Reconciling System Requirements and Runtime Behavior

M.S. Feather[1], S. Fickas[2], A. van Lamsweerde[3], and C. Ponsard[3]

[1] Computing Services Support Solutions, Los Angeles
[2] Computer Science Department, University of Oregon
[3] Département d'Ingénierie Informatique, Université Catholique de Louvain

Abstract

This paper considers the problem of runtime system deviations from requirements specifications. Such deviations may arise from lack of anticipation of possible behaviors of environment agents at specification time, or from evolving conditions in this environment. We discuss an architecture and a development process for monitoring system requirements at runtime to reconcile the requirements and the system's runtime behavior.
This process is deployed on three scenarios of requirements-execution reconciliation for the Meeting Scheduler system. The work builds on our previous work on goal-driven requirements engineering and on runtime requirements monitoring.

Keywords: Self-adapting systems, requirements monitoring, goal-driven requirements engineering, inconsistency management, obstacles, deviation analysis, system customization.

1. Introduction

Requirements engineering (RE) is concerned with the elicitation of high-level goals to be achieved by the system envisioned, the refinement of such goals and their operationalization into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software.

The RE process often results in specifications that are not realistic enough; the specifications are likely to be violated from time to time in the running system that implements them. Such inconsistencies between the expected and the actual system behavior arise even when the specification has been validated with the user and the system is implemented correctly. Reasons for this are that (i) the agents in the *environment* may behave in ways that were not anticipated or unfeasible to model at requirements time [21], and (ii) the post-deployment evolution of environmental conditions can make initially valid assumptions about the environment no longer valid [11].

Copyright 1998 IEEE. Published in the Proceedings of IWSSD9, April 1998 Isobe, Japan. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: +Intl. 908-562-3966.

Two complementary approaches can be followed to manage runtime violations of requirements [16, 17]:

- Anticipating as many as possible of them at specification time. Obstacles to goals/requirements/assumptions are derived from first-sketch specifications; more robust specifications are then derived from the obstacles identified [18].
- Detecting and resolving such violations at runtime. Resolution here consists in making on-the-fly, *acceptable* changes to the requirements. By "acceptable", we mean changes that satisfy the high-level goals underpinning the requirements being violated.

The dynamic approach naturally complements the static one. While obstacle analysis may prove highly cost-effective for obtaining robust specifications, *complete* identification of all possible obstacles may be unachievable. Besides, overly defensive specification may be too costly to implement and result in unnecessary complexity of the software. Dynamic analysis then comes into play.

The paper explores the dynamic approach by elaborating on the requirements monitoring paradigm suggested in [11]. The approach requires that alternative system designs be explicitly represented as system parameters and/or alternative refinement trees. Two kinds of parameters are extracted from the requirements specification: *monitored* parameters (for observing behavior), and *control* parameters (for changing behavior). Consider, for example, a meeting scheduling system and a ParticipantResponsive assumption asserting that a participant who receives an invitation to a meeting will email her time constraints within a week. A monitored parameter that can be derived from that assertion might be the participant's actual response time over a number of weeks; a control parameter might be the periodicity of reminders to be sent to this participant.

On-the-fly reduction of the gap between requirements and runtime behavior is achieved as follows:

- at specification time, specifications of event sequences to be monitored are generated from requirements specifications;

- at design time, an architecture of cooperating software agents is built in which alternative system designs are explicitly represented as system parameters and/or alternative design trees;
- at runtime, the system is observed by a generic monitor so that instances of the specified event sequences can be signalled to a generic reconciler that reasons across parameter values and/or design trees. In the former case, the value of the control parameter associated with the assertion being violated is changed (e.g., send reminders more frequently for this participant); in the latter case, a shift is made to some alternative design (e.g., give a phone call to this participant; or send email to the participant's secretary; etc.).

Our approach integrates the KAOS goal-driven specification methodology [4, 15] and the FLEA runtime event-monitoring system [2]. Section 2 therefore provides some background material on KAOS and on FLEA. Section 3 then proposes an architecture of a self-adapting system for requirements-behavior reconciliation. Section 4 illustrates the ideas on examples from the Meeting Scheduler system [15, 9]. Section 5 concludes by discussing the current status of this work together with future and related work.

2. Background

2.1 Goal-driven RE with KAOS

The KAOS methodology is aimed at supporting the whole requirements elaboration process -- from the high-level goals to be achieved by agents in the composite system to the operations, objects and constraints to be assigned to software and environmental agents. Thus WHY, WHO and WHEN questions are addressed in addition to the usual WHAT questions addressed by standard specification techniques. The methodology comprises a multi-paradigm specification language, a goal-driven elaboration method, and meta-level knowledge used for local guidance during method enactment. Hereafter we introduce some of the features that will be used later in the paper; see [4, 15] for details.

The underlying ontology

The following types of concepts will be used in the sequel.

