"From Natural Language to Formal Proof Goal"

Structured goal formalisation applied to medical guidelines.

1

Reduction

- to a purely descriptive form

2

Normalisation

- to a structured natural language version

3

Formalisation

- into the Goal Description Language

4

Attachment

- to the process model to verify

5

Translation

- to the logic of the verification tool

MSc Thesis
Ruud Stegers
March 2006

vrije Universiteit *amsterdam*

Supervisors: Dr. Annette Ten Teije
Prof. Dr. Frank van Harmelen

Artificial Intelligence
Knowledge Management and Knowledge Technology

Faculty of Sciences
Vrije Universiteit
Amsterdam

ABSTRACT

In the Protocure project, medical guidelines are formalised to enable the use of formal verification methods to improve those guidelines. To verify goals given in natural language, a translation is required to the formalism of the verification tool. The main concern is to assure equivalence of the translation and the original. In general this is a problem. When the domain expert and the expert on formal methods are different persons, neither has sufficient knowledge to check this equivalence. A method is required to assure equivalence also in these cases.

This thesis proposes a goal formalisation method in which the domain expert is involved in such a way, that the correctness of the result can be assured. By providing a common conceptual model of goals, the domain expert and the formal method expert share a frame of reference. Both have their own representation of this model: one in natural language, one in a newly introduced formal language GDL: the Goal Definition Language.

Throughout the thesis examples from the medical domain are provided, although the applicability of the method transcends the boundaries of any specific domain. The conclusion shows that the proposed method satisfies amongst others, requirements in area of correctness, traceability, variability and reusability.

ACKNOWLEDGEMENTS

I would like to thank the following people for their role in the realisation of this thesis

- My supervisors Annette ten Teije and Frank van Harmelen for their valuable feedback and the amount of freedom I was granted to proceed in the direction I deemed right.
- Jonathan Schmitt for completing the first proof based on my translation, and for providing 24/7 support, ideas and feedback on KIV and other subjects.
- Michael Balser for his help and insights on the formal semantics and KIV.
- The other people of the Protocure project for valuable discussions and ideas.
- Sebastiaan Spijker for sharing his knowledge and time on graphical design.
- José Hooijer for her input on medical matters, and her general loving support.

TABLE OF CONTENTS

Chapter 1
Introduction

1.1 Formal verification

Formal verification techniques can be used to make sure that a system adheres to specific formal requirements. Different methods are available to perform this task. Among those methods are symbolic execution and model checking. However, to be able to verify a system, goal are needed which the system should adhere to. Often a list of natural language goals is available which can be used for the verification of the system.

To use those natural language goals, a translation is required to the formalism of the verification tool. For time related goals, this may be some form of temporal logic. The main concern is to assure equivalence of the translation and the original. In general this is a problem. When the domain expert and the expert on formal methods are different persons, neither has sufficient knowledge to check this equivalence. A method is required to assure equivalence also in these cases.

1.2 Protocure

The Protocure project uses formalisation techniques on medical guidelines to allow formal verification of those guidelines[9][7]. In medical practice, guidelines are used to set a standard for treatments and to make sure that patients receive the best care for their specific situation: guidelines describe the way medical care should be conducted. Since guidelines are frequently used, errors may result in many instances of sub-optimal treatment. The Protocure project aims at finding techniques to identify errors in order to improve medical care.

No fixed set of goals is present for guidelines. The goals that a specific guideline should comply with may come from different sources, but are without exception in natural language. The problem of equivalence of the formalisation and the original is very real here: many of the sources assume extensive medical knowledge and at the same time, the target formalism — temporal logic — is far beyond the comprehension of any doctor.

In earlier work by Van Gendt [3], it was already decided that an intermediate representation was required to reduce the equivalence problem. The work has focused on using a specific element of Asbru — the formalisation language used in Protocure to formalise guidelines — to this end. However, this *Intention* element turned out to introduce problems of its own. It still required the medical expert to be involved with a formal representation. Also, it turned out that due to lack of formal semantics multiple interpretations of the same expression were in use. The conclusion drawn by Van Gendt was that the *Intention* would at least need to be changed in order to be suitable for the task.

1.3 Problem statement

To make sure that the effort put into the conversion task by the domain experts (i.e. the medical expert) is directed entirely on the content and interpretation of the goal, an easy-to-interpret representation should be used by this group throughout the whole conversion. Additionally, to assure proper and reproducible conversion results, the whole process should be split into smaller, well defined steps.

The process of formalising natural language goals in general, and for the Protocure project specific, will be the subject of this thesis: "What steps are required, and what representation is suitable to formalise natural language goals with the help of a domain expert". The goals of the conversion process itself are: *work towards a canonical form*, *identify an clarify ambiguities*, *provide correct translations*, *ensure traceability*, *enable reusability*, and *reduce variability*. These goal will be further elaborated on page 6 of chapter 2.

1.4 Structure

Chapter 2 will provide an outline of the answer to the problem statement. This chapter will also provide a detailed list of the requirements the whole conversion process should adhere to. Chapter 3, 4, 5 and 6 will provide detailed information on individual steps of the conversion. Throughout those chapters, four example goals from the medical domain will be used to illustrate each step. Finally, chapter 7 will evaluate the results, and test the outcome against the requirements.

The appendixes contain full specification of intermediate languages and their semantics.

Chapter 2
Goals in formal verification: an overview

Formal verification of systems requires not only a formal model of the system, but also one or more goals to verify. Two kinds of goals may be distinguished: general structural goals (reachability of states, termination) and domain related goals. The latter will be the subject of the following chapters.

Domain related goals may come from a domain expert or literature in the form of a natural language description. Those natural language descriptions will - just like the model itself - have to be formalised before they can be used for formal verification. This chapter will point out the difficulties when doing so, and provide a domain independent step-by-step description of the formalisation process.

2.1 Goals and formal verification

The main problem encountered when starting verification of goals for some formal system, is the ambiguity of those goals when they are specified in natural language. No matter what domain, or what source of the goals: there are always a lot of implicit assumptions and interpretations that must be made explicit before they can be used for formal verification. An ad-hoc method, in which the expert on the formal system makes the translation by hand directly into the logic of the target system, may work sometimes, but is error prone due to the obvious domain specific choices and interpretations that have to be made.

Incorporating a domain expert in the conversion process seems to be a necessity, however the gap between the natural language representation and the logic is far to big to close without help. A structured method, understandable for the domain expert and flexible enough for the formal methods expert seems to be required here. The domain expert should only be looking at a (structured) natural language version of the goal at hand, while the formal expert maintains a formal representation of the goal that is so close to the natural language that no errors are introduced in the mapping between those two. That way, at the end of the formalisation process both experts can be assured that the final goal is significant in the domain, and that the formal representation indeed means what it should mean.

To accommodate this process, both experts need some shared model of how goals can be expressed. They can then talk about the goal in terms of this general model, each falling back on their own representation of it for the details. In the next section such a model is presented and explained. In section 2.3, a global overview is given of the structured method to express a goal in terms of this model and how to use this to finally reach the target logic. In the next chapters, the individual steps are explained more thoroughly. Since all these steps are basically domain independent, in this chapter they will be described as such.

2.2 The Goal model

There are some considerations where taken into account designing the shared goal model.

1. **Ease of interpretation** The model must be easy to explain, so it can be understood and interpreted with only little training. It should enable both experts to reason about goals and discus these goals using a common structure and vocabulary.
2. **Expressible** The chosen goal model should have a simple mapping to both natural language and to a formal representation. The gap between the domain experts and experts in formal verification should naturally be bridged by the model.
3. **Expressive power** The model must be powerful enough to express a large variety of goals.
4. **Translatable** The model should be easy to translate into many target formalisms.

2.2.1 The observer

The chosen common goal model can best be understood in terms of an external observer. The observer monitors the execution of the process model that needs to be verified and continuously compares the observed behaviour to what is required by the goal. If behaviour is observed that does not adhere to the goal, the model is considered inconsistent with the goal: the verification fails. The actual job of the observer starts when some start event is observed. The observation of this start event only 'counts' when the circumstances are right: the state of the model should meet some conditions that are also specified in the goal description. After a valid start event has been observed, the model should strictly adhere to some prescribed behaviour until the observer sees some end event. A goal is thus expressed in terms of a start event, a pre-condition for this event, an end event, and some desired behaviour for in between.

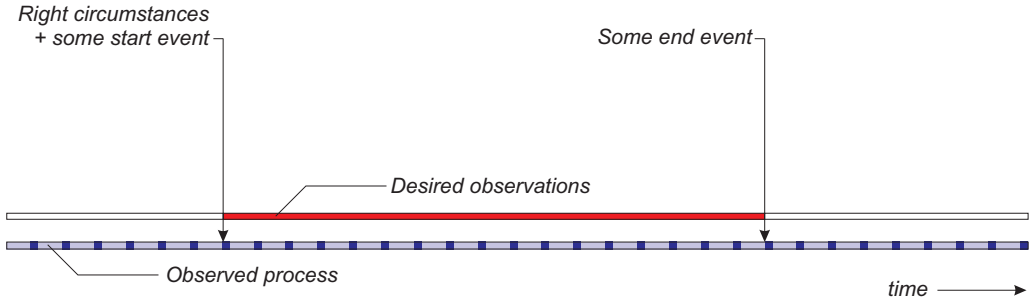


Figure 2.1: Shared goal model

Figure 2.1 illustrates this model. The blue line denotes the execution of the model that is being observed. After the start event has been observed under the

right circumstances, the red bar shows that the model should adhere strictly to the desired behaviour. The red bar stops at the moment the end event has occurred. From that point on the behaviour is no longer important: left and right of the event arrows the behaviour of the model is not relevant.

The simple nature of the common goal model follows from the first consideration. Despite this simple nature, the specification of the desired behaviour between the start and end event allows for sufficiently expressive goal descriptions. Since the model can easily be written in terms of a state machine, translation into any target formalism should be relatively simple: as long as a state machine can be modelled, the common goal model can also be expressed. The next chapters show that the common goal model also maps naturally to natural language — with that it does indeed brigde the gap between the domain expert and the formal systems expert.

2.2.2 Related work

Several others have worked on goal models in relation to the medical domain. The first work of relevance is the *intention* found in Asbru. Shahar introduces the intention as a high level goal associated with formalised guidelines[11]. The second work of interest is the work of John Fox, who has created an ontology of goals based on goals in breast cancer treatment[2].

Both works are an effort to find a vocabulary to express goals. However, the reasons to do so differ. Below both methods are described and evaluated with respect to the goal model.

Asbru Intention

Asbru's *intention* is part of the Asbru specification. Asbru uses a plan-hierarchy to formalise medical guidelines. An *intention* can be added to plans to specify the intended outcome of the plan (and its sub-plans). In the case of Asbru, the *intention* was originally meant to aid the plan selection process in a real-time environment. By adding information about the intentions and effects of plans, it should be possible to reuse plans in other environments.

For the *intention*, a tripled of words is used to describe the behaviour:

$$\left\{ \begin{array}{c} achieve \\ avoid \\ maintain \end{array} \right\} \times \left\{ \begin{array}{c} intermediate \\ overall \end{array} \right\} \times \left\{ \begin{array}{c} action \\ state \end{array} \right\}$$

It starts with a verb, followed by a temporal specification relative to the plan: goals using *intermediate* are applicable during the execution of the plan. The *overall* keyword denotes the end of the plan. Finally, the behaviour is either about a *state* or about an *action*. For example *achieve overall state* means that some state (condition) must be true at the end of the plan.

Although originally not designed for that reason, an attempt was made by Van Gendt [3] to express verification goals using the intention. For several reasons this attempt failed. Some of the problems where related to difficulties associating the concepts in the goal with the concepts in the process model. Since this problem

is to a large extend independent of the chosen language, this demonstrates that apart from a vocabulary, also a methodology is needed. Another type of problem however, were problems with the semantics of the *intention*. First of all, no formal semantics have been defined. Only some examples are available to demonstrate the use of the *intention*. Therefore, the semantics must follow from the structure and the words used. Unfortunately, the rigid nature of the triplets do not allow a suitable assignment of meaning according to the natural meaning of the words. Therefore, several combinations have been assigned non-intuitive meanings. Additionally, in the body of the intention, extra time annotations can be used. Although in theory this should enrich the expressive power, in practice a uniform interpretation has become very difficult to provide.

Based on these experiences, the conclusion was drawn that a simple, clear conceptual goal model is required for a successful translation. Additionally, a formal expression language is required which is based on this model, and of which the meaning stays very close to the natural meaning of the words used. The formal language that comes with the goal model of this thesis therefore contains specific expressions as keywords, exactly tailored to match the behaviour they describe.

Goal Ontology

In order to understand the requirements for modelling and managing clinical goals in breast cancer care, John Fox has studied CREDO, a decision support system for cancer care. The aim of formalising the goals in that system is to be able to recover from failed goals and clinical intervention.

The resulting ontology discriminates between *knowledge goals* and *action goals*. The knowledge goals are subdivided in *information acquisition* and *decisions between alternative hypotheses*. Knowledge goal are about handling information and deriving information by reasoning. Therefore verbs like *classify* and *predict* are found here. The action goals are about *achieving some state of the world* and *enacting tasks*. This is about interaction with the world. Verbs like *communicate* and *investigate* can be found in this category.

The chosen subdivision makes perfects sense in the context of internal goals for a decision support system. However, for verification tasks the separation turns out to be artificial, and even difficult to handle. Concepts like 'communicate' may have a very well defined meaning in the context of the given system, however outside that system, is has no natural formal meaning by itself.

The last example shows that a system independent expression language cannot contain any system dependent concepts. Again the goal model satisfies this requirement by providing a framework using universal, abstract concepts like *event* and *condition*. Only when the goal is applied to some specific model, the abstract events and conditions will be turned into concrete elements from the process model.

2.3 Structured conversion

Now that with the goal model, the domain expert and the expert in formal verification has a shared frame of reference, the next step is to develop a structured, step

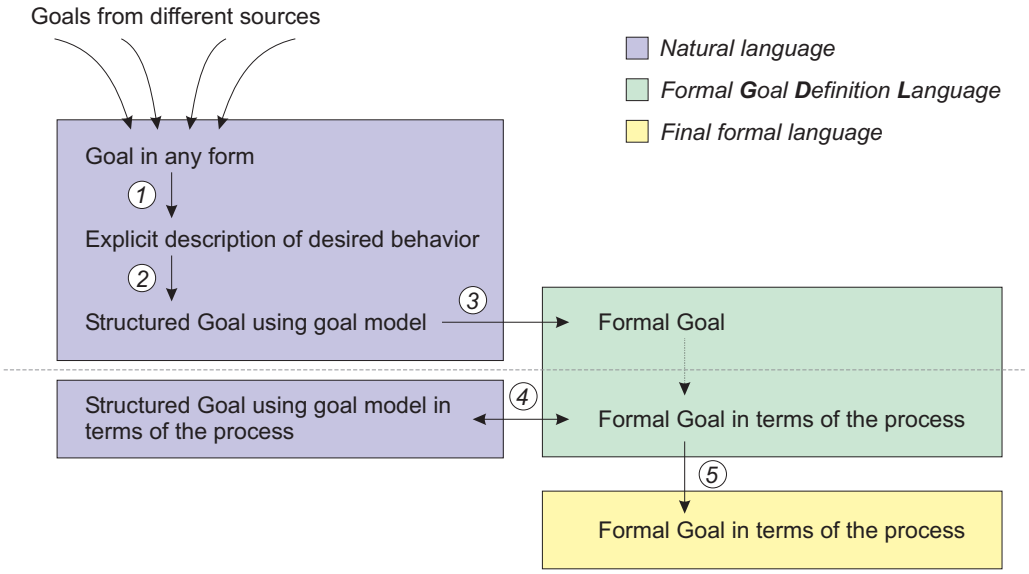


Figure 2.2: Life-cycle of a goal

by step conversion process. The main requirements of such a process are:

i. To direct the conversion process and for ease of interpretation, work towards canonical forms of the goal.

ii. Identify and clarify all assumptions and ambiguities present in the original goal.

iii. Ensure correctness of every change to the goal: the domain expert should be able to validate every change to ensure its validity for the domain.

iv. Ensure traceability. The conversion must be completely reproducible by means of the intermediate results and the documentation.

v. Enable reusability of work at different stages. Maintain generality for as long as possible.

vi. Reduce variability of the conversion result.

To achieve those goals, every domain goal is transformed and translated in five separate steps. This yields a life-cycle for goals which is depicted in figure 2.2. The five numbered arrows correspond to those five conversion steps. The steps are explained in the next section.

2.3.1 The actual steps

To illustrate the initial steps below, an example from the medical domain is provided for clarity. The formal part has been left out in this stage. More detailed examples and more details on the individual steps, are given in the next chapters.

> **Example:**
> "Percentage of people who had a Cholesterol or HDL-cholesterol measurement during the last 12 months."

Reduction ① Independently of the source of the goal, the first step is to make sure that the desired behaviour is explicitly described. This step aims at removing non essential information from the original formulation. (E.g. In the medical domain, *indicators* are goals that are used to measure quality of treatment afterwards. In some cases they don't immediately describe the desired behaviour, but merely specify how it should be measured: "The percentage of patients that .... during the last year". The first step extracts the unit of quality from this notation by explicitly writing down what needs to be observed during the diagnoses and treatment.) The first step is called the reduction step, since the goal is reduced to a purely descriptive form.

> "People should have a Cholesterol or HDL-cholesterol measurement every 12 months."

Normalisation ② After the reduction, the goal is rewritten in terms of behaviour that should be adhered to between some start and some end event: the goal model. The goal is transformed into a normal form here, hence the name 'normalisation step'. This is also the step in which the majority of the ambiguities and implicit assumptions are taken care of. After this step, the temporal relations of the resulting goal should be clear and only be interpretable in one way. It is important that the reduction and the normalisation steps are performed independently of any specific model to verify. It should just be another way of writing down the original goal. By not making model specific adjustments, the normalised goal may be reused for verification of more than one model.

> ^{G}[For people], ^{S}[at earliest 11 months after a cholesterol or HDL-cholesterol measurement], ^{E}[and at latest 13 months after that measurement], ^{B}[another cholesterol or HDL-cholesterol measurement must be performed once]

The groups between brackets correspond to the elements of the goal model. For more information, refer to page 14.

Formalisation ③ The normalised form is now being formalised. This step to *GDL* – the formal *Goal Definition Language* – is small and easy: *GDL* is explicitly designed to reflect the goal model and since the normalised natural language form is also designed to do that, the transition is straight forward. For an example of *GDL* refer to page 23.

Attachment ④ The (model independent) normalisation/formalisation is used in a collaborative effort of both experts to rewrite terms found in the original structured goal, to terms found in the model under investigation. Since the origin of the goals is undefined, there is no way to guarantee that the vocabulary used is the same. This mapping is a creative process requiring both the knowledge about the formalised model, and the domain knowledge of the expert. In this stage, the

goal actually becomes attached to a specific model. Essential to this step is that all changes are reflected both in the *GDL* version as well as in the structured natural language version. That way, both experts are talking about the same (adjusted) goal all the time, and changes proposed by the expert in formal verification, can be evaluated by the domain expert.

Translation ⑤ When a final version of the goal has been found, the remaining task is to translate it from *GDL* to the target formalisation. This should be a mechanical step to prevent undesired additional changes to the goal (which are not evaluated by the domain expert).

2.4 Conversion philosophy

The steps described above already show that the conversion process does not necessarily preserve the exact goal. Especially during the attachment phase, the goal is adjusted to fit on to the model under investigation. Whether or not this is acceptable depends on the goal.

The aim of formal verification is to find errors in a system. If formal requirements are available, then conversion should be possible without significant changes. If not, less precise sources such as natural language descriptions must be used. In the latter case, the conversion process may yield several goals which are different from the original but which still are significant and important goals for the domain. Given the choice between totally discarding a goal, or using an adjusted, but relevant version of it, the latter seems preferable.

Chapter 3
Reduction and Normalisation

In this chapter, the reduction and normalisation step, which were introduced in the previous chapter, are explained in detail. Although all steps are in principle applicable to any domain, some details are more easily explained using a domain specific perspective. Therefore, from this chapter on, the general view is abandoned to be able to provide background information on conversion in the medical domain, the subject of this thesis. Four examples of medical indicators will serve to illustrate the difficulties and techniques. The examples are a selection of real-world goals which will highlight different aspects of the conversion process.

Original - Examples

1. "Percentage of people who had a Cholesterol or HDL-cholesterol measurement during the last 12 months."

2. "Percentage of people with diabetes who have had one or more severe hypoglycaemia during the last 12 months."

3. "Percentage of women with breast cancer who had local recurrence within 5 years after breast-conserving surgery."

4. "The possibility of breast reconstruction should be discussed with all patients prior to mastectomy."

3.1 Reduction

As explained in the previous chapter, the reduction step aims at rewriting the original goal to a form which explicitly describes the desired outcome. It extracts the quality aspect captured by the goal and removes non essential annotations.

An example of such non essential annotations can be found in medical indicators. One style of writing indicators is by describing exactly how the quality should be measured: this style can be found in the first three example indicators. The extra time annotation is the main source of difficulties during the reduction phase. Consider example number 2. After finding out that hypoglycaemia is not a good thing, it may be tempting to rewrite the goal to: "For people with diabetes, hypoglycaemia should be avoided during the next year". However, if adhering to the guideline would cause all diabetes patients to have hypoglycaemia after 366 days, the rewritten goal would be satisfied. Obviously, this is not what the original indicator was aiming at, since in that case it will fail the next year. General rules for handling such time annotations will be provided below.

3.1.1 Handling time annotations

When comparing the first and the second indicator on page 11, there seems to be a high level of resemblance. Both mention 12 months and something that should or should not have happened. There is a crucial difference though, that becomes clear when we try to determine the exact quality aspect captured by these two indicators.

The normal use of an indicator is that — for the purposes of quality measurement in medical practice — once a year, the indicator is evaluated for every patient that was treated. This procedure is repeated every year. When Indicator 1 is measured every year, it tries to enforce that the Cholesterol levels should be measured every 12 months. Rewriting that as a goal yields: "Every twelve months, Cholesterol must be measured".

The second indicator on the other hand, tries to achieve the opposite. It checks that hypoglycaemia has not occurred during that year, and by repetition, every subsequent year. Rewriting that to a goal yields: "hypoglycaemia should never occur". The time annotation has vanished in this case since it has no medical significance. It was only there to accommodate the statistical process.

