An Integrated Formal Framework for Reasoning about Goal Interactions

Michael Winikoff*

Department of Information Science
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

Abstract. One of the defining characteristics of intelligent software agents is their ability to pursue goals in a flexible and reliable manner, and many modern agent platforms provide some form of goal construct. However, these platforms are surprisingly naive in their handling of *interactions* between goals. Most provide no support for detecting that two goals interact, which allows an agent to interfere with itself, for example by simultaneously pursuing conflicting goals. Previous work has provided representations and reasoning mechanisms to identify and react appropriately to various sorts of interactions. However, previous work has not provided a framework for reasoning about goal interactions that is generic, extensible, formally described, and that covers a range of interaction types. This paper provides such a framework.

1 Introduction

One of the defining characteristics of intelligent software agents is their ability to pursue goals in a flexible and reliable manner, and many modern agent platforms provide some form of goal construct [1]. However, these platforms are surprisingly naive in their handling of *interactions* between goals in that few implemented agent platforms provide support for reasoning about interactions between goals. Platforms such as Jason [2], JACK [3], 2APL [4] and many others don't make any attempt to detect interactions between goals, which means that agents may behave irrationally. Empirical evaluation [5] has shown that this can be a serious issue, and that the cost of introducing limited reasoning to prevent certain forms of irrational behaviour is low, and consistent with bounded reasoning.

There has been work on providing means for an agent to detect various forms of interaction between its goals, such as resource contention [6], and interactions involving logical conditions, both positive [7] and negative (e.g. [8]). However, this strand of work has not integrated the various forms of reasoning into a single framework: each form of interaction is treated separately. Although more recent work by Shaw and Bordini [9] does integrate a range of interaction reasoning mechanisms, it does so indirectly, by translation to Petri nets, which makes it difficult to extend, to determine whether the

* This work was partly done while the author was employed by RMIT University.

reasoning being done is correct, or to relate the reasoning back to the agent's goals and plans (traceability).

This paper provides a framework for extending BDI platforms with the ability to reason about interactions between goals. The framework developed improves on previous work by being generic and by being formally presented. Thus, the key criteria for evaluating our proposed framework is its ability to deal with the different types of interaction between goals. The sorts of goal interactions that we want to be able to model and reason about include the following.

Resources: goals may have resource requirements, including both reusable resources such as communication channels, and consumable resources such as fuel or money. Given a number of goals it is possible that their combined resource requirements exceed the available resources. In this case the agent should realise this, and only commit to pursuing some of its goals or, for reusable resources, schedule the goals so as to use the resources appropriately (if possible). Furthermore, should there be a change in either the available resources or the estimated resource requirements of its goals, the agent should be able to respond by reconsidering its commitments. For example, if a Mars rover updates its estimate of the fuel required to visit a site of interest (it may have found a shorter route), then the rover should consider whether any of its suspended goals may be reactivated.

Conditions: goals affect the state of the agent and of its environment, and may also at various points require certain properties of the agent and/or its environment. An agent should be aware of interactions between goals such as:

- After moving to a location in order to perform some experiment, avoid moving elsewhere until the experiment has been completed.
- If two goals involve being at the same location, schedule them so as to avoid travelling to the location twice.
- If there are changes to conditions then appropriate re-planning should take place. For example, if a rover has placed equipment to perform a long-running experiment but the equipment has malfunctioned, then the rover should respond to this.

In summary, the challenge is to provide mechanisms that allow for:

- Specification of the dependencies between goals/plans and resources/conditions. To be practical, dependencies must be specified in a local and modular fashion where each goal or plan only needs to specify the resources/conditions that it is directly affected by.
- Reasoning about conditions and resources so as to detect situations where there is interaction between goals.
- Having a means of specifying suitable responses to detected interactions. Possible responses include suspending or aborting a goal, changing the means by which a goal is achieved (e.g. travelling by train rather than plane to save money), and scheduling goals (e.g. to avoid double-booking a reusable resource).

Section 2 reviews the goal framework and agent notation that we will build on. Section 3 presents our framework for reasoning about goal interactions, and Section 4 completes the framework by extending the agent notation. In Section 5 we evaluate the framework by showing how it is able to deal with the various types of goal interaction under consideration. We conclude in Section 6.

2 Conceptual Agent Notation with Generic Goals

We now briefly present the Conceptual Agent Notation (CAN) [10, 11]. CAN is used as a representative for a whole class of BDI agent languages which define agent execution in terms of event-triggered plans, where multiple plans may be relevant to handle a given event, and where failure is handled by reposting events. It is similar to AgentSpeak(L) [12] in that it uses a library of event-triggered plans which have a specified trigger, context condition, and plan body. CAN differs from AgentSpeak(L) in that it provides additional constructs, and in that it uses a particular failure handling mechanism (event reposting) which is common to BDI languages.

In order to extend goals into "interaction-aware goals" that are able to detect and respond to interactions with other goals we will use a variant of CAN which uses the generic goal construct of van Riemsdijk *et al.* [1]. Their framework defines a goal type with certain default life-cycle transitions, and provides a mechanism for adding additional life-cycle transitions. A goal type is defined in terms of a set C of condition-response pairs $\langle c, S\rangle$ where c is a condition to be checked that, if true, changes the goal's state to S. For example, a goal to achieve p includes $\langle p, \text{DROPPED}\rangle$ which specifies that when p becomes true the goal should be dropped. Condition-response pairs come in two flavours: "continuous", checked at all times, and "end", checked only at the start/end of plan execution. A goal instance $\mathsf{g}(C, \pi_0, S, \pi)$ specifies a set of condition-response pairs C, an initial plan π_0, a current state S (e.g. ACTIVE, DROPPED, SUSPENDED), and a current plan π.

The default goal life-cycle of van Riemsdijk *et al.* [1] is that goals are adopted into a suspended state, and they are then repeatedly activated and suspended until they are dropped. Active goals are subjected to means-end reasoning to find an abstract plan for pursuing the goal, and this plan is then executed (as long as the goal remains active).

We integrate this generic goal construct into CAN, replacing its more limited goal construct. The resulting language defines an agent in terms of a set Π of plans of the form $e : c \leftarrow \pi$ where e is the triggering event, c is a context condition (a logical formula over the agent's beliefs), and π is a plan body (we will sometimes refer to plan bodies as "plans"):

$$\pi ::= \epsilon \mid a \mid e \mid \pi_1; \pi_2 \mid \pi_1 \| \pi_2$$

We denote the empty plan body by ϵ, and an event is written as e. For simplicity we define a generic action construct, a, which has a pre-condition pre_a and post-condition defined in terms of non-overlapping addition and deletion sets add_a and del_a. A number of previously defined CAN constructs can be viewed as special cases of this, for example $+b$ can be defined as an action with $pre_{+b} = true$, $add_{+b} = \{b\}$ and $del_{+b} = \emptyset$. Similarly, $-b$ has $pre_{-b} = \{b\}$, $add_{-b} = \emptyset$, $del_{-b} = \{b\}$ and $?c$ has $pre_{?c} = \{c\}$ and $add_{?c} = del_{?c} = \emptyset$. We assume that events e and actions a can be distinguished. An agent configuration is a pair $\langle B, G\rangle$ where B is the agent's current beliefs, and G is a set of goals.