- *Object:* an object is a thing of interest in the composite system whose instances may evolve from state to state. It is in general specified in a more specialized way -as an *entity, relationship,* or *event* dependent on whether the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are characterized by attributes and invariant assertions.
- *Action:* an action is an input-output relation over objects; action applications define state transitions. Actions are characterized by pre-, post- and trigger conditions.
- *Agent:* an agent is another kind of object which acts as processor for some actions. An agent *performs* an action if it is effectively allocated to it; the agent *HasAccess*/ *HasControl* to/over an object if the states of the object are observable/controllable by it. Agents can be humans, devices, programs, etc.
- *Goal:* a goal is an objective the composite system should meet. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition for satisfying the goal. The goal refinement structure for a given system can be represented by an AND/OR directed acyclic graph. Goals *concern* the objects they refer to.
- *Constraint:* a constraint is an implementable goal, that is, a goal that can be formulated in terms of states controllable by some individual agent. Goals must be eventually AND/OR *refined* into constraints. Constraints in turn are AND/OR *operationalized* by actions and objects through strengthenings of their pre-, post-, trigger conditions and invariants, respectively. Alternative ways of assigning responsible agents to a constraint are captured through AND/OR *responsibility* links; the actual assignment of agents to the actions that operationalize the constraint is captured in the corresponding *performance* links.
- *Assumption:* an assumption is a fact taken for granted about agents in the environment. Unlike goals, assumptions cannot be enforced in general and need not be refined. Assumptions often appear as auxiliary assertions needed to prove the correctness of goal refinements or operationalizations. They are tentatively true and are likely to change. (Note that a same assertion may be either an assumption or a constraint depending on the design; e.g., ParticipantConstraintsCorrect may be an assumption in one design or a constraint in another that would require the operationalization into actions like CheckParticipantConstraints).

Language constructs

Each construct in the KAOS language has a two-level generic structure: an outer semantic net layer [1] for *declaring* a concept, its attributes and its various links to other concepts; an inner formal assertion layer for *formally defining* the concept. The declaration level is used for conceptual modeling (through a concrete graphical syntax), requirements traceability (through semantic net navigation) and specification reuse (through queries) [6]. The assertion level is optional and used for formal reasoning [5].

The generic structure of a KAOS construct is instantiated to specific types of links and assertion languages according to the specific type of the concept being specified. For example, consider the following goal specification for a meeting scheduler system:

Goal *Achieve* [ParticipantsConstraintsKnown]
Concerns Meeting, Participant, Scheduler, ...
RefinedTo ConstraintsRequested, ConstraintsProvided
InformalDef *A meeting scheduler should know the constraints of the various participants invited to the meeting within C days after invitation.*
FormalDef $\forall$ m: Meeting, p: Participant, s: Scheduler
Invited (p, m) $\wedge$ Scheduling (s, m)
$\Rightarrow \Diamond_{\leq Cd}$ Knows (s, p.Constraints)

The declaration part of this specification introduces a concept of type "goal", named ParticipantsConstraintsKnown, refering to objects such as Participant or Scheduler, refined into subgoals ConstraintsRequested and ConstraintsProvided, and defined by some informal statement.

The assertion defining this goal formally is written in a real-time temporal logic borrowed from [14]. In this paper we will use some classical operators for temporal referencing [19]: **o** (in the next state), • (in the previous state), $\Diamond$ (eventually), ♦ (some time in the past), ❑ (always in the future), ■ (always in the past), $\mathcal{U}$ (always in the future *until*), $\mathcal{W}$ (always in the future *unless*). Real-time restrictions are indicated by subscripts [14]; e.g., $\Diamond_{\leq nu}$ means "some time in the future within *n* time units *u*".

In the formal assertion above, the predicate Invited(p,m) means that, in the current state, an instance of the Invited relationship links variables p and m of sort Participant and Meeting, respectively. The Invited relationship, Participant agent and Meeting entity are declared in other sections of the specification, e.g.,

Agent Participant
CapableOf CommunicateConstraints, ...
Has Constraints: **Tuple** [ExcludedDates: **SetOf** [*TimeInterval*],
PreferredDates: **SetOf** [*TimeInterval*]]

Relationship Invited
Links Participants {**card**: 0:N}, Meeting {**card**: 1:N}
DomInvar $\forall$p: Participant, m: Meeting
Invited (p, m) $\Leftrightarrow$ p $\in$ Requesting[-,m].ParticipantsList

In the declarations above, Constraints is declared as an attribute of Participant (this attribute was used in the formal definition of ParticipantsConstraintsKnown).

As mentioned earlier, operations are specified formally by pre- and postconditions, for example,

Action DetermineSchedule
Input Requesting, Meeting {**Arg**: m}; **Output** Meeting {**Res**: m}
DomPre Requesting (-,m) $\wedge \neg$ Scheduled (m)
DomPost Feasible (m) $\Rightarrow$ Scheduled (m)
$\wedge \neg$ Feasible (m) $\Rightarrow$ DeadEnd (m)

Note that the invariant defining Invited is not a requirement, but a domain description [13]; the pre-/postcondition of DetermineSchedule above are domain descriptions as well. The effective requirements are found in the *constraints* refining the goals and in the additional pre-/postconditions and invariants that *strengthen* the corresponding domain assertions so as to ensure all constraints specified [4, 15].

2.2 The FLEA monitoring system

The FLEA language provides constructs for expressing temporal combinations of events. Runtime code to monitor such combinations is automatically generated by the FLEA system. The runtime system comprises an historical database management system equipped with an inference engine, and a communication mechanism to gather events and distribute notifications of occurrences of event combinations. The rest of this section gives a quick overview of the capabilities of FLEA [2].

FLEA introduces events as special relations in which the first parameter is the time at which the event occurs. Other parameters are event attributes.