In the third indicator, there also is a time annotation present. However, this time annotation has an explicit reference to a point in time related to the treatment: the moment of the breast-conserving surgery. Given that this time annotation is specifically related to the treatment, it cannot be discarded as 'a time annotation supporting the process of measurement'. But the fact that the indicator refers to a percentage (over some time) implies that another time annotation is needed. For demonstration purposes, we may augment the third indicator to "Percentage of women with breast cancer who had local recurrence within 5 years after breast-conserving surgery, *during the last 12 months*".

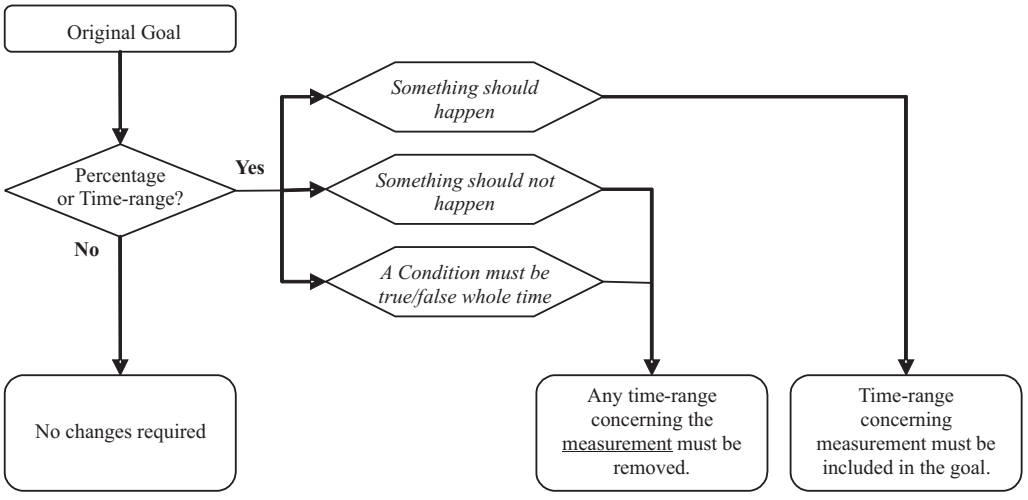


Figure 3.1: Decision table for time annotations

Local recurrence is not desired. Therefore we have situation again — just like in the second example indicator — in which something should never happen. Using the augmented version of the indicator it becomes clear the the previously added time annotation can be discarded. A simple check confirms this: removing the

second time annotation does not affect quality aspect of the indicator. However, removing the first does change the meaning. Figure 3.1 provides guidance for goals containing time annotations. Following the flow-chart, the third indicator leads to "Any time-range concerning the measurement must be removed". As shown before, caution must be taken when dealing with time groups: only time ranges related to the measurement should be removed.

The fourth example is trivial with respect to the reduction step: there are no time annotations to consider, nor are there any other superfluous elements[1] in the original goal. The correct reductions of all the examples are given below:

Reduction - Examples

1. "People should have a Cholesterol or HDL-cholesterol measurement every 12 months."

2. "For people with diabetes, hypoglycaemia should not occur."

3. "For women with breast cancer, local recurrence within 5 year after breast-conserving surgery should not occur."

4. "The possibility of breast reconstruction should be discussed with all patients prior to mastectomy."

3.2 Normalisation

After the superfluous (time) annotations have been removed and the goals have been reduced to their pure form, the next step will be to rewrite the goal into a structured form. An essential part of this rewrite is to define the exact domain of the goal along two scales: first, the exact group of patients the goal applies to. Second, the exact periods in time when the goal must hold. Determination of these domains corresponds to rewriting the goal in terms of the goal model introduced in chapter 2. (See figure 2.1, on page 4 for details).

The group of patients to which the goal applies, maps to *'the right circumstances'* shown in the picture: the goal should only hold for those parts of the plan that are applicable to that same group of patients. The exact period during which the goal should hold translates to the *start* and *end event*.

After the domain has been specified, the next step in the normalisation process is to write the goal in the structured form. This form is introduced in the next section.

During the normalisation, many assumptions and ambiguities are made explicit. Generally this requires domain knowledge. Therefore, assistance of a medical doctor was required to normalise example 1 – 4.

[1] Until now, no superfluous elements other than the percentage/time annotation combination have been found. However, there may be other classes of those elements; either in the medical domain, or in any other.

3.2.1 The normal form

The normal form consist of a couple of elements, in a fixed order, that coincide with the elements of the goal model. Since it is essential that the goal is written in understandable natural language, linguistic adjustments may be required to embed the groups in a readable sentence. Coloured, labelled brackets are added to the individual groups for easy identification. The general format is structured as shown below. The structure presented here may be nested as a whole in the *Behaviour* field, thereby allowing the expression of more complex, nested goals, without sacrificing the simple nature of the goal model.

^{G}[Group] ^{S}[Time-range Start] ^{E}[Time-range End] ^{B}[Desired Behaviour]

In the next section the different goals from the examples are rewritten into this form. Examples for the use of the structured form can be found there.

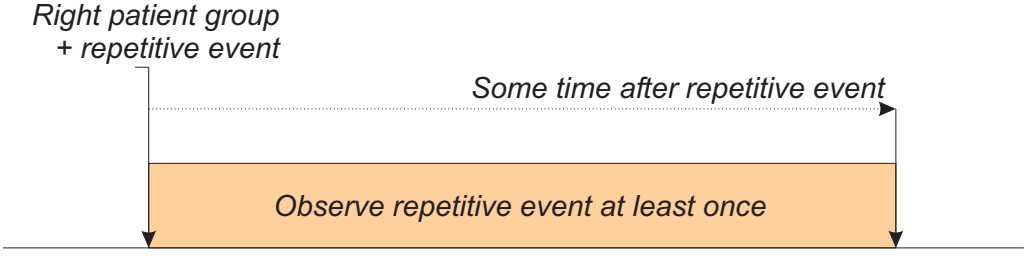
3.2.2 Patterns

Viewing a goal in term of the goal model is not always easy. It requires that one thinks of a sequence of events rather than just any behaviour. To clarify this way of thinking, the reduced examples of page 3.1.1 will be discussed here. From the individual formats, some general patterns will be derived that can help to transform others. It is impossible to show every conceivable pattern. However, an effort will be made to provide enough ground to be able to apply the same technique to other reductions.

Example 1

> "People should have a Cholesterol or HDL-cholesterol measurement every 12 months."

Example 1 shows a typical repetitive goal. In terms of the goal model, the start event is the measurement itself. The end event is defined as 12 months after that measurement — the start event. The desired behaviour is that during this period, another measurement is taken. It is repetitive in the sense that every time the desired behaviour is observed, it is at the same time the start event of another observation. Pattern 1 shows the pattern for a repetitive goal.

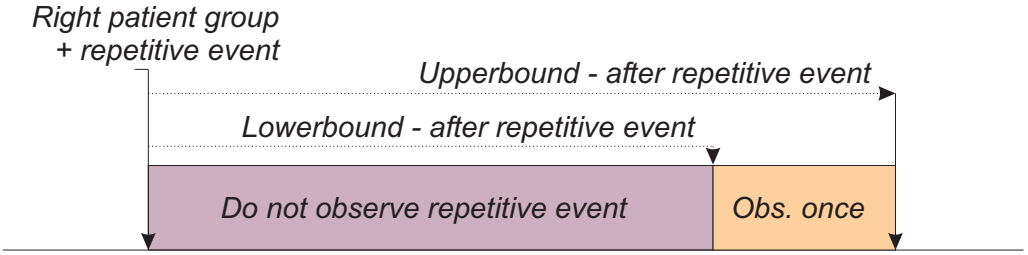


Pattern 1: Repetitive goal

The choice of the end point needs some further consideration. It concerns the interpretation of ‘every 12 months’. The first option is to read ‘at least once in every period of twelve months’ (literally translated). The other possibility is to read ‘exactly once every twelve months’. This is exactly the kind of ambiguity that has to be eliminated!

Modelling the first interpretation is straight forward according to pattern 1. The time displacement of the end event is 12 months, just like in the goal from the reduction. The second interpretation is somewhat more tricky. Formalising ‘exactly once every period of twelve months’ would mean that the goal is broken if the second measurement takes place after 367 days. For some goals (like checking some critical system within some period) this behaviour is desired, but for many medical purposes, a 367 day interval would also be fine.

The solution to this problem is to provide a precise interval in which the observation should occur. For example one, this could be ‘exactly once, occurring after at least 11 months and at most 13 months’. The exact boundaries of the interval depend on the property. This type is illustrated by the pattern of pattern 2



Pattern 2: Repetitive goal with explicit bounds

Note that the first interpretation is also vulnerable to the ‘367 days’ problem. This can solved by stretching the period to for example 13 months. What is reasonable here again depends on the goal.

With every repetitive goal, there should be some kind of boot-strapping. If only the repetition is defined, it may occur that the event will never happen and this will not cause the goal to fail (since a goal can only fail after the start event has been observed). There should be an additional specification that requires the repetitive event to occur at least once. The form of this extra goal depends entirely on the type of goal, but it is to be expected that it will usually resemble pattern 1. The main difference will be that the start event will have to be instantiated by some initial event — usually early in the process. The length of the observation period may be same as that of the repetition interval of the main goal, but — depending on the goal — it may also be much shorter.

The final normalisation of example 1 is according to pattern 2, with a bootstrap goal according to (adjusted) pattern 1:

Normalization - Example 1

^{G}[For people],^{S}[from the start of the diabetes care], ^{E}[within 12 months], ^{B}[cholesterol or HDL-cholesterol must be measured.]

^{G}[For people],^{S}[at earliest 11 months after a cholesterol or HDL-cholesterol measurement], ^{E}[and at latest 13 months after that measurement], ^{B}[another cholesterol or HDL-cholesterol measurement must be performed once.]

Notes:

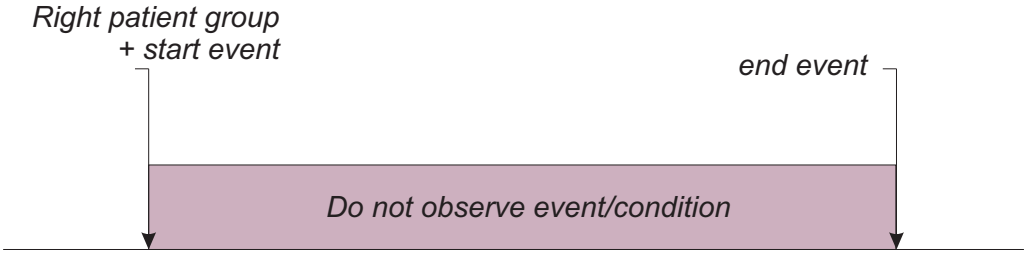
1 **Normalization** At least once during the treatment, cholesterol should be measured, but no later than 12 months after the start of the treatment.

2 **Normalization** 'Every year' has been formalized into 'exclusively within 11 to 13 months'. Medical Expert: *The relevance of a cholesterol measurement is limited, so more than one measurement a year should be avoided out of cost considerations.*

Example 2

"For people with diabetes, hypoglycaemia should not occur."

The second example aims at preventing something from happening. Pattern 3 depicts this behaviour. The difficulty here is to determine the start and end



Pattern 3: Avoid during the period

events. The question is how to translate 'never' (*should not occur*). The indicator was written with a medical setting in mind: therefore we may consider the goal applicable during the time the hospital is responsible for the care of this patient. To be precise: the period between the start and end of the care. The rest of the normalisation is straight forward:

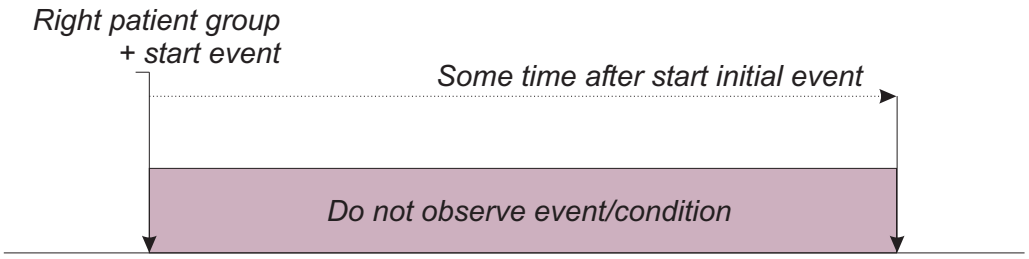
Normalization - Example 2

^{G}[For people with diabetes],^{S}[between the start of the diabetes care], ^{E}[and the end of the care], ^{B}[hypoglycaemia should not occur].

Example 3

> "For women with breast cancer, local recurrence within 5 year after breast-conserving surgery should not occur."

The third goal maps to pattern 4. This pattern is similar to pattern 3, with the



Pattern 4: Avoid during the period, relative

difference that the end event is specified in terms of some time after the start event. This pattern is easy recognisable in the reduced version of the goal. The different components of the normalised form can almost be taken literally from the text. Note the explicit specification of '*after successful completion* of breast-conserving surgery'. This refines the reference to this procedure in the original. A note is used to provide enough background to interpret this expression correctly.

Normalization - Example 3

^{G}[For women with breast-cancer],^{S}[after successful completion of breast-conserving surgery] ^{E}[until 5 years thereafter], ^{B}[local recurrence should not occur].

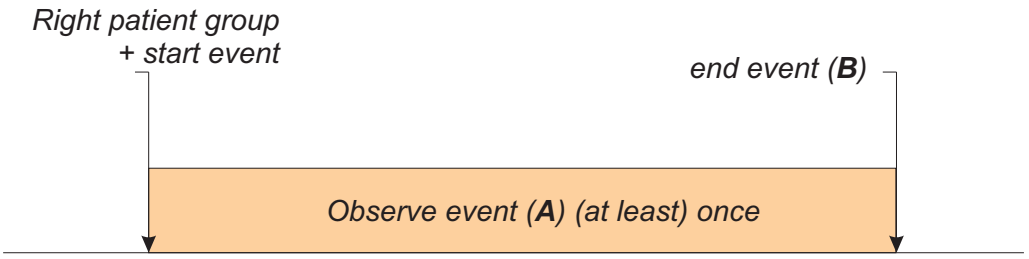
Notes:

1 **Normalization** 'Successful completion', is evaluated immediately after the procedure and reflects the surgeons opinion concerning the removal of the cancer. The long term success is not taken into account here.

Example 4

> "The possibility of breast reconstruction should be discussed with all patients prior to mastectomy."

The last example provide a whole new kind of goal. It requires that the patient is informed before surgery is started. Essential to realise here is that there is no requirement that surgery will eventually take place. Seen in terms of the common goal model, this means that if some event B is observed, another event A must already have been observed. Pattern 5 shows that. Although it is natural that the



Pattern 5: A should happen before B

goal can only fail once event B has happened — there is no way to be sure that in the future A will indeed not happen before the eventual occurrence of B — for clarity, it is recommended to explicitly specify that B is not required to happen.

The start event is not relevant for the ordering itself. Since no other clues are given, the start of the care is taken as start event. Taking all this into account, normalising example 4 will then result in:

Normalization - Example 4

^{G}[For women with breast-cancer],^{S}[after start of the medical care] but ^{E}[before commencing mastectomy], ^{B}[the possibility of breast reconstruction should have been discussed with the patient].

Notes:

1 **Normalization** '··· with all patients ···', has been rewritten to the group specification: 'women with breast-cancer': the goal comes from a document specifically concerning women with breast-cancer.

2 **Normalization** Mastectomy is not required to happen. Only *if* it happens, the patient should have been informed.

3 **Normalization** Since there is no harm in informing the patient more than once, no restriction has been added.

3.3 Conclusion

Several of the goals that were set for the conversion process in section 2.3 have been addressed. The reduction and the normalisation are still completely related to the natural language version of the goal. This makes it easy for the domain expert to verify correctness of any changes made to the goal with respect to the domain. Starting the process with the reduction also adds to this: it eliminates distracting, superfluous elements from the goal in a controlled way. The resulting purely descriptive from is — although not yet structured — the first step towards a canonical expression of the goal. (i, iii, iv on page 7).

The normalisation forces both experts to think about the goal in terms of start, end, and what lies in between. Rethinking a goal like this brings up questions about the exact meaning of the goal. By answering those questions, ambiguities are solved

automatically. Proper documentation in this step ensures traceability. The final result is the goal in a canonical form: the structured version of the goal model (i, ii, vi). Reusability is not yet an important issue in these steps (v).

Chapter 4
Formalisation

The formalisation step which is described in this chapter, will pave the road towards the attachment phase. After the formalisation, two equivalent representations of the same goal are available. One in (structured) natural language, the other in a formal expression. By having those, the formal methods expert and domain expert can discuss the same goal using their own representations.

The formal expression of the goal will be provided by *GDL*, the *Goal Definition Language*. This new language is specifically designed to reflect the structure of the goal model and will therefore also stay very close to the structured version. *GDL* is split into two parts:

- A general, domain independent part, *Generic GDL*, that represents the goal model itself.
- A task specific extension to *GDL* that defines the exact conditions and events that can be used. The extension is heavily dependent on the domain and on the formalisation language of the process. For Protocure, these processes are medical guideline expressed in Asbru. Therefore, the ***GDL***-Asbru extension of *GDL* has been developed.

This chapter has been divided in four sections. Section 4.1 will provide a foundation of definitions for later use. Section 4.2 discusses the elements of *Generic GDL* together with their semantics, followed by a section that ties the patterns of chapter 3 to *GDL*. Using all this, in section 4.4 the examples of the previous chapter will be taken through the formalisation. The formalisation is still independent of a specific process model. Therefore, ***GDL***-Asbru will be not be discussed until chapter 5.

4.1 Definitions

GDL expressions are evaluated on linear sequences of states. Such a sequences of states is referred to as an *interval*.

definition 1 *(Interval I) Let* $\bar{n} \in \mathbb{N}$. *An interval*

$$I = (\sigma_0, \ldots, \sigma_{\bar{n}})$$

consists of

- *an intial state* σ_0, *and*
- *a finite or infinite and possibly empty sequence of* transitions

$$(\sigma_{i-1}, \sigma_i)_{i=1}^{\bar{n}}$$

$|I| := \bar{n}$ *is defined as the* length *of an interval I. The interval with* $|I| = 0$, *only contains the initial state* σ_0.

To accommodate manipulation of intervals, a set of additional functions has been defined.

definition 2 *(Auxiliary definitions for Intervals)*

$$
\begin{aligned}
I(i) &:= \begin{cases} \sigma_i, & \text{if } i \leqslant \bar{n} \\ \sigma_{\bar{n}}, & \text{otherwise} \end{cases} \\
I|_i &:= \begin{cases} (\sigma_i, \ldots), & \text{if } i \leqslant \bar{n} \\ (\sigma_{\bar{n}}), & \text{otherwise} \end{cases} \\
I|^i &:= \begin{cases} (\sigma_0, \ldots, \sigma_i), & \text{if } i \leqslant \bar{n} \\ (\sigma_0, \ldots, \sigma_{\bar{n}}), & \text{otherwise} \end{cases} \\
I|^{-i} &:= \begin{cases} (\sigma_0, \ldots, \sigma_{\bar{n}-i}), & \text{if } i \leqslant \bar{n} \\ (\sigma_0), & \text{otherwise} \end{cases} \\
I|_i^j &:= (I|^j)_i \\
I|_i^{-j} &:= (I|^{-j})_i
\end{aligned}
$$

$I(i)$ selectes the ith state of an interval. $I|_i$ takes a portion of I, starting with σ_i. $I|^i$ takes a portion, starting with state 0 and ending with σ_i. $I|^{-i}$ removes the last i states of the interval. $I|_i^j$ takes a subinterval of I from σ_i to σ_j. In the same way $I|_i^{-j}$ returns the subinterval starting with i, and the last j states removed.

definition 3 *(Counting of events) The following function*

$$\lfloor . \rfloor_{.,.} : \mathbf{E} \times \mathrm{I} \times \mathrm{L} \mapsto \mathbb{N}$$

takes an event, an interval and a lower bound, and returns a number such that:

$$
\lfloor e \rfloor_{I,l} = \begin{cases} 0 & \text{if } |I| = 0 \ \vee \ |I| = l - 1 \\ \lfloor e \rfloor_{I|^{-1},l} & \text{if } |I| \neq 0 \ \wedge \ I, l, 0 \not\models_{\overline{\mathrm{e}}} e \\ (\lfloor e \rfloor_{I|^{-1},l}) + 1 & \text{if } |I| \neq 0 \ \wedge \ I, l, 0 \models_{\overline{\mathrm{e}}} e \end{cases}
$$

Definition 3 provides a function which returns the number of states in interval $I|_l$ in which the event occurs. The details of events and the interpretation of $I, l, 0 \models_{\overline{e}} e$ is discussed in detail in section 4.2.1 and further. For now, it is sufficient to read $I, l, 0 \models_{\overline{e}} e$ as 'Event e has occurred during the transition to the last state of I'.

definition 4 *(Definition of time) Let* $\textbf{\textit{Duration}}(\sigma_i)$ *be the duration of* σ_i*. The following function*

$$\|.\| : \mathbf{I} \mapsto \mathbf{T}$$

takes an interval and returns the duration of that interval:

$$\|I\| := \sum_{i=0}^{|I|} \textbf{\textit{Duration}}(I(i))$$

Finally, definition 4 provides a way to talk about durations of intervals, given the duration of individual states. T is assumed to be a standardised unit of time.

4.2 The Goal Definition Language

GDL is the formal counterpart of the structured natural language version. There are two ways to write *GDL*:

1. Machine readable XML of which the syntax can be found in appendix A.
2. A presentation syntax for use in documents (Appendix B).

The semantics have been defined on the presentation syntax, and that is what this section will be about. The complete definition of semantics can be found in appendix C. As the elements and structure of the presentation syntax are also present in the XML version, there is a deterministic translation possible between those two. For orientation, figure 4.1 shows a full example of *GDL* (using events from the ***GDL***-Asbru extension).

Goal Maintain-example
  Precondition
    always-true

  Time-specification
    From
      Transition treatment **enter** active $\overset{+}{\mapsto}$ 1 day
    Until
      Transition treatment **leave** active

  Maintain-during-period
    Param Lower-Blood-press $<$ 90

Figure 4.1: Full example of a goal using *maintain-during-period*.