Figures 1 and 2 provide formal semantics for this language (based on previously presented semantics for CAN [1, 10, 13]) in structured operational semantics style [14] where the premise (above the line) gives the conditions under which the transition below the line may take place. We define a number of different transition types. Firstly, $\rightarrow$ as

$$\frac{S = \textsc{Active}}{\mathsf{g}(C, \pi_0, S, \epsilon) \stackrel{e}{\Rightarrow} \mathsf{g}(C, \pi_0, S, \pi_0)}\ 1 \qquad \frac{\pi \stackrel{e}{\Rightarrow} \pi' \quad S = \textsc{Active}}{\mathsf{g}(C, \pi_0, S, \pi) \stackrel{e}{\Rightarrow} \mathsf{g}(C, \pi_0, S, \pi')}\ 2$$

$$\frac{\langle c, S', f\rangle \in C \quad B \models c \quad S \neq S' \quad ok(f, \pi)}{\mathsf{g}(C, \pi_0, S, \pi) \stackrel{u}{\Rightarrow} \mathsf{g}(C, \pi_0, S', \pi)}\ 3$$

$$\frac{g \in G \quad \langle B, g\rangle \stackrel{e}{\Rightarrow} \langle B', g'\rangle}{\langle B, G\rangle \stackrel{e}{\rightarrow} \langle B', (G \setminus \{g\}) \cup \{g'\}\rangle}\ 4 \qquad \frac{g \in G \quad g \stackrel{u}{\Rightarrow} g'}{\langle B, G\rangle \stackrel{u}{\rightarrow} \langle B, (G \setminus \{g\}) \cup \{g'\}\rangle}\ 5$$

$$\frac{\langle B, G\rangle \stackrel{u}{\rightarrow}{}^{*} \langle B, G'\rangle \quad \langle B, G'\rangle \stackrel{u}{\not\rightarrow} \langle B, G''\rangle}{\langle B, G\rangle \stackrel{u}{\twoheadrightarrow} \langle B, \{g | g \in G' \wedge g \neq \mathsf{g}(C, \pi_0, \textsc{Dropped}, \pi)\}\rangle}\ 6$$

$$\frac{\langle B, G\rangle \stackrel{u}{\twoheadrightarrow} \langle B, G''\rangle \quad \langle B, G''\rangle \stackrel{e}{\rightarrow} \langle B', G'\rangle}{\langle B, G\rangle \rightarrow \langle B', G'\rangle}\ 7$$

Fig. 1. Formal semantics for CAN with generic goals

being a transition over a *set* of goals (i.e. $\langle B, G\rangle$), and $\Rightarrow$ is defined as being a transition over a *single* goal/plan (i.e. $\langle B, g\rangle$ where $g \in G$). Furthermore, we use letters to denote particular transition types (e for execute, u for update) and a superscript asterisk ($*$) denotes "zero or more" as is usual. For conciseness we abbreviate $\langle B, g\rangle$ by just g, for example the bottom of rule 9 abbreviates $\langle B, e\rangle \stackrel{e}{\Rightarrow} \langle B', (\!|\Gamma|\!)\rangle$, and similarly for rules 1-3.

Figure 1 defines the semantics of goals. The first two rules specify that an active goal can be executed by replacing an empty plan with the initial plan π_0 (rule 1) or by executing the goal's plan (rule 2) which makes use of the rules for plan execution (Figure 2). The next rule (3) defines a single goal update: if an update condition holds, update the goal's state, subject to two conditions: firstly, the new state should be different ($S \neq S'$), secondly, the condition c should be active given the f tag[1] and the plan π; formally $ok(f, \pi) \equiv ((f = end \wedge \pi = \epsilon) \vee (f = mid \wedge \pi \neq \epsilon) \vee f = all)$. Rules 4 and 5 define respectively execution (rule 4) and update (rule 5) of a set of goals by selecting a single goal and respectively executing it or updating it. Rule 6 defines a complete update cycle $\stackrel{u}{\twoheadrightarrow}$ which performs all possible updates, and deletes goals with a state of "DROPPED". Rule 7 defines a single top-level transition step of a set of goals: first perform all possible updates ($\stackrel{u}{\twoheadrightarrow}$) and then perform a single execution step ($\stackrel{e}{\rightarrow}$). We require that all possible updates are done in order to avoid ever executing a goal that has a pending update to a non-active state.

Figure 2 defines a single execution step ($\stackrel{e}{\Rightarrow}$) for various CAN plan constructs. Rule 8 defines how an action a is executed in terms of its precondition and add/delete sets. Rule 9 defines how an event is replaced by the set of guarded relevant plan instances

[1] We have compressed the two sets C and E of van Riemsdijk *et al.* [1] into a single set of triples $\langle c, S, f\rangle$ where f is a flag specifying when the condition should be checked: when the plan is empty (end), when the plan is non-empty, i.e. during execution (mid) or at all times (all). E.g. $\langle c, S\rangle \in C$ in their framework translates to $\langle c, S, all\rangle$.

$$\frac{B \models pre_a}{\langle B, a\rangle \stackrel{e}{\Rightarrow} \langle (B \cup add_a) \setminus del_a, \epsilon\rangle}\ 8 \qquad \frac{\Gamma = \{c\theta{:}\pi\theta \mid (e'{:}c{\leftarrow}\pi) \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{e \stackrel{e}{\Rightarrow} (\!|\Gamma|\!)}\ 9$$

$$\frac{(c_i{:}\pi_i) \in \Gamma \quad B \models c_i\theta \quad \pi_i\theta \stackrel{e}{\Rightarrow} \pi'}{(\!|\Gamma|\!) \stackrel{e}{\Rightarrow} \pi_i\theta \triangleright (\!|\Gamma \setminus \{c_i{:}\pi_i\}|\!)}\ 10 \qquad \frac{P_1 \stackrel{e}{\Rightarrow} P'}{P_1; P_2 \stackrel{e}{\Rightarrow} \overline{P'; P_2}}\ 11 \qquad \frac{P_1 \stackrel{e}{\Rightarrow} P'}{P_1 \| P_2 \stackrel{e}{\Rightarrow} \overline{P' \| P_2}}\ 12$$

$$\frac{P_2 \stackrel{e}{\Rightarrow} P'}{P_1 \| P_2 \stackrel{e}{\Rightarrow} \overline{P_1 \| P'}}\ 13 \qquad \frac{P_1 \stackrel{e}{\Rightarrow} P'}{P_1 \triangleright P_2 \stackrel{e}{\Rightarrow} \overline{P' \triangleright P_2}}\ 14 \qquad \frac{P_1 \stackrel{e}{\not\Rightarrow} P' \quad P_2 \stackrel{e}{\Rightarrow} P_2'}{P_1 \triangleright P_2 \stackrel{e}{\Rightarrow} P_2}\ 15$$