Some events are external events called "basic events". They typically represent applications of actions by autonomous agents. Such events have to be declared, e.g.,

```
(defevent constraints-provided: external (string))
```

for the type of events of a participant (identified by a string) providing her constraints. Whenever such events are received by FLEA, it automatically timestamps them, adding that timestamp parameter as the first parameter of the relation storing that event in the underlying database. For example,

```
(constraints-provided 1365124 chp)
```

records that participant chp has provided his constraints at time 1365124.

For the experiments so far, events are exchanged between FLEA and the autonomous agents through a bus (currently implemented as a flat file which is also a log of all the system history). Later implementations may substitute socket connections for reasons of efficiency.

Beside external events, events may also be defined in terms of *temporal patterns* of events such as some specific event occurring after another (within some time, with no event in-between, etc.). Table 1 gives the various temporal patterns currently definable in FLEA.

Syntax	Meaning
then P Q	an event *P* followed by an event *Q*
then-excluding P Q R	an event *P* followed by an event *Q*, without any event *R* in-between
in-time P Q d	an event *P* followed by an event *Q* within time delay *d*
too-late P Q d	an event *P* **not** followed by an event *Q* within time delay *d*

Table 1: Temporal patterns in FLEA

Events can be counted, values of numerical attributes can be summed, etc. Such computations yield state information, which is represented as relations in the underlying database. Events can in turn be defined in terms of changes to relations' values. Thus, states and events can be defined in

terms of one another.

Example: counting reminders sent to a participant and detecting when more than 5 reminders have been issued.

More abstract events can be built from elementary events occurring in the system. In particular, Boolean expressions can be formed that combine event occurrences and state predicates.

The above features are useful for detecting violation of complex assertions, for keeping track of the frequency of violations and for deciding when it is time to react.

3. The requirements-runtime reconciliation process

Figure 1 shows the two levels involved in the requirements-behavior reconciliation process.

At *development level*, the goal-based specification is elaborated; KAOS assertions that could be violated are identified and systematically translated into FLEA event definitions; the KAOS specification is implemented as a system of parameterized cooperating agents. At *system execution level*, agent traces are observed by the monitor instantiated to the event definitions generated at development level; the violation file produced by the monitor is analyzed by the reconciler for appropriate response --that is, parameter tuning or shift to an alternative design in the KAOS AND/OR graph and its corresponding implementation. The response can be specific to incriminated agent instances (e.g., sending more reminders to the unresponsive participant axel).

To identify deviations between the specified and runtime behavior, the monitor has to observe the system in action. Two approaches can be considered: (i) restricting the observation to the automated part of the system, or (ii) observing as much as possible --including quantities from the environment. These two possibilities correspond to the notions of internal and external monitor, respectively [20]. The second approach requires appropriate interfaces to be designed for accessing the state of environmental agents.

The various steps involved in Figure 1 are now detailed before being illustrated in Section 4.

3.1 Development level

- *D1. Elaborate the goal refinement/operationalization graph, identify breakable assertions in the specification, and formalize them.* Formal refinement patterns may be used here to help discovering hidden goals/assumptions [5]. When constraints on single agents are reached in the refinement process, possibly with companion assumptions, the analyst has to ask questions such as "can this constraint/assumption be violated at runtime due to the behaviour of some agent instance associated with it?"; "if so, should one care about such violations to the point that the assertion needs to be monitored for reconciliation beyond some violation treshold?". In the sequel we will call *breakable assertions* those assertions retained for