As the example shows, three major parts can be distinguished. The first part is the *precondition*, which corresponds to the *group* (G) part of the structured representation and which — as shown here — is empty:

Normalization - Full example

^{G}[],^{S}[From one day after the start of the treatment] ^{E}[until the moment the treatment is discontinued], ^{B}[the blood pressure should remain below 90].

The second part is the time-specification. This specification defines the period in which the behaviour should hold. It always starts with a *from*, followed by an event (or a combination of events), to define the start of the period. The *from* part corresponds with the *start* (S) section in the structured version. For the end of the period, there is more than one possibility: in the example a simple *until* is chosen but amongst others, it is also possible to specify a duration. Obviously, this part corresponds as a whole to the *end* section (E).

The third part describes the actual behaviour. *Maintain-during-period* is one of five behaviours that have been defined. Section 4.2.4 goes into detail about the different possbilities here. If some behaviour is not covered in the current *GDL* specification, it can easily be added without breaking semantics of existing elements. The behaviour part of *GDL* is connected to the *behaviour* (B) part of the structured natural language version.

$$
\begin{aligned}
I \models \quad & \textbf{Goal } \mathcal{A}_{1:Name} \\
& \quad \textbf{Precondition} \\
& \quad\quad \mathcal{P}_{1:Condition} \\
& \quad \textbf{Time-specification} \\
& \quad\quad \textbf{From } \mathcal{P}_{2:Event} \\
& \quad\quad \mathcal{P}_{3:Time-delimiter} \\
& \quad \mathcal{P}_{4:Behaviour} \\
\text{iff } \forall\, i,j > i \,.\quad & I|^{i} \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge I|^{i},0,0 \models_{\overline{e}} \mathcal{P}_{2:Event} \\
& \wedge\ I,i,j \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter} \\
& \wedge\ \neg\exists\, i<k<j \,.\, I,i,k \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter} \\
& \Rightarrow I|^{j-1},i \models_{\overline{b}} \mathcal{P}_{4:Behaviour}
\end{aligned}
$$

Table 4.1: Semantics of the *goal* body

Table 4.1 shows the definition of the semantics of the goal. A goal is true for some interval I, if the predicate logic formula is true for that I. The formula itself is built around an implication. The all-predicate actually defines an i, a possible index of the starting state, and a j, a possible index of the state in which the end event occurs. If the left-hand side manages to define two states σ_i and σ_j such that they satisfy the precondition and the time-specification, then the right-hand side of the implication describes the required behaviour over the interval between those states.

As table 4.1 shows the semantics are defined recursively: the formula contains several $\models$ operators that define the meaning of smaller language elements. To increase expressiveness, and to be able to distinguish between language elements,

different versions of the $\models$ operator are defined: $\models_{\overline{c}}$ for conditions, $\models_{\overline{e}}$ for events and $\models_{\overline{b}}$ for behaviours. Each will be discussed separately in the following sections.

Section 4.2.2, *The start of the period*, section 4.2.3, *The end of the period* and section 4.2.4, *Behaviours* will go into detail on the different part of the *GDL* semantics. The next section will start by a description of events and conditions and how they relate to intervals.

4.2.1 Conditions and events

Before continuing with the discussion of the semantics of the goal model itself, the most basic elements of *Generic GDL* will be discussed: (abstract) conditions and events. How these relate to a given interval is the subject of this section.

Conditions

Conditions in *GDL* apply to a single state of a trace. For any state, the condition is either true, or false. However, for reasons of uniformity, conditions are primarily given semantics in relation to an interval.

$$I \models_{\overline{c}} \mathcal{P}_{1:condition}$$

The expression above means that condition $\mathcal{P}_1$ is true in the last state of I. Figure 4.2 illustrates an interval which satisfies the expression. Writing $I|^i \models_{\overline{c}} Condition$ can be used to express: "the condition is true in state σ_i".

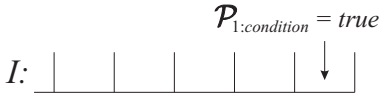


Figure 4.2: An interval which make $I \models_{\overline{c}} \mathcal{P}_1$ true.

Generic GDL allows any boolean combination of conditions. The following operators are supported: **()**, **not**, **and**, **or** and **xor**. Section C.1.4 on page 94 provides the semantics for each of those.

For individual atomic conditions (of which the majority is defined in *GDL* extensions), the semantics are referred back again to the semantics of this condition in relation to a single state. State $I(|I|)$ yields the last state of I:

$$\begin{aligned} I \models_{\overline{c}} & \quad \mathcal{P}_{1:Atomic-condition} \\ \text{iff } & \; I(|I|) \models \mathcal{P}_{1:Atomic-condition} \end{aligned}$$

As an example the (trivial) semantics of the *always-true* condition are provided here (note the σ instead of I):

$$\begin{aligned} \sigma \models & \quad \text{always-true} \\ \text{iff } & \; true \end{aligned}$$

Atomic events

Atomic events occur between transitions from one state to the other. As a consequence, an event — in contrast with a state — does not have a duration.

$$I \models \mathcal{P}_{1:Atomic-event}$$

The basic notation given here is both in writing and in meaning similar to that of conditions: $I \models_{\overline{e}} Event$ means that the event occurred during the transition to the last state of I. (Figure 4.3.) Also here, $I|^i \models_{\overline{e}} event$ can be used to test whether or not the event occurred in the transition to state σ_i

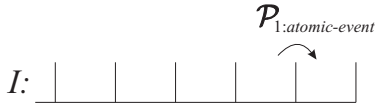


Figure 4.3: An interval which makes $I \models \mathcal{P}_{1:Atomic-event}$ true.

For complex expressions with atomic events, and for another class of events — the delayed events — more information is required. This is solved by providing an extended notation for events in general:

$$I, l, s \models_{\overline{e}} \mathcal{P}_{1:event}$$

The value of l represents the *lower bound*, s stands for *shift*. The exact meaning of the lower bound depends on the type of event and will be discussed separately form each class of events. The shift does not influence the semantics of the atomic event and therefore the discussion about the meaning of s will be postponed shortly until delayed events are treated in detail. Generally speaking, this notation for the semantics of events can best be read as "I models $\mathcal{P}_1$ taking the lower bound and the shift into account".

For atomic events, the lower bound limits the applicability of the operator to a specific range of states. The event only registers when it occurs during the transition to the last state with $l < |I|$. Figure 4.4 shows an interval which just satisfies the event. The shaded area marks the state left of the lower bound. As can be seen, if l would have been increased by one, the interval would not satisfy the event.

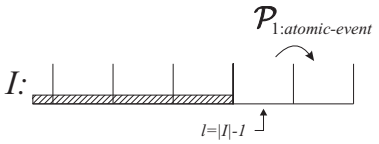


Figure 4.4: An interval which makes $I, l, * \models_{\overline{e}} \mathcal{P}_{1:Atomic-event}$ true.

As s is not relevant for atomic events, '$I, l, * \models_{\overline{e}}$ *Atomic-event*' can be read as: "The event occurred during the transition to the last state of I, and it occurred after the state transition to σ_l". This is expressed by the following semantics definition:

$$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Atomic-event}$$
$$\text{iff } l < |I| \ \wedge \ I \models \mathcal{P}_{1:Atomic-event}$$

The delayed event

Apart from specifying atomic events, it is useful to be able to specify a delay after some event as the start- or end event. Although by itself the delay is strictly not an event, the expiring of the delay is. The '$\overset{+}{\longmapsto}$' symbol is used to specify such a delay:

$$I, l, s \models_{\overline{e}} \mathcal{P}_{1:atomic-event} \overset{+}{\longmapsto} \mathcal{A}_{1:Delay}$$

The shift value s comes into play here. The value of $s \in \mathbb{N}$ determines in which state of I (counted from the end) the delay should expire. Figure 4.5 shows the default situation, when a delayed event is applied with $s = 0$. For delayed events, l determines that the event only holds when $l \leqslant |I|$.

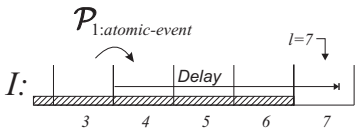


Figure 4.5: $I, 7, 0 \models_{\overline{e}} \mathcal{P}_{1:event} \overset{+}{\longmapsto} \mathcal{A}_{1:Delay}$

'$I, 7, 0 \models_{\overline{e}} Event \overset{+}{\longmapsto} Delay$' can be read as: "The delay after the occurrence of the event expires during the last state but not earlier than state σ_l (i.e. state 7)". For a value of $s = 1$ the situation changes slightly:

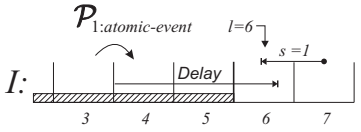


Figure 4.6: $I, 6, 1 \models_{\overline{e}} \mathcal{P}_{1:event} \overset{+}{\longmapsto} \mathcal{A}_{1:Delay}$

As figure 4.6 shows, '$I, 6, 1 \models_{\overline{e}} Event \overset{+}{\longmapsto} Delay$' can be read as: "The delay after the occurrence of the event expires during the second last state of I but not earlier than state σ_l (i.e. state 6)".

The formal definition of the delayed event is as follows:

$$I, l, s \models \mathcal{P}_{1:Atomic-event} \overset{+}{\longmapsto} \mathcal{A}_{1:Delay}$$
$$\text{iff } |I| - s \geqslant l \ \wedge \ \exists\, k \,.\, (I|^k \models \mathcal{P}_1) \wedge \ \|I_k^{-(s+1)}\| < \mathcal{A}_1 \leq \|I_k^{-s}\|$$

Informally it states, that if the length of I minus s ($|I| - s$, the target state) is greater or equal to the lower bound l, and if a state k exists in which event $\mathcal{P}_1$ just occurred, and if the delay after this event is exceeded in the target state, but not in the state before that, then I, l, s indeed models the delayed event.

The bounded delayed event

As special version of the delayed event is the *bounded delayed event*: '$\overset{+}{\mapsto}$'. This is where l is even more restrictive. For the regular version of the delayed event, the only criterion is that the delay expires during the last state (for $s = 0$) and that the lower bound lies before that state. However, in some cases it is useful to be able to specify that the event itself must also lie after the lower bound. If the event would occur before the lower bound, it would never make the expression true: not even when the delay would expire in the right state. Figure 4.7 shows an interval with a bound. The event falls just outside of the shaded area and is therefore valid. Would $l = 7$ be used like in figure 4.5, it would disqualify the interval.

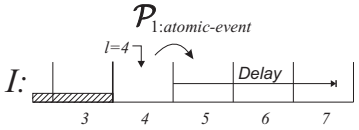


Figure 4.7: $I, 4, 0 \models_{\overline{e}} \mathcal{P}_{1:event} \overset{+}{\mapsto} \mathcal{A}_{1:Delay}$

The formal semantics of the bounded delay are very similar to that of the regular version, except for the more restrictive bounds check. The functions of the shift (s) remains unchanged:

$$I, l, s \models \mathcal{P}_{1:Atomic-event} \overset{+}{\mapsto} \mathcal{A}_1$$
$$\text{iff } \exists\, k > i \,.\, (I|^k \models \mathcal{P}_{1:Atomic-event}) \wedge \|I_k^{-(l+1)}\| < \mathcal{A}_1 \leq \|I_k^{-l}\|$$

The *Generic GDL* notation does not allow explicit specification of the lower bound. The the lower bound follows from the location the bounded delayed event is used in. In the discussion of the main format of *Generic GDL*, for each location the applicable lower bound will be mentioned.

Combinations of events

Atomic and delayed events can be combined into complex event combinations. The available operators are: **()**, **and**, **or** and **xor**. Since the semantics of *Not event* is in many cases not intuitive (e.g. 'Observe *Not event* 3 times'), the **not** operator has been left out. Combinations of events are evaluated individually per event and then connected again with propositional logic:

$$I, l, s \models \mathcal{P}_{1:Event} \textbf{ and } \mathcal{P}_{2:Event}$$
$$\text{iff } I, l, s \models \mathcal{P}_{1:Event} \wedge I, l, s \models \mathcal{P}_{2:Event}$$

Figure 4.8 shows an interval that satisfies a delayed event and an atomic event combined by **and**. The delay must expire in the last state of I, *and* the second event must occur during the transition to the last state.

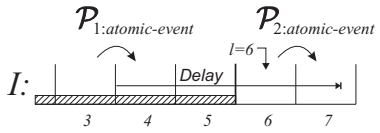


Figure 4.8: $I, 6, 0 \models_{\overline{e}} \mathcal{P}_{1:Atomic-event} \overset{+}{\longmapsto} \mathcal{A}_{1:Delay}$ **and** $\mathcal{P}_{2:Atomic-event}$

4.2.2 The start of the period

The start of the period is — according to the goal model — determined by the combination of the pre-condition and the start event[1] (***From*** ...). The pre-condition must be true when the start event occurs. The first line of predicate logic (of the semantics of page 24) shows this combination:

$$I|^i \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge I|^i, 0, 0 \models_{\overline{e}} \mathcal{P}_{2:Event}$$

This line is true for every segment $I|^i$ that satisfies both the condition and the event in the last state. The event is used without lower bound or shift: $l = 0$, and $s = 0$.

In figure 4.9 a graphical representation is provided for some pre-condtion *pre* and start event: X ***and*** $Y \overset{+}{\longmapsto}$ *5 minutes*. The shaded states are potentially part of the interval (if a suitable end is found).

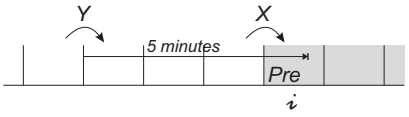


Figure 4.9: i in case of X ***and*** $Y \overset{+}{\longmapsto}$ *5 minutes*.

4.2.3 The end of the period

The end of the period can be specified by a choice of period-delimiters. Table 4.2 shows the options.

There are two groups of period delimiters: *until* and *duration*. The first is combined with an event that triggers the end. The duration takes a time value to determine the end. As with events, extra information is needed to have the end of the period reliably detected by all types of delimiters: the whole interval I, the intended start of the period i, and the proposed end the period j:

$$I, i, j \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$$

The definition above means informally: 'If true, the period delimiter marks a valid end in state j of I, given the intended period start i'. The third line of the

[1]For the remainder of this chapter, when *an event* is mentioned, this can be any combination of events.

4 **Period-delimiter**	
–	**Until** ▷ *Event*#8, *p.* 90
–	**Open-until** ▷ *Event*#8, *p.* 90
–	**Until end**
–	**Duration** [Time]`Numerical` [Unit]▷*Unit*#5, *p.* 90
–	**Open-duration** [Time]`Numerical` [Unit]▷*Unit*#5, *p.* 90

Table 4.2: The different options for the period delimiter.

goal semantics ensure that if some j marks a valid end, there is no other j which occurs earlier and that also marks a valid end:

$$\neg\exists\ i < k < j\ .\ I, i, k \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$$

How the individual period delimiters work out will be discussed now.

Until

The until operator does what one would expect: it marks the end of the period when an event occurs. The semantics are as follows:

$$I, i, j \models_{\overline{p}} \textbf{Until}\ \mathcal{P}_{1:Event}$$
$$\text{iff}\ I|^{j}, i, 1 \models_{\overline{e}} \mathcal{P}_{1:Event}$$

The first notable thing is the use of i as lower bound: for bounded delayed events used in the until clause, the bound is the start of the period. Effectively this means that in those cases, the event itself should occur after the start of the period. The second significant issue, is the shift of '1'.

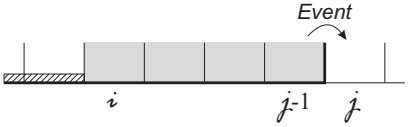


Figure 4.10: $I|^{j}, i, 1 \models_{\overline{e}} \mathit{Event}$ in case of a non-delayed end event.

For any non-delayed event the interval is terminated by until according to figure 4.10. (The shaded states constitute the period.) The figure shows that the last state in which the behaviour should hold is σ_{j-1}: the behaviour should hold until the moment the event occurs. For delayed events the situation is different. The delay may expire in the middle of a state. Therefore, the behaviour should apply at least to the part of the state where the delay has not yet expired. As a single state does not change over time, this means that the behaviour should hold during the whole

state. In order to maintain the fact that the last state of the period is σ_{j-1}, a shift value of 1 is needed as show in figure 4.11

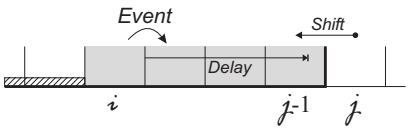


Figure 4.11: $I|^{j-1}, i, 1 \models_{\overline{e}} \mathit{Event}$ in case of a delayed end event.

Figure 4.11 will not change in case of bounded delayed events: the event occurs after i. However, would the intended start be one or more states to the right (i.e. i is bigger), then the event would occur before i and a bounded delay would not have made the formula true. Figure 4.12 shows an interval which would only satisfy a regular delayed event. However, a bounded delayed event with that value for the delay will not be accepted for that j.

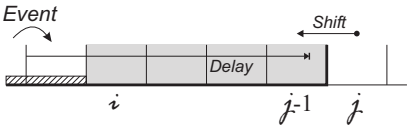


Figure 4.12: An interval failing $I|^{j}, i, 1 \models_{\overline{e}} \mathit{Event} \overset{+}{\longmapsto} \mathit{Delay}$

Open-until

The *open-until* operator is a slightly modified version of the standard *until*. The difference is important though. The semantics of the goal only ensure that the behaviour is enforced for every interval between i and the first occurring delimiter. However, when the delimiter does not occur any more, there is no interval. For some goals the behaviour should be enforced 'until the first delimiter or the end of the interval, whichever occurs first'. For example, introducing *open-until* in the example of figure 4.1, would mean that even if *treatment* would never leave active, the behaviour (*maintain blood-pressure below 90*) would still be enforced until the end of the interval.

$$I, i, j \models_{\overline{p}} \quad \textbf{Open-until}\ \mathcal{P}_{1:Event}$$
$$\text{iff}\ \ j = |I| + 1 \ \vee\ I|^{j}, i, 1 \models_{\overline{e}} \mathcal{P}_{1:Event}$$

For the semantics this means that the *open-until* is not only true for those j in which the normal *until* would be true, but also for $j = |I| + 1$. Selecting the period up to σ_{j-1} then yields the original end. The third line of the goal semantics ensures that the end of execution cannot be selected unless there is no earlier delimiter.

Until end

The *Until-end* delimiter is only true for $j = |I|+1$. No other event can be specified so every valid start make a period that run until the end of the interval. The semantics are straight forward:

$$I, i, j \models_{\overline{p}} \quad \textbf{Until end}$$
$$\text{iff } j = |I| + 1$$

Duration

Instead of specifying a specific event to close the period, it is also possible to provide a *duration*. From σ_i, until the state in which the duration expires, the behaviour should hold. Just like the shifted version of the delayed event, the *duration* makes sure the time expires one state early to compensate for $I|^{j-1}$.

$$I, i, j \models_{\overline{p}} \quad \textbf{Duration } \mathcal{A}_1$$
$$\text{iff } \left\|I|_i^{j-2}\right\| < \mathcal{A}_1 \leqslant \left\|I|_i^{j-1}\right\|$$

Open-duration

An open version of the duration is also provided, which — if the remaining duration of the interval is not enough to fit the requested duration into — accepts the end of the interval as end. The semantics are

$$I, i, j \models_{\overline{p}} \quad \textbf{Open-duration } \mathcal{A}_1$$
$$\text{iff } j = |I| + 1 \ \vee \ \left\|I|_i^{j-2}\right\| < \mathcal{A}_1 \leqslant \left\|I|_i^{j-1}\right\|$$

4.2.4 Behaviours

Once a value for i and j has been found which satisfy the left hand side of the implication of table 4.1 on page 24, behaviour is enforced over the interval $I|_i^{j-1}$. However, since for delayed events it is required to have information about states before σ_i (the event may occur before the start of the period), this part of the interval is passed unmodified and i is included with the operator:

$$I|^{j-1}, i \models_{\overline{b}} \mathcal{P}_{4:Behaviour}$$

Table 4.3 shows the behaviours that have been defined in *Generic GDL*. As will become clear in this section, $\models_{\overline{b}}$ *behaviour* will be expressed using $\models_{\overline{c}}$ and $\models_{\overline{e}}$. Now every behaviour in table 4.3 will be discussed.

maintain-during-period

If during the period some condition should be true the whole time, *maintain-during-period* should be used. As i defines the start of the interval, the condition should hold for every σ_k with $k \geqslant i$.

$$I, i \models_{\overline{b}} \quad \textbf{Maintain-during-period } \mathcal{P}_{1:Condition}$$
$$\text{iff } \forall\, k \geqslant i \,.\, I|^k \models_{\overline{c}} \mathcal{P}_1$$

2 **Behaviour**

–	**Maintain-during-period** ▷ *Condition*$_{\#7,\ p.\ 90}$
–	**Avoid-during-period** ▷ *Condition-or-Event*$_{\#6,\ p.\ 90}$
–	**Observe-during-period** ▷ *Condition*$_{\#7,\ p.\ 90}$
–	**Observe-during-period** [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Count]$_{\texttt{Num}}$ ▷ *Event*$_{\#8,\ p.\ 90}$
–	**Observe-during-period range** [Lowerbound]$_{\texttt{Num}}$ ([Count]$_{\texttt{Num}}$) ▷ *Event*$_{\#8,\ p.\ 90}$
–	**Achieve-at-end** ▷ *Condition*$_{\#7,\ p.\ 90}$
–	**Sub-goal** **Precondition** ▷ *Condition*$_{\#7,\ p.\ 90}$ **Time-specification** ▷ *Time-specification*$_{\#3,\ p.\ 89}$ ▷ *Behaviour*$_{\#2,\ p.\ 89}$

Table 4.3: The different behaviours defined in *GDL*

avoid-during-period

The *avoid-during-period* behaviour is exactly opposite to that of *maintain-during-period*. In contrast with the latter, *avoid-during-period* can also accept events as operand. For bounded delayed events, the bound is defined as i — the start of the period. Therefore in those cases, only delayed events of which both the event and the delay are in $I|_i$ should be avoided. The semantics when an event is provided is given here, the (very similar) version with a condition as operand can be found in C.1.3.

$$I, i \models_{\overline{b}} \textbf{Avoid-during-period}\ \mathcal{P}_{1:Event}$$
$$\text{iff } \neg\exists\ k \geqslant i\ .\ I|^k, i, 0 \models_{\overline{e}} \mathcal{P}_1$$

observe-during-period ...

