scenario description

The consultant has envisaged necessary to optimise resource allocation for two critical links (LC1, LC2). For these links he wants to set up specific conditions on when to trigger upper-threshold alerts and how new BW allocations are to be calculated as a consequence of upper-threshold crossings. For EF traffic, he wants to set up the following directives:

1. If the current load exceeds an upper threshold, the DRsM must increase the BW allocation and the thresholds (upper and lower) 20% of the dynamic range if the current load is below 60% of the dynamic range.

2. Contrary, if the current load is above 60% of the dynamic range, the increments described above must be in rates of the 10% of the dynamic range.

3. Additionally he wants the BW increased to this PHB to be deducted from the one allocated to AF traffic and that spare capacity be split equally amongst PHBs.

6. Scenario Execution

6.1 Scenario execution preconditions

Besides the population of the object distribution, the scenario preconditions are the documentations for system specification and the DRsM goal graphs.

Regarding the system specification documentation, we have modeled the traffic engineering dynamic resource management part of TEQUILA using standard off-the-shelf tools which UML models are direct inputs for the Behavior Manager component.

With regard to the goal elaboration process, we have elaborated the goal graph for the DRsM component making use of the Objectiver GUI attached to our solution. In Figure 6 we show a fragment of the goal-graph for this scenario. The procedure to elaborate KAOS goal-graphs is described in [12].

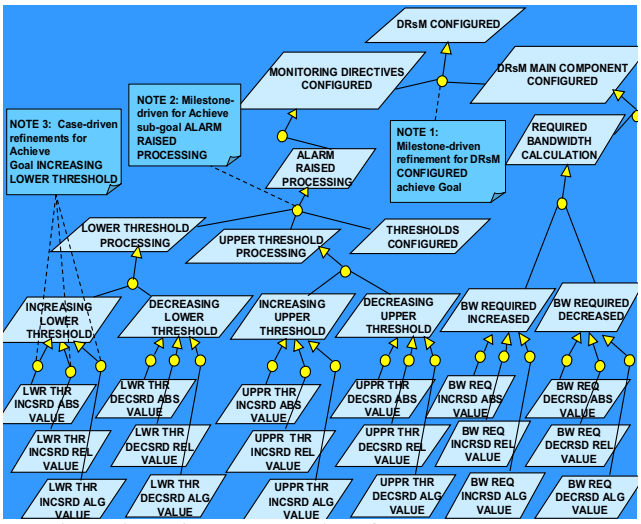

Figure 6. Partial representation of DRsM goal graph

The goal-graph provides the consultant with available refinements for the *DRsMConfigured* Achieve goal. The first refinement (Note 1 in Fig. 6) suggests a *milestone-driven* pattern as to refine Monitoring and DRsM Main Component directives. Subsequently, these goals are refined into other strategies towards the lowest-level goals. For example, for the ALARM RAISED PROCESSING goal, the graph suggests a new *milestone-driven* pattern (Note 2 in Fig. 6) to refine three sub-goals aimed at processing lower/upper threshold crossings (LOWER THRESHOLD PROCESSING and UPPER THRESHOLD PROCESSING) and configure the new thresholds previously processed (THRESHOLDS CONFIGURED). Further, case-driven patterns are used to refine these into lowest-level ones that represent the different alternatives suggested to the administrator. For example, the INCREASING LOWER THRESHOLD is refined into three strategies using *case-driven* refinements (Note 3 in Fig. 6) as suggesting different options to increase thresholds: by absolute, relative or algorithmic values.

6.2 Scenario execution details

Request for Policy Refinement submission. The consultant selects the strategies that better commit with his directives: BW allocation and thresholds need to be increased by relative values for EF traffic and the same relative value to be deducted from the AF PHB. Besides, the administrator opts to allocate any extra BW equally between the PHBs. For this, the consultant submits the following RPR through `i_GoalMgr`:

`Selected goals [1]:` G1: LWR THR INCRSD REL VALUE, G2: UPPR THR INCRSD REL VALUE, G3: THRESHOLDS CONFIGURED, G4: BW REQ INCRSD REL VAL *(EF Allocation increase)*, G5: BW REQ DECRSD REL VAL *(AF allocation decrease)*, G6: SPARE CAP EQ SPLIT, G7: LINK CONFIGURED
`GoalAttributes[1]:` G1: 0.20 * dynRange(EF,LC1,LC2), G2: 0.20 * dynRange(EF, LC1, LC2), G3: null, G4: 0.20 * dynRange (EF, LC1, LC2), G5: (0.20 * dynRange(EF), LC1, LC2), G6: null, G7: null
`Constraint[1]:`utilValue < 0.60*dynRange(EF,LC1,LC2) *(current load of LC1/LC2 below 60% of the dynamic range)*
`GoalAttributes[2]:` G1: 0.10 * dynRange(EF,LC1, LC2), G2: 0.10* dynRange(EF, LC1, LC2), G3: null, G4: 0.10 * dynRange (EF, LC1, LC2), G5: (0.10 * dynRange(EF), LC1, LC2), G6: null, G7: null
`Constraint[2]:`utilValue> 0.60*dynRange(EF,LC1,LC2) (*current load of LC1/LC2 above 60% of the dynamic range*).