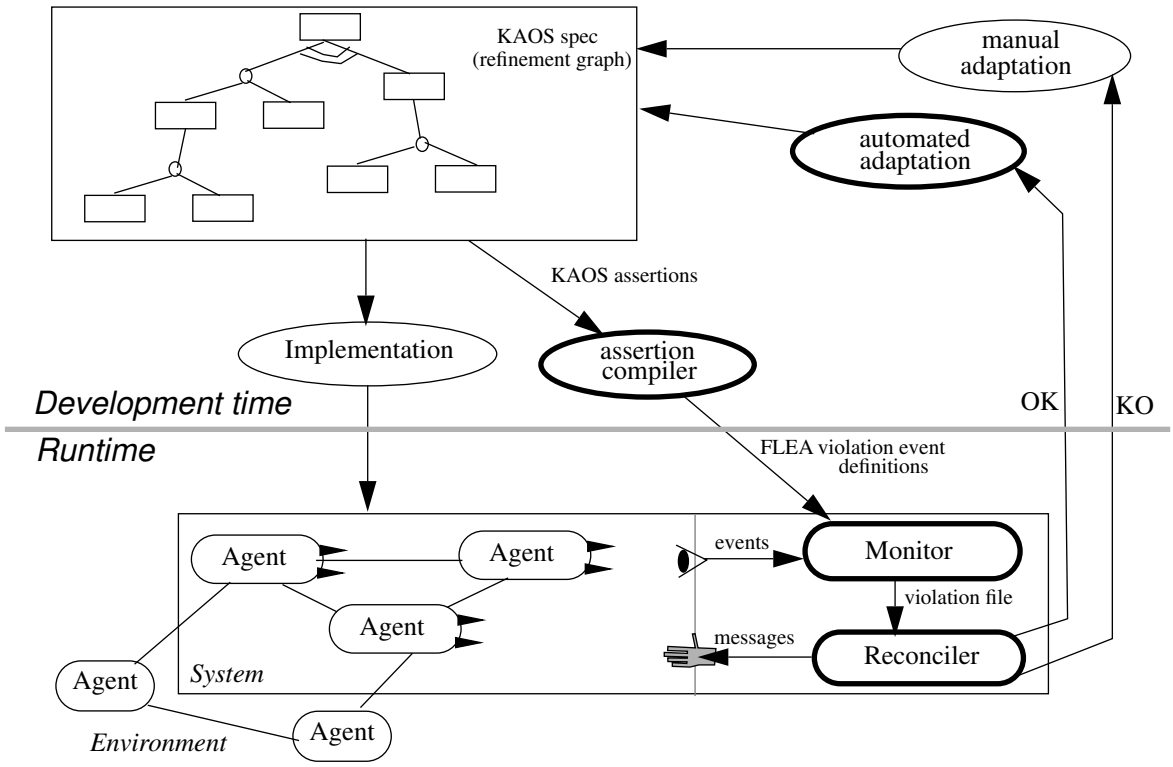

FIGURE 1 - Architecture of a self-adapting system

monitoring as a result of asking such questions. Breakable assertions are then formalized (if not done before) so as to make their temporal pattern explicit. This temporal pattern will be translated into a FLEA violation event definition in step D4 below.

- *D2. Check for monitorability and identify monitored parameters.* For internal monitoring, the objects involved in an assertion to be monitored must belong to the automated part of the system (or have a consistent image in it); for external monitoring, additional interface objects need to be introduced. Monitored parameters are then identified therefrom. In general, system adaptation should not take place after a single assertion violation by some specific agent instance; a deviation can be accidental and/or occasional. Moreover, too prompt adaptations could encourage human agents to deviate too easily. Thus only recurring deviations should trigger adequate reconciliation. Therefore, appropriate thresholds have to be defined and deviation statistics have to be gathered to detect the event of a threshold being reached. This can easily be achieved through the counting and summing facilities provided by FLEA.
- *D3. Identify reconciliation tactics.* For each breakable assertion, a choice has to be made between (i) enforcing it by introduction of restoration actions on control parameters to be identified from the assertion, or (ii) finding an alternative assertion to achieve the same parent goal in the refinement graph.
- *D4. Translate breakable assertions into FLEA*. Definitions of event sequences to be monitored are generated from real-time temporal logic assertions. This can be done systematically through the use of *transformation rules*.

  A first step is to map predicates onto events. Predicates capture relationships in KAOS assertions; for the purpose of detecting violations, we are interested in the events of a predicate changing its truth value. We therefore associate an event type *P** with each predicate *P* appearing in a breakable assertion; P* captures the class of events of P becoming true, that is, changing its truth value from *false* to *true*.

  KAOS assertion patterns can then be translated into FLEA event definition patterns by use of transformation rules. We give a sample of typical ones.

(*Avoid* assertion) $P \Rightarrow \Box \neg Q$
↓ negation
$P \wedge \Diamond Q$
↓ violation event
then P^* Q^*

(*Maintain* assertion) $P \Rightarrow Q\ \mathcal{W} R$
↓ negation
$P \wedge \neg (Q\ \mathcal{W} R)$
↓ violation event
then-excluding P^* $(\neg Q)^*$ R^* **or** (P^* **and not** Q)

(*Achieve* assertion) $P \Rightarrow \Diamond_{\leq nu} Q$
↓ negation
$P \wedge \Box_{>nu} \neg Q$
↓ violation event
too-late P^* Q^* N

In the *Maintain* translation rule above, the temporal logic assertion specifies that the predicate *Q* must remain permanently true from any current state in which *P* holds unless predicate *R* is/becomes true. The FLEA violation event definition is therefore a disjunction; the first disjunct captures the events of *P* becoming true followed by *Q* becoming false without *R* becoming true in-between; the second disjunct captures the case of *P* becoming true in some (current) state in which *Q* is false. In the *Achieve* translation rule, *N* is the result of converting the *n* time units *u* (e.g., second, minute, day) into FLEA time units.

The next section will illustrate the translation process on various KAOS assertions.

- *D5. Elaborate traceable architecture and implementation.* Functional goals/constraints assigned to software agents need to be mapped to modules in an agent-based architecture, and traceability links must be established between monitored/control parameters in the specification and the implementation, respectively.

3.2 Runtime level

- *R1. Make state information accessible to the monitor.* An appropriate communication channel must be established between the monitor and each software agent to be monitored in the architecture. Two alternative tactics can be followed to make the history of monitored parameters available.
  - *Passive communication:* every software agent provides an interface that can be questioned by the monitor. The latter has to poll every agent at some appropriate sampling rate, depending on time granularity, to make sure that no meaningful state transition is missed. This requires a huge amount of historical data to be stored. Another problem is lack of robustness; missing a single transition may result in incorrect perception of subsequent states.
  - *Active communication:* every software agent notifies an event to the monitor when a monitored parameter changes --e.g., a predicate becoming true (that is, a relationship instance being created); an entity being created; a KAOS event occurring; an event count being increased; a change of value for some object attribute; etc. The price to pay here is that some additional agent programming is required to signal relevant state transitions (see Section 4).

