Chapter 5

Problem Solving Behaviour in *GARP*

This chapter describes how the conceptual framework for qualitative prediction of behaviour can be transformed into a design model for the construction of a computer program. A design model in *KADS* consists of three views. The functional view (5.1) describes the functions that must be realised by the artifact. In particular, this section discusses some of the additional functionality that is required, besides problem solving, for implementation of the artifact (for example, the interaction with the user).

The most interesting problem to solve in this chapter is the construction of a behavioural view of the artifact. Algorithms must be developed that realise the problem solving behaviour required by the inferences in the conceptual model. Our solution to this problem is described in two sections. Section 5.2.1 describes how the meta-classes from the conceptual model are mapped onto design elements, and section 5.2.2 describes how these design elements are used by algorithms for realising the problem solving behaviour that is specified by the knowledge sources.

The physical view describes how the design elements and algorithms are composed into the different physical modules which constitute the actual artifact (5.3). The implemented program is called *GARP*, which is an acronym for General Architecture for Reasoning about Physics.[1]

The last section describes two prediction models, one of the cooling mechanism of a refrigerator and one of heart diseases. Both models are implemented in *GARP* and illustrate important aspects of the problem solving behaviour manifested by *GARP*.

5.1 Functional Description of *GARP*

In *KADS* a functional decomposition takes as input:

- a model that represents the problem solving expertise,
- a model of the external requirements on the artifact as a result of the environment in which it has to operate, and

[1] *GARP* is implemented in *SWI-Prolog* [139]. Both *GARP* and *SWI-Prolog* are distributed without charge for non-commercial purposes. Please contact the author for more details.

- a model of cooperation (modality) that represents how the system interacts with its environment, in particular with its user.

The research reported here does not include an analysis of the external requirements and a model of cooperation. However, guidelines for the design and implementation can be derived from the purpose that the artifact must serve.

A major problem in qualitative reasoning is the construction of prediction models. An artifact that implements an integrated framework for qualitative prediction of behaviour is particularly interesting when it supports the development of these models. The following three requirements are essential for providing such support:

- prototyping,
- cognitive plausibility, and
- adequate computational performance.

KADS defines the notion of *interpretation model* as a model of expertise that describes inference, task and strategic knowledge of a certain problem solving task. During the development of a knowledge based system, an interpretation model guides the acquisition and structuring of the domain specific knowledge relevant to the application. We take this approach one step further, because we implement such an interpretation model. The resulting artifact, *GARP*, embodies a shell for qualitative prediction of behaviour. This reasoning shell can be filled by a knowledge engineer with domain knowledge and as such can be used for developing and experimenting with different prediction models. The latter is relevant for theory formation concerning qualitative reasoning.

The knowledge engineer should be able to use the reasoning shell for examining different aspects of the domain knowledge. As prediction models are often large, it is relevant that these studies can be performed on small subsets of the total model. The implementation should therefore facilitate prototyping of partial prediction models.

Related to prototyping is the ability to interact with the control on the prediction inferences. Qualitative problem solvers, in particular when insufficient domain knowledge has been represented, often encounter ambiguity and because of that explosively generate behaviour states. Providing the user with partial control of the reasoning process, allows focusing of the prediction engine without requiring all the necessary knowledge to be present in the artifact. In fact, each intervention of the knowledge engineer may be considered as an indication that a specific piece of knowledge is missing in the simulation model.

The knowledge engineer is concerned with modelling the *knowledge* that is used in a reasoning process. The development of a prediction model should therefore not be hampered by implementation details. Moreover, the understandability of the behaviour of the artifact increases when the required input, the reasoning process itself, and the output of the artifact, are presented to the user in a *cognitively plausible* way. In other words, it is important that the knowledge engineer has access to the program at the conceptual level. In particular, the program should provide *conceptual tracing* of the reasoning steps that it carries out, i.e. notify the user about its reasoning activities in terms of the knowledge entities present in the conceptual model.

Finally, the problem solver must be implemented in a computationally adequate way, i.e. the artifact should provide solutions within a reasonable amount of time.

5.1.1 Principal Design Decisions

The important constraints on the functional decomposition that follow from the above mentioned requirements are:

- *Type-to-type mapping*

  This constraint embodies the idea that the design model, and consequently the implementation of the artifact, should follow the knowledge types represented in the conceptual model as closely as possible. This is particularly relevant for providing conceptual access to the artifact. With a type-to-type mapping the realisation of this functionality is less complicated, because the transformation step between the artifact and the conceptual model is relatively small. In general, all functionalities that follow from required interactions between the artifact and the user benefit from a type-to-type mapping.

- *Mixed control and intervention*

  The user should be able to intervene in the control of the reasoning process. The user must therefore have control of the problem solving functions and of the reasoning process as a whole. The user should be able to manipulate the reasoning process completely, but should also be able to leave parts of the control, or all of it, to the artifact. It is in this respect important that inference packages (functional units) are put together in such a way that they allow controlling the inference capabilities in sufficient detail. The type-to-type mapping can in this respect be used for determining the appropriate grainsize for the inference steps.

- *Visualisation of the problem solving process*

  Functions must be defined for visualisation of the reasoning process that is carried out, its intermediate results, and its final output. The level of detail at which the information is presented to the user is crucial. Although the presented information must contain a sufficient amount of detail, it should not be represented to the user *below the knowledge level*, i.e. it should not consider machine specific aspects of the reasoning process, but only focus on the *knowledge* that is manipulated by the prediction engine. The type-to-type mapping provides the necessary information for finding the appropriate level of detail for the visualisation.

- *Feedback on solvability*

  The usability of the artifact for prototyping greatly improves when the artifact provides feedback about the type of faults that it encounters during its reasoning process. In particular, feedback should be given on the solvability of a prediction problem.

- *Modification access*

  For prototyping and building prediction models it is relevant that the knowledge engineer has 'easy' access to modification facilities for changing the knowledge represented in the artifact.

5.1.2 Functional Decomposition of *GARP*

Following the design decisions described above and the conceptual framework described in the previous chapter, the functional decomposition of *GARP* starts with three top-level functions:[2] <*problem solving*>, <*interface*> and <*storage*>. Figure 5.1 depicts how these functions are decomposed into subfunctions. This figure also shows the input and output relations and the division of the control between *GARP* and the user. In the following sections these aspects of the functional decomposition will be further discussed. It should be noted that there is no explanation function. As a result *GARP* cannot explain its reasoning process.

5.1.2.1 Storage Functions

The storage functions result from both the conceptual model and the additional constraints imposed on the design and implementation of *GARP*. The top-level function <*storage*> can be decomposed into the subfunctions shown in table 5.1.

Storage function	**Represented knowledge**
<*store input system*>	The prediction problems that have to be solved.
<*store library knowledge*>	The general knowledge that can be used to solve prediction problems.
<*store P.S. trace*>	A ‘permanent’ record of the problem solving results.
<*store intermediate P.S. results*>	The intermediate results of the problem solving process.
<*store P.S. output*>	The output of the problem solving process.

Table 5.1: Storage functions

The output of the problem solving process is a subset of the intermediate results, both their storages can therefore be addressed by a single functional unit (see figure 5.1). All the storage functions are controlled by *GARP*. The user has access to the knowledge stored by these functions, only by calling upon other functions. The storage functions provide input for (=storage functions output), and receive output from (=storage functions input), both the interface functions and the problem solving functions.

5.1.2.2 Interface Functions

The interface functions result mainly from the additional requirements imposed on the implementation of *GARP*. They are essential for the interaction between *GARP* and the user. The top-level function <*interface*> can be decomposed into the functions <*visualise*>, <*modification access*> and <*user control*>. The function <*visualise*> is concerned with showing the user what is going on inside the problem solver. It can be decomposed into the following functions:

- <*show errors*> notifies the user about the assessment of solvability (see 5.1.2.3).

[2] We write names of functions as follows: <*function name*>.

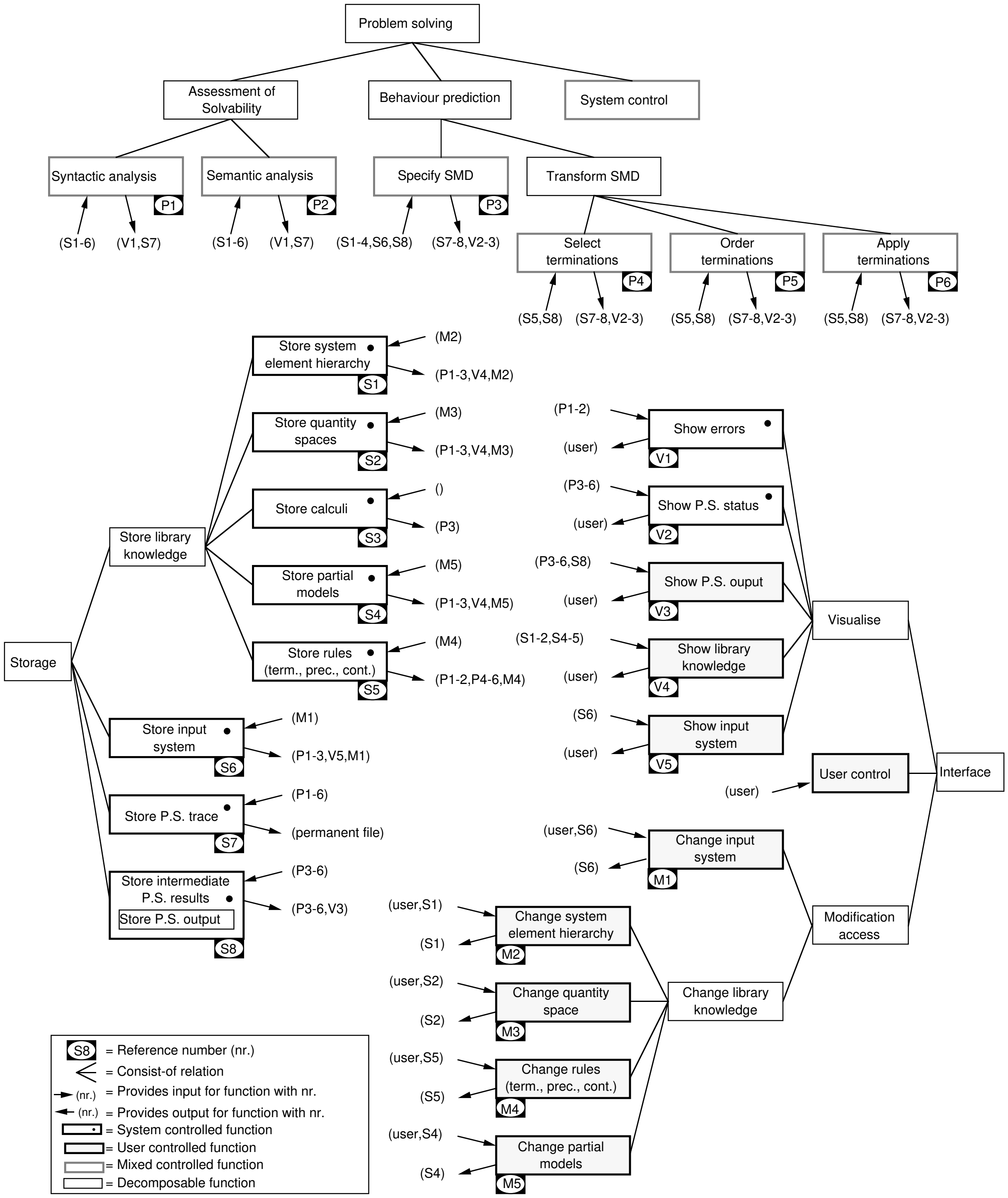

Figure 5.1: Functional decomposition of *GARP*

- <*show P.S. status*> informs the user about what kind of inference is being carried out by *GARP*. This function gets input from all problem solving functions about their problem solving activities (=status).
- <*show P.S. output*> displays all the output that is derived by *GARP*. This function also gets input from all problem solving functions, but in addition may get input from the storage function <*store intermediate P.S. results*>. The former is used for showing output during problem solving whereas the latter is used for showing output after problem solving is finished or interrupted.
- <*show library knowledge*> and <*show input system*> allow the user to read through all the knowledge that is present in the library of *GARP*. These functions get their input from the corresponding storage functions.