Component executions. Initially, the ***Goal Manager*** processes the above RPR and interacts with the Objectiver package to abstract the details of the goal selection. In our scenario the Goal Manager has abstracted the temporal prescription and temporal association out 43 `Goal` instances.

The ***Requirements Manager*** initially abstracts the temporal relationships amongst the lowest-level goals out of the 43 `Goal` instances. Following on with this, it produces the correct LTL requirements by instantiating the formulae that specializes the temporal relationships amongst the lowest-level goals. Figure 7 shows a fragment of the objects produced in run-time by the Requirements Manager during the execution of our scenario. The most relevant here is the formal definition of the LTL formulae included in the `Formal Definition` field of the structure (bottom part of Fig.

7). Additionally, the structure includes identifiers for the LTL formulae arguments.

```
struct globaldefs.NameAndStringValue_T[1]
    string name: TemporalLogic
    string value: LTL
struct globaldefs.NameAndStringValue_T[2]
    string name: PatternMapping
    string value: AbsenceBased
struct globaldefs.NameAndStringValue_T[3]
    string name: FormalDefinition
    string value: [] ( ( m1 && ! m3 && <> m3 ) -> (! m2 U m3) ) || [] (m4 -> [] (! m5) ) || [] (m6 -> [] (! m7) )
```

Figure 7: Fragments of ReqMgr output

The ***Search Manager*** produces the system trace executions and applies the Translation Primitives to abstract the policy fields. The left part of Figure 8 shows the visual representation of a fragment of the system trace execution corresponding to one of the threshold-related directives of our scenario (upper threshold crossing). On the right part of Fig. 8 we show the result of the application of the Translation Primitives and the Search Manager algorithms, to the system trace execution shown in the left part.

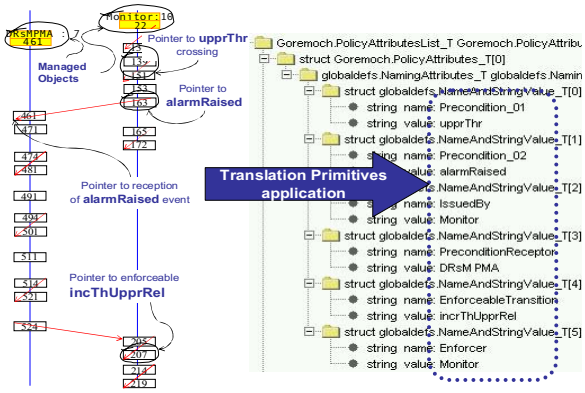

Figure 8: System trace execution and SearchMgr output

The ***Policy Encoder*** finally encodes, compiles and stores the policies after receiving the policy fields and the administrative constraints. We have populated the Inventory component with a basic object distribution ad-hoc to our scenario, nevertheless the mechanisms and the principle to automatically deploy policies from policy fields, have been preserved. Figure 9 shows two out of the 12 policies refined in our scenario. The over_DR_UpprThsLC1 policy commits to the administrative directive given to increase the upperThreshold (10% of the dynamic range) when the bandwidth utilization is higher than 60% of the dynamic range in LC1. The under_DR_UpprThsLC1 policy differs to the former in the value for the increase (20% of the dynamic range) when the bandwidth utilization is lower than 60% of the dynamic range.

```
Source | Code | Meta Policies
1  inst oblig /Private/PolicyNoTypes/Obligs/overDR_UpprThsLC1 {
2      on alarmRaised ( upprThr , utilValue , Link , EF ) ;
3      subject s = /Links/LC1/Router/drsmPMA ;
4      target t = /Links/LC1/Router/DRsM/Monitor ;
5      do t . incrThUpprRel ( Link , EF , "0.10*dynamicRange" ) ;
6      when utilValue > 0.6 * t . dynRange ; }
7  inst oblig /Private/PolicyNoTypes/Obligs/underDR_UpprThsLC1 {
8      on alarmRaised ( upprThr , utilValue , Link , EF ) ;
9      subject s = /Links/LC1/Router/drsmPMA ;
10     target t = /Links/LC1/Router/DRsM/Monitor ;
11     do t . incrThUpprRel ( Link , EF , "0.20*dynamicRange" ) ;
12     when utilValue < 0.6 * t . dynRange ; }
```

Figure 9: Sub set of the refined policies of our scenario

7. Discussion and Future Work

Regarding system behaviour, we have considered unconditional executions of actions in response to events, considering that all states and transitions are permissible. Future work will be directed to consider non permissible states and their relationships with Maintain/Avoid goals.

So far we have considered situations when just one execution trace is produced. Moreover, considering the generation of multiple traces may allow analysis to choose between different transition plans, possibly for different conditions and hence different policies.

Having in mind that the policy refinement process is carried out taking into account system behavior modelling, future research will be directed to measure and analyse the impact of administrative guidelines on the performance of the managed systems. This is a challenging issue that is fully domain-dependent. It may possibly imply to adapt other mechanisms to relate system performance with goal specialisation.