Passive and active communication can also be combined through transition monitoring with regular checkpoints for resynchronization. Note that in both cases only *relevant* events need to be watched/notified, that is, those corre-

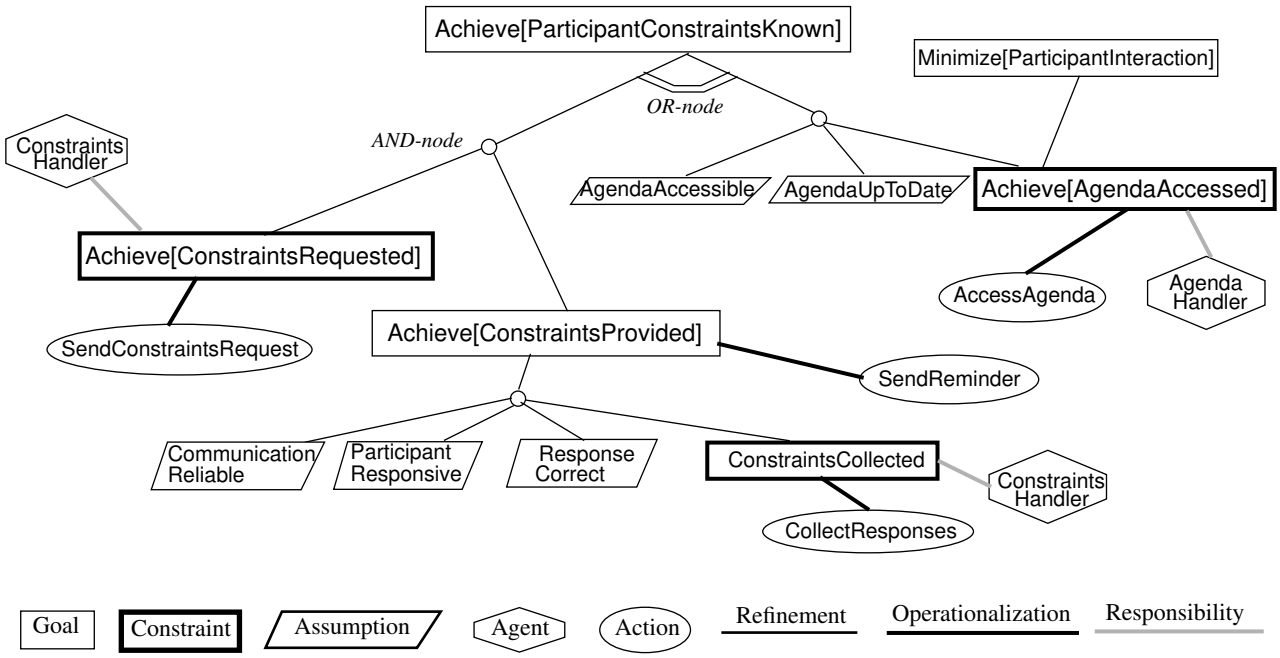

FIGURE 2 - Refinement/operationalization of *Achieve[ParticipantConstraintsKnown]*

sponding to state transitions in the monitored parameters derived from the breakable assertions identified.

- *R2. Update FLEA event definitions (if necessary).* The event sequences to be monitored must always be consistent with the breakable assertions in the *current* specification. The monitor has therefore to be reconfigured after customized shift to some alternative design; a new KAOS-to-FLEA translation has to be performed accordingly.
- *R3. Reconcile system requirements and runtime behaviour.* The violation file produced by the monitor is analyzed; the reconciliation tactic chosen in step D3 is applied. The specification and the implementation are adjusted accordingly; the monitor is reconfigured if necessary.

4. Illustration on the Meeting Scheduler system

Figure 2 shows the semantic network for the goal Achieve[ParticipantConstraintsKnown] in the Meeting Scheduler specification [15, 9]. The refinement/operationalization subtree captures two alternative "designs" to achieve this goal: getting a participant's constraints through explicit requests, or through access to her electronic agenda. Each design relies on a different set of assumptions.

Consider the first alternative. It is AND-refined into two subgoals: Achieve[ConstraintsRequested] and Achieve[ConstraintsProvided]. The first subgoal is a constraint assignable to the ConstraintsHandler agent. It states that a request for time constraints should be sent to every participant invited to the meeting:

$\forall$ m: Meeting, p: Participant

• $\neg$ Invited(p,m) $\wedge$ Invited(p,m)

$\Rightarrow \Diamond_{\leq Yd}$ ConstraintsRequested(p,m)

This constraint is operationalized through the action SendConstraintRequest.

The second subgoal can be refined according to a milestone-driven tactic [5] --the request must be transmitted by a communication agent; the request must be answered in time and in a correct way by the participant; the response must be transmitted by the communication agent; the responses must be collected by the scheduler. Note that some of these assertions are assumptions about agents in the environment --if electronic mail is used for communication, the first assumption is about an automated agent in the environment; the second assumption is about expected behavior of human agents.

4.1 Reconciliation by parameter tuning

We first consider the assumption ParticipantResponsive and replay the various steps described in the previous section.

Development level

D1. Identify breakable assertions in the specification, and formalize them. The assumption ParticipantResponsive is identified as a breakable assertion to be monitored. It can be formalized by

$\forall$ m: Meeting, p: Participant

RequestReceived(p,m) $\Rightarrow \Diamond_{\leq Xd}$ ConstraintsProvided(p,m)

D2. Check for monitorability and identify monitored

parameters. This assertion is internally monitorable because the two relationships involved in the assertion, that is, RequestReceived and ConstraintsProvided, can be accessed within the automated part of the system through corresponding monitorable events.