When the visualisation is active, it always presents its output to the user. The user determines the amount of detail that each problem solving function presents to the function <*show P.S. output*>. The interface functions <*show errors*> and <*show P.S. status*> are controlled by *GARP* and are always active whenever *GARP* is active. The <*show library knowledge*> is also completely controlled by the user.

The second group of interface functions is concerned with modifying the knowledge base of *GARP*. There is a modification function for changing the input system and for changing each part of the library knowledge, except for the function <*store calculi*>. The modification functions do not allow changes to the calculi. The input for the modification functions comes from the related storage functions and their output results in actual changes in the knowledge stored by these storage functions. All the modification functions are controlled by the user.

The user's control over *GARP* is managed by the function <*user control*>. This function has access to all the functions that can or must be controlled by the user.

5.1.2.3 Problem Solving Functions

The function <*assessment of solvability*>, results from the additional requirements imposed on the implementation. It can be decomposed into two functions. The <*syntactic analysis*> analyses whether knowledge is presented to *GARP* in the appropriate language. The <*semantic analysis*> analyses whether the knowledge written in this language can be understood in terms of the knowledge already present in *GARP*.[3] Both functions can be controlled by *GARP* or by the user, but the <*syntactic analysis*> should always precede the <*semantic analysis*>. *GARP* automatically invokes the assessment functions when it receives new knowledge. The two functions get their input from the storage function <*store input system*> and <*store library knowledge*>. The output of both functions is presented to the user via the interface function <*show errors*>, either as a notification that the knowledge is understood or as feedback about the problems encountered.

The second problem solving function, <*behaviour prediction*>, follows directly from the conceptual model. Each function refers to a knowledge source in the inference structure. These problem solving functions can be controlled by *GARP* or by the user.

[3] Being solvable in terms of being syntactically and semantically correct does not imply that the problem can be solved by *GARP* within a reasonable amount of time.

The knowledge sources *assemble* and *compute* are realised by a single functional unit (<*specify*>), because they are closely related with respect to control. An assemble step, for example, often requires a computation for determining whether the additional parameter relations introduced by a partial model are consistent with the already known mathematical model.

The function <*specify*> requires input from all storage functions, except from <*store rules*> and <*store P.S. trace*>. The transformation functions get their input from the storage functions <*store rules*> and <*store intermediate P.S. results*>.

All problem solving functions present information about their status and the derived output to the user by respectively using the interface functions <*show P.S. status*> and <*show P.S. output*>. In addition they present their output to the storage functions <*intermediate P.S. results*> and possibly to <*store P.S. trace*>. The amount of information presented to <*show P.S. output*> and <*store P.S. trace*> is controlled by the user.

The function <*system control*> is concerned with the task and strategic knowledge from the conceptual model. It has access to all the functions that must be controlled by *GARP*. The function itself is controlled by the function <*user control*>.

5.2 Behaviour Description of *GARP*

This section describes the methods that are used in *GARP* for realising the problem solving behaviour required by the functions in the functional decomposition. In particular we are faced with the nontrivial problem of transforming the problem solving functions, resulting from the conceptual model, into algorithms that, by manipulating design elements, realise the machine specific behaviour needed for implementing those functions. The major part of what follows is therefore concerned with

- describing the design elements that are used to implement the meta-classes (5.2.1), and
- how these design elements are manipulated by algorithms in order to realise the machine specific behaviour (5.2.2).

Some important properties of the developed methods can be pointed out. Firstly, the algorithm for specification implements an efficient depth-first search, that not only tests whether conditions can be inferred from the knowledge present in the system model description (SMD), but that in addition uses an assumption mechanism for finding all partial models that are consistent with that knowledge. Secondly, the computation of parameter relations, which is part of the specification, proposes an advanced method for coping with problems related to transitivity reasoning. Thirdly, the transformation to successive states of behaviour allows an explicit selection and ordering of possible terminations.

In section 5.2.5 the required behaviour for the other functions from the functional decomposition is discussed. However, the emphasis of our work is concerned with realising machine specific behaviour for problem solving competence, and not so much with the interaction of the artifact with the user and the environment. We will therefore discuss these other functions only briefly.

5.2.1 Design Elements

For implementing *GARP* we chose a *Prolog* environment. The declarative nature of *Prolog* allows a direct (one-to-one) mapping between meta-classes and design elements. The design elements are therefore relatively easy to read and are not confused with implicit procedural notions.

The central design decision that determined the characteristic of the design elements used in *GARP*, was to ensure that at all times in the problem solving process, all the knowledge taken into account by *GARP* could be made available to the user in a way that closely matched the knowledge entities defined in the conceptual model. Recall that the one-to-one mapping was an essential constraint imposed on the implementation of *GARP*. This design decision turns out to be well supported by the declarative nature of *Prolog*.

The examples presented in this section are all taken from the actual implementation of *GARP*.[4]

5.2.1.1 System Elements

The hierarchy of generic system elements is represented by the predicate *isa* with the generic names of the lower and the higher concepts in the isa-hierarchy. The predicate *instance* is used for creating instances from the generic isa-hierarchy. Its arguments specify the instance name for representing the real-world entity and the generic name that refers to the isa-hierarchy. The predicate *has_attribute* is used for modelling relations between instances. Its arguments specify the instance names of two system elements and the relation that holds between them. The relation is represented directly in the predicate by the name of the second argument (see table 5.2). Relations can be used in two ways. They can either model a structural relation between two instances or an attribute of an instance. In the latter case the attribute is represented as an atom, for example $has\_attribute(Con1, openness, open)$. Additional examples of how these design elements are used are given in table 5.7. A isa-hierarchy is shown in table 5.3.

System element facets	**Design element**
- generic	isa(SubName, SuperName).
- instance	instance(InstanceName, GenericName).
- relation	has_attribute(InstanceName1, relation, InstanceName2).

Table 5.2: Design elements for system elements

It is important to notice that all instances are created and further managed by *GARP*. In the *instance* predicate, for example, *GARP* will create a unique identifier for the in-

[4] *GARP* is implemented in *SWI-Prolog* [139]. The following guidelines help to understand the examples. Predicate names are written in small characters, followed by a bracket, and ending with a bracket followed by a point, thus: $predicate\_name()$. Variables start with a capital, followed by a string of characters, thus: $This\_is\_a\_variable12$. Variables with exactly the same name appearing within a single predicate, share their information (this is called unification). A stand alone underscore represents a variable that does not share information with other variables. An underscore has two uses in *GARP*: representing default and unknown (or not specified) knowledge. When encountering an underscore *GARP* first tries to insert the default, if default knowledge is not present it treats the underscore as representing unknown or unspecified knowledge. A list of predicates or variables starts with '[' and ends with ']'. A List as a whole can also be represented as a variable.

stance name, by extending the generic name with a unique number,[5] by which it can recognise the instance at any time and any place in the prediction model.

isa(entity, nil).
isa(heat_exchangeable_object, entity).
isa(physical_object, heat_exchangeable_object).
isa(substance, heat_exchangeable_object).
isa(container, physical_object).
isa(gas, substance).
isa(liquid, substance).
isa(solid, substance).
isa(water, liquid).
isa(ice, solid).
isa(path, entity).
isa(fluid_path, path).
isa(heat_path, path).

Table 5.3: Design elements implementing a generic isa hierarchy

5.2.1.2 Parameters and Parameter Values

A parameter is represented by a four place predicate with the same name as the parameter and as arguments a reference to the system element the parameter applies to, the instance name of the parameter, the type of the parameter and a reference to the quantity space that specifies the values the parameter may have:

$$parametername(InstanceName, ParInstanceName, Type, Qspace).$$

The type can be either *discrete, qualitative* or *quantitative*. The type qualitative is default. None of the examples that we have implemented in *GARP* uses the type *quantitative*. The quantity space of a parameter is further discussed in the next section.

The value of a parameter is represented by the predicate *value*. Its arguments represent the instance name of the parameter to which it belongs, the quantitative value the parameter has (optional), the qualitative value the parameter has, and the value of the derivative of the parameter.

$$value(ParInstanceName, Quantitative\_Value, Qualitative\_Value, Derivative).$$

Examples of how design elements for parameters and parameter values are used, are given in table 5.7.

5.2.1.3 Quantity Space

The design element for quantity spaces is the predicate *quantity_space*. Its arguments are a unique symbol by which it can be referred to, an instance name of the parameter for

[5] In the text we will write a $Variable12$ with a capital and a numerical extension when we refer to an instance name that is going to be, or should have been, created by $GARP$.

which the quantity space is instantiated, and a list of alternating points and intervals:

$$quantity\_space(zp\_max, X, [point(zero), plus, point(max\_plus(X))]).$$

The identifier (zp_max) can be used in the parameter predicate for referring to the type of quantity space that the parameter uses. In the list of alternating points and intervals (third argument), the points are made specific for a certain parameter as soon as the instance of the quantity space is created by *GARP* for that parameter:

$$quantity\_space(zp\_max, Height18, [point(zero), plus, point(max\_plus(Height18))]).$$

This means that in each instance of a quantity space the points are unique for a certain parameter and in principle unrelated to points in other quantity space instances, except for *point(zero)* which is the same for each quantity space that uses it.[6] Parameter relations between quantity spaces have to be defined in order to further relate these points (see also section 4.2.1.4).