If something should happen for a specific number of times, this can be specified using *observe-during-period*. Three different version are available. Below the semantics are given for the case where an exact number of occurrences of some event is required. Semantics for the other syntactical options are given in appendix C.

$$
\begin{aligned}
I, i \models_{\overline{b}} \quad & \textbf{Observe-during-period} = \mathcal{A}_{2:Count} \\
& \quad \mathcal{P}_{1:Event} \\
\text{iff } & \lfloor \mathcal{P}_1 \rfloor_{I,i} = \mathcal{A}_2
\end{aligned}
$$

The lower bound i is passed to the count function: in case of bounded delayed events only those instances are counted of which the event lies after the start event.

The fact the state count will only increase over time can be used to add additional knowledge to the semantics. In the example of above, if '$\lfloor \mathcal{P}_1 \rfloor_{I,i} = \mathcal{A}_2$' it is also true that '$\forall\ k \geqslant i\ .\ \lfloor \mathcal{P}_1 \rfloor_{I|^k,i} \leqslant \mathcal{A}_2$'. Such knowledge may be exploited during verification to detect failure early.

achieve-at-end

Achieve-at-end may used when some condition must be true by the end of the interval. More specifically: the condition must be true in the last state of I:

$$
\begin{aligned}
I, i \models_{\overline{b}} \quad & \textbf{Achieve-at-end} \\
& \quad \mathcal{P}_{1:Condition} \\
\text{iff } & I \models_{\overline{c}} \mathcal{P}_1
\end{aligned}
$$

Sub-goal

Instead of a real behaviour, it is also possible to provide a sub-goal. The behaviour that is described by the sub-goal is applied to the interval selected by the encompassing goal. The sub-goal may be nested multiple levels deep. As can be seen in the formal semantics, the goal stands completely on itself: the states before i are discarded right away. This means that both for bounded delay events, and regular events, the events itself must be within $I|_i^{j-1}$ of the goal one level higher.

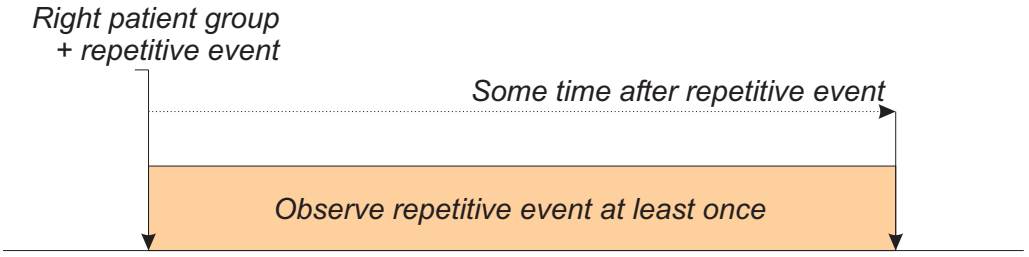
$$
\begin{aligned}
I, i \models_{\overline{b}} \quad & \textbf{Sub-goal} \\
& \quad \textbf{Precondition} \\
& \quad\quad \mathcal{P}_{1:Condition} \\
& \quad \textbf{Time-specification} \\
& \quad\quad \textbf{From } \mathcal{P}_{2:Event} \\
& \quad\quad \mathcal{P}_{3:Time-delimiter} \\
& \quad \mathcal{P}_{4:Behaviour} \\
\text{iff } & \forall\ i', j' > i'\ . \quad (I|_i)|^{i'} \models_{\overline{c}} \mathcal{P}_{1:Condition} \ \wedge\ (I|_i)|^{i'}, 0, 0 \models_{\overline{e}} \mathcal{P}_{2:Event} \\
& \qquad\qquad \wedge\ I|_i, i', j' \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter} \\
& \qquad\qquad \wedge\ \neg\exists\ i' < k' < j'\ .\ I|_i, i', k' \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter} \\
& \qquad\quad \Rightarrow (I|_i)|^{j'-1}, i' \models_{\overline{b}} \mathcal{P}_{4:Behaviour}
\end{aligned}
$$

Now that every element of table 4.1 has been discussed, and the semantics have been defined, it is time to apply *GDL* to the patterns of chapter 3.

4.3 Patterns in *GDL*

In the previous chapter, several patterns have been identified. This section will show the expression of those patterns in *GDL*. These expressions will then be used in the next section to formalise the running examples. The part of the template that needs to be filled with the specifics of the actual goal, have been underlined: Right patient group. For more information on when to use a given pattern refer to section 3.2.2. For application examples, refer to the example goals in section 4.4

4.3.1 Pattern 1: Repetitive goal



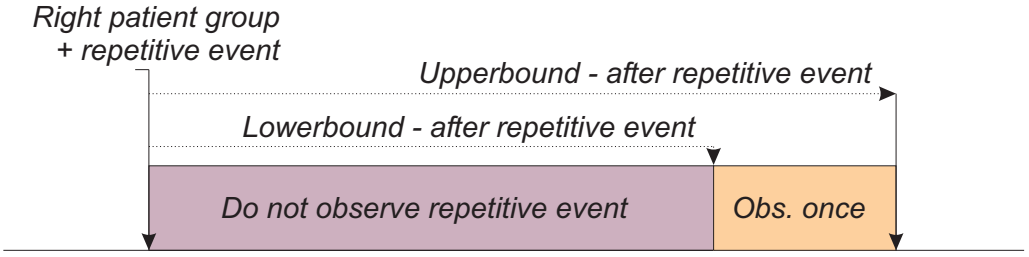
As the figure above shows, the basic repetitive pattern uses the same event twice: the first time as start-event, the second time as the event that needs to be observed within some time. The schematic naturally leads to the application of the *observe-during-period* behaviour. The required number of counts is specified as $\geqslant 1$. The end event has been defined as an offset from the start event which points to the *From + duration* format of the time specification. The template therefore looks like this:

Goal Pattern Template 1: *Repetitive event*
Precondition
Right patient group

Time-specification
From Repetitive event
Duration Offset from repetitive event

Observe-during-period $\geqslant 1$
Repetitive event

4.3.2 Pattern 2: Repetitive goal with explicit bounds



On page 15, a second way of expressing repetitive events was introduced. This version is somewhat more complicated since it actually consists of two behavioural descriptions in one. The first part ensures that the repetitive events are at least some minimum time apart (the purple part of the schematic). The second part defines a specific interval in which the event must be observed once. The intervals follow each other in time and when applied repeatedly, a regular repetition results.

The dualistic nature of the pattern is expressed in a translation into two separate goals. One for each interval. The first interval wants to avoid the repetitive event: *avoid-during-period*. The time specification takes the *From + open-duration* form. The open-duration is used to ensure that the event is not repeated even in case the given duration is longer than the remaining execution time. The second interval is clearly of the *observe-during-period* type with *= 1* as count specification. The start event has an offset that is the size of the lower bound from the schematic. Due to this offset, the duration should be the difference between the upper bound and the lower bound. Altogether that leads to those two templates:

Goal Pattern Template 2a: *Repetitive event with bounds*
Precondition
Right patient group

Time-specification
From Repetitive event
Open-duration Lower bound

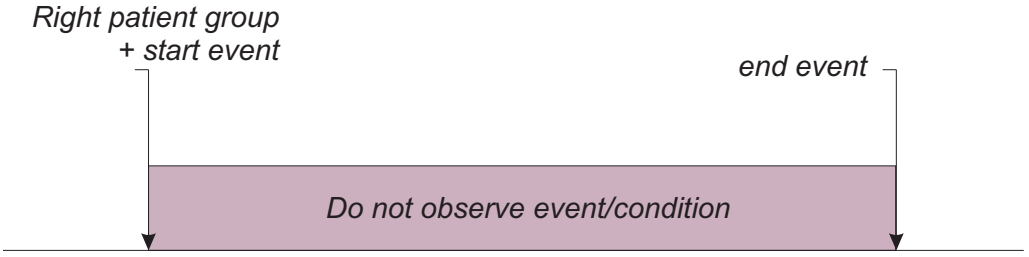
Avoid-during-period
Repetitive event

Goal Pattern Template 2b: *Repetitive event with bounds*
Precondition
Right patient group

Time-specification
From Repetitive event $\overset{+}{\longmapsto}$ lower bound
Duration upper bound - lower bound

Observe-during-period = 1
Repetitive event

4.3.3 Pattern 3 and 4: Avoid during the period



Pattern 3 (shown here) and pattern 4 (see page 17) translate trivially into *GDL*:

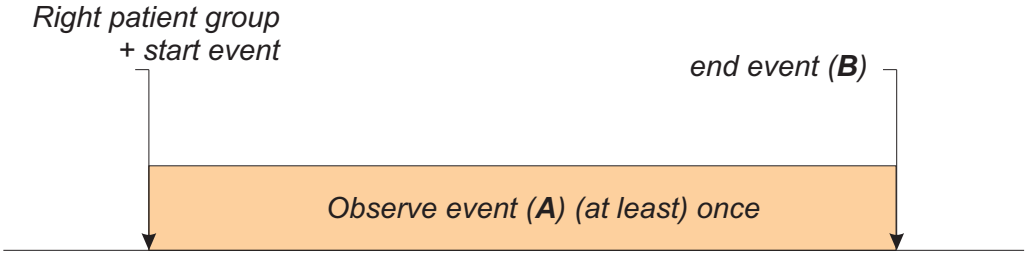
Goal Pattern Template 3: *Avoid*
Precondition
Right patient group

Time-specification
From Start event
Until End event

Avoid-during-period
Event or condition to avoid

Pattern 4 only differs from pattern 3 by the time specification: *From + duration* instead of *From - until*. Therefore, only a template for pattern 3 is given.

4.3.4 Pattern 5: A should happen before B



The last pattern describes an ordering between events. This ordering is expressed using the *observe-during-period* behaviour. 'A should happen before B' can be read as 'At the moment B occurs, A should already have been observed'. If B would never happen, then it does not matter whether or not A has happened. This is exactly the kind of behaviour *until* enforces: if a valid end is found, the behaviour should hold. (Note that an *open-until* would for that reason be completely inappropriate here. An open-until always yields an end point: either the event, or the end of the interval.) Knowing this, the pattern obviously translates to *observe-during-period*:

Goal Pattern Template 5: *A before B*
Precondition
Right patient group

Time-specification
From Start event
Until Event B

Observe-during-period $\geqslant 1$
Event A

In the next section the pattern will be applied to the running examples of previous chapters.

4.4 Formalisation

Now that the formal language has been defined, and the connection between *GDL* and the goal model has been established, the actual formalisation step may take place. It is important to realise that in this stage the formalisation will not be complete: the atomic conditions and events themselves are not yet rewritten to a *GDL* extension (i.e. ***GDL***-Asbru). The reason for this lies again in the independence of the goal to any specific model in this stage of the conversion.

Unfolding condition- and event combinations is part of the formalisation. The connectives that have been defined in *GDL* (Element 7, 8, page 90), must be introduced in the natural language. Consider the following event specification:

"11 months after a cholesterol or HDL-cholesterol measurement"

According to the specification both events need an offset of their own. Keeping that in mind, the formalisation becomes:

Cholesterol measurement $\overset{+}{\longmapsto}$ 11 months **or**
HDL-cholesterol measurement $\overset{+}{\longmapsto}$ 11 months

By application of the patterns, the task of partial formalisation is reduced to unfolding the content of the structured version, and inserting it to the variable fields in the template.

4.4.1 Example 1

From page 3.2.2 on, the first example has normalised to a repetitive goal together with a bootstrapping goal. The bootstrapping part looks a bit like template 1: *Repetitive goal*. It has been adapted by replacing the (repetitive) start event by a regular event.

Normalisation - Example 1.1: bootstrap

^{G}[For people],^{S}[from the start of the diabetes care], ^{E}[within 12 months], ^{B}[cholesterol or HDL-cholesterol must be measured.]

Formalisation - Example 1.1: Bootstrap

Goal Example 1.1
Precondition
For people

Time-specification
From the start of the diabetes care
Duration 12 months

Observe-during-period $\geqslant 1$
Cholesterol measurement **or** HDL-cholesterol measurement

Notes:

1 **Formalization** *duration* has been chosen instead of *open-duration* since there is no requirement that cholesterol must be measured even when the care takes less than 12 months

The repetition part is modelled as a *repetitive goal with explicit bounds* (pattern 2, and templates 2a and 2b).

Normalisation - Example 1.2: Repetition

^{G}[For people],^{S}[at earliest 11 months after a cholesterol or HDL-cholesterol measurement], ^{E}[and at latest 13 months after that measurement], ^{B}[another cholesterol or HDL-cholesterol measurement must be performed once.]

For the formalisation, the combined events (**or**) of both the *start* and *behaviour* parts in the structured version of example 1.2 have been separated. The duration of the second interval was calculated according to template 2.

Formalisation - Example 1.2a: Repetition

Goal Example 1.2a
Precondition
For people

Time-specification
From
cholesterol measurement **or**
HDL-cholesterol measurement
Open-duration 11 months

Avoid-during-period
Cholesterol measurement **or** HDL-cholesterol measurement.

Formalisation - Example 1.2b: Repetition

Goal Example 1.2b
Precondition
For people

Time-specification
From
cholesterol measurement $\overset{+}{\longmapsto}$ 11 months **or**
HDL-cholesterol measurement $\overset{+}{\longmapsto}$ 11 months
Duration 2 months

Observe-during-period = 1
Cholesterol measurement **or** HDL-cholesterol measurement.

4.4.2 Example 2

Normalization - Example 2

^{G}[For people with diabetes],^{S}[between the start of the diabetes care], ^{E}[and the end of the care], ^{B}[hypoglycaemia should not occur].

The formalisation of example 2 is a straight forward substitution exercise on template 3: *Avoid during the period.*

Formalisation - Example 2

Goal Example 2
Precondition
For people with diabetes

Time-specification
From the start of the diabetes care
Until the end of the diabetes care

Avoid-during-period
hypoglycaemia

4.4.3 Example 3

Normalization - Example 3

^{G}[For women with breast-cancer],^{S}[after successful completion of breast-conserving surgery] ^{E}[until 5 years thereafter], ^{B}[local recurrence should not occur].

Formalisation - Example 3

Goal Example 3
Precondition
For women with breast-cancer

Time-specification
From successful completion of breast-conserving surgery
Duration 5 year

Avoid-during-period
local recurrence

4.4.4 Example 4

Normalization - Example 4

^{G}[For women with breast-cancer],^{S}[after start of the medical care] but ^{E}[before commencing mastectomy], ^{B}[the possibility of breast reconstruction should have been discussed with the patient].

Example 4 is formalised according to pattern/template 5: *A before B*.

Formalisation - Example 4

Goal Example 4
Precondition
For women with breast-cancer

Time-specification
From the start of the medical care
Until start of mastectomy

Observe-during-period $\geqslant 1$
discuss possibility of breast reconstruction with the patient

4.5 Conclusion

Generic GDL has been introduced as the formal expression of the goal model. On the formal side it provides a canonical form that stays very close to the structured natural language version. Due to this close tie, the domain expert is able to actually verify each and every change that is made on the formal side. (i, iii of page 7)

The patterns make that the formalisation step is usually no more than a simple matter of inserting parts of the structured version into the correct pattern. This leads - even without documentation - to a high degree of traceability and a great reduction in variability. Although some elements of the goal may sometimes be expressed in more way in *Generic GDL*, the (names of) language elements have been chosen in such a way that there is always a 'intuitive' way to do it. (iv, vi)

Since the actual events and conditions are strictly kept independent of any target model, the result of the first three steps is fully reusable for different models. Multiple attachment steps may be based on the formalised goal. (v)

In some cases it may turn out that the normalisation is not specific enough to choose a specific *Generic GDL* element. (e.g. whether to use an *until*, or a *open-until*.) Application of *Generic GDL* to the goal therefore raises questions and by answering them residual ambiguities are eliminated. The structured version must be adjusted accordingly so the domain expert can verify the choices made. (ii, iii)

Chapter 5
Attachment

Chapter 4 has yielded a partially formalised version of the structured natural language goal. This goal already contains the structure of the goal, however it is still independent of any specific process model. The actual event and conditions still need formalisation.

This second formalisation step is not trivial for multiple reasons. For example consider the following condition:

"The patient has been informed about breast-reconstruction."

Assume the goal needs to be expressed in ***GDL***-Asbru. Depending on the process model, there may be more than one way to express this. If the process model uses a boolean parameter to store whether or not the patient has been informed, the condition above would be expressed as follows:

Param PatientInformed = true

In other cases it may be more suitable to use the state of the plan to express the condition. Assume there is a plan call *InformPatient* which takes care of informing the patient. Then the above condition may look like this:

Planstate InformPatient = completed

Founding out which attachment options exist, and determining which version to choose, makes the attachment difficult. To keep the domain expert (i.e. the medical expert) involved, both versions of the goal — the structured natural language version, and the formal *GDL* expression — are rewritten in parallel to fit onto the selected process model: both version should represent the same goal at all times.

In the next section, problems that may be encountered during the attachment will be discussed and whenever possible, solutions are provided. In section 5.3 ***GDL***-Asbru will be introduced, and the actual attachment will be performed on the examples of the previous chapters.

5.1 Difficulties

The aim of the attachment step is simple: express the goal in terms of the process model so it can be verified. However, when doing so, several difficulties are encountered. Some of these were found by Van Gendt [3] in a study on how to apply indicators to guidelines. Since the problem is a special case of applying goals to a process model, the results are also applicable to the attachment step. The following problems were identified:

1. "The parameter referred to in the indicator is not used at all in the protocol."
2. "The act required by the indicator is not used in the protocol."

3. "More medical information is needed for modelling."

4. "Parameters in the protocol are not given a value that is comparable to the value mentioned in the indicator."

Missing concepts

The first two problems can be summarised as missing concepts. Missing concepts are not always a real problem. Remember that the original goal may come from any source and some may not be relevant for the model. Therefore, before proceeding any further with the attachment, the domain expert needs to establish that the selected goal is indeed relevant for the model.

In those cases where the goal is deemed relevant for the model, a missing concept may shed the first light on an anomaly in the model. The question "why is this concept not in the model?" must always be asked. Sometimes the answer can be found in the origin of the model itself. A model based on "Treatment for diabetes patients" may omit to refer to the fact that the patients have diabetes: it has not been modelled explicitly. The goal may be adjusted according to the knowledge that 'patient has diabetes' is always true during the execution.

Another possibly cause for missing concepts can be found in the level of detail of the model. A model generally represents an abstracted version of a part of the domain. When a real-world process is being modelled, a lot of 'obvious' facts can be expected to be omitted. In these cases where such a fact would be required, it may — depending on the domain and model — be an option to enhance the model to include the missing concept. Of course this can only be done if the additive is truly 'common practice' or 'common knowledge'. If not, it is a missing concept that points to an anomaly. The domain expert is the only one that can make this distinction.

A specific instance of the abstraction problem concerns the difference between process goal and result goals. A process goal is only about the actions that follow from the model. Therefore, it only needs a description of the order and conditions for those actions. A result goal addresses the changes in the world that follow from those actions. The connection between actions and results must be in the model to be able to attach such a goal. If this relation has not been specified, and there is no other source where this information can be found, the attachment will fail [1].

For a solution for a missing concept in the precondition, consider the following example:

> "^{G}[For patients with diabetes], ^{S}[from the start of the diabetes care], ..."

[1] It is important to realise there is a difference between "Treatment X will not be completed until result Y has been accomplished" and "Treatment X will result in Y". The former is a process goal and may be expressed using the complete condition of Asbru. The latter is a result goal and needs the relation between the actions of X and result Y. Verifying the latter is a subject that will not be discussed in this thesis.

There may be models that don't allow expression of 'patients with diabetes' as a condition (e.g. there may not be a parameter expressing that fact). The solution proposed here uses the fact that the start group and the precondition are closely related: combined they define the start of the period. Using that, is may be possible to rewrite the precondition into the start event. For the given example, this might be :

"$^{G}[]^{S}$[From the start of the diabetes treatment]"

Since only patients with diabetes will be treated for diabetes, such a rewrite is most likely valid in the given case. The details of the model, and the opinion of the expert determine whether or not such a rewrite is available. Special care should be taken that no relevant part of the execution is excluded from verification.

Substitution

In some cases, a missing concept is actually a case of substitution. This is basically the third problem found by Van Gendt. There are three categories of substitution. In the first case a more specific instance of the concept in the goal is available in the model. E.g. the goal may refer to anti-angina medication whereas the model may prescribe some specific sort of this type of medication. It is up to the domain expert to decide whether or not the specific concept is really an instance of the more general one. In the case that it is, the goal may be adjusted.

In the case of the inverse situation (where the model contains the more general concept), the domain expert needs to check whether the goal has been formulated too narrow. If it is, the more general concept may be inserted in the goal. If it is not, this is a sign of an anomaly in the model.

The third category of substitution occurs when both the goal and the model refer to a specific concept which can easily be exchanged. E.g. Both can mention some specific kind of anti-angina medication. If both have have the same *function* in relation to the goal - which is up to the domain expert do decide - then the goal can be adjusted.

Compatibility

The last problem found by Van Gendt is about compatibility of values. In a sense, this is again a problem regarding the detail of the model, only now concerning values instead of concepts. E.g. The goal may refer to a temperature in degrees Celsius while the model uses Fahrenheit. In such simple cases, a straight-forward transformation of a numerical value will solve the problem. The problem becomes more difficult when the *model* uses qualitative scales like 'high', 'medium', 'low'. Even if an accurate mapping to numerical values is available, comparing these with a specific value in the goal may be problematic: the precision is not enough. When such a problem arises, again this may point to an anomaly.

One possibility to cope with compatibility issues is to change the goal into a more restrictive version. If the model is consistent with the more restrictive version of the goal, it would also be consistent with the original. If it is not consistent, a less

restrictive version may be used. If this one is not consistent either, then the original may also be considered inconsistent. If it is consistent, it is up to the domain expert to draw conclusions.