Fig. 2. Formal semantics for CAN plan constructs

$(\!|\Gamma|\!)$. Rule 10 selects an applicable plan instance from a set of plans, using the auxiliary construct $\triangleright$ to indicate "try π, but if it fails, use the set of (remaining) relevant plans". Rule 11 simply defines the semantics of sequential execution ";", rules 12 and 13 define parallel execution "$\|$", and rules 14 and 15 define "try-else" ($\triangleright$). The function denoted by an overline (e.g. $\overline{\pi_1; \pi_2}$) cleans up by removing empty plan bodies: $\overline{\epsilon; \pi} = \overline{\epsilon \| \pi} = \overline{\pi \| \epsilon} = \pi$, and $\overline{\epsilon \triangleright \pi} = \epsilon$, otherwise $\overline{\pi} = \pi$.

Note that the semantics model failure as an inability to progress, i.e. a failed plan body π is one where $\pi \not\Rightarrow \pi'$. This simplifies the semantics at the cost of losing the distinction between failure and suspension, and creating a slight anomaly with parallelism where given $\pi_1 \| \pi_2$ we can continue to execute π_2 even if π_1 has "failed". Both these issues are easily repaired by modelling failure separately (as is done by Winikoff *et al.* [10]), but this makes the semantics considerably more verbose.

We can now define a (very!) simple Mars rover that performs a range of experiments at different locations on the Martian surface. The first plan below for performing an experiment of type X at location L firstly moves to the appropriate location L, then collects a sample using the appropriate measuring apparatus.

$$exp(L, X) \ :\ \neg locn(L) \ \leftarrow\ goto(L)\ ;\ sample(X)$$
$$exp(L, X) \ :\ locn(L) \quad \leftarrow\ sample(X)$$

We assume for simplicity of exposition that $goto(L)$, and $sample(X)$ are primitive actions, but they could also be defined as events that trigger further plans. The action $goto(L)$ has precondition $\neg locn(L)$ and add set $\{locn(L)\}$ and delete set $\{locn(x)\}$ where x is the current location.

3 Reasoning about Interactions

We provide reasoning about interactions between goals by:

1. Extending the language to allow goal requirements (resources, conditions to be maintained etc.) to be specified (Section 3.1).
2. Providing a mechanism to reason about these requirements, specifically by aggregating requirements and propagating them (Section 3.2).

3. Defining new conditions that can be used to specify goal state transitions, and adding additional state transition types that allow responses to detected interactions to be specified. These are then used to extend CAN with interaction-aware goals (Section 4).

3.1 Specifying Requirements

There are a number of ways of specifying requirements. Perhaps the simplest is to require each primitive action to specify its requirements. Unfortunately this is less flexible since it does not allow the user to indicate that a plan, perhaps combining a number of actions, has certain requirements. We thus extend the language with a construct $\tau(\pi, R)$ which indicates that the plan π is tagged ("τ") with requirements R. It is still possible to annotate actions directly, $\tau(a, R)$, but it is no longer the only place where requirements may be noted.

However, in some cases, the requirements of a goal or plan can only be determined in context. For example, the fuel consumed in moving to a location depends on the location, but also on the current location, which is not known ahead of time. We thus provide a second mechanism for dynamic tagging where the requirements of a goal/plan are provided in terms of a procedure that computes the requirements, and a condition that indicates when the procedure should be re-run. This is denoted $\tau(\pi, f, c)$ where f is a function that uses the agent's beliefs to compute the requirements, and c is a recomputation condition. Once the requirements have been propagated (see next section) this becomes $T(\pi, R, f, c)$ (the difference between τ and T is discussed in Section 3.2) we need to retain f and c so the requirements can be re-computed (if c becomes true). Otherwise $T(\pi, R, f, c)$ behaves just like $T(\pi, R)$.

We define R as being a pair of two sets, $\langle L, U\rangle$, representing a lower and upper bound. For convenience, where a requirement R is written as a set $R = \{\ldots\}$ then it is taken to denote the pair $\langle R, R\rangle$. Each of the sets can be defined in many ways, depending on the needs of the domain and application. Here we define each set as containing a number of the following requirement statements:

- $re(r/c, t, n)$ where the first argument in the term is either r or c, denoting a reusable or consumable resource, t is a type (e.g. fuel), and n is the required amount of the resource.
- $pr(c)$ where c is a condition that must be true at the *start* of execution (i.e. a pre-condition)
- $in(c)$ where c is a condition that must be true *during* the *whole* of execution (including at the start). For the computation of summaries we also define a variant in_s which means that c must be true *somewhere* during the execution but not necessarily during the whole execution.

In the Mars rover example we have the following requirements:

1. $goto(L)$ computes its requirements based on the distance between the destination and current location. This needs to be re-computed after each goto. We thus specify the requirements of the $goto(L)$ action as $\tau(goto(L), f(L), c)$ where $f(L)$ looks up the current location $locn$ in the belief base, and then computes the distance between

it and L; and where $c = \Delta locn(x)$ (informally, Δc means that the belief base has changed in a way that affects the condition c; formally, if B is the old belief base and B' the updated belief base, then $\Delta c \equiv \neg(B \models c \Leftrightarrow B' \models c)$);
2. $sample(X)$ requires that the rover remains at the desired location, hence we specify an in-condition (in) that the location ($locn$) remains L: $\tau(sample(X), \{in(locn(L))\})$.

We thus provide requirements by specifying the following plan body (for the first plan), where f is a function that is given a location L and computes the fuel required to reach the location.
$\tau(goto(L), f(L), c); \tau(sample(X), \{in(locn(L))\})$

3.2 Propagating Requirements

We define a function Σ that takes a plan body and tags it with requirements by propagating and aggregating given requirements. We use ε to denote the empty requirement. The function returns a modified plan which contains tags of the form $T(\pi, R)$: this is different from $\tau(\pi, R)$ in that τ is used by the user to provide requirements for a plan, not including the requirements of the plan's sub-plans, but T *does* include the requirements of sub-plans. Observe that Σ is defined compositionally over the plan, and that computing it is not expensive in the absence of recursive plans [5].