5.2.1.4 Parameter Relations

Parameter relations are represented by a number of predicates. The name of the predicate refers to the specific constraint that the relation imposes on the two parameters (see table 5.4).

The arguments of the inequality, subtraction and addition relations may refer to both instantiated parameter names and instantiated point values of quantity spaces. If, for example, $Height18$ and $Height19$ are two instantiated parameter names and $max\_plus(Height18)$ and $max\_plus(Height19)$ are two instantiated point values (of two different quantity spaces), then inequality relations can be specified as given in table 5.5.

The two arguments for the predicate representing inequalities between derivatives are more restricted. The relations d_equal, $d\_greater$, and $d\_smaller$ are primarily used for specifying the derivative of a parameter. Parameter $Height18$ is increasing, for example, is modelled as:

$$d\_greater(Height18, zero).$$

Giving a parameter a value and a derivative by using inequality relations is another way of specifying the value predicate. Instead of the $d\_greater$ mentioned above and the relation with point $max\_plus(Height18)$ as given in table 5.5 we also could have written:

$$value(Height18, \_\, , max\_plus(Height18), plus).$$

Using the parameter relations has the advantage that the assumption mechanism can be used. The inequality relations between parameters that are consistent with the knowledge already present in an SMD, but that cannot be derived from that knowledge, are assumed to be true by the specification inference. This mechanism is further discussed in section 5.2.3.

The predicates min and $plus$[7] are context sensitive and can therefore be used both for inequalities between qualitative values and between derivatives (see table 5.6). Notice

[6] Notice that any name can be used for representing points and intervals. We could, for example, have defined $point(max\_plus(Height18)$ as $point(maximum(Height18)$ or $point(max(Height18)$.

[7] The predicate names min and $plus$ should not be confused with values from a quantity space that may have the same name.

Parameter relations	**Design element**
Inequalities	equal(Arg1, Arg2). greater(Arg1, Arg2). greater_or_equal(Arg1, Arg2). smaller(Arg1, Arg2). smaller_or_equal(Arg1, Arg2).
Subtraction *Addition*	min(Arg1, Arg2). plus(Arg1, Arg2).
Inequalities for derivatives	d_equal(Arg1, Arg2). d_greater(Arg1, Arg2). d_smaller(Arg1, Arg2).
Correspondences	v_correspondence(Par1, Val1, Par2, Val2). dir_v_correspondence(Par1, Val1, Par2, Val2). q_correspondence(Par1, Par2). dir_q_correspondence(Par1, Par2).
Proportionalities	prop_pos(Par1, Par2). prop_neg(Par1, Par2).
Influences	inf_pos_by(Par1, Par2). inf_neg_by(Par1, Par2).
Implications	iff(List1, List2). if(List1, List2).

Table 5.4: Design elements implementing parameter relations

Related entities	**Instantiated relation**
Two parameters	$equal(Height18, Height19)$
Two quantity space points	$equal(max\_plus(Height18), max\_plus(Height19))$
Parameter and Q. space point	$equal(Height18, max\_plus(Height18))$

Table 5.5: Examples of inequality relations

that the subtraction and the addition relation are always used as arguments in inequality relations.

The predicates representing the correspondence relations must be interpreted as follows:

Value correspondence: a certain value ($Val1$) of one parameter ($Par1$) always appears together with a certain value ($Val2$) of another parameter ($Par2$), and therefore, if one of the values is known the other can immediately be derived ($v\_correspondence$).

Directed value correspondence: is a value correspondence that only holds if the second argument is known ($dir\_v\_correspondence$). If only the first parameter value is known then the relation does not give *any* additional information. The $substance\_flow$ through a pipe, for example, will be $zero$ if the $flow\_area$ of the pipe is $zero$ (and not the other way around):

$$dir\_v\_correspondence(substance\_flow14, zero, flow\_area12, zero)$$

Contexts of min **and** $plus$	**Relates**
$equal(min(Height18, Height19), zero)$.	Interval values
$d\_equal(plus(Height18, Height19), zero)$.	Derivatives

Table 5.6: Context sensitive use of min and $plus$

Quantity space correspondence: is a value correspondence that holds for all values in the quantity spaces ($q\_correspondence$), i.e. for each value of parameter ($Par1$) there is a corresponding value in the quantity space of parameter ($Par2$). Recall that an essential precondition for using a quantity correspondence is that the quantity spaces of the two parameters have an equal number of points and intervals in their quantity spaces.

Directed quantity space correspondence: is a quantity space correspondence, specifying that parameter ($Par1$) can be determined only when the corresponding values(s) of parameter ($Par2$) are known ($dir\_q\_correspondence$).

The predicates representing the proportionality relations and the influences must be interpreted as follows:

Proportionalities: a change in parameter ($Par2$) causes a change in parameter ($Par1$) in the *same* direction ($prop\_pos$), or in the *opposite* direction ($prop\_neg$).

Influences: if the qualitative value of $Par2$ is greater than $zero$, then $Par1$ tends to increase (inf_pos_by) or decrease (inf_neg_by) as a result of $Par1$.
If the qualitative value of $Par2$ is less than $zero$, then $Par1$ tends to decrease (inf_pos_by) or to increase (inf_neg_by).
If the value of $Par2$ is $zero$ then it has no effect on the derivative of $Par2$.

Finally, there are two predicates for representing implications which can be used for modelling conditional statements between two lists of relations. The relation:

$$if(List\_of\_Relations1, List\_of\_Relations2)$$

is directed, meaning (contrary to other directed relations) that the *first* list of relations ($List\_of\_Relations1$) has to be true in order for the second list of relations ($List\_of\_Relations2$) to hold. Both lists may contain either a single relation or a whole conjunction of relations. For example:

$$iff([equal(Arg2, Arg3), d\_equal(Arg3, zero)], [prop\_pos(Arg1, Arg2)]).$$

The predicate iff represents the undirected implication. The implication allows the expression of relations that do not have a separate status in $GARP$.

5.2.1.5 Input Systems

The design element for input systems is an assembly of other design elements and is represented by the predicate *smd* with five arguments. These arguments represent the name of the input system, the system elements, the parameters, the parameter values, the

```
smd( input_system( 'U-tube filled with water' ),
     sysem_elements([
         instance( Con1, container ),
         has_attribute( Con1, openness, open ),
         instance( Con2, container ),
         has_attribute( Con2, openness, open ),
         instance( Water1, water ),
         instance( Water2, water ),
         has_attribute( Con1, contains, Water1 ),
         has_attribute( Con2, contains, Water2 ),
         instance( Path, fluid_path ),
         has_attribute( Path, connected, Con1 ),
         has_attribute( Path, connected, Con2 ) ]),
     parameters([
         height( Water1, Height1, _, zp_max ),
         height( Water2, Height2, _, zp_max ) ]),
     par_values([
         value( Height1, _, plus, _ ),
         value( Height2, _, plus, _ ) ]),
     par_relations([
         greater( Height1, Height2 ) ]),
     system_structures([]) ).
```

Table 5.7: A design element implementing an input system for the U-tube

parameter relations and the system structures. The example given in table 5.7 represents a possible input system for the U-tube. The list of system elements describes the physical situation of a U-tube, namely that it consists of two containers ($Con1$ and $Con2$), both containing water ($Water1$ and $Water2$), and connected by a fluid path ($Path$). The list of parameters specifies a height for each column of water. The reference to the quantity space (zp_max) defines that the values of these parameters can be $zero$ (point), $plus$ (interval) or a certain $maximum$ (point) (more parameters can of course be introduced by the qualitative inference engine during the behaviour analysis). The list of values specifies that at the beginning both $Height1$ and $Height2$ have value $plus$. Finally, the list of parameter relations specifies that, although both parameters are in an interval directly above $zero$, they are $unequal$. $Height1$ is currently greater than $Height2$.

In $GARP$ each quantity space is specific to a single parameter. The input system for the U-tube, as defined in table 5.7, leaves open whether the maximum values for $Height1$ and $Height2$ are equal, i.e. $GARP$ allows the representation of partially ordered parameter values. The missing knowledge may lead to ambiguity in the final behaviour prediction. Adding an inequality relation between the maximum values of both quantity spaces, such as:

$$equal(max\_plus(Height1), max\_plus(Height2))$$

would remove this ambiguity.

The last argument of the *input_system* predicate (*system_structures*) allows partial models to be prespecified in the input system. In this way the input system can already be specified in a certain (partial) state of behaviour. The notion of system structures will be further discussed in the next section.

5.2.1.6 Partial Models

The design element for modelling partial models is, similar to that used for input systems, an assembly of other design elements. It is represented by the predicate *system_structures* and has four arguments. The first argument specifies the name of the partial model being represented by the predicate. The name is a predicate in itself. It has one argument that is a list of the *instantiated names* of the system elements that are subject to the behaviour represented by the partial model. In the example presented in table 5.8 the name specifies that the partial model is a liquid flow involving two containers (*Con*1 and *Con*2) and a path (*Path*). The name can be used by other partial models as a conditional statement in both their *isa* and their *system_structures* argument.

```
system_structures( liquid_flow( (Path, Con1, Con2) ),
    isa([ process_model ]),
    conditions([
        system_elements([
            instance( Path, fluid_path ),
            has_attribute( Path, connected, Con1 ),
            has_attribute( Path, connected, Con2 ) ]),
        parameters([
            pressure( Liquid1, Pressure1, _, zp ),
            pressure( Liquid2, Pressure2, _, zp ) ]),
        par_relations([
            greater( Pressure1, Pressure2 ) ]),
        system_structures([
            open_contained_liquid( (Con1, Liquid1) ),
            open_contained_liquid( (Con2, Liquid2) ) ]) ]),
    givens([
        parameters([
            amount( Liquid1, Amount1, _, zp ),
            amount( Liquid2, Amount2, _, zp ),
            flow_rate( (Con1, Con2), Flow_rate, _, zp ) ]),
        par_relations([
            equal( Flow_rate, min( Pressure1, Pressure2 ) ),
            inf_neg_by( Amount1, Flow_rate ),
            inf_pos_by( Amount2, Flow_rate ) ]) ]) ).
```

Table 5.8: Design element implementing a partial model of the liquid-flow process

The second argument of the *system_structures* predicate represents the supertype relation (*isa*) that exists between partial models. The top-nodes can be either *process_model*, *single_description_model*, *composition_model*, *decomposition_model*, or *agent_model*. The super-type relation specifies the place of the partial model in one of these hierarchies by either referring to the top-node or to some other partial model in the hierarchy. Multiple inheritance is achieved by having the name of more than one partial models present in the list of the *isa* argument.