When anomalies are discovered during the attachment of a goal (e.g. missing concepts), these may prevent the further use of such a goal[2]. However, slightly different versions of the goal may come to mind that can be modelled and which are also relevant in the domain. When all goals come from a carefully selected test-set, using such a deviating goal is probably not an option. In cases where good goals are hard to find (the medical domain), such an opportunistic approach may pay off in a richer set of goals.

5.2 *GDL*-Asbru

The formalisation has provided a frame of *Generic GDL* with natural language descriptions for events and conditions. The *GDL* related part of the attachment consists of replacing those natural language parts with domain specific event and conditions expressed in a *GDL*-extension. Since the models used for the examples are in Asbru, a set of Asbru specific event and conditions will be used: ***GDL***-Asbru.

The domain specific extensions are based on the features provided by the modelling language. Since Asbru is mainly about plans and parameters, these are targeted in the domain events and conditions. (For a short introduction to basics of Asbru see *Asbru* on page 47.) Table 5.1 shows the domain conditions which are available. The first two, which are very similar, compare the value of a parameter with either a value or with the value another parameter. This type of condition has **Param** as prefix. The **Planstate** prefix allows testing whether or not a plan is in a given state. Since there is no natural ordering between states only $=$ and $\neq$ is allowed for the operator (since *Generic GDL* does allow boolean combinations of conditions, this won't limit the expression).

11 **Domain-Condition**	
–	**Param** [Param]$_{\texttt{String}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Value]$_{\texttt{Str/Num}}$
–	**Param** [Param]$_{\texttt{String}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Param]$_{\texttt{Str}}$
–	**Planstate** [Plan]$_{\texttt{String}}$ [Operator]$_{\triangleright BoolOpr_{\#14,\ p.\ 91}}$ [State]$_{\triangleright State_{\#16,\ p.\ 91}}$

Table 5.1: ***GDL***-Asbru Domain conditions

Consider the following examples:

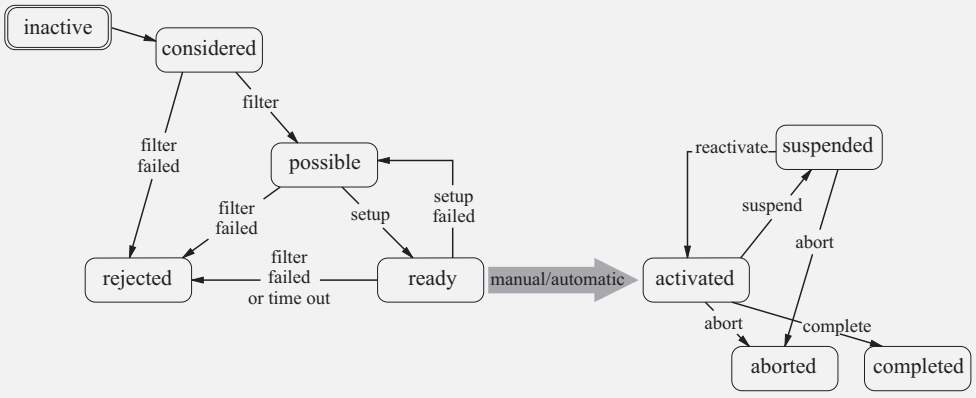
Param blood-pressure $<$ 90

[2]The fact that anomalies have been found during the formalisation process makes by definition that the goal has served its purpose: finding anomalies is the whole point of verification.

Asbru

Asbru is a plan-specification language defined as part of the Asgaard/Asbru project. It allows representation of plans for the medical domain. Guidelines are represented as a hierarchy of plans. Specific features include the use of temporal patterns in specifications, bounding intervals for parameters and temporal patterns, and a rich set to define the order of plan execution.

The plan is a central concept which contains actions and other plans, and which defines which of those plans and actions are executed in what order: sequential, any-order, parallel or unordered. Which plans are optional and which are mandatory is part of the specification. Among the possible actions is the 'ask' which initiates input of a variable. Every plan has a state-based life-cycle which is depicted here:



Transitions from one state to another may be guarded or initiated by conditions. The picture shows the different transitions and guards. The filter precondition continuously guards the left hand side of the picture. The setup precondition guards the transition to *ready*. Some transitions are coupled with the parent plan. e.g. If the parent aborts, every child also aborts.

Parameters are another feature of Asbru. A parameter allows storage of values and is used in conditions. Scales and calculated values are amongst the options. The *ask* action is used to initiate a measurement of a value.

For more information on Asgaard/Asbru refer to [11][8].

Planstate inform-patient = completed

The first condition evaluates to true if the parameters in the Asbru process model is less than 90. The second condition can be used to test whether plan 'inform-patient' is in de the *completed* state.

Table 5.2 shows the Asbru related events. The **Transition** event monitors state transitions of plans. A plan may enter or leave a state. When it does, **Transition** will register the event. The **ParamUpdate** and **ParamUpdateTo** events monitor the content of parameters. The former registers an event when the value of a parameter is updated. This does not necessarily mean that the value must be changed by the update. The latter event takes this a step further by putting a constraint on the new value: only if the new value satisfies the condition, the event will register.

12 **Domain-Event**	
–	**Transition** [Plan]$_{\texttt{String}}$ [Dir]$_{\triangleright DirOpr_{\#15,\ p.\ 91}}$ [State]$_{\triangleright State_{\#16,\ p.\ 91}}$
–	**ParamUpdate** [Param]$_{\texttt{Str}}$
–	**ParamUpdateTo** [Param]$_{\texttt{Str}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Value]$_{\texttt{Str/Num}}$

Table 5.2: ***GDL***-Asbru Domain events

The following events describe the update of the 'blood-pressure' parameter, and the transition to completed of the 'inform-patient plan':

ParamUpdate blood-pressure
Transition inform-patient **enter** completed

5.3 The attachment

The attachment process will be explored and demonstrated by going through the process of attaching the 4 examples of the previous chapters. Since less concrete guidelines can be given for the attachment than for the other steps, the documentation of this step is even more important. All changes should be documented in such a way that both the formal methods expert and the domain expert are able to reconstruct and validate the result.

Any changes to the goal that are introduced during the attachment, will be mentioned in the notes, and marked in the structured version by underlining the relevant part.

5.3.1 Example 1

Ref: Formalisation 4.4.1, p. 39

Example 1 will be attached to the Asbru formalisation of the Diabetes protocol. The version of 28.8.2005 is used. Although it is impossible to provide a standard

way to perform the attachment, an element-by-element inspection will probably work in most cases. Starting with the time-group, it appears that *'For people'* does not provide any distinction when related to this model: the process model is about people that are under care of a doctor for (possible) diabetes. *'For people'* contains this group, and it can therefore be left out.

The start group only requires a further formalisation step into ***GDL***-Asbru. *'the start of the diabetes care'* coincides with the start of the execution of the guideline and thus with the start of the Asbru model. The *start* element is inserted.

The end element does not need any attention and therefore only the behaviour group needs further inspection. Both 'cholesterol' and 'HDL-cholesterol' need to be attached. There are two parameters in the model that contain those values and there is a plan named *'Cholesterol tests'* which asks for those values. There are no other locations where those values are being asked for[3]. To make sure that every measurement is used, the parameters themselves are used for the attachment instead of the plan. A measurement is formalised as a parameter update. In this case there is no need to change the structured version on this point.

The repetition part of the normalisation and formalisation do not contain any other elements than the bootstrap part. Therefore, the same considerations apply as stated above. These are the results of the attachment of example 1.

Attachment - Example 1.1: Bootstrap (DMT2 Protocol 28.8.2002)

^{G}[ ]^{S}[From the start of the diabetes care], ^{E}[within 12 months], ^{B}[cholesterol or HDL-cholesterol must be measured.]

Goal Example 1.1
Precondition
always-true

Time-specification
From start
Duration 12 months

Observe-during-period $\geqslant$ 1
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

[3] At this stage there is still discussion about the so called 'implicit ask'. The main question is whether using a value in a condition will automatically trigger an ask on that parameter when it is encountered. A doctor would probably do so when the available value is too old to be trusted.

Notes:

1 **Attachment** The attachment target is the Asbru formalisation of the DMT2 protocol, version 28.8.2002. The ***GDL***-Asbru extension is used.

2 **Attachment** 'For people' is left out. In *GDL* this translates to *always-true* in the precondition. In the structured version the group is made empty.

3 **Attachment** 'Start of the diabetes care' coincides with the start of the Asbru execution. Therefore *start* is inserted as start event.

4 **Attachment** Cholesterol measurement $\mapsto$ **ParamUpdate** total-cholesterol

5 **Attachment** HDL-Cholesterol measurement $\mapsto$ **ParamUpdate** HDL-cholesterol

Attachment - Example 1.2: Repetition (DMT2 Protocol 28.8.2002)

^{G}[$\_\_$]^{S}[At earliest 11 months after a cholesterol or HDL-cholesterol measurement], ^{E}[and at latest 13 months after that measurement], ^{B}[another cholesterol or HDL-cholesterol measurement must be performed once.]

Goal Example 1.2a
Precondition
always-true

Time-specification
From
ParamUpdate cholesterol **or**
ParamUpdate HDL-cholesterol
Open-duration 11 months

Avoid-during-period
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

Goal Example 1.2b
Precondition
always-true

Time-specification
From
ParamUpdate cholesterol $\overset{+}{\longmapsto}$ 11 months **or**
ParamUpdate HDL-cholesterol $\overset{+}{\longmapsto}$ 11 months
Duration 2 months

Observe-during-period = 1
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

Notes:

1 **Attachment** Note 1, 2, 4 and 5 of the attachment of the bootstrap part (*Example 1.1*) also apply here.

5.3.2 Example 2

Normalization - Example 2

^{G}[For people with diabetes],^{S}[between the start of the diabetes care], ^{E}[and the end of the care], ^{B}[hypoglycaemia should not occur].

Example 2 is a typical result goal. With this goal it must be possible to verify that the steps in the model add to the prevention of hypoglycaemia. To do so, a parameter is needed that continuously shows whether hypoglycaemia would have occurred given the actions. Although a boolean *hypoglycemia* is available in the model, this parameter does not represent continuously the current status of the patient: it only serves as a storage for the hypoglycaemia status when it is requested by means of an 'ask' in the model. It may even be that the 'ask' is never reached. Therefore, attachment to this parameter is not correct.

To problem found here, occurs with every purely action oriented process model in combination with a result goal. While the model only prescribes the order of actions, the goal is about the effects of those actions. To attach this category of goals anyway, an extra model should be added to the attachment: a model that describes the effects of the actions, on the subject of those actions. Then there could be two partial attachments, which combined form a full attachment.

If such a model would be present for diabetes patients (e.g. a model of a typical diabetes patient), then 'hypoglycaemia' could be attached to this other model[4]. The goal could then be verified for this 'typical patient'. (Of course, both models would need to interact in such a way that the 'typical patient' responds according to the actions prescribed in the first model.)

5.3.3 Example 3

Normalization - Example 3

^{G}[For women with breast-cancer],^{S}[after successful completion of breast-conserving surgery] ^{E}[until 5 years thereafter], ^{B}[local recurrence should not occur].

[4]Work is being done on modelling patient behaviour with respect to treatment. People from the *Radboud Universiteit Nijmegen* together with people from the *Universität Augsburg* have succeeded in modelling small parts of a 'prototypical' patient and used this information in the KIV symbolic verifier. When this work evolves it may allow full attachment of result goals.

Just like the previous example, example 3 is result goal. Also in this case there is no detailed and reliable information available that allows verification of the goal: whether or not local recurrence will occur cannot be derived from the existing model alone.

5.3.4 Example 4

Ref: Formalisation 4.4.4, p. 42

Example 4 will be attached to the 23.11.2005 Asbru model of Chapter 1 of the breast-cancer guideline. This chapter is about diagnosis and treatment of women with DCIS. Inquiry with the medical expert has shown that the group specification ('woman with breast cancer') may in this case be considered equal to 'women diagnosed with DCIS'. Therefore, the addition *'For women with breast-cancer'* is left out.

It was also found that in this model, the goal may easily be limited to the treatment phase of the care. Whether or not to adjust the start-group is mainly a matter of personal preference. Doing so would most likely reduce the proof effort: a smaller part of the plan hierarchy needs to be examined. At the same time it does alter the essence of the goal slightly: it is no longer enough when during diagnosis the options are discussed with the patient. The patient now explicitly needs to be informed during the treatment phase. In this example — for demonstration purposes — the start group will be adjusted accordingly. The parent plan that contains all the treatment steps is called *ch1-treatment*. The start group of the structured version becomes: ^{S}[After the start of the BC treatment]. The corresponding ***GDL***-Asbru expression is: '**Transition** ch1-treatment **enter** active'.

Now there are two concepts left for attachment: the act of commencing mastectomy and discussing the options with the patient. For mastectomy there is a plan in the hierarchy with that same name. This plan has a child called *mastectomy-proper* which models the actual mastectomy. The latter is perfectly suitable to attach the start of of performing mastectomy to: '**Transition** mastectomy-proper **enter** active'.

Concerning the text 'the possibility of breast reconstruction should have been discussed': informing the patient must have been completed before proceeding with mastectomy. The plan *patient-information-reconstruction* models the action and '**Planstate** patient-information-reconstruction = completed' expresses the fact that is must indeed be in the completed state. That finalises the attachment:

Attachment - Example 4 (BC Ch1 23.11.2005)

^{G}[ ],^{S}[After the start of the BC treatment] but ^{E}[before commencing mastectomy], ^{B}[the possibility of breast reconstruction should have been discussed with the patient].

Goal Example 4
Precondition
always-true

Time-specification
From Transition ch1-treatment **enter** active
Until Transition mastectomy-proper **enter** active

Observe-during-period
Planstate patient-information-reconstruction = completed

Notes:

1 **Attachment** The attachment target is the Asbru formalisation of the Dutch "Richtlijn - Behandeling van het mammacarcinoom", Chapter 1, version 23.11.2005. The ***GDL***-Asbru extension is used.

2 **Attachment** 'Woman with breast cancer' is considered to be equal to 'women diagnosed with DCIS', which is the domain of the chapter. Therefore, the group specification has been discarded. Medical Expert: *DCIS is an early form of breast cancer.*

3 **Attachment** A treatment plan is available in the process model. Since the actions prescribed should according to the medical expert be part of the treatment, the start event has been adjusted accordingly.

4 **Attachment** *mastectomy-proper* models the actual act of performing the mastectomy while its parent, the *mastectomy* plan, groups several treatment steps. Therefore, *mastectomy-proper* fits best on 'performing mastectomy'.

5 **Attachment** Since the operand of *Observe-during-period* has become a condition instead of an event, the count requirement has been removed.

6 **Attachment** Start of the BC treatment $\mapsto$ **Transition** ch1-treatment **enter** active

7 **Attachment** commencing mastectomy $\mapsto$ **Transition** mastectomy-proper **enter** active.

8 **Attachment** the possibility of breast reconstruction should have been discussed with the patient $\mapsto$ **Planstate** patient-information-reconstruction = completed

5.4 Conclusion

The attachment is the most difficult step of the conversion process. Mapping concepts correctly to the associated element in a formal model requires careful evaluation of the exact function of the concept within the goal and in the model. The previous chapter isolated the individual concepts within the goal. Since that simplifies attachment into isolated concept-to-concept mappings, the variability is reduced greatly. (vi of page 7)

If there would be any doubt regarding the specific meaning of any concept in the goal, now it would become clear. In those cases it is essential to go back to the normalisation and clarify those concepts by means of additional notes. (ii)

By making the different reasons for failure of the attachment clear, these cases can be recognised in an early stage which prevents errors. Also by updating both the formal version and the structured version at the same time, the correctness of each and every step can be verified. Adding the proper documentation ensures traceability. (iii, iv)

The result of the attachment will be the source for the final translation. When different verification tools are used for one model, multiple translations may be required. None of the work previously done has to be repeated to get to those translations: although by now it is dependent of a specific model, the attachment result is fully reusable for different target formalisms. (v)

Chapter 6
Translation

6.1 Mechanical translation

Once the attachment has been completed, only the translation to the formalism of the verification tool is left. This translation should be a strictly mechanical step: once the validity of the translation function has been established, mechanically applying it will ensure that the semantics of the goal will not change. This is essential since changes in this stage would be impossible to detect and validate by the domain expert. The XML version of *GDL* allows for automatic translations.

The mechanical nature of the translation makes it trivial from a process perspective. No considerations, other than that the translation should have the same semantics as *GDL*, need to be taken into account.

Section 6.2 will introduce the translation function of *Generic GDL* and ***GDL***-Asbru to KIV — one of the tools used in the Protocure project. This translation is interesting both as an example of the translation step, and as a unconventional way to express goals in KIV that yields some interesting properties.

Section 6.3 contains the KIV translations for examples 1 and 4 of the previous chapters, followed by a section on optimisation. Finally section 6.5 contains the conclusion.

6.2 *GDL* to KIV

KIV is one of the verification tools used in the Protocure project. Amongst others, KIV offers the possibility to perform verification of parallel programs by symbolic execution. For a brief introduction to KIV, refer to *KIV* on page 56. For more information one the implementation of parallel programs, refer to *Temporal logic in KIV* on page 57.

For the Protocure project, people from the formal methods group at the Universität Augsburg have implemented a subset of Asbru as a (parallel) program in KIV. With this implementation, it has become possible to verify Asbru models through symbolic execution. During previous chapter the goals were attached to such models. The goal and the initial state of the model will be combined in one sequent which has to be proven.

Several techniques have been, and are being developed to make verification of large sets of parallel programs feasible. Amongst others, there are quite successful efforts to reduce the state explosion associated with parallel programs by using plan abstractions in which the plan state is no longer important [10].

In the next section, a brief introduction to the current Asbru implementation in KIV will be given. Only the parts relevant for this chapter will be described[1]. After this introduction, the ideas and concepts behind the KIV expression of *GDL* are explained in section 6.2.1.

[1]Since the implementation is ongoing work, the specification will most likely be subject to changes in the (near) future.

KIV

KIV, the *Karlsruhe Interactive Verifier*, is an advanced tool for the development of correct software, including formal specification and verification. The KIV system originates from 1986 at the University of Karlsruhe. Currently, branches of KIV are located at Karlsruhe[6], Saarbrücken[5], and Augsburg[4].

KIV implements a sequent calculus based on *First-* and *Higher order logic*, extended with *Dynamic logic* and *Temporal logic*. A sequent consists of two parts: the antecedent, and the succedent. The disjunction of the formulas on the succedent should follow from the conjunction of formulas on the antecedent:

$$(\phi_0, \ldots, \phi_n \vdash \psi_0, \ldots, \psi_m) \Leftrightarrow (\phi_0 \wedge \ \ldots \wedge \ \phi_n \Rightarrow \ \psi_0 \vee \ \ldots \vee \ \psi_m) \tag{6.1}$$

Constructing a proof in KIV is done by rewriting the sequent into one or more simpler sequent which at one point can be proven trivially. (Either by a contradiction on the antecedent, or when a term appears both on the antecedent and on the succedent.) Formula 6.2 shows an example sequent:

$$A \ \vee \ B, \ \neg B \vdash A \tag{6.2}$$

Proving this sequent would require splitting $A \ \vee \ B$. This results in two new sequent, which — if both proven — provide proof of the original:

$$B, \ \neg B \vdash A \ \ \textit{and} \ \ A, \ \neg B \vdash A \tag{6.3}$$

The first sequent in formula 6.3 contains a contradiction on the antecedent and is therefore true. The second sequent on the other hand contains A on both sides which makes it also true. By repeatedly applying simplifications, a proof tree is built. When all nodes are closed, the root node may be considered proven as well, and so is formula 6.2.

A set of standardised simplification rules has been implemented, and a powerful automatic simplifier is able to automatically reduce the complexity of first order logic formulas to a large extend. Many proof branches are automatically closed this way.

In the cases where the automatic simplifier is not able to perform any further simplifications, user interaction is required. The user may — amongst others — manually select simplification rules, backtrack or apply lemmas.

Programs are either represented as dynamic logic formulas, or as temporal logic formulas. For these logics there are also simplification rules available which can be applied automatically. However, support for automatic application of those rules is not yet as comprehensive as for first order logic.

In the verification of programs *induction* playes an important role: in short, if at some point in the trace an invariant can be found which makes the goal hold, the rest of the trace may be considered consistent.

Temporal logic in KIV

One of the logics supported by KIV is Temporal Logic. Temporal logic was added by Michael Balser of the Universität Augsburg, and described in his PhD-Thesis [1].

With temporal logic it becomes possible to express formulas that relate to future states. Amongst the implemented operator are *always* ($\Box$), *eventually* ($\Diamond$) and *next* ($\bullet$ and $\circ$). Programs can be specified using a Pascal-like program syntax. Internally, the programs are represented as temporal formulas. Support for parallel formulas (and therefore, support for parallel programs) is added via the *parallel* operator ($\|$).

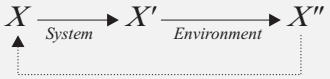
The *step* rule advances the program on the sequent to the next step (symbolic execution). In case of multiple execution options, multiple branches are spawned in the proof-tree. For example, this may be the case when two or more programs are running in parallel and multiple interleaving options exist.

To illustrate a step, here the behaviour of the *if* is shown and a simplified explanation is provided:

$$\textbf{if } \phi \textbf{ then } \psi_1 \textbf{ else } \psi_2 \;\Leftrightarrow\; (\phi \;\wedge\; \psi_1) \;\vee\; (\neg\phi \;\wedge\; \psi_2) \tag{6.4}$$

The *if* itself does not actually require execution. The *step* first rewrites it to the form on the right hand side of the equivalence. Then the *step* is applied to ψ_1 and ψ_2. The first instruction in ψ_1 (and likewise in ψ_2) is 'executed' by rewriting it to the execution result. The rest of the program in ψ_1 is taken to the next state. E.g. assume $\psi_1 : [N := 5;\ \psi_1']$, then applying *step* to ψ_1 yields: $N' = 5 \;\wedge\; \circ\, [\psi_1']$ (simplified). This replaces ψ_1 (and likewise ψ_2) on the right side of eqn. 6.4. The resulting formula causes a case-split due to the 'or' which yields two branches. If ϕ is known, then one of the branches will be closed immediately. Otherwise, both branches remain open.