$\Sigma(\epsilon) = T(\epsilon, \varepsilon)$
$\Sigma(a) = T(a, \{pr(pre_a)\})$
$\Sigma(e) = T(e, \langle L_1 \sqcap \ldots \sqcap L_n, U_1 \sqcup \ldots \sqcup U_n \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
and $\pi_1 \ldots \pi_n$ are the plans relevant for e.
$\Sigma(\pi_1; \pi_2) = T(\pi'_1; \pi'_2, \langle L_1 \fatsemi L_2, U_1 \fatsemi U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\pi_1 || \pi_2) = T(\pi'_1 || \pi'_2, \langle L_1 \,[\!]\, L_2, U_1 \,[\!]\, U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\pi_1 \triangleright \pi_2) = T(\pi'_1 \triangleright \pi'_2, \langle L_1, U_1 \fatsemi U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\langle\!|\Gamma|\!\rangle) = T(\langle\!|\Gamma'|\!\rangle, \langle L_1 \sqcap \ldots \sqcap L_n, U_1 \sqcup \ldots \sqcup U_n \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
and $\Gamma = \{b_1{:}\pi_1, \ldots, b_n{:}\pi_n\}$ and $\Gamma' = \{b_1{:}\pi'_1 \ldots, b_n{:}\pi'_n\}$.
$\Sigma(\tau(\pi, \langle L, U \rangle)) = T(\pi', \langle L' \oplus L, U' \oplus U \rangle)$, where $T(\pi', \langle L', U' \rangle) = \Sigma(\pi)$
$\Sigma(\tau(\pi, f, c)) = \Sigma(T(\pi, f(B), f, c))$, where B is the agent's beliefs.
$\Sigma(T(\pi, R)) =$ if $\pi' = T(\pi'', \varepsilon)$ then $T(\pi'', R)$ else π', where $\pi' = \Sigma(\pi)$

The requirements of an action are simply its pre-condition. The requirements of an event are computed by taking the requirements of the set of relevant plans and combining them: the best case is the minimum of the available plans ($\sqcap$), and in the worse case (represented by the upper bound) we may need to execute all of the plans and so we take the (unspecified sequential) maximum of the requirements of the available plans using $\sqcup$. The requirements for $\pi_1; \pi_2$ and $\pi_1 \| \pi_2$ are determined by computing the requirements of π_1 and of π_2 and then combining them appropriately with auxiliary functions $\fatsemi$ and $[\!]$ which are both variants on $\sqcup$: $\fatsemi$ treats pre-conditions of the first requirement set as pre-conditions, since we *do* know that they occur at the start of execution; and $[\!]$ is like $\sqcup$ except that, because execution is in parallel, we cannot reuse resources. The lower bound requirements for $\pi_1 \triangleright \pi_2$ are just the (lower bound) requirements for π_1, since, if all goes well, there will be no need to execute π_2. However, in the worse case (upper bound) both π_1 and π_2 will need to be executed (sequentially, hence $\fatsemi$). The

requirements for a user-tagged plan, $\tau(\pi, R)$ are determined by computing the requirements of π and then adding ($\oplus$) this to the provided R. Finally, when re-computing the requirements of $T(\pi, R)$ we simply replace R with the newly computed requirements.

The case for events (e) is interesting: in the worst case, we may need to execute all of the available plans. In the best case, only one plan will be executed. However, when computing the initial estimate of requirements we don't know which plans are applicable, so we overestimate by using all relevant plans, and later update the estimate (see below).

The function Σ is defined in terms of a number of auxiliary functions: $\oplus$, $\sqcup$, $\sqcap$, $\wr$ and $[]$. By defining what can appear in R as well as these functions the agent designer can create their own types of reasoning. We have defined an example R in the previous section, and briefly and informally given the intended meaning of the auxiliary functions above. The appendix contains precise formal definitions of these auxiliary functions.

We integrate requirements propagation into the operational semantics of CAN by defining operational semantics for the $T(\pi, R)$ construct which captures the process of updating requirements:

$$\frac{\pi \Rightarrow \pi' \quad \pi \neq \epsilon \quad R \neq \varepsilon}{T(\pi, R) \Rightarrow \Sigma(\pi')} \qquad \frac{\pi \neq \epsilon}{T(\pi, \varepsilon) \Rightarrow \pi} \qquad \frac{}{T(\epsilon, R) \Rightarrow \epsilon}$$

The first rule is the general case: if π executes one step to π' then $T(\pi, R)$ can also be stepped to $\Sigma(\pi')$. The next rule specifies that tagging with an empty requirement set can be deleted. The final rule allows an empty plan body with requirements to be resolved to the empty plan body. Finally, we modify the goal initialisation rule (first rule in figure 1) to compute requirements by replacing the right-most π_0 with $\Sigma(\pi_0)$:

$$\frac{S = \text{ACTIVE}}{\mathsf{g}(C, \pi_0, S, \epsilon) \stackrel{e}{\Rightarrow} \mathsf{g}(C, \pi_0, S, \Sigma(\pi_0))}$$

Alternatively, as is done by Thangarajah *et al.* [6], we could modify the agent's plan set by replacing π with $\Sigma(\pi)$ at compile time.

Returning to the Mars rover, let $\pi = \tau(goto(L), f, c); \tau(sample(X), \{in(locn(L))\})$ then the following requirements are computed (recall that $T(\pi, R)$ where R is a set is short for $T(\pi, \langle R, R \rangle)$, and we assume that f returns 20 for the fuel requirement of reaching L from the starting location):

$$\begin{aligned}
\Sigma(\pi) &= T(\pi_2; \pi_3, \{re(c, \mathit{fuel}, 20), in_s(locn(L)), pr(\neg locn(L))\}) \\
\pi_2 &= T(goto(L), \{re(c, \mathit{fuel}, 20), pr(\neg locn(L))\}, f, c) \\
\pi_3 &= T(sample(X), \{in(locn(L))\})
\end{aligned}$$

After the first action (π_2) has completed we have: $\Sigma(\pi') = \pi_3$.

4 Using Requirements to Deal with Interactions

The work of the previous sections allows us to specify requirements, and to compute and update them. In this section we consider how this information can be used to avoid

undesirable interactions and (attempt to) ensure desirable interactions. For goals, we do this by defining a new goal type, an "interaction-aware goal", which has additional condition-response triples. But first, we define a number of new conditions, and new responses.

4.1 Conditions

The language of conditions is extended with new constructs: rok ("resources are ok"), $culprit$, and $interfere$.