D3. Identify reconciliation tactics. Consider a participant not responding in time to some constraint request. The following adaptations might be envisaged: increasing the frequency of reminders; emailing the request to another participant who knows the time constraints of the incriminated participant (e.g., her secretary); switching to an alternative design based on the participant's electronic agenda. For the first tactic, the control parameter is the frequency of reminders, to appear as parameter in the SendReminder action (under control of the ConstraintsHandler agent). For the second tactic, the control parameter is the participant whom to send the reminder to, to appear as parameter in that same action.

We could make the first reconciliation tactic more precise through the following domain-specific *adaptation rule*:

(a1) **IF** the average number of reminders-per-meeting to this participant becomes greater than U
THEN increase the frequency of reminders
AND start sending reminders sooner

The second reconciliation tactic could be made more precise through the following rule:

(a2) **IF** the average number of reminders-per-meeting to this participant becomes greater than V (V > U)
THEN after the first reminder send next reminders to alternative contact person

We will see below how these rules can be implemented in FLEA.

D4. Translation into FLEA. The assumption ParticipantResponsive formalized in step D1 above has an *Achieve* pattern; the third transformation rule in Section 3.1 is therefore applicable. The preliminary step of converting KAOS predicates *P* from this formalization into FLEA event types *P** results in the following FLEA event type declarations:

```
(defevent request-received :external (string string))
(defevent constraints-provided :external (string string))
```

where the two parameters of type string represent the meeting and participant involved, respectively.

The transformation rule for *Achieve* assertions then directly yields the following event violation definition:

```
(defevent constraints-provided-violation
          :definition (participant meeting)
  (too-late  (request-received $ participant meeting)
             (constraints-provided $ participant meeting)
             X[1]) )
```

In this definition, participant and meeting are introduced as event attributes to associate the events request-received and constraints-provided pairwise for a same participant and meeting.

D5. Elaborate traceable architecture and implementation. The ConstraintsHandler software agent is derived from the KAOS specification; it has the parameterized operations SendConstraintsRequest and SendReminder among its methods (as discussed in D3).

Runtime level

R1. Make state information accessible to the monitor. Every time a request for constraints is received or a response is sent out, the ParticipantMailer agent notifies the event to the monitor. The event is picked up through the FLEA bus and incorporated into the FLEA event database.

Example:

```
(request-received 01/11/97 Oregano_meeting axel)
(request-received 01/11/97 Oregano_meeting martin)
(request-received 01/11/97 Oregano_meeting jeff)
(constraints-provided 02/11/97 Oregano_meeting martin)
(constraints-provided 04/11/97 Oregano_meeting jeff)
```

(For simplicity, timestamps are converted into dates.)

R2. Update FLEA event definitions. This step is not necessary here since there is no shift to an alternative design.

R3. Reconcile system requirements and runtime behaviour. Assume a time-out of 5 days, and participant axel not having returned his constraints for the Oregano meeting before this time-out. After five days, the following event will be generated by FLEA and stored in the violation file:

```
(constraints-provided-violation 06/11/97 Oregano_meeting axel)
```

Let us show how this information can be used for reconciliation according to the adaptation rules (a1) and (a2) above. Assume that external events are generated when an invitation is sent out and when a reminder is sent out, according to the following definitions:

```
(defevent invitation-sent :external (string string))
(defevent reminder-sent :external (string string))
```

Then the average number of reminders sent per meeting can be generated according to the following definition:

```
(defrel participant-avgnb-reminder :definition (participant avg)
    (/   (countof (reminder-sent * participant) =))
         (countof (invitation-sent * participant) =))
         avg) )
```

Events can be generated for the adaptation rules (a1) and (a2) above. The definition of a single event type to handle the first rule would look like:

```
(defevent constraint-handler-adaptation-1-needed
                :transition (participant newahead newfreq)
  (and  (start (> (participant-avgnb-reminder participant) U))
        (+ 1 (start-reminders-ahead participant =) newahead)
        (+ 2 (frequency-of-reminders participant =) newfreq)) )
```

The constraint-handler-adaptation-1-needed event will be generated when the participant-avgnb-reminder value starts to exceed U, with the event's parameters newahead set to one

1. X should be converted into FLEA time units.

more than the current value (assumed to be accessible in the relation start-reminders-ahead), and newfreq to two more than the current value.

Imagine that previously axel has rarely been late to respond with his time constraints, but now changes his behavior and is frequently late to respond. As axel's average climbs above the U threshold, the system will detect this and increase the frequency of reminders (e.g., from 2 every week to 4 every week) and start sending them sooner (e.g., 7 days before the deadline rather than 6 days before). This is due to the generation of the event:

(constraint-handler-adaptation "axel" 7 4)

A similar treatment for adaptation rule (a2) can be used to change to automatically redirecting reminders to axel's secretary should he continue to be unresponsive.

The example above also shows the need for some higher-level language to specify control meta-rules such as (a1) and (a2) --see the discussion in Section 5.

4.2 Reconciliation by shifting to an alternative design

An alternative refinement for the goal Achieve[ParticipantConstraintsKnown] is to get the participant's time constraints by accessing her electronic agenda (see Figure 2). There are two assumptions here about participants: (i) they give access to their agenda, and (ii) their agenda is up to date. Both assumptions can be monitored on a participant-by-participant basis. As violations are detected, they are used to trigger a switch to explicitly asking those participants for their constraints, rather than rely upon their inaccessible and/or outdated agendas.

Space limits preclude a complete exposition, but it is worth noting two points.