The third and the fourth argument represent the *conditions* and *givens* that are specific for the partial model. The *conditions* specify which entities must be present before the *system_structure* is applicable, the *givens* specify the new information (consequences) that can be added to the SMD that the partial model applies to when the conditions are true. Both the *conditions* and the *givens* may, similar to the input system, contain lists of system elements, parameters, parameter values, parameter relations and system structures.[8]

The *system_structures* in the *conditions* differs from the *isa* because it represents an *applies-to* relation instead of a *supertype*. The $liquid\_flow((Path, Con1, Con2))$ in the example is not a subtype of $open\_contained\_liquid((Con1, Liquid1))$, but it applies to this partial model, i.e. it refers to other *system_structures* that must be present in order for the *system_structure* itself to be applicable. In a way, the applies-to hierarchy, together with the list of system elements represents the structural *view* on the real-world system to which the partial model that has them as conditions, adds new properties.

There are three places where inconsistencies can occur: (1) the *givens* of a partial model can be inconsistent with the knowledge already present in the SMD, and the knowledge specified in partial models that are related by (2) a supertype relation, or (3) by an applies-to relation, may be inconsistent. In *GARP*, these inconsistencies are all interpreted as incorrect domain models. Different behaviours should always be specified by *exclusive* (sub hierarchies of) partial models. Inconsistencies, as described above should not occur (see also section 5.2.3).

5.2.1.7 Transformation Rules

All transformation rules are represented by the predicate *rule*. The first argument of the predicate specifies its type, either *termination*, *precedence*, or *continuity*. The second argument of the predicate is the name of the transformation rule. Similarly to the name argument of the *system_structures* predicate, this argument may include instance names that refer to knowledge (such as, system elements or parameter names) which is relevant for using the precedence *rules*.

The predicate for termination rules has in addition to its type and name argument two other arguments, the *condition* and the *result*. The *condition* argument specifies under which conditions the rule is applicable, whereas the *result* argument specifies how the entities in the *condition* argument, that turned out to be present in the current SMD, will have changed in the next SMD. Both the *condition* and the *result* arguments consist of lists of predicates representing system elements, parameters, parameter values and parameter relations (system structures may not be used). In addition, the *condition* argument may

[8] Proportionality relations and influences may not be defined as conditions. Also it is unusual to define a complete *system_structure* in the givens of another *system_structure*. Especially from an incremental approach to modelling, it would be more appropriate to define this new *system_structure* as a separate entity, possibly with only the other *system_structure* as a condition, or as being a subtype of the latter.

include specific requirements about quantity spaces. The *quantity_space* argument can be used for:

- testing if a value is an *interval* or a *point*, and
- finding the adjacent value in the quantity space.

In table 5.9 an example is shown of the termination rule *to_point_below*(*Par*). The knowledge that is modelled by this *rule* is explained in table 4.7 in the previous chapter (section 4.2.1.14). Basically it defines that the parameter value *Interval* must change in the next SMD to *Point*.

```
rule( termination, to_point_below( Par ),
      condition([
          par_values([
              value( Par, Q, Interval, min ) ]),
          quantity_space([
              interval( Par, Interval ),
              meets( Par, Point, Interval ) ]) ]),
      result([
          par_values([
              value( Par, Q, Point, _ ) ]) ]) ).
```

Table 5.9: Design element implementing a termination rule

The predicate representing precedence rules has two arguments in addition to its type and name argument. The *condition* argument specifies the conditions under which the rule applies whereas the *action* argument specifies the manipulation that must be carried when the rule is applicable. The *condition* argument may contain lists representing predicates of parameters, parameter values, and parameter relations (system elements and system structures may not be used). In addition, the *termination* argument must specify the two terminations that are going to be ordered by the rule. The *action* argument of the precedence rules can manipulate these terminations in two ways: *merging* or *removing*.[9] In table 5.10 an example is shown of the precedence rule *merge_correspondence*($Par1$, $Par2$). It contains the knowledge that two terminations towards an interval can be treated as one termination if the parameter values are corresponding values. More details concerning the knowledge represented in this rule can be found in table 4.8 in the previous chapter (section 4.2.1.14).

The *condition* argument of the predicate representing the continuity rules may only contain a list of predicates representing parameter values. The *givens* argument may contain lists of predicates representing parameter values and parameter relations. Usually, the continuity rule specifies the degree of freedom that one allows for parameter values and derivatives that do not change between two SMD's. The *givens* argument in this

[9]In most cases one termination is removed in favour of keeping the other, but removing both terminations is also possible.

```
rule( precedence, merge_correspondence( Par1, Par2 ),
        condition([
             terminations([
                  to_interval_below( Par1 ),
                  to_interval_below( Par2 ) ]),
             par_values([
                  value( Par1, _, Value1, _ ),
                  value( Par2, _, Value2, _ ) ]),
             par_relations([
                  correspondence( Par1, Value1, Par2, Value2 ) ]) ]),
        action([
             merge([
                  to_interval_below( Par1 ),
                  to_interval_below( Par2 ) ]) ]) ).
```

Table 5.10: Design element implementing a precedence rule

respect refers to conditions that should be satisfied in the new state of behaviour in order for the transformation to be valid. The example given in table 5.11 defines that if the parameter was decreasing in the old SMD, then it should keep its qualitative value and be either steady or decreasing, but not increasing, in the next SMD.

```
rule( continuity, undefined( Par ),
        condition([
             par_values([
                  value( Par, Q, I1, min ) ]) ]),
        givens([
             par_values([
                  value( Par, Q, I1, _ ) ]),
             par_relations([
                  d_smaller_or_equal( Par, zero ) ]) ]) ).
```

Table 5.11: Design element implementing a continuity rule

5.2.1.8 System Model Descriptions and their Ordering

The SMD is represented by the predicate *smd* and refers to all the knowledge that has been inferred for a single state of behaviour. It has arguments for representing knowledge about all the system elements, parameters, parameter values, parameter relations, and partial models that have been derived for a certain input system.[10] In addition, it has arguments to represent the input system to which the knowledge it represents applies, which SMD's are predecessors ($from\_relation$), and which ones are successors ($to\_relation$). The latter also represents the *cause* of the transformation, i.e. an enumeration of the used termination and precedence rules that lead to the new state of behaviour. The total set of SMD's represents the behaviour description of the input system.

5.2.2 Problem Solving Methods

An important aspect of the behavioural view in the design model is the set of problem solving methods, used for realising the problem solving potential. This section describes these methods as they are used for $GARP$. In particular, it shows how the design elements, described above, are used by specific algorithms in order to realise the problem solving goals. Section 5.2.3 describes how $GARP$ searches for partial models that apply to an input system. Sections 5.2.3.4 and 5.2.3.5 focus on the constraint satisfaction that is needed to support the specification inference. Finally, section 5.2.4 explains how $GARP$ identifies successive states of behaviour.

5.2.3 Specification of a System Model Description

An input system contains a description of real-world entities in terms of system elements, parameters, parameter values, parameter relations and system structures. This description can be:

- the initial input system supplied by the user, or
- the result of transforming an SMD (section 5.2.4).

The task carried out by the specification method is to develop a full description of an SMD in terms of partial models that apply to the input system. Developing such a full SMD starts by first adding the input system to this description and then proceeds by gradually augmenting it with instances of *applicable* partial models.

5.2.3.1 Depth-first Specification

Developing a graph of possible behaviours can be done in a *depth-first* or a *breadth-first* way. The breadth-first method is used in the component centred approach by taking the cross-product of all qualitative states. This results in the generation of all possible states of behaviour (=total-envisionment). The depth-first method is used in the first implementation of the process centred approach (GIZMO) where only those behaviours are generated, for a particular configuration of system elements, that match the conditions introduced by the input system and the applicable partial models (=attainable envisionment).[11].

Given the constraints that were put on the functional decomposition (see section 5.1.1) a step by step development of the behaviour graph, resulting in an attainable envisionment must be preferred. For supporting the development of prediction models it is beneficial

[10]The knowledge present in each of the system structures is also present in overall lists of system elements, parameters, etc. This knowledge is represented twice for efficiency.

[11]The QPE implementation [73; 68] is essentially a breadth-first method

that the knowledge engineer can focus on, and interact with the prediction of a single state of behaviour. This appears to be more intuitive and therefore easier to handle than a breadth-first approach. However, the algorithm developed for the specification inference in *GARP* is such that if it is presented with an underspecified input system (for example, an input system that contains only a list of system elements) then it will find all possible behaviours of such an input system. While retaining the advantages of the depth-first method, a total-envisionment is then generated.

Constructing an efficient depth-first search algorithm introduces two problems that have to be dealt with. Firstly, finding applicable partial models essentially means retrieving generic partial models from the library and matching their description with what is already included in the SMD. However, the SMD will be extended with new instances when partial models are included and thereby allow other partial models to be applicable that were not applicable previously. Furthermore, multiple instances of partial models may be found for different sets of system elements. Hence, the complete library must be taken into account at all times. Simply matching partial model descriptions against what is already included in the SMD means that the same instances will be found over and over again, which is not difficult to recognise afterwards, but which can be expensive. To avoid this cost, subsets of instances, that match with partial model conditions, need to be generated in such a way that each subset is unique. However, we do not want to generate all possible subsets, but prefer to be guided by the conditions that match with them. This is accomplished in the algorithm by distinguishing between new and known instances. The new instances are used one by one to find all partial models that have one matching condition. All other conditions must match with the known instances. Any instance that is used in this manner will from then on be a known instance. By this means a set of *candidates* is generated, i.e. partial models that apply to the instances of both system elements and partial models included in the SMD.

Secondly, for a candidate to be applicable, its parameter relations and values must be derivable from, or at least be consistent with, those already in the SMD (see section 5.2.3.4). When values or relations are not derivable, but can be assumed, there may be mutually exclusive candidates, depending on which assumptions are made, each resulting in a possible SMD. The specification algorithm must therefore distinguish carefully between derivable and assumable candidates and determine which of these candidates exclude each other. This is done by associating an assumption-level with each assumption and marking the (in)consistency of any other assumable candidate at that or any further level. Upon backtracking an alternative candidate is chosen that has *always* been marked as inconsistent with that or any further level. Hence at a single level only mutually exclusive alternatives are found.

It is advantageous to postpone the assumption of candidates until no other candidates can be included in the SMD. This is not only computationally more efficient, it will also clarify which partial models depend on the assumptions being made.

Finally, the specification inference must result in a sound description of an SMD, in particular in terms of parameter values and parameter relations. Two constraint satisfaction methods are therefore necessary to implement the specification inference.

- Inequality reasoning determines the consistency and derivability of parameter values and inequality relations. As this determines the applicability of partial models, this method is carried out in parallel with the specification inference.

- Resolving influences and proportionality relations establishes the behaviour of parameters. As all influences on a specific parameter are known only when the SMD is completed, this method is applied after the specification inference.

These methods are described in further detail in the sections 5.2.3.4 and 5.2.3.5. In particular, we propose an advanced method for coping with the problems related to transitivity in inequality reasoning.