The new condition $rok(G)$ means that there are enough resources for all of the goals in G. Informally we define $rok(G)$ by computing the resource requirements of the active goals in G and comparing it with the available resources. If the available resources exceed the resource requirements of the active goals, then clearly $rok(G)$ is true. If not, then we need to work out which goals should be suspended. We define the condition $culprit(g)$ to indicate that the goal g is responsible for a lack of sufficient resources. Informally, $culprit(g)$ is true if removing g from G makes things better[2] i.e., $culprit(g) \equiv rok(G \setminus \{g\}) \wedge \neg rok(G)$. See the appendix (definitions 4 and 5) for the (correct) formal definitions of rok and $culprit$.

The new condition $interfere(g)$ is true if g is about to do something that interferes with another goal. Informally, this is the case if one of the actions that g may do next (denoted $na(g)$, defined in the appendix) has an effect that is inconsistent with another goal's in-condition (where both goals are active). Formally[3] $interfere(g) \equiv \exists g' \in (G \setminus \{g\}), c \in getin(g'), a \in na(g)\ .\ g.S = g'.S = \textsc{Active} \wedge \mathit{eff}_a \supset \neg c$ where G is the agent's goals, we use $g.S$ to refer to the state of the goal g, we use $\supset$ to denote logical implication (to avoid confusion with transitions), and we define $getin$ to return the in-conditions of a goal, plan or requirements set:

$$\begin{aligned} getin(\mathsf{g}(C, \pi_0, S, \pi)) &= getin(\pi) \\ getin(T(\pi, \langle L, U\rangle)) &= \{c \mid in(c) \in L\} \\ getin(\pi) &= getin(\Sigma(\pi)), \text{ if } \pi \neq T(\pi', R) \end{aligned}$$

We also define eff_a to be a logical formula combining add_a and del_a as a conjunction of atoms in add_a and the negations of atoms in del_a. For example, for $goto(L)$ we have $\mathit{eff}_{goto(L)} = locn(L) \wedge \neg locn(x)$.

We can also define a similar condition that detects interference with pre-conditions. In order to avoid suspending goals unnecessarily, we only consider interference to be real if the pre-condition being affected currently holds. In other words, if the precondition c of goal g' does *not* currently hold, then there is not a strong reason to suspend goal g which makes c false because c is *already* false. This gives $interfere_{pre}(g) \equiv \exists g' \in (G \setminus \{g\}), c \in getpre(g'), a \in na(g)\ .\ B \models c \wedge g.S = g'.S = \textsc{Active} \wedge \mathit{eff}_a \supset \neg c$ where $getpre$ retrieves the pre-conditions, similarly to $getin$ (see appendix definition 2).

---

[2] In fact, as discussed in the appendix, this isn't entirely correct.

[3] We use a period "." to denote "such that", i.e. $\exists a \in A, b \in B\ .\ p$ is read as "there exists a in A and b in B such that p".

4.2 Responses

Responses to interactions can be either "subtle": influencing existing choices, but not changing the semantics, i.e. "subtle" responses can be viewed as refining the semantics by reducing non-determinism. Alternatively, responses can be "blunt" responses which change the semantics.

So-called "subtle" responses apply where there is a choice to be made in the execution. This is the case in the following places: when selecting which (top-level) goal to execute, when selecting which plan to use from a set of alternatives ($(\!|\Gamma|\!)$), and when selecting which parallel plan to execute ($\pi_1 \| \pi_2$). Note that only the first case involves goals: the second and third involve plan bodies.

Influencing the choice of goal can be done by a range of means, including suspending goals and giving certain goals higher priority. Suspending goals can be done using the generic goal mechanism. In order to allow a goal to be given a higher priority we define a new *response* (not goal state) PICKME (below).

Influencing the selection of a plan from a set of possible plans ($(\!|\Gamma|\!)$) can be done by modifying the selection rule (it can't be done using the generic goal mechanism because plan selection occurs within a single goal). For example, we could require that an applicable plan is not selected if a cheaper plan exists. This can be formalised by adding to the rule for plan selection the following additional condition (the relation $\prec$ and function $getres$ are defined in the appendix in definitions 1 and 3):

$$\frac{(c_i{:}\pi_i) \in \Gamma \quad B \models c_i\theta \quad \neg\exists (c_j{:}\pi_j) \in \Gamma . getres(\pi_j) \prec getres(\pi_i)}{(\!|\Gamma|\!) \stackrel{e}{\Rightarrow} \pi_i\theta \triangleright (\!|\Gamma \setminus \{c_i{:}\pi_i\}|\!)}$$

However, we do not consider plan selection to be particularly useful in preventing resource issues, because the set of applicable plans will typically not contain a wide range of options.

The third case, influencing the scheduling of parallel plans ($\pi_1 \| \pi_2$) we consider to be less useful and leave it for future work.

Turning now to the so-called "blunt" responses we have a number of possible responses including: (a) dropping a goal, and (b) adding a new goal. The former may be used to permanently eliminate a goal that cannot be achieved (although suspension may be a more sensible response). The second may be used to create a new goal (or plan), for example, if a resource shortage is detected, a plan may be created to obtain more of the resource (e.g. re-fuelling).

We thus define the following additional responses:

- $!\pi$ which executes π (we can define synchronous and asynchronous variants of this)
- PICKME which specifies that this goal should be given priority when selecting which goal to execute (but, since more than one goal may be flagged as PICKME, cannot guarantee that the goal will be selected next). More generally, we could have a priority mechanism and have responses that raise/lower the priority of the goal.

These are defined formally as follows. Although they appear in condition-response triples, the semantics of these two constructs aren't just changing the state of the goal,

and so we revise the existing rule so it does not apply to these two responses:

$$\frac{\langle c, S', f\rangle \in C \quad S' \in asd \quad B \models c \quad S \neq S' \quad ok(f, \pi)}{\mathsf{g}(C, \pi_0, S, \pi) \overset{u}{\Rightarrow} \mathsf{g}(C, \pi_0, S', \pi)}\ 3'$$

where $asd = \{\text{ACTIVE}, \text{SUSPENDED}, \text{DROPPED}\}$.

Because we want a PICKME to only last while the corresponding condition is true, we do not update the goal's state to PICKME, but instead modify the selection rule (rule 4) by adding the following additional condition (premise, where we use $\supset$ to denote logical implication) which requires that if any active goals are prioritised, then the selected goal must be a prioritised one: $(\ (\exists \mathsf{g}(C', \pi_0', \text{ACTIVE}, \pi') \in G\ .\ \langle c', \text{PICKME}\rangle \in C' \ \wedge\ B \models c') \supset (\langle c, \text{PICKME}\rangle \in g.C \wedge B \models c)\ )$. Where g is the goal being selected, and where we use $g.C$ to denote the C set of g (i.e. $g = \mathsf{g}(C, \pi_0, S, \pi)$).