System specification is a critical issue. Although SPIN does not build the entire state space to find system execution traces, we have envisaged necessary to introduce additional specification management procedures in order to avoid the state explosion problem for very large-scale systems.

Even when policy refinement is an off-line process, real systems may need refinements of thousands of policies. Experimental results show that the overhead introduced by our solution is not considerable in comparison with the policy storage time. For example, the 12 policies in our scenario were refined in less than two seconds. The storing of a policy takes in average 800ms, anyhow these comparisons are merely indicative of the feasibility of the solution.

8. Related Work

To the best of our knowledge, at the time of this publication, there is no evidence of any complete functional solution for goal-oriented policy refinement. Moreover, functional solutions for policy refinement

are rather scarce. POWER [13] is one of the few policy refinement approaches hitherto implemented. It is a tool that enables administrator consultants to refine policies from pre-defined templates tailored for specific use. Our functional solution is a goal-oriented approach in which the consultant selects the goals that better satisfy his needs, having the possibility to formulate any combination of goals as required, with no pre-definition of policy template choices.

Reference [1] proposes to transform both, policy and system behavior specifications into a formal notation based on Event Calculus (EC). Abductive reasoning is used to derive strategies that would achieve high-level goals. From these strategies, policies are encoded. The EC-based approach and our framework differ in the way system behavior is analyzed and in the way policy information is abstracted. While EC and abduction is used in the former to infer the sequences of actions that achieve particular goals, our approach goes through automated state exploration to obtain system trace executions that fulfils lower-lever goals. The EC-based approach does not address explicit temporal execution of goals [12] whilst this is one of the pillars of our approach; system trace executions are obtained from finite state verification properties considering not only such temporal relationships but the temporal prescription of the goals. We encode policies systematically using a set of Translation Primitives applied to system execution traces while in the EC-based approach these are encoded using the generated strategies.

9. Conclusions

We have presented a functional solution for goal-oriented policy refinement grounded linear temporal logic and reactive systems analysis techniques. We have used KAOS methodology [3] and the goal-based approach to policy refinement introduced [1]. We have presented the technical foundations of our approach and demonstrated how these were used in our implementation prototype.

First of all, we have presented a formal and automated approach to abstract *temporal logic (LTL) formulae characterizing goal fulfillment*. In this sense, we described how, the temporal logic foundations of goals elaborated through the KAOS methodology, must be linked to patterns for finite-state verification properties, in favor of policy refinement.

Secondly, we have presented a formal and practical approach to abstract policies from system trace executions obtained through automated reactive systems analysis tools (i.e. model checking). This approach has been specialized in a set of *Translation Primitives* whose feasibility to refine policies in a systematic manner has been demonstrated in the paper.

Another contribution of this work has been the identification and formalization of the technical foundations mentioned above in favor of policy refinement. Additionally, we show how these foundations are integrated into a functional environment and demonstrate its application in the field of DiffServ QoS Management.

As far as it is reflected in the literature, policy refinement is at its initial stage. In this sense, we hope that our functional solution may contribute to solve the policy refinement problem.

Acknowledgments

We thank Christophe Ponsard and Philippe Massonet for their support with Objectiver APIs.

10. References

[1] A.K. Bandara, E.C. Lupu, J. Moffett, A. Russo; "A goal-based approach to policy refinement" IEEE Intl. Workshop on Policies for Distributed Systems and Networks, 2004

[2] E. Delor, R. Darimont, A. Rifaut. “Software quality starts with the modelling of goal-oriented requirements”. Intl. Conference of Software & Systems Engineering, 2003

[3] R. Darimont and A. van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," Symp. on Found. of Soft. Eng. (FSE) 1996

[4] J. Rubio-Loyola, J. Serrat, M.Charalambides, P. Flegkas, G. Pavlou, A. Lluch. “Using linear temporal model checking for goal-oriented policy refinement frameworks” IEEE International Workshop on Policies for Distributed Systems and Networks Stockholm, Sweden June 6-8, 2005

[5] Z. Manna, A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springr-Verlag, 1992

[6] M. B. Dwyer, G. S. Avrunin and J. C. Corbett. “Property specification patterns for finite-state verification” Workshop on Formal Methods in Software Practice 1998

[7] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman. "Tools for doamin-based policy management of distributed systems", NOMS, Friorence, Italy, 2002

[8] Danciu, V., Kempter, B., “From processes to policies–concepts for large scale policy generation”, IEEE/IFIP NOMS *2004*

[9] G. Holzmann. The SPIN Model Checker: Primer Reference Manual. A. Wesley. ISBN 0-321-22862-6. 2004

[10] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. “Interactive verification of UML state machines”. Intl. Conference on Formal Engineering Methods 2004

[11] P. Trimintzios et al. "A management and control architecture for Providing IP Differentiated Services in MPLS-based Networks,". IEEE Comms Magazine, 2001.

[12] A. Bandara et al. ”Policy Refinement for DiffServ Quality of Service Management” Int. Symposium on Integrated Network Management (IM) France, May 2005.

[13] M. Casassa, A. Baldwin, C. Goh. “POWER prototype: towards integrated policy-based management” IEEE/IFIP NOMS 2000