5.2.3.2 Extending the Set of Design Elements

During specification, the most important objects used in the algorithm are maintained as the sets:

- *Instances*: the instances of system elements and partial models included in the SMD, where we distinguish between:
  - *NewInstances*: instances not yet used to find applicable partial models.
  - *KnownInstances*: instances already used to find applicable partial models.
- *BaseRelations*: inequality relations equivalent to the parameter relations and values existing in the SMD.
- *DerivableRelations*: the closure of BaseRelations under the inference rules for inequality reasoning (see section 5.2.3.4).
- *Candidates*: partial models that apply to Instances, but whose parameter values and relations have not yet been tested.
- *AssumableCandidates*: partial models whose parameter relations are not in DerivableRelations, but consistent with BaseRelations.

5.2.3.3 The Specification Algorithm

To start with, let the instances of system elements and partial models (=their names with instantiated system element names) from the input system (either provided by the user or resulting from a transformation step, 5.2.4) be the set NewInstances. Let KnownInstances, Candidates and AssumableCandidates be empty sets. BaseRelations is the set of inequality relations equivalent to parameter relations and values from the input system or the previous state.

1. *Finding the Candidates*: for each i in NewInstances:
   (a) Find all partial models S that apply to i and instances from KnownInstances. Find generic parent structures for S that apply to KnownInstances or already existing instances of these parents. Add S and its *new* parents to the set of Candidates. As a partial model inherits all attributes from its super-concepts, it and its *new* parents form a single Candidate.
   (b) Remove i from NewInstances and add it to KnownInstances.

2. *Test the Candidates*: for each partial model in Candidates, consider the parameter values and relations in its conditions:

   (a) If derivable from BaseRelations (thus in DerivableRelations), include it in the SMD. Create instances for the partial model and system elements in its givens and add these to NewInstances. Add the parameter relations and values to BaseRelations.[12] Influences and proportionality relations are collected, but cannot be resolved before the SMD is completed.

   (b) If inconsistent with the BaseRelations, ignore the Candidate.

   (c) If assumable, i.e. adding the relations is consistent with the BaseRelations, move the Candidate to AssumableCandidates,

3. If NewInstances is non-empty: goto 1

4. *Test AssumableCandidates*. Reconsider the assumptions (inequality relations) that are necessary for each assumable candidate:

   (a) If now derivable from BaseRelations, as a result of the inclusion of other candidates, add the partial model as in 2a. Mark it as being consistent with the current assumption level.

   (b) When going through this loop for the first time (i.e no assumptions have yet been made):

      i. *then:* if now inconsistent with the BaseRelations, ignore the Candidate,

      ii. *else:* (i.e. when assumptions have been made) if now inconsistent with BaseRelations, mark the partial model as being inconsistent with the current assumption level.

5. If NewInstances is non-empty: goto 1

6. *Make an assumption*: At this point, there are no more NewInstances that can be used to find new Candidates. The Candidates found so far are either:

   - included in the SMD, because their conditions are derivable from the SMD, or
   - ignored because their conditions are inconsistent with the SMD, or
   - in the set AssumableCandidates because one or more conditions is not derivable from, but consistent with the SMD.

   If AssumableCandidates is non empty, create a new assumption level and *repeat*:

   (a) Make a copy of the current SMD.

   (b) When going through this loop for the first time (i.e each time when a copy of the SMD is made):

      i. *then:* assume the parameter-relations and values for some partial model in AssumableCandidates. The partial model and its givens are added to the copied SMD as in 2a,

[12] This may cause a contradiction, which usually is a result of an erroneous model and implies that the SMD is invalid.

ii. *else:* find an assumable candidate that was always marked inconsistent with the current assumption level or any further level in step 4. Add this candidate to the copied SMD.

(c) The copied SMD is completed by going through the steps from 1 and eventually returned as a result.

When the specification algorithm is finished the SMD that has been subject of the reasoning process should marked as being *interpreted* (=augmented SMD).

5.2.3.4 Inequality Reasoning

The qualitative calculus, as originally proposed in [57; 92] cannot derive the result of the addition of a positive and a negative quantity, even if additional information about the ordering between the magnitude of these quantities is known. This leads, among other reasons, to *spurious behaviour*, which is a well known problem [127].[13] As a simple example, consider the energy system of figure 5.2. In this closed system the total amount of energy exchange must be zero. Suppose that a working washing machine consumes more

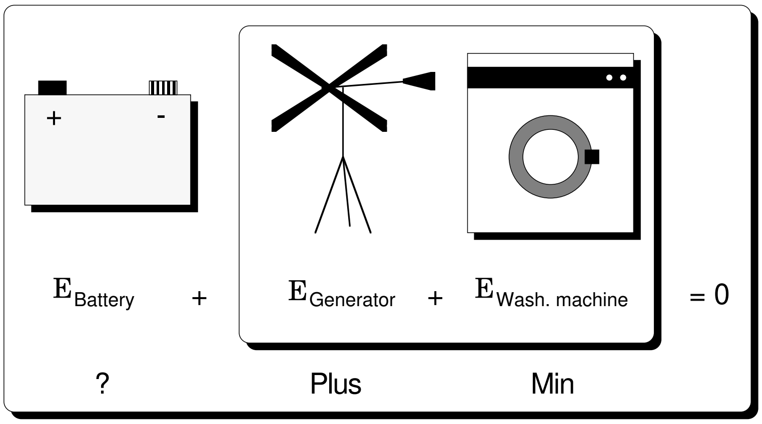

Figure 5.2: Energy system

energy than the windmill can produce. We would then like to derive that the accumulator (battery) must release energy in such a case, i.e. if the washing machine is working. For us it is easy to see that substitution of:

$$E_{generator} + E_{washingmachine} < 0$$

in:

$$E_{battery} + E_{generator} + E_{washingmachine} = 0$$

will provide that result. However the qualitative calculus mentioned above does not allow that. When evaluating:

$$E_{battery} + E_{generator} + E_{washingmachine}$$

[13] If we define spurious behaviour as predicted behaviour that can never occur in the real-world, there can be two causes for spurious behaviour: a lack of proper inference rules (calculus) or incomplete information in the model (ambiguity). This section is concerned with the first type of spurious behaviour.

the result of:

$$E_{generator} + E_{washingmachine}$$

$(plus + min)$ is ambiguous, thus we may assume $E_{battery}$ to be positive, negative or zero. Especially when conservation of energy, flow or force should be modelled, this is a serious problem.

Forbus [71] maintains a partial ordering of inequality relations between quantities. When provided with proper inference rules for maintaining this partial ordering it is possible to eliminate this kind of spurious behaviour. Various proposals concerning the derivation of such a partial ordering have been made. Simmons [120] applies a set of axioms for reasoning about inequalities and arithmetic expressions, for example:[14]

$$a > b \rightarrow a + c > b + c$$

As Simmons recognises, applying such axioms on every quantity in the system will lead to an infinite set of new expressions. Therefore the additional constraint is imposed that expressions (e.g. $a+b$ and $b+c$) must already exist in the system, thereby impeding some necessary derivations that require expressions not in the system as an intermediate result. For example, when searching for the inconsistency in:

$$a > c, b > d, a + b < c + d$$

we need the intermediate result:

$$a + b > c + b$$

where $c + b$ does not already exist as an expression. In Simmons' approach, an inequality relation between quantities is determined by applying these axioms in a breadth-first way whenever some part of the system queries that relation. This is probably preferable if there is a large, but fixed, set of equations in a single world. However, for our purposes this is not a satisfactory method. During the course of the reasoning process new relations will be added to the partial ordering. Therefore a search path that was previously a dead end, may later provide the desired result. This means that the same paths must be inspected over and over again. Moreover, finding a contradiction $(N > N)$ in the partial ordering in effect means inspecting the transitive closure. Searching for inequality relations between quantities whenever the system needs to establish the derivability of that relation is thus far more inefficient than maintaining the closure under the applied inference rules in the first place, so that all queries can be answered without redundant search.

The method Forbus [73] uses for reasoning with inequalities maintains the closure for inferences concerning transitivity. However, these inferences are oriented towards finding inconsistencies and do *not* apply to arithmetic summation. This method applies an Assumption based Truth Maintenance System ($ATMS$) [54], which enables re-usability of derivations in multiple contexts (caching). This is of course more advantageous if:

- derivations are expensive, and
- there is a large amount of overlap in derivations between different problem solving contexts.

[14] [120] also provides axioms for reasoning about multiplications. These are not applied in our approach, as the related functions are usually expressed in terms of influences and proportionality relations.

In $GARP$ these two criteria are only partially fulfilled. The specification inference employs a form of backtracking when it makes assumptions. Each assumption is made in a different problem solving context, but these contexts share (a copy of) the derivations made so far, including those for inequality reasoning. This will not cover all the overlap in derivations between different contexts, but the inference rules, described below, are fairly inexpensive to execute. Finally, the use of the $ATMS$ involves some overhead, especially the prevention of inferences that are useless because the antecedents do not occur in any problem solving context. Therefore, we concluded that the application of the $ATMS$ in our approach is not worthwhile at this level of inference.

The specification of an SMD, makes explicit the derivability, assumability or incompatibility of partial model conditions, i.e. parameter values and parameter relations. In order to reason about inequalities, parameter values are rewritten as inequality relations between the parameter and points on its quantity space. Furthermore, the ordering of points on any quantity space is asserted as a set of inequalities. We also assert that a parameter cannot receive a value, or change to a value, beyond a value in its quantity space. This rewriting system allows the specification and reasoning about quantity spaces containing any number of values.

In order to solve problems such as the energy problem above, we need axioms for transitivity, elimination, and substitution with regard to inequality relations and arithmetic summation. As mentioned, these axioms are cumbersome because search can go in many directions, unless it is restricted to result in arithmetic expressions within the system. The technique we use combines the axioms for transitivity and reasoning about arithmetic summation in a simple and efficient inference mechanism, based on summation of inequality relations, in such a way that the restriction to expressions actually in the system is not necessary to prevent search from going out of bounds. Table 5.12 summarises the possible inferences (A,B,C and D are single quantities or sums of quantities).

	$A = B$	$A >= B$	$A > B$
$C = D$	$A + C = B + D$	$A + C >= B + D$	$A + C > B + D$
$C >= D$	$A + C >= B + D$	$A + C >= B + D$	$A + C > B + D$
$C > D$	$A + C > B + D$	$A + C > B + D$	$A + C > B + D$

Table 5.12: Inference rules for inequality reasoning

In order to guarantee progress we impose three additional constraints: [15]

1. the result of a summation can be simplified (at least one quantity occurs on both sides of the result, thus in A & C and in B & D in table 5.12);

2. the result may not contain the same quantity twice on one side (which would result in ever growing sums);

3. a relation cannot be combined with any of its antecedents.