We now turn to $!\pi$. A response of the form $!\pi$ transforms the goal from $\mathsf{g}(C, \pi_0, S, \pi')$ to the variant $\mathsf{g}_\pi(C, \pi_0, S, \pi')$:

$$\frac{\langle c, !\pi\rangle \in C \quad B \models c}{\mathsf{g}(C, \pi_0, S, \pi') \overset{u}{\Rightarrow} \mathsf{g}_\pi(C, \pi_0, S, \pi')}\ 16$$

We then define the semantics of this as follows:

$$\frac{\pi \overset{e}{\Rightarrow} \pi_1}{\mathsf{g}_\pi(C, \pi_0, S, \pi') \overset{e}{\Rightarrow} \overline{\mathsf{g}_{\pi_1}(C, \pi_0, S, \pi')}}\ 17$$

where $\overline{\mathsf{g}_\epsilon(C, \pi_0, S, \pi)} = \mathsf{g}(C, \pi_0, S, \pi)$, and for $g = \mathsf{g}_\pi(\ldots)$ with $\pi \neq \epsilon$ we have $\overline{g} = g$.

4.3 Interaction-Aware Goals

Finally, we are in a position to define a new goal type which uses the conditions and responses defined, along with the underlying infrastructure for specifying and propagating requirements, in order to deal with interactions as part of the agent's goal reasoning process.

We extend goals into interaction-aware goals by simply adding to their C set the following condition-response triples, where $culprit$ is short for $culprit(g)$ with g being the current goal, and similarly for $interfere$. The condition $notculprit$ differs from $\neg culprit$ in that it includes the current goal g in the computation of resources (whereas $culprit$ treats it as not having any resource requirements, since it is suspended). Formally $notculprit(\mathsf{g}(C, \pi_0, \text{SUSPENDED}, \pi)) \equiv \neg culprit(\mathsf{g}(C, \pi_0, \text{ACTIVE}, \pi))$. Similarly, $notinterfere$ differs from $\neg interfere$ by considering the current goal as being hypothetically active, i.e. $notinterfere(\mathsf{g}(C, \pi_0, \text{SUSPENDED}, \pi)) \equiv \neg interfere(\mathsf{g}(C, \pi_0, \text{ACTIVE}, \pi))$.

$$\mathcal{I} = \{\langle culprit, \text{SUSPENDED}, all\rangle, \langle notculprit, \text{ACTIVE}, all\rangle,$$
$$\langle interfere, \text{SUSPENDED}, all\rangle, \langle notinterfere, \text{ACTIVE}, all\rangle\}$$

An alternative, if there is a plan π_r which obtains more of a needed resource, is to use it instead of suspending: $\mathcal{I}' = \{\langle culprit, !\pi_r, all\rangle, \ldots\}$.

5 Motivating Scenarios Revisited

We now consider how the different forms of reasoning discussed at the outset can be supported. We define

$$gexp(l, x) \equiv g(\mathcal{I} \cup \{\langle locn(l), \textsc{PickMe}, all\rangle\}, exp(l, x))$$

that is, $gexp(l, x)$ is an interaction-aware goal which uses the initial plan body (which is actually just an event) $exp(l, x)$. Finally, we suppose that the Mars rover has been asked to perform three experiments: experiment 1 of type T_1 at location L_A (i.e. $g_1 = gexp(L_A, T_1)$) experiment 2 of type T_1 at location L_B (i.e. $g_2 = gexp(L_B, T_1)$), and experiment 3 of type T_2 at location L_A (i.e. $g_3 = gexp(L_A, T_2)$).

Let us now briefly consider how the Mars rover deals with the following cases of interaction:

1. **A lack of resources causes a goal to be suspended, and, when resources are sufficient, resumed:** since the goals are interaction-aware, suspension and resumption will occur as a result of the conditions-responses in $\mathcal{I}$. Specifically, should the resources available be insufficient to achieve all goals, then some of goals will be suspended by the $\langle culprit, \textsc{Suspended}, all\rangle$ condition-response triple. Note that since updates are performed one at a time, this will only suspend as many goals as are needed to resolve the resource issue.
   If further resources are obtained, then the suspended goals will be re-activated ($\langle notculprit, \textsc{Active}, all\rangle$). In the case of reusable resources, the suspension/resumption mechanism will realise scheduling of the reusable resources amongst goals: once a goal has completed and releases the (reusable) resources it has been using, another goal that requires these resources can then resume.
2. **A lack of resources, instead of suspending, may trigger a plan to obtain more resources:** if the goals are defined using $\mathcal{I}'$ rather than $\mathcal{I}$, then a lack of resources will cause a plan body π_r to be used to obtain more resources. In this domain, where the main resource is fuel, a sensible choice for π_r would be to re-fuel.
3. **Once the Mars rover has moved to location L_A, it avoids moving again until the sampling at L_A has completed:** once goal g_1 has executed $goto(L_A)$ then, as discussed at the end of Section 3.2, its requirement is updated to include the in-condition $locn(L_A)$. Should goal g_2 get to the point of being about to execute its action $goto(L_B)$, then this next action interferes with the in-condition, and goal g_2 will then be suspended, using the condition-response triple $\langle interfere, \textsc{Suspended}, all\rangle$, preventing the execution of $goto(L_B)$. Once g_1 has concluded the experiment, then it no longer has $locn(L_A)$ as an in-condition, and at this point g_2 will be re-activated ($\langle \neg interfere, \textsc{Active}, all\rangle$).
4. **Once it has moved to location L_A, the rover also performs g_3 before moving elsewhere:** when it reaches L_A the PICKME response of g_3 (and g_1) is triggered which prioritises selecting these goals over g_2, and thus the rover will remain at L_A until g_1 and g_3 are both completed.

As can be seen, interaction-aware goals — which are defined in terms of the additional condition and response types, which themselves rest on the resource specification and propagation mechanism defined in Section 3 — are able to deal with a range of goal-interaction scenarios.

6 Discussion

We have provided a framework for reasoning about goal interactions that is: **generic**, i.e. can be customised to provide the reasoning that is needed for the application at hand; **presented formally**, and hence precisely, avoiding the ambiguity of natural language; and that **integrates** different reasoning types into one framework. We have also defined a wider range of conditions and responses than previous work.

Our work can be seen as a rational reconstruction of earlier work [6–8] which formalises and makes precise the English presentation in these papers. However, we do more than just formalise existing work: we provide a generic framework that allows for other forms of reasoning to be added, and for the existing forms to be integrated.