Reconciliation tactics

Suppose that the default design is based on the use of electronic agendas. In case of repeated failure for some participant, one solution is to dynamically switch to the alternative design of getting constraints through explicit request to *this* participant, while other participants will continue to be dealt with via their agendas. To accommodate this, the OR-refinement tree should be annotated with the following invariant:

$\forall$ m: Meeting
$A1 \cup A2 = \{p: participant \mid Invited(p,m)\}) \wedge A1 \cap A2 = \emptyset$
where A1= {p: Participant | ConstraintsRequested(p,m)}
A2= {p: Participant | AgendaAccessed(p,m)}

System adaptation

At runtime, the software agents responsible for the various design alternatives need to know exactly which participants they are controlling. The two sets in the above assertion must therefore be updated at runtime through adaptation events that indicate an alternative switch for some specific agent. Such events can be derived from the above failure event:

```
(defevent constraints-handler-notify
            (t1 message participant meeting)
    (and  (agenda-failure t1 participant meeting)
          (<bind message to "add">) )

(defevent agenda-handler-notify
            (t1 message participant meeting)
    (and  (agenda-failure t1 message participant meeting)
          (<bind message to "remove">) )
```

4.3 Example with a *Maintain* pattern

The examples so far considered *Achieve* assertions, that is, assertions having the pattern $P \Rightarrow \Diamond Q$. *Maintain* goals/assumptions are frequently found in KAOS specifications; their pattern is $P \Rightarrow \Box Q$ or $P \Rightarrow Q \; \mathcal{W} R$. We illustrate the development level process in such cases.

D1. Identifying and formalizing breakable assertions

It might be desirable to require that the time constraints provided by a participant for a meeting remain valid unless changes are explicitly notified:

$\forall$ m: Meeting, p: Participant
ConstraintsProvided(p,m)
$\Rightarrow$ ConstraintsValid(p,m) $\mathcal{W}$ ChangeNotified(p,m)

It might be worth monitoring this assertion in order to discourage repeatedly unreliable behaviour.

D2. Check for monitorability and identify monitored parameters. While the ConstraintsProvided and ChangeNotified relationships can be accessed within the automated part of the system, the ConstraintsValid relationship needs some extra interface to monitor it externally. For instance, a list of no-shows might be maintained by a MinutesManager software agent. The event type that captures invitees appearing in this list might be named constraints-invalid.

D3. Identify reconciliation tactics. Consider a meeting invitee having sent no constraints change and not showing up. Various reconciliation tactics can be envisaged, e.g., sending a commitment message upon initial receipt of the constraints; publicizing the "black list" of no-shows; assigning a penalty when such repeated behaviour exceeds some acceptable treshold; etc. The shift from one design to the other might be based on the participant's status --e.g., the last alternative above might be selected for students.

D4. Translate breakable assertions into FLEA. The relevant events to be monitored are declared in a way similar to Section 4.1. The *Maintain* transformation rule in Section 3.1 is applicable and yields the following FLEA violation event definition, after dropping the second, impossible disjunct:

```
(defevent constraints-valid-violation
          :definition (participant meeting)
    (then-excluding (constraints-provided $ participant meeting)
                    (constraints-invalid $ participant meeting)
                    (change-notified $ participant meeting) )
```

5. Discussion

The novelty of this approach lies in the combination of goal-based requirements-time reasoning, event-based runtime monitoring, and system self-adaptation tactics. Discussion will focus on this combination, and skip mention of alternatives for the individual elements of the combination (i.e., alternative requirements approaches, alternative treatments of temporal database reasoning).

5.1 Related work

Similar efforts exist towards deriving real-time monitors from system requirements documentation in the SCR framework [20]. The similarity with our approach is the automated generation of a monitor. The difference is that [20] simulates the whole composite system through finite state automata to detect deviations (a state of the FSA represents an equivalence class of histories of the system). This approach is more operational and suited for testing. We are also focusing on soft assertions to detect deviations as opposed to strong requirements (although we could). Finally, [20] does not consider the problem of reconciliation after deviations have been detected.

The dynamic inconsistencies we consider are related to the concept of deviation between a running process and its process model as studied in [3]. As in [20], state machines are used in [3] both for modeling the human environment and the process support system.

The Bridget system [12] tackled the bridging of the design-use gap for form-based interfaces. In this approach, design-time assumptions about, say, the completion of fields in forms, were cast into run-time "expectation agents". These were able to detect mismatches between assumptions and use, and report these to developers and/or users. The KAOS-FLEA marriage takes this idea a step further, the key being KAOS' explicit treatment of the requirements process. The Bridget team has more recently worked on a design environment, Argo [23], and experiments are underway to try to connect FLEA with Argo.

Durney has looked at the problem of requirements adaptation. He has identified a set of requirements strengthening and weakening transformations based on scenarios generated through static analysis [7]. We see promise in tying this work to "scenarios" generated through runtime monitoring as well.

The execution environment in the general architecture proposed in this paper for runtime monitoring and reconciliation bears some resemblance to environments having different operating modes, as found in many control systems. The main difference lies in what such modes capture here, that is, goals, assumptions and designs that are translated systematically into event definitions, and can be adapted at runtime to reduce monitored deviations while preserving the higher-level goals those goals/assumptions/designs contribute to.