Some additional inference rules are necessary, expressing that some results are stronger than others. For example, when $a > b$ and $a >= b$ are both derived, $a > b$ is preferred.

[15] The implementation represents sums as bitmaps, which allows for fast testing of these constraints.

For the energy system above, it is now possible to determine the value of the battery's energy dissipation (figure 5.3).

$$
\begin{array}{l}
E_{\text{Battery}} + E_{\text{Generator}} + E_{\text{Wash. machine}} = 0 \\
\qquad\qquad\qquad\qquad 0 > E_{\text{Generator}} + E_{\text{Wash. machine}} \\
\hline
E_{\text{Battery}} + \cancel{E_{\text{Generator}} + E_{\text{Wash. machine}}} > \cancel{E_{\text{Generator}} + E_{\text{Wash. machine}}} \\
\qquad\qquad\qquad\qquad 0 > E_{\text{Generator}}
\end{array}
$$

Figure 5.3: Solution for energy system

These inferences are applied in an exhaustive breadth-first way, incrementally when a new parameter-relation or value is added to the SMD. Thereby the closure under the inference rules is generated, so that it is always known if a partial model's conditions are derivable. Computing the *full* closure can still be costly. Two optimisations greatly reduce these costs. Firstly, most quantities will be related to zero. It is not necessary to derive all transitive relations between quantities less than and greater than zero, in fact the largest part of the closure. If such a relation is mentioned as a condition, the derivability is established easily. Note that this optimisation will not prevent us from finding a contradiction, e.g. when $a < 0$ and $a = 0$ or $a > 0$. A second optimisation that reduces the closure of inequality relations is accomplished by treating two equal quantities as a single quantity.

By combining axioms for reasoning about transitivity and arithmetic summations, and under the constraints described above, it is possible to establish the derivability, assumability, or incompatibility of system structure conditions at reasonable costs, thereby avoiding the problems associated with the approaches of [120; 73] when applied to the same problem. In this way we are able to reduce spurious behaviour and allow quantity spaces using any number of values.

5.2.3.5 Resolving Influences and Proportionality Relations

Influences and proportionality relations are collected during the specification of an SMD and evaluated after a full specification is accomplished. In contrast with the treatment of sums in inequality relations through inequality reasoning, the summation of influences and proportionality relations must be evaluated according to a weaker calculus:

- two or more positive influences on a parameter combine to a single positive influence,
- two or more negative influences on a parameter combine to a single negative influence, and
- the combination of (one or more) positive and (one or more) negative influences is ambiguous.

Resolving influences and proportionality relations is done through a straightforward constraint satisfaction algorithm. When evaluation is not ambiguous or undetermined, the resulting value is provided to the inequality reasoner in order to find further consequences. Occasionally this leads to an inconsistency, thereby invalidating the SMD. When no more progress can be made and some relations are unresolved, two assumptions can be made:

- The value of a parameter is *zero* if that parameter directly influences (influence relation) other parameters, but is not influenced itself.
- The derivative of a parameter is *zero* if that parameter indirectly influences (proportionality relation) other parameters, but is not influenced itself.

Both assumptions specify that if a parameter is not influenced itself, it can not influence other parameters either.

5.2.4 Finding Successive System Model Descriptions

The transformation inference is concerned with identifying successive states of behaviour. Section 5.2.4.1 describes how the termination rules are used for finding potential terminations. Section 5.2.4.2 describes how the precedence rules are applied for ordering the applicable terminations. Finally, section 5.2.4.3 describes how the resulting set of terminations is used for finding a termination to an already existing SMD or how they initialise the specification of a new SMD. The algorithm as a whole is depicted in figure 5.4.

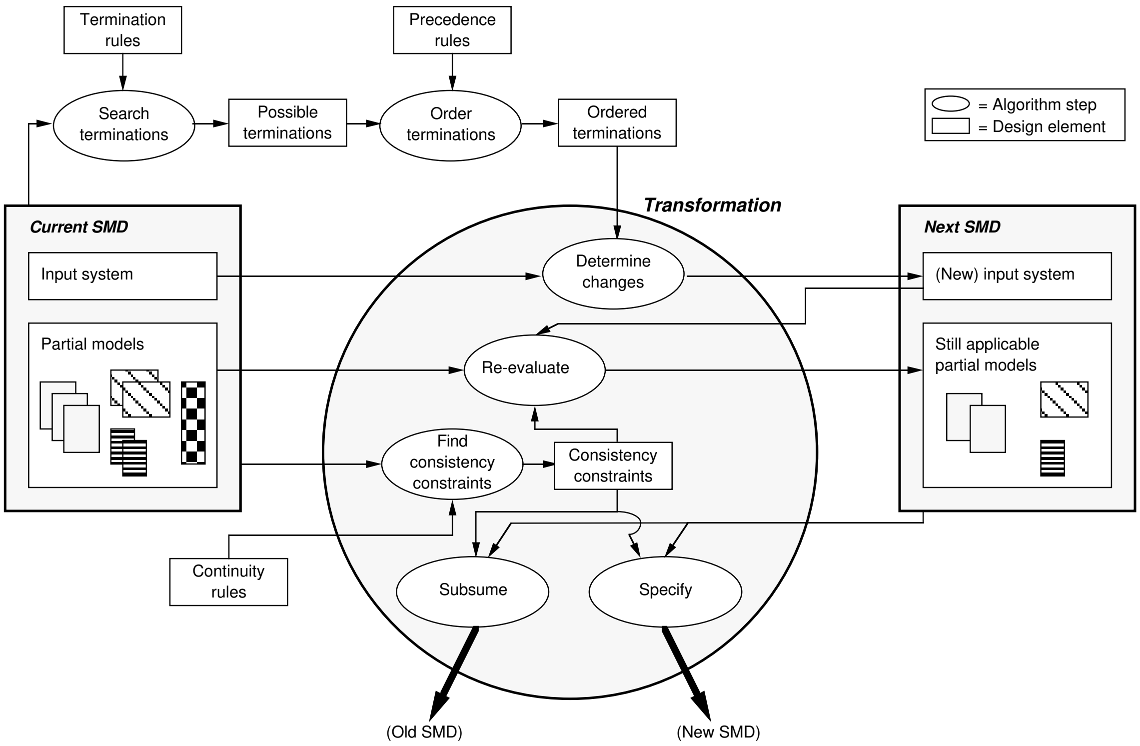

Figure 5.4: Transformation algorithm

5.2.4.1 Search for Possible Terminations

Finding potential terminations requires a relatively simple match between SMD and the termination rules. The input consists of:

- a fully specified SMD, and
- the termination rules in the library.

The algorithm goes as follows:

- For the SMD that is subject of the termination inference:

  1. Match the conditions of each rule from the list of termination rules with the SMD. If the match succeeds then save the instantiated rule in the list of possible terminations of the SMD.
  2. When no more termination rules can be found then mark the SMD as *termination* (also if no terminations could be found at all).

With respect to the parameter relations it is relevant that the algorithm performs a direct match and does not include any transitivity inferencing. In other words, terminations apply only to the knowledge that is directly *present* in the SMD.

5.2.4.2 Order Possible Terminations

The ordering of possible terminations takes as input:

- a terminated SMD with a list of possible terminations, and
- the precedence rules present in the library.

For ordering possible terminations it is important to distinguish between terminations that belong together, since they represent aspects of a single transformation (merge), terminations that exclude each other (remove), and terminations that can neither be merged nor removed. The algorithm consists of the following steps:

1. Find a precedence rule that has as condition argument two terminations which appear in the list of terminations of the SMD. If such a precedence rule is found then go to 2. If no (more) applicable precedence rules can be found then go to 4.
2. Find out whether the additional conditions of the precedence rule match with the contents of the SMD. If the conditions match then go to 3, else go back to 1.
3. Carry out the action part of the rule:

   - *In the case of merge:*

(a) Merge both the condition and the result arguments of one termination rule into the condition and result arguments of the other termination rule (the two separate terminations are merged into a single composite termination). The name of the composite termination will be a list that contains the names of each individual termination and the name of the precedence rule that merged them.[16]
(b) Remove the two individual terminations from the list of possible terminations and add the composite termination to the remaining list.
(c) Go back to 1.

- *In the case of remove:*
  (a) Remove the termination(s) specified in the remove action argument of the precedence rule from the list of possible terminations.
  (b) Go back to 1.

The order between merging and removing is not explicitly controlled by the algorithm, but depends on the order in which the rules are stored in the library. Generally it is better to first enumerate all merging rules and then all removing rules in the library. This approach guarantees that *all* terminations related to a termination that must be removed, are indeed removed.

4. At this moment the list of possible terminations cannot be modified any further. Each composite or individual termination in the list is a termination that can occur independently of the other terminations in the list. This implies that the terminations may occur simultaneously or separately. In effect, this means that all possible combinations of the terminations are potential terminations (=cross-product), therefore: add to the list of possible terminations all combinations of terminations from that list and go to 5.

5. Mark the SMD as being ordered (also if no terminations were present.)

5.2.4.3 Apply Transformation: Subsumption, Specification and Continuity

For each ordered SMD that is object of the apply transformation algorithm the following information is available:

- the input system on which the SMD is based,
- the partial models that are part of the SMD,
- the list of possible terminations (each termination in this list specifies: (1) the aspects from the old SMD that will terminate (condition), and (2) how these aspects will reappear in the new SMD (result)),
- the continuity rules present in the library, and

[16] It could be that the merged knowledge is inconsistent, because of errors in the domain knowledge. This will not be detected here, but when the resulting termination is used for specifying the next state of behaviour.

- the total list of *all* system elements, parameters, parameter values and parameter relations that are present in the SMD.[17]

The problem to solve is to determine on the basis of this information what aspects of the old (current) SMD still hold in the next SMD. It will, for example, not be sufficient to simply copy the information from the old SMD into the new SMD and meanwhile replace the information from the old SMD that is mentioned in the condition argument of the termination by the information present in the result argument of the termination, because such an approach ignores the dependencies between partial models. If, for example, partial model A is based on the parameter relation $X > Y$ and partial model B is based on A (e.g. by an *applies to* or *is supertype of* dependency). Then simply substituting $X > Y$ by $X = Y$ (if this was the termination) would be insufficient, because partial models A and B must also be removed.

This dependency does not apply to the contents of the input system. The knowledge represented in the input system is not derived from any other knowledge during the specification step. The first step of the algorithm is therefore as follows:

1 Take the input system from the old SMD and remove those aspects from the input system that match the information in the condition argument of the termination. Merge the remaining aspects of the input system with the information in the result argument of the termination and add this to the new SMD as its input system.