In addition to work on reasoning about interactions between an agent's goals, there has also been work on reasoning about interactions between the goals of different agents [15, 16]. This work has a somewhat different flavour in that it is concerned with the cost of communication between agents. However, in some other aspects, such as the use of requirements summaries, it is similar to the single agent case.

Also related is the work by Horty and Pollack [17] which looked at the cost of plans in context (i.e. taking into account the agent's other plans). Although the paper is ostensibly concerned with cost, they do also define various notions of compatibility between plans. However, their plans are composed only of primitive actions.

Thangarajah *et al.* [18] consider the goal adoption part of goal deliberation: should a candidate goal (roughly speaking, a desire) be added to the agent's set of adopted goals? They embed the goal adoption problem in a BDI setting into a soft constraint optimisation problem model and discuss a range of factors that can be taken into account in making decisions. However, while promising, this is early work: the presentation is informal and a precise definition of the mapping to soft constraint optimisation problems is not given.

There are three main directions for future work that we would like to pursue: implementation, evaluation, and extending to further interaction scenarios.

What this paper presents can be seen as an extended BDI programming language with interaction-aware goals. One area for future work is how to implement this extended language using a standard BDI platform (such as Jason, Jadex, JACK etc.) that doesn't have a generic goal construct, or resource/condition management. One possibility is to transform the agent program, Π, into a variant that uses existing constructs (such as maintenance goals) to realise the desired behaviour. Another possibility, if the platform provides an API for manipulating the state of goals, is to realise generic goals by two parallel goals: one that executes the plan π, and another (with higher priority) that monitors for conditions and updates the first goal's state. Finally, a third approach is to use a meta-interpreter [19]. An implementation would allow for an evaluation to be done in order to assess the benefits, and also the real practical computational cost.

An interesting scenario which we have not yet investigated is "achieve then maintain", where a particular condition is achieved (e.g. booking a hotel), but then for some period of time (e.g. until the travel dates) the condition is maintained and updated should certain changes take place (e.g. budget reductions or changes to travel dates).

References

1. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: A unifying framework. In: Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2008) 713–720
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007) ISBN 0470029005.
3. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia (1998) Available from *http://www.agent-software.com*.
4. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
5. Thangarajah, J., Padgham, L.: Computationally effective reasoning about goal interactions. Journal of Automated Reasoning (2010) 1–40
6. Thangarajah, J., Winikoff, M., Padgham, L., Fischer, K.: Avoiding resource conflicts in intelligent agents. In van Harmelen, F., ed.: Proceedings of the 15th European Conference on Artificial Intelligence, IOS Press (2002) 18–22
7. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and exploiting positive goal interaction in intelligent agents. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), ACM Press (2003) 401–408
8. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and avoiding interference between goals in intelligent agents. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI). (2003) 721–726
9. Shaw, P.H., Bordini, R.H.: Towards alternative approaches to reasoning about goals. In Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M., eds.: Declarative Agent Languages and Technologies (DALT). (2007) 164–181
10. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), Toulouse, France (2002) 470–481
11. Sardiña, S., Padgham, L.: A BDI agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems **23**(1) (2011) 18–70
12. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W.V., Perrame, J., eds.: Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Springer Verlag (1996) 42–55 LNAI, Volume 1038.
13. Sardina, S., Padgham, L.: Goals in the context of BDI plan failure and planning. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2007) 16–23
14. Plotkin, G.: Structural operational semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University (1981 (reprinted 1991))
15. Clement, B.J., Durfee, E.H.: Identifying and resolving conflicts among agents with hierarchical plans. In: AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Technical Report WS-99-12. (1999)
16. Clement, B.J., Durfee, E.H.: Theory for coordinating concurrent hierarchical planning agents using summary information. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence. (1999) 495–502
17. Horty, J.F., Pollack, M.E.: Evaluating new options in the context of existing plans. Artificial Intelligence **127**(2) (2001) 199–220

18. Thangarajah, J., Harland, J., Yorke-Smith, N.: A soft COP model for goal deliberation in a BDI agent. In: Proceedings of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef). (September 2007)
19. Winikoff, M.: An AgentSpeak meta-interpreter and its applications. In: Third International Workshop on Programming Multi-Agent Systems (ProMAS), Springer, LNCS 3862 (post-proceedings, 2006) (2005) 123–138

A Definitions

Definition 1 ($\prec$). *We define an ordering on requirement sets as follows. We say that R_1 is less than R_2 ($R_1 \preceq R_2$) if, intuitively, R_2 requires more than R_1. Formally, we define this by recognising that for a given condition c we have that $in_s(c) \preceq pr(c) \preceq in(c)$, i.e. a requirement that a condition hold for some unspecified part of the execution is less demanding than insisting that it hold at the start, which in turn is less demanding than insisting that it hold during the whole of execution (including at the start). We thus define $R_1 \preceq R_2$ to hold iff:*

- $re(f,t,n_1) \in R_1 \supset (r(f,t,n_2) \in R_2 \wedge n_1 \leq n_2)$
- $in(c) \in R_1 \supset (in(c') \in R_2 \wedge c' \supset c)$
- $pr(c) \in R_1 \supset ((pr(c') \in R_2 \vee in(c') \in R_2) \wedge c' \supset c)$
- $in_s(c) \in R_1 \supset ((in_s(c') \in R_2 \vee pr(c') \in R_2 \vee in(c') \in R_2) \wedge c' \supset c)$

We next define na ("next action") which takes a plan body and returns a set of possible next actions. Note that na is an approximation: it doesn't attempt to predict which actions might result from a set of plans (Γ). A more accurate approach is to wait until an action is about to be executed before checking for interference.

$$\begin{aligned} na(a) &= \{a\} \\ na(\pi_1;\pi_2) &= na(\pi_1) \\ na(\pi_1 \| \pi_2) &= na(\pi_1) \cup na(\pi_2) \\ na(\pi_1 \triangleright \pi_2) &= na(\pi_1) \\ na(e) &= \emptyset \\ na((\Gamma)) &= \emptyset \end{aligned}$$

Definition 2 ($getpre$). *$getpre$ returns the pre-condition of a goal/plan.*

$$\begin{aligned} getpre(\mathsf{g}(C,\pi_0,S,\pi)) &= getpre(\pi) \\ getpre(T(\pi,\langle L,U\rangle)) &= \{c \mid pr(c) \in L\} \\ getpre(\pi) &= getpre(\Sigma(\pi)), \text{ if } \pi \neq T(\pi',R) \end{aligned}$$