To allow proof-abstractions even when using parallel programs, variables on the sequent are updated in two stages during the *step*: X → X' → X" (notice the upper-case). The first transition, called the *system step* is defined by the program. If the program wants to set the value of a variable for the next state, then it assigns the new value to X'. If no new value is assigned to a variable, the old value is retained automatically: $X' = X$. The second transition is called the *environment step*. During the environment step, values are not retained by default. This means that formulas on the sequent should explicitly specify the relation between X' and X''. Once the whole *step* has been completed, the new X gets the value of the previous X''.



Apart from the described *dynamic* variables, there are also static variables: once known, they never change. Static variables are written lower-case by convention.

6.2.1 Asbru in KIV

Asbru evolves around plans which transition from state to state (see *Asbru* on page 47). To keep track of the state in KIV, the state information of each plan in the hierarchy is stored in a hash-table like structure: `AS` (*Asbru State*). During the *step*, the environment takes care of the state transitions when applicable (according to Asbru semantics). On the sequent, known states of the individual plans are visible like this: `AS['treatment'] = inactive`. Since `AS` is just a regular variable, it is also possible to refer to `AS'['treatment']` and `AS''['treatment']`.

A clock, which progresses during execution, is part of the implementation. The clock is named `AC` and each *step* it is incremented by one. The clock is abstract in the sense that one 'tick' is not bound by default to real-world time. If this is required, this has to be defined explicitly. For the rest of this chapter it is assumed that one tick coincides with one second.

The *Patient Data History* — `PDH` — contains the patient record: the last known measured values of the patient. An entry is defined by a clock value, and a variable name: `PDH[Clock]['blood-pressure']`. Two fields can accessed. The first is `.value`. Referring to this field, yields the value of the variable. An additional `.update` field has been added to support detection of writes. If a value was just written, `.update` is true. Since `AC` contains the current value of the Asbru clock, `PDH[AC]['blood-pressure'] .value` accesses the last recorded blood pressure.

For the remainder of the chapter, the antecedent of the sequent can be visualised to contain the program and a list of formulas representing the plan states, patient data, the clock and other Asbru related information. In other formulas, these values can be referenced. The basic format for any proof regarding programs is a follows:

$$[\textbf{Program}] \wedge \textbf{StateInfo} \vdash \textbf{Goal}$$

At all times, for every reachable state, the goal must hold.

6.2.2 Basic principles of the translation

Any goal that is expressed in ***GDL***-Asbru is about values and events regarding plan-states and parameters. In this section gradually a goal expression is built. To provide an idea of how individual events will be translated, here the translation is given for the transition event:

$$\frac{T\left(\textbf{Transition } \mathcal{A}_{1:Plan} \textbf{ enter } \mathcal{A}_{2:State}\right)}{\texttt{AS['}\mathcal{A}_1\texttt{']} \neq \texttt{T(}\mathcal{A}_2\texttt{)} \wedge \texttt{AS''['}\mathcal{A}_1\texttt{']} = \texttt{T(}\mathcal{A}_2\texttt{)}}$$

Effectively the formula says: 'If currently the state of $\mathcal{A}_1$ is not yet $\mathcal{A}_2$, and in the next state it is, then this formula is true'. (The double primed version will be the non-primed version in the next state.) For each ***GDL***-Asbru event and condition, such a translation is available which can be inserted in the rest of the translation. In the next section, the general structure of the translation of *Generic GDL* is introduced.

A simple behaviour

Using the event and condition translations, the goal body itself can be translated. Instead of trying to find a single temporal logic formula to put on the succedent, an approach with trigger variables has been chosen. For example, assume a variable called `StartEventTrigger` existst which is false until the start event has been detected. Using this variable, some behaviour may be described. E.g. A blood pressure lower than 70 should be maintained after the start event. This could be written as:

```
StartEventTrigger ⇒ PD[‘Blood-pr’] < 70
```

If a variable called `EndEventTrigger` would become true once the end-event has occurred, then then the expression could be extended to:

```
StartEventTrigger ⇒ PD[‘Blood-pr’] < 70 ∨ EndEventTrigger
```

In plain English: ‘After the start event, the blood pressure needs to be below 70 unless the end event has been detected. This is exactly what the goal model is about. Since the formula should always be true instead of just in one state, the *always* operator (□) needs to be added. The final behavioural description then becomes:

```
□ (StartEventTrigger ⇒ PD[‘Blood-pr’] < 70 ∨ EndEventTrigger)
```

This behaviour coincides with the ***Maintain-during-period*** *Blood-pr* $<$ *70* behaviour. Now, the task left is to provide the administrative formulas for the `StartEventTrigger` and the `EndEventTrigger` according to the *GDL* semantics.

StartEventTrigger

During the execution of the program, there needs to be detection of the start event. The actual detection has already been shown at the start of section 6.2.2, however, `StartEventTrigger` needs to be updated accordingly. In pseudo-code:

```
If Start Event Then
   StartEventTrigger'' := true
```

`StartEventTrigger''` is used here for a good reason: in each and every state, `StartEventTrigger` has a well defined value. When it is *false* — which it is while the start event has not yet occurred — it is impossible to make it *true*: `StartEventTrigger` can not be *true* and *false* in the same state. The only possibility is to make `StartEventTrigger` *true* in the next state. This is exactly what happens when assigning *true* to the double primed version. Of course now the event detection formulas need to be formulated in such a way that an event is detected in advance. E.g. the translation for **Transition** evaluates to *true* when the event will occur during the next *step*.

Since by default the environment will not preserve the values of dynamic variables between states, this needs to be explicitly added. Therefore, this constraint to the environment will be added in the *else* part: if the start event is not detected, then the value of `StartEventTrigger` in the next state will be equal to the value in the current state.

```
If Start Event Then
   StartEventTrigger'' := true
Else
   StartEventTrigger'' := StartEventTrigger'
```

To make the if-clause comply with the *GDL* semantics, the precondition needs to be incorporated in the formula. The semantics dictate that the precondition is true in the state that comes right after the event. The condition should therefore be checked for the patient data of the next state: `PDH''`.

A final addition to the if-clause concerns the clock. For goals that specify a duration for the time-limiter, the time of the start event needs to be recorded. A variable `PS-TReP` (*Period Start-Time Reference Point*) is used for this purpose. It gets its value once the event is detected. If not, the old value is propagated. Since `PS-TReP` should not be reset if the start event would occur again, an extra check of `StartEventTrigger` is added to enforce single triggering.

```
If ¬StartEventTrigger ∧ Precondition" ∧ Start Event Then
   StartEventTrigger'' := true
   PS-TReP'' := AC''
Else
   StartEventTrigger'' := StartEventTrigger'
   PS-TReP'' := PS-TReP'
```

Now the only thing left to add is the initial value for the `StartEventTrigger`. `PS-TReP` does not need an initial value since the formula assures assignment before it is used any further. Altogether this yields the following expression in KIV notation:

```
(:  Initial system state :)
¬StartEventTrigger,

(:  Start event detection :)
□ (      ¬StartEventTrigger ∧ Precondition" ∧ Start Event
    ⊃    StartEventTrigger'' ∧ PS-TReP'' = AC''
    ;       StartEventTrigger'' = StartEventTrigger'
         ∧  PS-TReP'' = PS-TReP')
```

EndEventTrigger

The administration formulas for the `EndEventTrigger` are very similar to those of the `StartEventTrigger`. The main difference is the absence of the precondition

and the PS-TReP variable. Next to the event itself, StartEventTrigger is on the if-clause. This is required, since the EndEventTrigger is only relevant once the start event has occurred. Finally, an extra clause is added stating that if the end event has occurred, from the next state on, EndEventTrigger will always be true. This extra knowledge allows KIV to close the proof immediately after the end event. The following definition will result:

```
(:  Initial system state :)
¬EndEventTrigger,

(:  End event detection :)
□  (       StartEventTrigger ∧ End Event
      ⊃    EndEventTrigger'' ∧ • (□ EndEventTrigger)
      ;    EndEventTrigger'' = EndEventTrigger')
```

6.2.3 Events, conditions and behaviours

This section will discuss the translation of some *GDL* events, conditions and behaviours. This selection of events and conditions allows translation of example 1 and 4 of the previous chapter and illustrates the way *GDL* can be expressed in KIV.

Event translations

Some events require bookkeeping (e.g. the delayed event needs to keep record of the the expired time since the occurrence of the event itself). Therefore, event translations consist of two parts. The first part is called *support* and is put between the other formulas on the antecedent. This part is responsible for taking care of administrative tasks like taking care of time or counting. It keeps track of support variables, and allows to 'detect' the event as a whole correctly. If *support* is not defined, it is considered to be empty.

The second part is called *test*. The *test* part of the translation is inserted at the location where the occurrence of the event is actually tested (E.g. at the location of the bold printed End Event in the *EndEventTrigger* section). Here the support variables (if any) are put to use. First here are the translations of the two necessary Asbru events:

T (**Transition** $\mathcal{A}_{1:Plan}$ **enter** $\mathcal{A}_{2:State}$)

Test: (AS['$\mathcal{A}_1$'] $\neq$ T($\mathcal{A}_2$) $\wedge$ AS''['$\mathcal{A}_1$'] = T($\mathcal{A}_2$))

T (**ParamUpdate** $\mathcal{A}_{1:Parameter}$)

Test: (PDH''[AC]['$\mathcal{A}_1$'] .update)

The *GDL start* element denotes the start of the execution. Formally, the implementation shown below breaks with the *GDL* semantics with respect to the first state: *GDL* includes the first state in the interval, while the translation starts checking the behaviour from the second state. However, given the details of the KIV/Asbru translation, this will not influence the outcome of the verification[2].

T (**start**)

Support: FirstState, □ (¬FirstState'')
Test: (FirstState)

The translation of delayed events for use in the *From* part of the time specification is given below (*DSE* stands for *Delayed Start Event*). Note that this translation cannot be applied to delayed events used as operand to the *(open-) until*, nor as operand to any behaviour. This is caused by the fact that the provided translation, detects the expiring of the delay one state ahead. For *(open-) until* and operands to for example *avoid-during-period*, detection in the state itself is required.

$T_{From}\left(\mathcal{A}_{1:Event} \overset{+}{\longmapsto} \mathcal{A}_{2:Time}\right)$

Support: ¬DSE-Occurred,
□ (¬DSE-Occurred ∧ $T(\mathcal{A}_1)$
⊃ DSE-Occurred'' ∧ DSE-TReP'' = AC
; DSE-Occurred'' = DSE-Occurred'
∧ DSE-TReP'' = DSE-TReP')
Test: (DSE-Occurred''
∧ diff(AC, DSE-TReP'') < seconds($\mathcal{A}_2$)
∧ diff(AC'', DSE-TReP'') ⩾ seconds($\mathcal{A}_2$))

In the *support* part, the actual event is detected, and once it happens, the time of the occurrence is stored in DSE-TReP. The test part checks whether the delay will expire during the next state. When used as start event, this means that from that state on the behaviour will be checked. The requirement for this to work is that every step in the Asbru/KIV model is of equal length (e.g. 1 second). For simplicity it is assumed here that $\mathcal{A}_2$ contains the time of the delay in seconds, or that the given time will be converted to seconds.

Although not really an event, but nevertheless used as an end event, is a *duration* specification. The *duration* evaluates to true when the duration expires during the current state. This makes the EndEventTrigger true in the subsequent state, and therefore the behaviour is checked up-to and including the state in which the duration expires:

[2]A translation which would take the first state into account is the following: StartEventTrigger ∧ □ (StartEventTrigger''). However, this would replace start event detection formulas entirely. For clarity, the structure that has been introduced so far will be maintained, which leads to the slightly deviating semantics.

$T\,(\textbf{Duration}\ \mathcal{A}_{1:Time})$

Test: (diff(AC, PS-TReP'') $<$ seconds($\mathcal{A}_1$)
$\wedge$ diff(AC'', PS-TReP'') $\geqslant$ seconds($\mathcal{A}_1$))

Condition translation

For the examples of the previous chapter only one condition is required, which is shown here:

$T\,(\textbf{Planstate}\ \mathcal{A}_{1:Plan} = \mathcal{A}_{2:State})$

Test: (AS['$\mathcal{A}_1$'] = T($\mathcal{A}_2$))

Behaviour translations

Here the translations are provide for *avoid-during-period* and *observe-during-period*. However, since the exact semantics are depending on which time delimiter has been chosen (e.g. *until* vs. *open-until*), the context of the translation is added to the translation function.

As with events, some behaviours require bookkeeping. Therefore, for behaviours the translation has been split into a *support* part and a *check* part. The *support* part will be put on the antecedent again. The *check* part belongs on the succedent and described the core behaviour.

$T_{open-duration}\,(\textbf{Avoid-during-period}\ \mathcal{A}_{1:Event})$

Check: $\Box$ (StartEventTrigger $\Rightarrow$ $\neg$T($\mathcal{A}_2$) $\vee$ EndEventTrigger'')

Avoid-during-period does not require bookkeeping of any kind. Since *open-duration* is used, any occurrence of the event *before* the end event is seen causes immediate failure of the goal. If the end event occurs simultaneously with the guarded event, the goal is not broken.

The *observe-during-period* in a non-open environment, requires that the number of occurrences is checked once the end event has actually been observed. This requires counting of the occurrences from the moment `StartEventTrigger` becomes true. The final check is made in the state where the `EndEventTrigger` is about to become true. In this state the count cannot be changed any more, since for an event to be counted it must occur completely between the start and the end.

$T_{until+duration}$ (**Observe-during-period** $\geqslant$ n $\mathcal{A}_{1:Event}$)

```
Support: ObsCount = 0,
         □ (        StartEventTrigger ∧ T(A1)
                 ∧  ¬EndEventTrigger''
              ⊃  ObsCount'' = ObsCount' + 1
              ;  ObsCount'' = ObsCount')
Check:   □ (     ¬EndEventTrigger ∧ EndEventTrigger''
              ⇒  ObsCount ⩾ n)
```

Observe-during-period in a non-open environment with a condition as operand is translated similarly. Since the exact count is not important, a boolean is used instead of a number. Since `Observed` may be changed during the last interval, `Observed''` is checked.

$T_{until+duration}$ (**Observe-during-period** $\mathcal{A}_{1:Condition}$)

```
Support: ¬Observed,
         □ (        StartEventTrigger ∧ T(A1)
                 ∧  ¬EndEventTrigger
              ⊃  Observed''
              ;  Observed'' = Observed')
Check:   □ (     ¬EndEventTrigger ∧ EndEventTrigger''
              ⇒  Observed'')
```

Since the translations shown are sufficient to translate the examples with, the other *Generic GDL* behaviours will not be translated here. However, if required, these other behaviour can be translated using similar structures as the provided translations.

6.3 Translation

After attachment, two out of four examples are left for translation: example 1 and example 4. First the translation of example 1 will be discussed, followed by the translation of example 4. Due to the relatively simple nature of the 4th example, some additional support formulas will be introduced there.

6.3.1 Example 1

During the previous chapters, example 1 has been expressed using three separate goals. Each of these goal must hold in order for example goal one to hold. The three different goals may be proven individually, or all at once. For the latter it is required to put them together in one sequent. For clarity, here the different goals will be kept separately.

Bootstrap

Attachment - Example 1.1: Bootstrap (DMT2 Protocol 28.8.2002)

Goal Example 1.1
Precondition
always-true

Time-specification
From start
Duration 12 months

Observe-during-period $\geqslant 1$
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

Translation - Example 1.1: Bootstrap (DMT2 Protocol 28.8.2002)

```
(:  Initial system state :)
¬StartEventTrigger, ¬EndEventTrigger, FirstState, ObsCount = 0,

(:  Support for 'start' in From:)
□ (¬FirstState''),

(:  Start event detection :)
□ (     ¬StartEventTrigger ∧ true ∧ FirstState
      ⊃ StartEventTrigger'' ∧ PS-TReP'' = AC''
      ;    StartEventTrigger'' = StartEventTrigger'
         ∧ PS-TReP'' = PS-TReP'),

(:  End event detection :)
□ (        StartEventTrigger
         ∧ diff(AC, PS-TReP'') < seconds(31536000)
         ∧ diff(AC'', PS-TReP'') ⩾ seconds(31536000)
      ⊃ EndEventTrigger'' ∧ • (□ EndEventTrigger)
      ;  EndEventTrigger'' = EndEventTrigger'),

(:  Support part of the behaviour :)
□ (        StartEventTrigger ∧ ¬EndEventTrigger''
         ∧    (PDH''[AC]['cholesterol'] .update )
            ∨ (PDH''[AC]['HDL-cholesterol'] .update)
      ⊃ ObsCount'' = ObsCount' + 1
      ;  ObsCount'' = ObsCount')
⊢
(:  The check part of the behaviour :)
□ (¬EndEventTrigger ∧ EndEventTrigger'' ⇒ ObsCount ⩾ 1'')
```

The translation of the bootstrap goal is reasonably straight forward. It consists of a combination of the translation elements discussed before. Verification of this goal would look as follows. In the first state, the condition within *start event detection* would be true. This causes the *then* part to be added to the sequent. The result will be that after the next step, `StartEventTrigger` will be on the sequent, just like `PS-TReP = 1`[3]. Both `StartEventTrigger` and `PS-TReP` will maintain this value for the rest of the proof due to the fact that from now on only the *else* part of *start event detection* will be executed.

`EndEventTrigger` is false, and as long as the duration has not expired, this value will be copied by the *else* part of *end event detection*. If at some stage during the steps that follow, either 'cholesterol' or 'HDL-cholesterol' is updated, the condition within *support part of the behaviour* will become true. `ObsCount` will be incremented each time this happens.

After 31536000 one-second steps (or some induction), the *end event detection* will make `EndEventTrigger''` true. As a result, the left hand side of the implication on the succedent becomes true and therefore, `ObsCount` must be greater or equal than 1. If this is not the case, there is no way to close the proof. If it is, □ `EndEventTrigger` will allow KIV to close the proof in the subsequent state (the left hand side of the implication will always be false). Although this addition is not strictly required, it does shorten the proof effort considerably.

Repetition

One aspect that has been avoided so far, is how to handle repetitive events. The semantics of *GDL* describe that any interval between a valid start event, and the subsequent end event should adhere to the behaviour. In the *bootstrap* part of the goal, a second occurrence of *start* is impossible, and therefore the given translation is valid. For the *repetition* part of example 1 however, something more is required.

Attachment - Example 1.2a: Repetition (DMT2 Protocol 28.8.2002)

Goal Example 1.2a
Precondition
always-true

Time-specification
From
ParamUpdate cholesterol **or**
ParamUpdate HDL-cholesterol
Open-duration 11 months

Avoid-during-period
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

[3]Actually, the assigned value will be a function of AC during the execution, however conceptually it contains the value 1.

For every occurrence of the StartEvent, a separate set of `EndEventTrigger`, `PS-TReP` and possibly other support variables are needed. For each of those variables, correct bookkeeping must be done. One way to achieve this, is to rewrite the translations which were used earlier with a universal quantifier:

```
□ (∀ n < CurrentNumberOfEvent .  ⌝EndEventTrigger(n) ...  )
```

Using the ∀ quantifier on the antecedent however, causes great problems for the simplifier. Therefore, another approach was taken: the required formulas are dynamically added to the sequent when a new start event occurs. The behaviour on the succedent can use the all-quantifier without any problems. This technique is demonstrated by the following translation of example 1.2a:

Translation - Example 1.2a: Repetition (DMT2 Protocol 28.8.2002)

```
(:  Initial system state :)
EventCount = 0,

(:  Start event detection :)
□ (         true
          ∧     (PDH''[AC][‘cholesterol’] .update )
             ∨  (PDH''[AC][‘HDL-cholesterol’] .update)
       ⊃     EventCount'' = EventCount' + 1
          ∧  ∃  n .  n = EventCount'


                (:  Start event bookkeeping :)
             ∧  PS-TReP''(n) = AC
             ∧  • (□ (PS-TReP''(n) = PS-TReP'(n)))
             ∧  □ (StartEventTrigger''(n))


                (:  End event detection :)
             ∧  □ (         diff(AC, PS-TReP''(n)) <
                                        seconds(28908000)
                          ∧  diff(AC'', PS-TReP''(n)) ⩾
                                        seconds(28908000)
                       ⊃     EndEventTrigger''(n)
                          ∧  • (□ EndEventTrigger(n))
                       ;  EndEventTrigger''(n) =
                                        EndEventTrigger'(n))
       ;  EndEventCount'' = EndEventCount'),
```

⇑ *Continued on the next page*

```
Continued ...
  ⇓

  ⊢
  (:  The check part of the behaviour :)
  □ (  ∀  n < EventCount .
                 StartEventTrigger(n)
              ⇒    ¬(     PDH''[AC]['cholesterol'] .update
                        ∨   PDH''[AC]['HDL-cholesterol'] .update)
                 ∨  EndEventTrigger''(n)
```

The translation roughly contains the same elements that were defined before, however this time only the start event detection is implemented as a top level formula. The whole systems evolves around the `EventCount` variable. If the start event is not detected, the value of this variable is passed on to the next state unchanged. The situation changes when the first start event is detected. At that point, `EventCount'' = EventCount' + 1` is put on the sequent. The second formula that is put on the sequent, is the existentially quantified formula. The quantifier is a way to make KIV fill in a real number for `n` in the rest of the formula (instead of some temporary static variable like `eventCount0`). By immediately stating that `n = EventCount'`, the quantifier can immediately be simplified away, and the contained formula is put on the sequent:

```
PS-TReP''(0) = AC,
• (□ (PS-TReP''(0) = PS-TReP'(0))),
□ (StartEventTrigger''(0)),
...
```

The part shown here takes care of the bookkeeping for the start event. Since these formulas will only appear on the sequent once the start event has occurred for that index (i.e. 0), no *if-then-else* is required here.

The first line sets the `PS-TReP''(0)` variable like before. The second line makes that for every subsequent state, the value is copied. The next (•) is required to avoid double assignment to the double primed version in the current state. The third line sets `StartEventTrigger''(0)` to true for the rest of the execution. Although strictly speaking the use of `StartEventTrigger` is not required, is does keep the format closer to the format originally presented.

The rest of the formulas in the scope of the quantifier result in the regular formulas of *end event detection*, however this time with a numerical index. This process can be repeated endlessly, each time yielding variables with a new index.

The succedent contains a quantified version of the behaviour. For every single start event that has been detected, required behaviour is enforced.