The next problem is to ensure continuity between the old SMD and the new SMD. If we would directly specify the above derived input system then discontinuity between the old and new SMD may (and is very likely) to occur. The following step is therefore required:

2 Take for each parameter value from the old SMD that is not mentioned in the condition argument of the termination and find a continuity rule that specifies how the value should reappear in the next SMD (if it appears in that SMD). The resulting list of continuity constraints is not visualised in the reasoning process of *GARP* (except for notification of inconsistency when it prevents certain partial models from being added to the new SMD).

The third step in the transformation algorithm concerns the specification of the new SMD, based on the newly defined input system and the continuity constraints. One way of realising this is by presenting the new input system to the specification algorithm as described in section 5.2.3. However, this approach neglects all the information derived in the old SMD, which is undesirable both from a conceptual and from a computational point of view. The information in the old SMD provides a focused view on the total set of partial behaviour models in the library. Examining the partial models in this subset for their applicability to the next SMD shortens the inference process and matches better with the intuitive understanding of how a certain state of behaviour transforms into the next state of behaviour. The algorithm therefore proceeds as follows:

3 Re-evaluate each partial model (*system_structures*) from the old SMD with respect to its applicability in the new SMD (this inference is basically the same as that

[17] This total list also includes the information present in the input system and in each partial model.

described in section 5.2.3). The result of this inference is a partially augmented SMD, representing all the information from the old SMD that either changed (via the input system), or stayed constant but *was* consistent with the changes.

The last part of the algorithm is concerned with further specifying the partially augmented SMD. There is however a possibility that the partially augmented SMD is a subset of an already existing SMD (notice that by definition this can never be the SMD that just terminated). The algorithm therefore continues with a subsume step. If the subsume step fails a new specification follows:

1. Find all the existing SMD's that contain the partially augmented SMD as a subset. If there is no such existing SMD then go to 2, else create a transformation link between each existing SMD that obeys this constraint and the old SMD that was subject of the transformation algorithm as defined in 3.

2. Fully specify the partially augmented SMD as defined in the specification algorithm described in section 5.2.3. For each SMD that results from this specification create a transformation link between the old SMD, that was subject of the transformation algorithm, as defined in 3. If no new SMD results from the specification algorithm, then go to 4.

3. A transformation link between two SMD's is created by adding the termination (notice that this is the set of termination and precedence rules that were used) that was used for the transformation algorithm as the cause in the to argument of the SMD that terminated. Also include the identifier of the new SMD in the to argument of the terminated SMD and the identifier of the terminated SMD in the $from$ argument of the new SMD. Go to 4.

4. Remove the termination from the list of possible terminations. Mark the SMD as being closed (transformed), when all terminations have been object of the algorithm, otherwise redo the whole procedure for the next termination.

The apply transformation algorithm must be applied for each termination in the list of possible terminations that belongs to an SMD.

5.2.5 Other Functions and their Required Behaviour

In table 5.13 additional functions are enumerated that are required for implementing *GARP*. Most of these functions are realised by traditional software engineering techniques. The operating system that controls the software is used for realising the storage functions. The editors provided by the operating system are used for realising the different modification access functions, they can be called from within the artifact. In other words, while running *GARP* it is possible to call editors and change the knowledge used by *GARP*.

The visualisation function uses 'pretty-print' routines that format the knowledge present in *GARP* such that it can be read and understood by the user. The function for user control is realised by a command interface that facilitates access to the functionalities that are controlled by user. The system control function uses a loop that controls

Function name	**Method used**
Syntactic analysis	Syntax checker (*SWI-Prolog* feature)
Semantic analysis	Grammar and reference checker
System control	Control loop
User control	Command interface
Visualise	'Pretty-print' routines
Modification access	Calls to editor(s)
Store library knowledge	Operating system file management
Store input system	Operating system file management
Store intermediate P.S. results	Assert routines (*SWI-Prolog*)
Store P.S. trace	Write protocol files

Table 5.13: Overview of other functions and their operationalisations

the *GARP* functions. The syntactic analysis is carried out by the syntax checker of *SWI-Prolog*. However, the semantic analysis uses a specific grammar that reads the knowledge presented to *GARP* and tests whether

- all the design elements have their required semantics, and
- all the relevant references between design elements are adequately represented.

5.3 Physical Modules

The physical (software) modules of the implementation of *GARP* can be divided into:

- modules that represent the knowledge that is used,
- modules that implement the reasoning process, and
- a module that implements the traditional software engineering functionalities (which is not discussed further).

The design elements for representing the declarative knowledge are enumerated in table 5.14. These modules basically relate to the storage functions from the functional decomposition, except for the function $<$*store calculi*$>$. This function has been implemented as an integrated part of the algorithms described below.

The modules that contain the algorithms do not have a one-to-one relation with the functional decomposition. Instead, the algorithms are distributed across a number of modules. They are briefly described below:

Class This module contains the specify algorithm. It calls upon other algorithms for detailed inferences.

Initquantity Algorithm to create parameter specific instances of quantity spaces.

Interface This module contains all the algorithms needed to facilitate the interaction with the user. In addition, it contains 'pretty-print' routines for implementing the visualisation function and a grammar for implementing the semantic analysis function.

Module name	Description of the module
Input system	A library of input systems that can be used by *GARP* for behaviour prediction.
Isa	The isa-hierarchy of generic system elements.
Library	All the partial models that *GARP* can use for generating SMD's.
Quantity space	A list of generic quantity spaces from which instances can be created.
Rules	The termination, precedence and continuity rules for SMD transformation.
GARP scratch	A module representing a permanent file that *GARP* uses for storing the history of the problem solving process.
GARP database	The *SWI-Prolog* database that *GARP* uses to store its intermediate problem solving results.

Table 5.14: Modules containing data structures

Intern This module translates the inequality relations, present in the knowledge presented to *GARP*, into machine specific counterparts, in order to allow fast computation.

Methods This module implements the top level control structure. It interacts with the interface module and specific algorithms that carry out detailed inferences (for example, *Class* and *Reclass*). It also contains large parts of the algorithms taking care of the transformation between SMD's.

Pl-bit A module containing C-code for efficient reasoning about inequalities. It forms the basis for *Solve*.

Pllib A module that contains all kinds of small prolog algorithms (in particular list manipulations) that are often used by other modules.

Reclass Algorithms for re-evaluating the knowledge in the old SMD. This module calls upon algorithms in *Class*.

Selector As SMD's can become very large *Prolog* predicates, efficient shortcuts are needed for sharing variables. This module contains algorithms for this purpose.

Semantic This module specifies the inheritance algorithm, which is used for inheritance between (1) partial models and (2) between system elements.

Solve Algorithms for solving inequalities.

Types Type declarations for *GARP* to discriminate between parameter relations. In particular between inequalities, correspondences, influences, and proportionalities.

The minimising coupling and maximising coherence (cf. [148]), that was applied for most of the declarative part, could not be maintained for the algorithms (also knowledge redundancy is not always guaranteed). The construction of algorithms was mainly guided by the *control* on the inference process and less by the type of design elements and algorithms that were used. As a result the module composition given above is characterised according to the different types of inference competence that must be realised, instead of the different types of data that are used.

5.4 Examples of Prediction Models in *GARP*

This section describes two models that have been implemented in *GARP*: the cooling mechanism of a refrigerator and two heart diseases. The model of the cooling mechanism of the refrigerator is primarily based on inequality relations between the different temperatures in this system. The model of the heart diseases is more orientated towards the values of specific parameters. For each value of the important parameters of the heart a partial behaviour model is specified, which results in states of behaviour that are easily understood by the names of the applicable partial models. Both prediction models show the need for an integrated set of modelling primitives for representing partial behaviours.

5.4.1 Model of a Refrigerator

This section illustrates the problem solving behaviour performed by *GARP* on a model of the cooling mechanism of a refrigerator. Figure 5.5 visualises the important physical objects of the refrigerator. The following behaviour description applies to this structure:

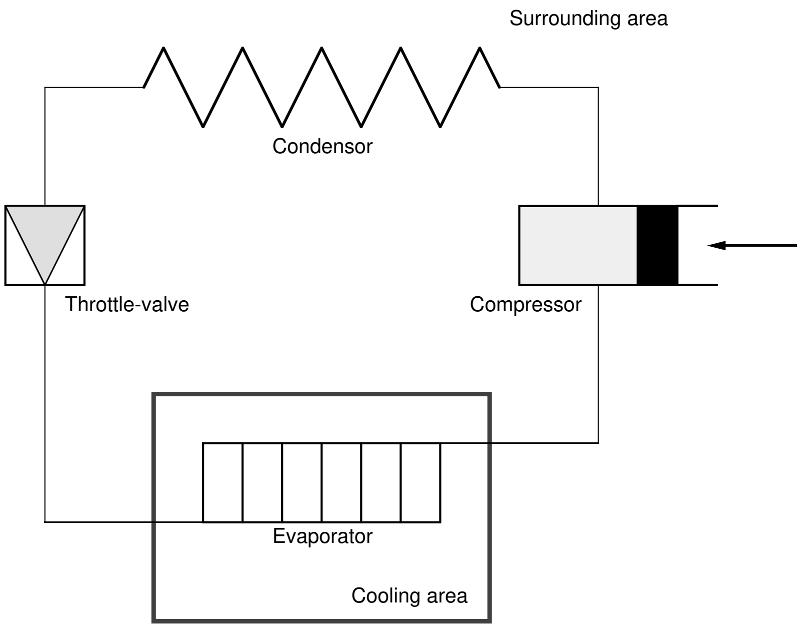

Figure 5.5: A model of the refrigerator

> *In a refrigerator based on the compression principle, gas is sucked out of the evaporator and compressed by the compressor. The compressed gas is then transformed into liquid in the condensor by cooling it with air or water. Next, the liquid goes through the throttle-valve, which decreases its pressure, and arrives in the evaporator. In the evaporator the liquid evaporates as a result*

of this low pressure and, by doing so, withdraws heat. This is where the actual cooling takes place.

At a first glance the refrigerator typically seems to be a system that can be modelled with the component centred approach, in particular, the behaviour of the compressor and the throttle-valve. However, the evaporator and the condensor cannot be represented by this approach, because their behaviour depends on the interaction with the environment (heat exchange) in which they operate (see section 2.3.1.1). This problem can be avoided when process models are used in combination with behaviour models of components.

It is also not possible to model the behaviour of the refrigerator using only the modelling primitives provided by the processes centred approach. In particular, because this approach does not have modelling primitives for representing the effects *caused by* components. For modelling the behaviour of the refrigerator the ontology for partial behaviour models presented in section 4.2.1.6 is essential.