Definition 3 ($getres$). *Calculating resource requirements only uses active goals, we ignore goals that are suspended or are executing responses triggered by $!\pi$.*

$$\begin{aligned} getres(\mathsf{g}(C,\pi_0,S,\pi)) &= getres(\pi), \text{ if } S = \textsc{Active} \\ getres(\mathsf{g}(C,\pi_0,S,\pi)) &= \varepsilon, \text{ if } S \neq \textsc{Active} \\ getres(\mathsf{g}_\pi(C,\pi_0,S,\pi)) &= \varepsilon \\ getres(T(\pi,\langle L,U\rangle)) &= \{re(f,t,n) \mid re(f,t,n) \in U\} \\ getres(\pi) &= getres(\Sigma(\pi)), \text{ if } \pi \neq T(\pi',R) \end{aligned}$$

Definition 4 **(rok).** *In defining $rok(G)$ we need to sum the resource requirements of the set of goals, and then check whether the available resources are sufficient. As discussed by Thangarajah et al. [6], there are actually a number of different cases. Here, for illustrative purposes, we just consider the case where there are sufficient resources to execute the goals freely as being an rok situation. We thus define the collected resource requirements of a goal set $G = \{g_1, \ldots, g_n\}$ as being $getres(G) = U_1 [] \ldots [] U_n$ where $U_i = getres(g_i)$. Finally, we define $rok(G) \equiv getres(G) \preceq \mathcal{R}$ where $\mathcal{R}$ is the available resources.*

Definition 5 **($culprit$).** *In defining $culprit(g)$ one situation to be aware of is where removing a single goal is not enough. In this situation the definition given in the body of the paper will fail to identify any goals to suspend. To cover this case we need a slightly more complex definition. Informally, the previous definition is correct except where there does not exist a single goal that can be removed to fix the resource issue ($\neg\exists g \in G.rok(G \setminus \{g\})$). In this case we consider $culprit(g)$ to be true if removing g and one other goal will fix the problem. This generalises to the situation where one must remove n goals to fix a resource issue:*

$$culprit(g) \equiv \exists n \,.\, (\;\; (\exists G' \subseteq G.|G'| = n \wedge rok(G \setminus G') \wedge \neg rok(G) \wedge g \in G') \\ \wedge \;\; (\neg\exists G'' \subseteq G.|G''| < n \wedge rok(G \setminus G'') \wedge \neg rok(G)))$$

We now turn to defining the various auxiliary functions that are needed. We assume that requirements definitions, R_i, are *normalised*, i.e. that they contain (a) exactly one $re(f, t, n)$ for each resource type t that is of interest (where n may be 0); and (b) exactly one in, one in_s and one pr. We also assume that resource reusability is consistent, i.e. that a resource type t is not indicated in one place as being consumable and in another as being reusable.

The intended meaning of the auxiliary functions (based on where they are used in the definition of Σ) is as follows: $\oplus$ adds resources without changing the intervals; $\sqcup$ is used to collect the upper bound for a set of plans which are executed sequentially in an unknown order; $\sqcap$ computes the minimal (lower bound) requirements of a set of alternative plans; $⨟$ corresponds to a sequential join of two intervals, and $[]$ corresponds to the parallel composition of two intervals. Formally, they are defined as follows:

$$\begin{aligned} &R_1 \oplus R_2 = \\ &\quad \{re(f, t, n_1 + n_2) \mid re(f, t, n_1) \in R_1 \wedge \; re(f, t, n_2) \in R_2\} \cup \\ &\quad \{in(c_1 \wedge c_2) \mid in(c_1) \in R_1 \wedge in(c_2) \in R_2\} \cup \\ &\quad \{in_s(c_1 \wedge c_2) \mid in_s(c_1) \in R_1 \wedge in_s(c_2) \in R_2\} \cup \\ &\quad \{pr(c_1 \wedge c_2) \mid pr(c_1) \in R_1 \wedge pr(c_2) \in R_2\} \end{aligned}$$

$$\begin{aligned} &R_1 \sqcup R_2 = \\ &\quad \{re(r, t, \max(n_1, n_2)) \mid re(r, t, n_1) \in R_1 \wedge \; re(r, t, n_2) \in R_2\} \cup \\ &\quad \{re(c, t, n_1 + n_2) \mid re(c, t, n_1) \in R_1 \wedge \; re(c, t, n_2) \in R_2\} \cup \\ &\quad \{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6) \mid in(c_1) \in R_1 \\ &\qquad\quad \wedge in(c_2) \in R_2 \wedge in_s(c_3) \in R_1 \wedge in_s(c_4) \in R_2 \\ &\qquad\quad \wedge pr(c_5) \in R_1 \wedge pr(c_6) \in R_2\} \end{aligned}$$

$$
\begin{aligned}
&R_1 \sqcap R_2 = \\
&\quad \{re(f,t,\min(n_1,n_2)) \mid re(f,t,n_1) \in R_1 \wedge\ re(f,t,n_2) \in R_2\} \cup \\
&\quad \{in(c_1 \vee c_2) \mid in(c_1) \in R_1 \wedge in(c_2) \in R_2\} \cup \\
&\quad \{in_s(c_1 \vee c_2) \mid in_s(c_1) \in R_1 \wedge in_s(c_2) \in R_2\} \cup \\
&\quad \{pr(c_1 \vee c_2) \mid pr(c_1) \in R_1 \wedge pr(c_2) \in R_2\}
\end{aligned}
$$

$$
\begin{aligned}
&R_1 \fatsemi R_2 = \\
&\quad \{re(r,t,\max(n_1,n_2)) \mid re(r,t,n_1) \in R_1 \wedge\ re(r,t,n_2) \in R_2\} \cup \\
&\quad \{re(c,t,n_1+n_2) \mid re(c,t,n_1) \in R_1 \wedge\ re(c,t,n_2) \in R_2\} \cup \\
&\quad \{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5) \mid in(c_1) \in R_1 \wedge\ in(c_2) \in R_2\ \wedge in_s(c_3) \in R_1 \\
&\qquad\qquad \wedge\, in_s(c_4) \in R_2 \wedge pr(c_5) \in R_2\}\ \cup \{pr(c) \mid pr(c) \in R_1\}
\end{aligned}
$$

$$
\begin{aligned}
&R_1 \,[\!]\, R_2 = \\
&\quad \{re(f,t,n_1+n_2) \mid re(f,t,n_1) \in R_1 \wedge\ re(f,t,n_2) \in R_2\} \cup \\
&\quad \{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6) \mid in(c_1) \in R_1 \\
&\qquad\qquad \wedge\, in(c_2) \in R_2 \wedge in_s(c_3) \in R_1 \wedge\ in_s(c_4) \in R_2 \wedge pr(c_5) \in R_1 \wedge pr(c_6) \in R_2\}
\end{aligned}
$$