Repetition revisited

Attachment - Example 1.2b: Repetition (DMT2 Protocol 28.8.2002)

Goal Example 1.2b
Precondition
always-true

Time-specification
From
ParamUpdate cholesterol $\overset{+}{\longmapsto}$ 11 months **or**
ParamUpdate HDL-cholesterol $\overset{+}{\longmapsto}$ 11 months
Duration 2 months

Observe-during-period = 1
ParamUpdate cholesterol **or ParamUpdate** HDL-cholesterol

The translation of example 1.2b follows the same pattern as example 1.2a. However, this goal uses a combination of delayed start events. Since any number of events may happen, dynamic bookkeeping is required. For every event, a new `DSE-TReP(n)` is created. To be able to detect expiring of the delay of *any* of the events that happened earlier, for each occurrence a dedicated expiration detection formula is added to the sequent. This formula evaluates to a variable `DSE-Expired(n)`, which can be used for the *test*.

```
DSE-Count = 0,
□ (     Event
    ⊃     DSE-Count'' = DSE-Count' + 1
       ∧ ∃  n .  n = DSE-Count'

                 (:  Time related bookkeeping :)
              ∧  DSE-TReP''(n) = AC
              ∧  • (□ (DSE-TReP''(n) = DSE-TReP'(n)))

                 (:  Expiration detection :)
              ∧  ¬DSE-Expired(n)
              ∧  □  (        diff(AC, DSE-TReP''(n)) < Delay
                          ∧  diff(AC'', DSE-TReP''(n)) ⩾ Delay
                       ⊃  DSE-Expired''(n)
                       ;  DSE-Expired''(n) = DSE-Expired')
       ;  DSE-Count'' = DSE-Count')
```

Once the first event has been detected, `DSE-Count''` is assigned the incremented value of `DSE-Count'`. Additionally `DSE-TReP''(0)` is assigned, and a formula is added to retain the value during subsequent states. The detection formulas which

are added to the sequent initially set `DSE-Expired(0)` to *false*. Further it checks with `DSE-TReP''(0)` whether the delay has expired. `DSE-Expired''(0)` is set to *true* once it is. Otherwise, the previous value is retained.

The test part is build around an existential quantifier as can be seen in the translation. Once it has been detected, the formulas are again like in example 1.2a, using `EventCount` to keep track of the number of detected start events. Counting the number of guarded events (by the *observe-during-period* should be done independently for each interval. Therefore the support code is also part of the dynamically added formulas.

Translation - Example 1.2b: Repetition (DMT2 Protocol 28.8.2002)

```
(:  Initial system state :)
EventCount = 0, DSE-Count = 0,

(:  Delayed start event support :)
□ (      (PDH''[AC][‘cholesterol’] .update )
      ∨  (PDH''[AC][‘HDL-cholesterol’] .update)
   ⊃     DSE-Count'' = DSE-Count' + 1
      ∧  ∃  n .  n = DSE-Count'

            (:  Time related bookkeeping :)
         ∧  DSE-TReP''(n) = AC
         ∧  • (□ (DSE-TReP''(n) = DSE-TReP'(n)))

            (:  Expiration detection :)
         ∧  ¬DSE-Expired(n)
         ∧  □  (        diff(AC, DSE-TReP''(n)) <
                                          seconds(28908000)
                  ∧  diff(AC'', DSE-TReP''(n)) ⩾
                                          seconds(28908000)
                ⊃  DSE-Expired''(n)
                ;  DSE-Expired''(n) = DSE-Expired')
   ;  DSE-Count'' = DSE-Count')

(:  Start event detection :)
□  (     true ∧ (∃ n .  ¬DSE-Expired(n) ∧ DSE-Expired''(n))
      ⊃     EventCount'' = EventCount' + 1
         ∧  ∃  n .  n = EventCount'

            (:  Start event bookkeeping :)
         ∧  PS-TReP''(n) = AC
         ∧  • (□ (PS-TReP''(n) = PS-TReP'(n)))

         ⇑                          Continued on the next page
```

Continued ...

```
                    ⇓
                    ∧  □ (StartEventTrigger''(n))

                       (:  End event detection :)
                    ∧  □ (        diff(AC, PS-TReP''(n)) <
                                                  seconds(5256000)
                              ∧  diff(AC'', PS-TReP''(n)) ⩾
                                                  seconds(5256000)
                           ⊃     EndEventTrigger''(n)
                              ∧  • (□ EndEventTrigger(n))
                           ;  EndEventTrigger''(n) =
                                                EndEventTrigger'(n))

                       (:  Support part of the behaviour :)
                    ∧  ObsCount(n) = 0
                    ∧  □ (        StartEventTrigger(n)
                              ∧     (PDH''[AC][‘cholesterol’]
                                                        .update)
                                 ∨  (PDH''[AC][‘HDL-cholesterol’]
                                                        .update)
                              ∧  ¬EndEventTrigger''
                           ⊃  ObsCount''(n) = ObsCount'(n) + 1
                           ;  ObsCount''(n) = ObsCount'(n))
        ;  EndEventCount'' = EndEventCount'),
⊢
(:  The check part of the behaviour :)
□ (  ∀  n < EventCount .
                 StartEventTrigger(n)
             ⇒     ¬(     PDH''[AC][‘cholesterol’] .update
                      ∨   PDH''[AC][‘HDL-cholesterol’] .update)
                 ∨  EndEventTrigger''(n)
```

6.3.2 Example 4

Example 4 is translated again for a single occurrence of the start event. Given the structure of the plan, during the execution, treatment will only become active once.

The rest of the translation is very straight forward. The elements that were introduced in the previous sections can easily be recognised. Since duration is not used, `PS-TReP` has been left out of the translation. The *observe-during-period* requires a boolean support part which has been added to the sequent.

Attachment - Example 4 (BC Ch1 23.11.2005)

Goal Example 4
Precondition
always-true

Time-specification
From Transition ch1-treatment **enter** active
Until Transition mastectomy-proper **enter** active

Observe-during-period
Planstate patient-information-reconstruction = completed

Translation - Example 4 (BC Ch1 23.11.2005)

```
(:  Initial system state :)
¬StartEventTrigger, ¬EndEventTrigger, ¬Observed,

(:  Start event detection :)
□ (        ¬StartEventTrigger ∧ true
        ∧  AS[‘ch1-treatment’] ≠ active(@, @, @)
        ∧  AS’’[‘ch1-treatment’] = active(@, @, @)
     ⊃  StartEventTrigger’’
     ;  StartEventTrigger’’ = StartEventTrigger’),

(:  End event detection :)
□ (        StartEventTrigger
        ∧  AS[‘mastectomy-proper’] ≠ active(@, @, @)
        ∧  AS’’[‘mastectomy-proper’] = active(@, @, @)
     ⊃  EndEventTrigger’’ ∧ • (□ EndEventTrigger)
     ;  EndEventTrigger’’ = EndEventTrigger’),

(:  Support part of the behaviour :)
□ (        StartEventTrigger ∧ ¬EndEventTrigger
        ∧  AS[‘patient-info-reconstruction’] = completed
     ⊃  Observed’’
     ;  Observed’’ = Observed’)

⊢
(:  The check part of the behaviour :)
□ (¬EndEventTrigger ∧ EndEventTrigger’’ ⇒ Observed’’)
```

Notes:

1 **Translation** The translation was made under the assumption that plan 'treatment' will enter the active state only once.

6.4 Optimisation

Despite of all the automatic support of KIV, proofing a real-world goal is still a tedious job. During verification it may not be possible to close a branch of the proof tree. There may be several causes for this. The most important being that the goal does not hold, but other possibilities are that the simplification rules are not sufficient, or that the wrong rule has applied earlier in the trace. To help to distinguish between those cases, the following formulas may be added to the antecedent of the sequent (for example 4):

```
¬GoalFailed,
□ (     ¬(¬EndEventTrigger ∧ EndEventTrigger'' ⇒ Observed'')
    ⊃ GoalFailed''
    ;  GoalFailed'' = GoalFailed')
```

These formulas will result in the addition of the `GoalFailed` variable on the sequent. The condition part of the *if* in the formula is the negation of the goal on the sequent. Therefore, if the goal is broken, `GoalFailed''` will put one the sequent. This additional variable may help to distinguish between a failed goal and method-related causes.

Since extra formulas are added to the sequent, it is very important to make sure that the additional formulas will not influence the goal itself. In the formulas shown here, this is ensured by the fact that nothing is written on the succedent, and no other variables are assigned than `GoalFailed` (and of course `GoalFailed` is not part of the model).

6.5 Conlusion

The attachment result provides a good starting point to create the final translation to the required formalism. The KIV translation has been provided to demonstrate one possible target. However, at this moment, work is being done on providing translations to the SMV model checker.

The KIV translation maintains the separation between *Generic GDL* and model specific extensions. The translations for *Generic GDL*element like the behaviours and delayed events will therefore remain valid even though another *GDL* extension is used: for any new extension only the *test* part for events and conditions need to be added. This way reusability is assured again (v, on page 7).

The chosen method to express the goal in KIV allows modular proof translation. Modular translations are easily automated, which in turn assures the same translation result every time. The correctness of the resulting translation then follows from the correctness of the individual parts of the translation (iii, iv, vi).

Application of the proposed translation in KIV to actual goals with actual guidelines, has already yielded the first fully completed proof. In practice, proofing the goal has become easier since the original goal can be recognised on the sequent. The current phase within the goal model — before the start event, after the end event and in between — can be read from the sequent at all times. Additionally, many of the resulting branches of the proof tree can be closed automatically, or trivially by hand, which yields a 'clean' proof tree and makes the proof feasible. Extra information that simplifies the proof effort may be provided via the `GoalFailed` construct.

Chapter 7
Conclusion

This thesis has proposed a method of formalising natural language goals in such a way that the domain expert is involved in every step that may change the meaning of the goal. In every interpretation of natural language, choices will have to be made, and implicit assumptions will have to be made explicit. Involvement of the domain expert is essential since the domain expert is the only one who can decide whether the proposed interpretation or choice is valid and acceptable or not.

To assure a common vocabulary between the domain expert and the formal methods expert, the goal model was introduced in chapter 2. By providing both a structured natural language expression and a formal expression for this goal model, the formalisation itself has been reduced to a fairly straight forward exercise: the structured natural language expression of the goal simply needs to be expressed using the formal syntax which has similar structure. Before that task however, the original goal must first be rewritten in terms of the goal model. After the formalisation the goal needs to be fitted (attached) to the model under investigation and translated to the target formalism. The process described here almost automatically leads to the steps described in chapters 3 to 6.

7.1 Quality of conversion

To be able to assure the quality of the resulting conversion, in chapter 2, six requirements for the conversion process were formulated:

i. To direct the process and for ease of interpretation, work towards canonical forms of the goal.

ii. Identify and clarify all assumptions and ambiguities present in the original goal.

iii. Ensure correctness of every change to the goal: the domain expert should be able to validate every change to ensure its validity for the domain.

iv. Ensure traceability. The conversion must be completely reproducible by means of the intermediate results and the documentation.

v. Enable reusability of work at different stages. Maintain generality for as long as possible.

vi. Reduce variability of the conversion result.

In this section, each of those requirements will be evaluated.

7.1.1 Canonical form

The first step towards a canonical form is the result of the *reduction*. By explicitly rewriting any goal in a pure descriptive form, a universal starting point for the rest of the conversion process has been created. Although this form is not rigidly structured in any way, it does provide an important point of convergence.

The *normalisation* steps yields a structured natural language form. This structure, imposed by the four bracketed groups, is a 1:1 reflection of the goal model. Guidelines are provided which assist in the rewrite of the goal to this form. With these guidelines, and the fact that it still concerns natural language it has become feasible for the domain expert to perform the rewrite, and to check the equivalence of the rewrite and the original.

The formal expression language for the goal model is *Generic GDL*. *GDL* has globally the same structure as the structured natural language version. This makes the transition to the formal version easy. With the structured natural language version and *GDL*, the requirement of a canonical form has been satisfied.

7.1.2 Ambiguities

During any formalisation process, it is essential that any ambiguity which might be present in the original version, has been resolved by the end. The biggest reduction in ambiguity is achieved during the normalisation. This requires to rethink a goal in terms of the goal model. Since most natural language goals do not yet explicitly specify the start and end of a period, this automatically raises questions. Practical experiences with the application of the *normalisation* to real-life goals (Protocure workshop of 12-okt-2005, Augsburg) has confirmed this. Almost naturally the question 'what exactly does this mean' came up.

The next step — the *formalisation* — requires to choose a specific kind of required behaviour. While the time frame has already been determined in detail by the *normalisation* step, the different options for the 'behaviour' in *GDL* induce more rethinking of the goal. Especially differences like 'observe once' or 'observe exactly once' will surface. After formalisation, the behaviour will also be clear of ambiguities.

During the *attachment* of the goal, conceptual ambiguities will be resolved. Since finding equivalent concepts in the goal and the model first requires exact knowledge about the meaning of the concept within the goal, doing so raises a discussion and flushes the final ambiguities out. Also this has been demonstrated convincingly during the earlier mentioned workshop.

As described here, the different steps target different kinds of ambiguities. By following the steps every type of ambiguity will be addressed and the original goal will almost naturally have been clear of ambiguities.

7.1.3 Correctness

Essential during the conversion is to ensure correctness of intermediate results at all times. The main instrument to achieve this, is the continuous involvement of the domain expert. By allowing the domain expert to evaluate every change, correctness is maintained at all times. To make this involvement possible, a natural language

version of the current goal is maintained throughout the process. This allows the expert to focus on the meaning of the goal in a familiar form.

The close tie between the structured version and the *GDL* expression, makes is possible to maintain two equivalent versions of the same goal: one in natural language and one formal version. By assuring that every change in the formal version is immediately reflected in the natural language version at all times, the domain expert can validate every proposed change.

Once the attachment is complete, no more changes are allowed. The *translation* is a mechanical step which does not introduce any more changes. It is also the step where *GDL* is abandoned. Therefore, this step will not and can not be checked by the domain expert. However, the formal methods expert should ensure that the translation function that is used produces correct code. By means of the formal *GDL* semantics this can be confirmed.

Combining the input of the domain expert, and the knowledge of the formal methods expert, correctness of the conversion result can be ensured.

7.1.4 Traceability

The traceability requirement has been added to the list to make sure that any result can be reconstructed and the correctness can also be verified by other (domain) experts. The main tool to achieve this is by adding documentation to the intermediate result after each step. This way, not only the result of every single step is preserved, but also notes one issues that where encountered during that step are available afterwards.

Since the whole process has been subdivided in steps with specific tasks, this reduces the amount of documentation required: in the context of the task, many transformations are straight forward and don't need to be explained. Traceability follows from the method. However, it is up to both the domain expert, and the expert in formal methods, to identify the choices which are not straight forward, and to add proper documentation in those cases.

7.1.5 Reusability

To prevent duplicate work, reusability is an important aspect, also in the design of the goal model and *GDL*. Although crested to express a wide variety of goals, the goal model does not prescribe the exact kind of events that must be supported nor does it do so for conditions. The goal model only provides a frame to describe a temporal relation between events and conditions in general. This principle is also reflected in the dualistic design of *GDL*. *Generic GDL* is strictly confined to elements which are required to express the structure of the goal model, while a process model specific extension must be added to be able to express the actual events and conditions which are supported by that model. This dualistic nature allows translations of some target platform to be reused for different *GDL* extensions: the only thing that needs to be added is a new translation for specific events and conditions. The modular KIV translation illustrates this.

In the process the dualistic nature is reflected in the separation between the *formalisation* and the *attachment*: the former is only about structure, the latter is about filling in events and conditions. Because of this, reusability is possible in two different stages: the formalisation result may be attached to different process models, and attached goals may be translated to different target formalisms (different verification tools). During the conversion process, generality is preserved for as long as possible.

In the Protocure project, an example of reuse of the formalisation result, is attachment of a goal to more than one chapter of the formalised breast cancer guideline. An example of reuse of an attached goal would be the translation both to KIV and to the SMV model checking environment.

7.1.6 Variability

The final requirement is reduced variability. This is important for two reasons. First, given some goal, one would expect the same (overall) formalisation result independent of who performs the formalisation. Ideally there should be a one-to-one mapping between natural language goals, and formalised goals. The method should support finding the right formalised goal. Second, when the conversion process is designed such that at each stage there is a logical way to proceed, the whole process becomes much easier. The only variance in formalisations will result from differences in interpretation of the original goal. Unfortunately, this by itself can not be prevented although the involvement of the domain expert at least guarantees that the chosen interpretation makes sense and is relevant in the context of the domain.

Several properties of the conversion steps described in this thesis add to the invariability of the results. First, there is the task-oriented subdivision of the conversion process. Every step consist of a specific task. The fixed order of executing those tasks already reduces the variability. The canonical forms the are required in several stages also lead the expert into performing the task in a specific way. Finally, the vocabulary of the goal model in general and *GDL* specific do not allow many different ways to express a single goal. Every step, up to and including the formalisation tries to achieve convergence to the unique *GDL* expression. The attachment does not yield big changes to the formalisation result. Finally, the translation will — due to the mechanical nature of the task — always yield the same result given some *GDL* expression.

7.2 Future work

The conversion method proposed in this thesis satisfies the requirements which were set in advance. Practical application has already yielded promising prospects. Further use in the Protocure project will show whether the goal model has indeed enough expressive power to encapsulate new goals that will surface.

One possibly useful addition to *Generic GDL* might be the introduction of a *guarded event*. Such a guarded event would have a condition coupled to the event. The event would then only register when the condition is satisfied. Currently this behaviour can only be expressed on a limited basis for the start event. Whether or

not such an addition should be added depends on experiences in practical application, and the costs with respect to the level of complexity. At all times the intuitive nature of the goal model should be preserved.

GDL and the conversion process also need to proof themselves outside the Protocure project. One possible field of application might be the verification of Smart Cards running Java. At the Universität Augsburg, a KIV implementation of the Java VM has been built which is used to perform this kind of verification[12]. As many goals are only available in natural language and need to be formalised, *GDL* and the associated conversion process might be applied here. It would require defining a ***GDL***-Java extension which allows reference to specific Java related events and conditions. Once created, such an extension would be extremely useful in the verification of software in general.

An obvious subject for future research follows from the observation in the *Attachment* that result goals in general are hard to verify. The first steps to try so anyway, which are currently under investigation, focus on modelling effects next to the procedural models. Using those effect models in a formal environment causes difficulties for two reasons. First, given the tools used, there can be no contradiction in the effects model. Second, the effects model is built by hand and only represents one single patient group: the goal will only be proven for patients that behave exactly like the model. This leaves a large amount of patients for which the model is not verified. Additionally, the effect model needs to interact with the process model. In the current efforts the models are still very tied together. This makes reuse of effect models difficult. Work on incorporating effect knowledge should be continued, primarily focussing on finding a way to solve the inconsistency problem, followed by a solution for the close coupling of the models.

Support of uncertainty in the effect model would also be an interesting research area. Then instead of trying to get a yes/no answers for one prototypical patient, it might be possible to calculate a predicted success percentage of the goal for every possible execution. Having those numbers, a very directed effort is possible to improve the process for specific groups of patients. Ideally the statistical data required for the calculation would be taken from a database with actual patient data. Obviously, to be able to use such an approach, a sound theory is required on how to calculate reliable success rate based on available data. However, this is not trivial.

On the side of formal verification, promising results have already been recorded. Proof abstractions which make symbolic execution of large plan hierarchies feasible by folding the plan hierarchy into a much smaller trees seem to be within reach, also on a larger scale. Additionally, proposed combinations of model checking and symbolic execution, utilising the strong point of both methods will most likely pay off. Perhaps the goal model also can add to the reduction of the size of the proof trees: the structure of the goal model should make it easier to identify branches which will not influence the goal. In this area there are many possibilities for further research.

Finally, the academic results from the Protocure project should find their way to the everyday work of guideline designers. If verification tasks would be an integral part of the design process and verification would taken into account at an early stage, goals could be formulated in parallel and the conversion steps could be embedded in

the guideline life cycle. By the combined effort of people that design the guidelines and people that verify them, the quality of guidelines and the subsequent medical care can be taken to an even higher level.

REFERENCES

[1] Michael Balser. *Verifying Concurrent Systems with Symbolic Execution*. PhD thesis, University of Augsburg, Augsburg, 2005.

[2] John Fox, Alyssa Alabassi, Elizabeth Black, Chris Hurt, and Tony Rose. *Modelling clinical goals: a corpus of examples and a tentative ontology*. In *Studies in health technology and informatics*, 101, pages 31–45, 2004.

[3] Marjolein van Gendt. *The power of Medical Quality Indicators*. Master's thesis, Vrije Universiteit, Amsterdam, 2004.

[4] *Interactive Theorem Proving* [online]. Available from: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/research/kiv/.

[5] *KIV at the DFKI GmBH* [online]. Available from: http://www.dfki.uni-sb.de/vse/projects/kiv.html.

[6] *KIV Home Page at the University of Karlsruhe* [online]. Available from: http://i11www.ira.uka.de/~kiv/KIV-KA.html.

[7] Mar Marcos, Michael Balser, Annette ten Tije, Frank van Harmelen, and Christoph Duelli. *Experiences in the formalisation and verification of medical protocols*. In *Artificial Intelligence in Medicine: 9th European Conference on Artificial Intelligence in Medicine*, pages 132–141, 2003.

[8] Silvia Miksch. *Plan management in the medical domain*. In *AI Communications*, 4, 1999.

[9] *Protocure II* [online]. Available from: http://www.protocure.org.

[10] J. Schmitt and M. Balser. *Interactive Verification of Asbru - A Tutorial*. Technical Report 2006-3, University of Augsburg, February 2006. Available from: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

[11] Yuval Shahar, Silvia Miksch, and Peter Johnson. *The Asgaard project: A task-specific framework for the application and critiquing of time-oriented clinical guidelines*. In *Artificial Intelligence in Medicine*, 14, pages 29–51, 1998.