5.2 Current status and future work

We are currently implementing a prototype self-adapting system for the Meeting Scheduler system. Experimentation with this prototype is needed to show the feasibility and limitations of our approach. We plan to build a KAOS-FLEA generator on top of the GRAIL/KAOS environment [6], by making use of the attributed grammar mechanisms supported by the abstract syntax tree engine used by GRAIL [22]. A pre-compiled library of KAOS-to-FLEA transformation rules should be made available for frequent goal/assumption patterns. Guidelines for adapting specialized assertions should also be provided.

The requirements-runtime reconciliation process needs much further attention. One approach under investigation is to capture adaptation/reconciliation tactics as control meta-rules, formalized in some high-level declarative language, from which control code can be synthesized.

Fickas has begun work on a second example problem to test the architecture we propose. The problem is one of configuring an enterprise network[1]. It pushes our work further in the area of non-functional requirements. *Reliability* is an issue; monitored parameters capture availability of and connectivity to specific resources/subnets over a WAN, and control parameters capture alternative enterprise-internetworking designs. *Security* is an issue; monitored parameters capture access patterns and known attack signatures, and control parameters capture firewall settings and access control lists. *Performance* is an issue; monitored parameters capture LAN routing and cycle resources, and control parameters capture bandwidth reservation policies, host priority scheduling, etc. Early results suggest a good fit with the interacting agents model of composite system design [10], KAOS goals to represent enterprise networking requirements, the generation of network monitoring through FLEA, and the runtime adaptation of the system through consideration of alternative designs.

The enterprise networking problem is also being used to explore a tighter linkage between static and dynamic analysis. In particular, how can we test that any specific self-adatation architecture is valid? One approach that is currently being explored is the generation of "adatation tests" from KAOS specifications. Instead of verifying the correctness of the specification itself, these tests target the adaptation strategy embodied in the runtime system. Early results here again point to a good fit with the components we are attempting to integrate.

Acknowledgement. Fickas' work on validation of self-adapting systems is supported by ARPA as part of the QUORUM group. The work of van Lamsweerde and Ponsard was partially supported

1. This work is supported by and in cooperation with the Internet

by the “Communauté Française de Belgique” (FRISCO project, Actions de Recherche Concertées Nr. 95/00-187 - Direction générale de la Recherche). Thanks are due to the IWSSD reviewers for helpful feedback.

6. References

[1] R.J. Brachman and H.J. Levesque (eds.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[2] D. Cohen, M. S. Feather, K. Narayanaswamy and S. Fickas, “Automatic Monitoring of Software Requirements”, *Proc. 19th International Conference on Software Engineering*, Boston, May 1997.

[3] G. Cugola, E. Di Nitto, A. Fuggetta and C. Ghezzi, “A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems”, *ACM Transactions on Software Engineering and Methodology* Vol. 5 No. 3, July 1996, 191-230.

[4] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-directed requirements acquisition”, *Science of Computer Programming*, Vol. 20, 1993), pp. 3-50.

[5] R. Darimont and A. van Lamsweerde, “Formal Refinement Patterns for Goal-Driven Requirements Elaboration”, *Proc. FSE4 - Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineerin*g, San Francisco, October 1996.

[6] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, “GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering”, *Proc. ICSE’97 - 19th Intl. Conference on Software Engineering*, Boston, May 1997, 612-613.

[7] B. Durney, “Requirements Transformations”, PhD Thesis, Computer Science Department, University of Oregon, 1994

[8] M. Feather. “An Implementation of Bounded Obligations*”. Proc. KBSE’93, 8th Knowledge-Based Software Engineering Conference, Chicago, Illinois, USA*. 1993

[9] M. Feather, S. Fickas, A. Finkelstein and A. van Lamsweerde, “Requirements and Specification Exemplars”, *Automated Software Engineering*, Vol. 4 No. 4, October 1997, 419-438.

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

[11] S. Fickas and M. Feather, “Requirements Monitoring in Dynamic Environments”, *Proc. RE’95 - 2nd International Symposium on Requirements Engineering*, York, IEEE, 1995.

[12] A. Girgensohn, D.F. Redmiles and F.M. Shipman, III. Agent-Based Support for Communication between Developers and Users in Software Design. In *Proc. The Ninth Knowledge-Based Software Engineering Conference*, Monterey, September 1994.

[13] M. Jackson and P. Zave, “Domain Descriptions”, *Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

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

[15] 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 International Symposium on Requirements Engineering*, York, IEEE, 1995.

[16] A. van Lamsweerde, “Divergent Views in Goal-Driven Requirements Engineering”, *Proc. Viewpoints’96 - ACM SIGSOFT Workshop on Viewpoints in Software Development*, October 1996.

[17] A. van Lamsweerde, E. Letier, C. Ponsard. “Leaving Inconsistency”, *Proc. ICSE’97 workshop on “Living with Inconsistency”*, May 17, 1997

[18] A. van Lamsweerde, E. Letier. “Integrating Obstacles in Goal-Driven Requirements Engineering”, *Proc. ICSE’98 - 20th International Conference on Software Engineering, Kyoto, April 1998.*

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

[20] D. Peters. “Deriving Real-time Monitors from System Requirements Documentation”, Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE’97) Doctoral Consortium, January 1997

[21] C. Potts, “Using Schematic Scenarios to Understand User Needs”, *Proc. DIS’95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, University of Michigan, August 1995.

[22] Reps, T. and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[23] J.E. Robbins, D.M. Hilbert and D.F. Redmiles. Extending Design Environments to Software Architecture Design. In *Proc. The Eleventh Knowledge-Based Software Engineering Conference*, Syracuse, September 1996.