[12] Kurt Stenzel. *Verification of Java Card program*. PhD thesis, University of Augsburg, Augsburg, 2005. Available from: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

Appendix A
GDL: XML Specification
A.1 Generic *GDL*

A.1.1 The goal body

1 Element **goal**

Childname	*Element*	*Occurence*
precondition	#2, p. 83	*Once*
time-specification	#3, p. 83	*Once*
Once		
‖ maintain-during-period	#8, p. 84	*Once*
‖ observe-during-period	#9, p. 84	*Once*
‖ avoid-during-period	#10, p. 84	*Once*
‖ achieve-at-end	#8, p. 84	*Once*
‖ sub-goal	#11, p. 84	*Once*

2 Element **precondition**

Childname	*Element*	*Occurence*
abstract-condition	#12, p. 85	*Once*

A.1.2 Time groups

3 Element **time-specification**

Childname	*Element*	*Occurence*
from	#4, p. 83	*Once*
time-delimiter	#5, p. 83	*Once*

4 Element **from**
until
open-until

Childname	*Element*	*Occurence*
abstract-event	#16, p. 85	*Once*

5 Element **time-delimiter** — *Abstract Element*

Childname	*Element*	*Occurence*
Once		
‖ until	#4, p. 83	*Once*
‖ open-until	#4, p. 83	*Once*
‖ until-end	#6, p. 83	*Once*
‖ duration	#7, p. 84	*Once*
‖ open-duration	#7, p. 84	*Once*

6 Element **until-end**

7 Element **duration**
open-duration

Attribute name	*Content*
value	`Numerical`
unit	*'year', 'day', 'hr', 'min', 'sec', 'milli-sec', 'micro-sec'*

A.1.3 Goals

8 Element **maintain-during-period**
achieve-at-end

Childname	*Element*	*Occurence*
abstract-condition	#12, p. 85	*Once*

9 Element **observe-during-period**

Childname	*Element*	*Occurence*
Once		
‖ abstract-condition	#12, p. 85	*Once*
‖ abstract-event	#16, p. 85	*Once*
Attribute name	*Content*	
Only with abstract-event		
type	*'less', 'less-or-equal', 'equal', 'more-or-equal', 'more', 'not-equal', 'range'*	
lowerbound	`Numerical - with 'range'`	
count	`Numerical`	

10 Element **avoid-during-period**

Childname	*Element*	*Occurence*
Once		
‖ abstract-condition	#12, p. 85	*Once*
‖ abstract-event	#16, p. 85	*Once*

11 Element **sub-goal**

Childname	*Element*	*Occurence*
precondition	#2, p. 83	*Once*
time-specification	#3, p. 83	*Once*
Once		
‖ maintain-during-period	#8, p. 84	*Once*
‖ observe-during-period	#9, p. 84	*Once*
‖ avoid-during-period	#10, p. 84	*Once*
‖ achieve-at-end	#8, p. 84	*Once*
‖ sub-goal	#11, p. 84	*Once*

A.1.4 Conditions

12 Element **abstract-condition** — *Abstract Element*

Childname	*Element*	*Occurence*
Once		
‖ always-true	#13, p. 85	*Once*
‖ condition-not	#14, p. 85	*Once*
‖ condition-combination	#15, p. 85	*Once*
‖ abstract-domain-condition	#18, p. 85	*Once*

13 Element **always-true**

14 Element **condition-not**

Childname	*Element*	*Occurence*
abstract-condition	#12, p. 85	*Once*

15 Element **condition-combination**

Childname	*Element*	*Occurence*
abstract-condition	#12, p. 85	*Twice*
Attribute name	*Content*	
type	*'and', 'or', 'xor'*	

A.1.5 Events

16 Element **abstract-event** — *Abstract Element*

Childname	*Element*	*Occurence*
Once		
‖ event-combination	#17, p. 85	*Once*
‖ abstract-domain-event	#21, p. 86	*Once*

17 Element **event-combination**

Childname	*Element*	*Occurence*
abstract-event	#16, p. 85	*Twice*
Attribute name	*Content*	
type	*'and', 'or', 'xor'*	

A.2 *GDL*-Asbru

A.2.1 Conditions

18 Element **abstract-domain-condition** — *Abstract Element*

Childname	*Element*	*Occurence*
Once		
‖ parameter-condition	#19, p. 86	*Once*
‖ plan-state-condition	#20, p. 86	*Once*

19 Element **parameter-condition**

Attribute name	*Content*
parameter-name	`String`
type	*'less', 'less-or-equal', 'equal', 'more-or-equal', 'more', 'not-equal'*
Once	
‖ second-parameter-name	`String`
‖ value	`String/Numerical`

20 Element **plan-state-condition**

Attribute name	*Content*
plan-name	`String`
type	*'equal', 'not-equal'*
state	*'inactive', 'considered', 'possible', 'ready', 'rejected', 'activated', 'suspended', 'aborted' or 'completed'*

A.2.2 Events

21 Element **abstract-domain-event** — *Abstract Element*

Childname	*Element*	*Occurence*
Once		
‖ transition	#22, p. 86	*Once*
‖ param-update	#23, p. 86	*Once*
‖ param-update-to	#24, p. 87	*Once*

22 Element **transition**

Attribute name	*Content*
plan-name	`String`
direction	*'enter', 'leave'*
state	*'inactive', 'considered', 'possible', 'ready', 'rejected', 'activated', 'suspended', 'aborted' or 'completed'*
count	`Numerical`

23 Element **param-update**

Attribute name	*Content*
parameter-name	`String`

24 Element **param-update-to**

Attribute name	*Content*
parameter-name	`String`
type	*'less', 'less-or-equal', 'equal', 'more-or-equal', 'more', 'not-equal'*
value	`String/Numerical`

Appendix B
GDL: Presentation syntax
B.1 Generic *GDL*

1 **Goal**

– **Goal** [Name]$_{\texttt{Str}}$
  Precondition
    ▷ *Condition*$_{\#7,\ p.\ 90}$
  Time-specification
    ▷ *Time-specification*$_{\#3,\ p.\ 89}$
  ▷ *Behaviour*$_{\#2,\ p.\ 89}$

2 **Behaviour**

– **Maintain-during-period**
  ▷ *Condition*$_{\#7,\ p.\ 90}$

– **Avoid-during-period**
  ▷ *Condition-or-Event*$_{\#6,\ p.\ 90}$

– **Observe-during-period**
  ▷ *Condition*$_{\#7,\ p.\ 90}$

– **Observe-during-period** [Operator]$_{\triangleright \mathit{CompOpr}_{\#13,\ p.\ 91}}$ [Count]$_{\texttt{Num}}$
  ▷ *Event*$_{\#8,\ p.\ 90}$

– **Observe-during-period range** [Lowerbound]$_{\texttt{Num}}$ ([Count]$_{\texttt{Num}}$)
  ▷ *Event*$_{\#8,\ p.\ 90}$

– **Achieve-at-end**
  ▷ *Condition*$_{\#7,\ p.\ 90}$

– **Sub-goal**
  Precondition
    ▷ *Condition*$_{\#7,\ p.\ 90}$
  Time-specification
    ▷ *Time-specification*$_{\#3,\ p.\ 89}$
  ▷ *Behaviour*$_{\#2,\ p.\ 89}$

3 **Time-specification**

– **From**
  ▷ *Event*$_{\#8,\ p.\ 90}$
▷ *Period-delimiter*$_{\#4,\ p.\ 90}$

4 **Period-delimiter**

– | **Until**
▷ *Event*#8, *p.* 90

– | **Open-until**
▷ *Event*#8, *p.* 90

– | **Until end**

– | **Duration** [Time]Numerical [Unit]▷*Unit*#5, *p.* 90

– | **Open-duration** [Time]Numerical [Unit]▷*Unit*#5, *p.* 90

5 **Unit**

– | 'year', 'day', 'hr', 'min', 'sec', 'milli-sec' or 'micro-sec'.

6 **Condition-or-Event**

– | ▷ *Condition*#7, *p.* 90

– | ▷ *Event*#8, *p.* 90

7 **Condition**

– | (▷ *Condition*#7, *p.* 90)

– | ▷ *Condition*#7, *p.* 90 **and** ▷ *Condition*#7, *p.* 90

– | ▷ *Condition*#7, *p.* 90 **or** ▷ *Condition*#7, *p.* 90

– | ▷ *Condition*#7, *p.* 90 **xor** ▷ *Condition*#7, *p.* 90

– | ▷ *Atomic-condition*#9, *p.* 91

8 **Event**

– | (▷ *Event*#8, *p.* 90)

– | ▷ *Event*#8, *p.* 90 **and** ▷ *Event*#8, *p.* 90

– | ▷ *Event*#8, *p.* 90 **or** ▷ *Event*#8, *p.* 90

– | ▷ *Event*#8, *p.* 90 **xor** ▷ *Event*#8, *p.* 90

– | ▷ *Atomic-event*#10, *p.* 91

– | ▷ *Atomic-event*#10, *p.* 91 $\overset{+}{\longmapsto}$ [Time]Numerical [Unit]▷*Unit*#5, *p.* 90

– | ▷ *Atomic-event*#10, *p.* 91 $\overset{+}{\mapsto\!\!\!\mid}$ [Time]Numerical [Unit]▷*Unit*#5, *p.* 90

9 **Atomic-condition**

– | always-true

– | ▷ *Domain-Condition*$_{\#11,\ p.\ 91}$

10 **Atomic-event**

– | **start**

– | **ConditionToTrue:** ▷ *Condition*$_{\#7,\ p.\ 90}$

– | **ConditionToFalse:** ▷ *Condition*$_{\#7,\ p.\ 90}$

– | ▷ *Domain-Event*$_{\#12,\ p.\ 91}$

B.2 *GDL*-Asbru

11 **Domain-Condition**

– | **Param** [Param]$_{\texttt{String}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Value]$_{\texttt{Str/Num}}$

– | **Param** [Param]$_{\texttt{String}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Param]$_{\texttt{Str}}$

– | **Planstate** [Plan]$_{\texttt{String}}$ [Operator]$_{\triangleright BoolOpr_{\#14,\ p.\ 91}}$ [State]$_{\triangleright State_{\#16,\ p.\ 91}}$

12 **Domain-Event**

– | **Transition** [Plan]$_{\texttt{String}}$ [Dir]$_{\triangleright DirOpr_{\#15,\ p.\ 91}}$ [State]$_{\triangleright State_{\#16,\ p.\ 91}}$

– | **ParamUpdate** [Param]$_{\texttt{Str}}$

– | **ParamUpdateTo** [Param]$_{\texttt{Str}}$ [Operator]$_{\triangleright CompOpr_{\#13,\ p.\ 91}}$ [Value]$_{\texttt{Str/Num}}$

13 **CompOpr**

– | ‘$<$’, ‘$\leqslant$’, ‘$=$’, ‘$\geqslant$’, ‘$>$’ or ‘$\neq$’.

14 **BoolOpr**

– | ‘$=$’ or ‘$\neq$’.

15 **DirOpr**

– | ‘**enter**’ or ‘**leave**’.

16 **State**

– | ‘inactive’, ‘considered’, ‘possible’, ‘ready’, ‘rejected’, ‘activated’, ‘suspended’, ‘aborted’ or ‘completed’.

Appendix C
GDL: Formal semantics
C.1 Generic *GDL*

C.1.1 Goal body

$I \models$ **Goal** $\mathcal{A}_{1:Name}$
Precondition
$\mathcal{P}_{1:Condition}$
Time-specification
From $\mathcal{P}_{2:Event}$
$\mathcal{P}_{3:Time-delimiter}$
$\mathcal{P}_{4:Behaviour}$

iff $\forall\, i, j > i\ .\quad I|^i \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge I|^i, 0, 0 \models_{\overline{e}} \mathcal{P}_{2:Event}$
$\wedge\ I, i, j \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$
$\wedge\ \neg\exists\, i < k < j\ .\ I, i, k \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$
$\Rightarrow I|^{j-1}, i \models_{\overline{b}} \mathcal{P}_{4:Behaviour}$

C.1.2 Period delimiters

$I, i, j \models_{\overline{p}}$ **Until** $\mathcal{P}_1$
iff $I|^j, i, 1 \models_{\overline{e}} \mathcal{P}_1$

$I, i, j \models_{\overline{p}}$ **Open-until** $\mathcal{P}_{1:Event}$
iff $j = |I| + 1 \vee I|^j, i, 1 \models_{\overline{e}} \mathcal{P}_{1:Event}$

$I, i, j \models_{\overline{p}}$ **Until end**
iff $j = |I| + 1$

$I, i, j \models_{\overline{p}}$ **Duration** $\mathcal{A}_1$
iff $\left\|I|_i^{-2}\right\| < \mathcal{A}_1 \leqslant \left\|I|_i^{-1}\right\|$

$I, i, j \models_{\overline{p}}$ **Open-duration** $\mathcal{P}_1$
iff $j = |I| + 1 \vee \left\|I|_i^{-2}\right\| < \mathcal{P}_1 \leqslant \left\|I|_i^{-1}\right\|$

C.1.3 Behaviour

$I, i \models_{\overline{b}}$ **Maintain-during-period**
$\mathcal{P}_{1:Condition}$
iff $\forall\, k \geqslant i\ .\ I|^k \models_{\overline{c}} \mathcal{P}_1$

$I, i \models_{\overline{b}}$ **Avoid-during-period**
$\mathcal{P}_{1:Condition}$
iff $\neg\exists\, k \geqslant i\ .\ I|^k \models_{\overline{c}} \mathcal{P}_1$

$I, i \models_{\overline{b}}$ **Avoid-during-period**
$\mathcal{P}_{1:Event}$
iff $\neg\exists\ k \geqslant i\ .\ I|^k, i, 0 \models_{\overline{e}} \mathcal{P}_1$

$I, i \models_{\overline{b}}$ **Observe-during-period**
$\mathcal{P}_{1:Condition}$
iff $\exists\ k \geqslant i\ .\ I|^k \models_{\overline{c}} \mathcal{P}_1$

$I, i \models_{\overline{b}}$ **Observe-during-period** $= \mathcal{A}_{2:Count}$
$\mathcal{P}_{1:Event}$
iff $\lfloor \mathcal{P}_1 \rfloor_{I,i} = \mathcal{A}_2$

Idem for "$<$", "$\leqslant$", "$\neq$", "$\geqslant$", "$>$".

$I, i \models_{\overline{b}}$ **Observe-during-period range** $\mathcal{A}_{1:Lowerbound}$ $(\mathcal{A}_{2:Count})$
$\mathcal{P}_{1:Event}$
iff $\mathcal{A}_1 \leqslant \lfloor \mathcal{P}_1 \rfloor_I < \mathcal{A}_1 + \mathcal{A}_2$

$I, i \models_{\overline{b}}$ **Achieve-at-end**
$\mathcal{P}_{1:Condition}$
iff $I \models_{\overline{c}} \mathcal{P}_1$

$I, i \models_{\overline{b}}$ **Sub-goal**
Precondition
$\mathcal{P}_{1:Condition}$
Time-specification
From $\mathcal{P}_{2:Event}$
$\mathcal{P}_{3:Time-delimiter}$
$\mathcal{P}_{4:Behaviour}$
iff $\forall\ i', j' > i'\ .\quad (I|_i)|^{i'} \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge (I|_i)|^{i'}, 0, 0 \models_{\overline{e}} \mathcal{P}_{2:Event}$
$\wedge\ I|_i, i', j' \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$
$\wedge\ \neg\exists\ i' < k' < j'\ .\ I|_i, i', k' \models_{\overline{p}} \mathcal{P}_{3:Period-delimiter}$
$\Rightarrow (I|_i)|^{j'-1}, i' \models_{\overline{b}} \mathcal{P}_{4:Behaviour}$

C.1.4 Conditions

Composite Conditions

The operator precedence (starting with the strongest binding) is: **()**, **not**, **and**, **or**, **xor**:

A **xor** B **and not** C **or** D $\Leftrightarrow$ A **xor** ((B **and** (**not** C)) **or** D)

$I \models_{\overline{c}}$ $\mathcal{P}_{1:Atomic-condition}$
iff $I(|I|) \models \mathcal{P}_{1:Atomic-condition}$

$I \models_{\overline{c}} (\mathcal{P}_{1:Condition})$
iff $I \models_{\overline{c}} \mathcal{P}_{1:Condition}$

$I \models_{\overline{c}}$ **not** $\mathcal{P}_{1:Condition}$
iff $I \not\models_{\overline{c}} \mathcal{P}_{1:Condition}$

$I \models_{\overline{c}} \mathcal{P}_{1:Condition}$ **and** $\mathcal{P}_{2:Condition}$
iff $I \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge I \models_{\overline{c}} \mathcal{P}_{2:Condition}$

$I \models_{\overline{c}} \mathcal{P}_{1:Condition}$ **or** $\mathcal{P}_{2:Condition}$
iff $I \models_{\overline{c}} \mathcal{P}_{1:Condition} \vee I \models_{\overline{c}} \mathcal{P}_{2:Condition}$

$I \models_{\overline{c}} \mathcal{P}_{1:Condition}$ **xor** $\mathcal{P}_{2:Condition}$
iff $(I \models_{\overline{c}} \mathcal{P}_{1:Condition} \vee I \models_{\overline{c}} \mathcal{P}_{2:Condition})$
$\wedge \neg(I \models_{\overline{c}} \mathcal{P}_{1:Condition} \wedge I \models_{\overline{c}} \mathcal{P}_{2:Condition})$

Atomic *GDL* conditions

$\sigma \models$ always-true
iff $true$

C.1.5 Events

Composite events

The operator precedence is equal to those of conditions, augmented with the offset operators: $\overset{+}{\longmapsto}$, $\overset{+}{\mapsto\!\!\!|}$, **()**, **and**, **or**, **xor**:

A **xor** B **and** C $\overset{+}{\longmapsto}$ 5 **or** D $\Leftrightarrow$ A **xor** ((B **and** (C $\overset{+}{\longmapsto}$ 5)) **or** D)

$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Atomic-event}$
iff $l < |I| \wedge I \models \mathcal{P}_{1:Atomic-event}$

$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Atomic-event} \overset{+}{\longmapsto} \mathcal{A}_1$
iff $\exists\, k\, .\, (I|^k \models \mathcal{P}_{1:Atomic-event}) \wedge \|I_k^{-(s+1)}\| < \mathcal{A}_1 \leq \|I_k^{-s}\|$

$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Atomic-event} \overset{+}{\mapsto\!\!\!|} \mathcal{A}_1$
iff $\exists\, k > l\, .\, (I|^k \models \mathcal{P}_{1:Atomic-event}) \wedge \|I_k^{-(l+1)}\| < \mathcal{A}_1 \leq \|I_k^{-l}\|$

$I, l, s \models_{\overline{e}} (\mathcal{P}_{1:Event})$
iff $I, l, s \models \mathcal{P}_{1:Event}$

$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event}$ **and** $\mathcal{P}_{2:Event}$
iff $I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event} \wedge I, l, s \models_{\overline{e}} \mathcal{P}_{2:Event}$

$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event}$ **or** $\mathcal{P}_{2:Event}$
iff $I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event} \vee I, l, s \models_{\overline{e}} \mathcal{P}_{2:Event}$

$$I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event} \textbf{ xor } \mathcal{P}_{2:Event}$$
$$\text{iff } (I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event} \vee I, l, s \models_{\overline{e}} \mathcal{P}_{2:Event})$$
$$\wedge \neg(I, l, s \models_{\overline{e}} \mathcal{P}_{1:Event} \wedge I, l, s \models_{\overline{e}} \mathcal{P}_{2:Event})$$

Atomic *GDL* events

$$I \models \textbf{ start}$$
$$\text{iff } |I| = 0$$

$$I \models \textbf{ ConditionToTrue: } \mathcal{P}_{1:Condition}$$
$$\text{iff } (I^{-1} \not\models \mathcal{P}_{1:Condition}) \wedge (I \models \mathcal{P}_{1:Condition})$$

$$I \models \textbf{ ConditionToFalse: } \mathcal{P}_{1:Condition}$$
$$\text{iff } (I^{-1} \models \mathcal{P}_{1:Condition}) \wedge (I \not\models \mathcal{P}_{1:Condition})$$

C.2 *GDL*-Asbru

The definition of events and conditions of ***GDL***-Asbru are provided as a general Condition or Event function. In the translation specification these named functions need to be implemented. The required behaviour is given in natural language.

C.2.1 Domain conditions

$$\sigma \models \textbf{ Param } \mathcal{A}_{1:Param} < \mathcal{A}_{2:Value}$$
$$\text{iff } \sigma(\mathcal{A}_{1:Param}) < \mathcal{A}_{2:Value}$$

Idem for "$\leqslant$", "$=$", "$\geqslant$", "$>$" or "$\neq$".

$$\sigma \models \textbf{ Param } \mathcal{A}_{1:Param} < \mathcal{A}_{2:Param}$$
$$\text{iff } \sigma(\mathcal{A}_{1:Param}) < \sigma(\mathcal{A}_{2:Param})$$

Idem for "$\leqslant$", "$=$", "$\geqslant$", "$>$" or "$\neq$".

$$\sigma \models \textbf{ Planstate } \mathcal{A}_{1:Plan} = \mathcal{A}_{2:State}$$
$$\text{iff } \sigma[\mathcal{A}_{1:Plan}] = \mathcal{A}_{2:State}$$

Idem for "$\neq$".

C.2.2 Domain events

$$I \models \textbf{ Transition } \mathcal{A}_{1:Plan} \textbf{ enter } \mathcal{A}_{2:State}$$
$$\text{iff } I \models \textbf{ ConditionToTrue: Planstate } \mathcal{A}_{1:Plan} = \mathcal{A}_{2:State}$$

$$I \models \textbf{ Transition } \mathcal{A}_{1:Plan} \textbf{ leave } \mathcal{A}_{2:State}$$
$$\text{iff } I \models \textbf{ ConditionToFalse: Planstate } \mathcal{A}_{1:Plan} = \mathcal{A}_{2:State}$$

$I \models$ **ParamUpdate** $\mathcal{A}_{1:Param}$
iff *There was a write to parameter* $\mathcal{A}_1$ *during the transition to the last state of* I*.*

$I \models$ **ParamUpdateTo** $\mathcal{A}_{1:Param}$ $\mathcal{A}_{2:Operator}$ $\mathcal{A}_{3:Value}$
iff $I \models$ **ParamUpdate** $\mathcal{A}_{1:Param}$
$\wedge$ $I(|I|) \models$ **Param** $\mathcal{A}_{1:Param}$ $\mathcal{A}_{2:Operator}$ $\mathcal{A}_{3:Value}$