5.4.1.1 System Elements

The hierarchy of system elements represents a functional abstraction of the objects and substances of the refrigerator (see figure 5.6). In this hierarchy of system elements the

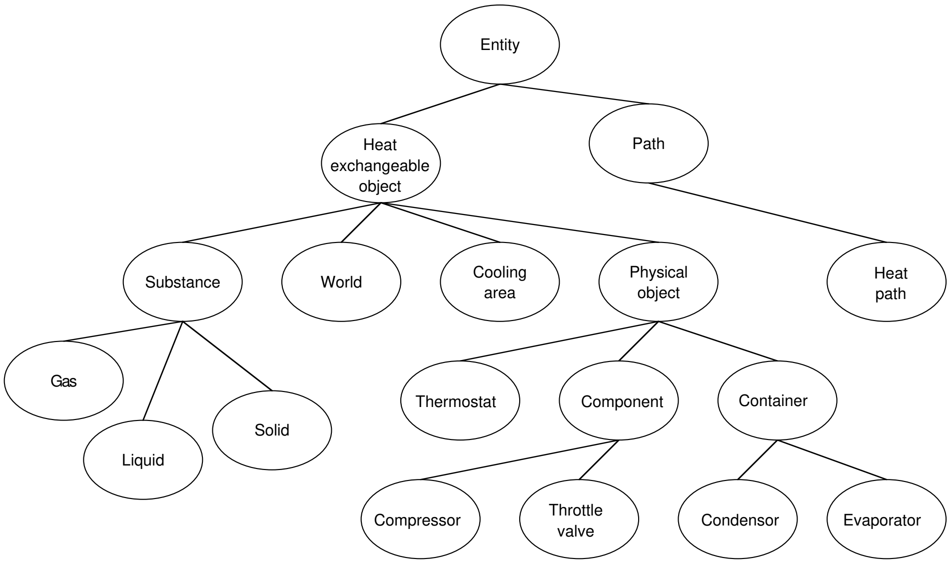

Figure 5.6: Hierarchy of system elements for the refrigerator

compressor and the throttle valve are modelled as components, whereas the condensor and the evaporator are modelled as containers. The thermostat is modelled as a physical object.

5.4.1.2 Input System, Parameters, Values and Relations

A large part of the input system (see table 5.15) is used for instantiating the structural description of the refrigerator. Instances have to be created for the outside world, the cooling area, the thermostat, the compressor, the throttle valve, the evaporator, the condensor and the substances contained by these last two containers. Two 'heat-paths' are needed for relating the world to the condensor and the evaporator to the cooling area.

Only the parameters that we want to give values or additional constraints must be defined in the input system. All the other parameters are created by *GARP* during the reasoning process as it finds the partial behaviour models that apply to the input system. In total 16 parameters are used for predicting the behaviour of the model described here (see table 5.16).

For reasoning about the behaviour of the refrigerator the parameter relations between the temperatures are important. In fact, it must be predicted that:

$$equal(Temp\_world, Temp\_C\_area)$$

changes to:

$$greater(Temp\_world, Temp\_C\_area)$$

This allows for modelling the quantity space of the temperature as a single interval. All temperatures therefore use the quantity space:

$$quantity\_space(p, \_ , [plus]).$$

except for the temperature of the cooling area. This temperature uses the quantity space:

$$quantity\_space(minimum\_p, X, [point(minimum(X)), plus]).$$

The parameter relations specify that at the beginning of the behaviour prediction all temperatures are equal. The list of parameter values specifies that the temperature of the cooling area is *plus*.

5.4.1.3 Static Models

The hierarchy of partial behaviour models is depicted in figure 5.7. There are seven partial behaviour models that describe the static properties of the system elements: five single description models and two composition models. The substance model is used for describing the basic parameters, values, and relations of the substances in the compressor and the evaporator (see table 5.17). In particular, it defines the proportional dependency between the heat and the temperature. The behaviour model for the cooling area and the world are essentially similar to the behaviour model for substances. Although the behaviour model for the world slightly differs with respect to temperature. The temperature of the world is modelled as always being constant. Each influence on the heat of the world is countered with an opposite influence on the heat. As a result, the changes in the heat and temperature of the world are neglected with respect to the behaviour of the refrigerator.

The closed container model specifies that a closed container is a container with relation $has\_attribute(Container, openess, closed)$. The closed contained substance is a composition model that specifies the behaviour of a substance when it is contained by a container with a fixed contain volume. It requires both a behaviour model for the substance and a behaviour model for the closed container to exist before it can be applied (it also requires a contain relation between the container and the substance). The closed contained

```
smd( input_system( 'Simple Refrigerator' ),
        system_elements([
            instance( World, world ),
            instance( C_area, cooling_area ),
            instance( Condensor, condensor ),
            has_attribute( Condensor, openness, closed ),
            instance( Substance_Con, substance ),
            has_attribute( Condensor, contains, Substance_Con ),
            instance( Evaporator, evaporator ),
            has_attribute( Evaporator, openness, closed ),
            instance( Substance_Evap, substance ),
            has_attribute( Evaporator, contains, Substance_Evap ),
            instance( Path_Con, heat_path ),
            has_attribute( Path_Con, connected, World ),
            has_attribute( Path_Con, connected, Substance_Con ),
            instance( Path_Evap, heat_path ),
            has_attribute( Path_Evap, connected, C_area ),
            has_attribute( Path_Evap, connected, Substance_Evap ),
            instance( Compressor, compressor ),
            has_attribute( Compressor, from, Evaporator ),
            has_attribute( Compressor, to, Condensor ),
            instance( T_valve, throttle_valve ),
            has_attribute( T_valve, from, Condensor ),
            has_attribute( T_valve, to, Evaporator ),
            instance( Thermo, thermostat ),
            has_attribute( Thermo, from, C_area ),
            has_attribute( Thermo, to, Compressor ) ]),
        parameters([
            temperature( World, Temp_world, _, p ),
            temperature( C_area, Temp_C_area, _, minimum_p ),
            temperature( Substance_Con, Temp_Sub_Con, _, p ),
            temperature( Substance_Evap, Temp_Sub_Evap, _, p ) ]),
        par_values([
            value( Temp_C_area, _, plus, _ ) ]),
        par_relations([
            equal( Temp_world, Temp_C_area ),
            equal( Temp_Sub_Con, Temp_Sub_Evap ),
            equal( Temp_world, Temp_Sub_Con ),
            equal( Temp_C_area, Temp_Sub_Evap ) ]),
        system_structures([]) ).
```

Table 5.15: Input system for reasoning about the refrigerator

Property of system element	Parameter
Outside world	Heat Temperature
Cooling area	Heat Temperature
Substance in condensor	Heat Temperature Pressure
Substance in evaporator	Heat Temperature Pressure
Compressor	Compressing rate
Throttle valve	Expansion rate
From cooling area to evaporator	Flow rate
From condensor to outside world	Flow rate
Substance in condensor	Condensing rate
Substance in evaporator	Evaporation rate

Table 5.16: Parameters for reasoning about the refrigerator

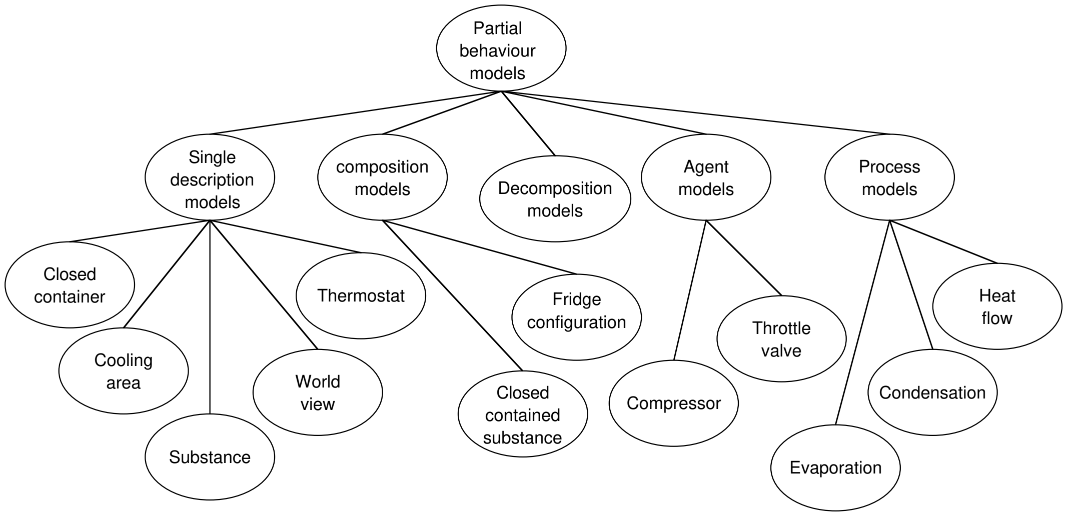

Figure 5.7: Hierarchy of partial models for the behaviour of the refrigerator

```
system_structures( substance( Sub ),
    isa([ single_description_model ]),
    conditions([
        system_elements([
            instance( Sub, substance ) ]) ]),
    givens([
        parameters([
            heat( Sub, Heat, _, p ),
            temperature( Sub, Temperature, _, p ) ]),
        par_values([
            value( Heat, _, plus, _ ),
            value( Temperature, _, plus, _ ) ]),
        par_relations([
            prop_pos( Temperature, Heat ) ]) ])).
```

Table 5.17: Single description model for substances

substance is a composition model because it specifies static behaviour that results from a configuration of system elements.

The thermostat behaves similarly to a switch. The switch is on when the value of the temperature of the cooling area is *plus* and off when the value of the temperature is *minimum*. The behaviour of the thermostat is used for determining the activity of the compressor.

5.4.1.4 Process and Agent Models

The knowledge represented in the heat-flow process is similar to the one defined by Forbus [70]. Our process model differs in the sense that it allows only heat-flows between *heat exchangeable objects* (see table 5.18).

There are two agent models in this model of the refrigerator: one for the compressor and one for the throttle valve.[18] The agent model for the compressor is given in table 5.19. It specifies that when the thermostat is active that the compressor increases the pressure in the closed contained substance into which it compresses the substance (the condensor). The throttle valve is modelled similar to the compressor. It is an agent that decreases the pressure of the substance in the evaporator.

In addition to the heat-flow process there are two other processes: condensation and evaporation. Both are used to the strengthen the effect of the heat-flows. In the case of the condensor, the condensation of gas into liquid generates energy. In the case of the evaporator, the evaporation of liquid into gas requires energy. During both processes the temperature of the substance is kept constant.

[18] The behaviour models that describe these components when they are not active are of type single description models. However, they specify only behavioural properties when they are active and therefore are modelled only as agent models.

```
system_structures( heat_flow( (Path, Object1, Object2) ),
    isa([ process_model ]),
    conditions([
        system_elements([
            instance( Object1, heat_exchangeable_object ),
            instance( Object2, heat_exchangeable_object ),
            instance( Path, heat_path ),
            has_attribute( Path, connected, Object1 ),
            has_attribute( Path, connected, Object2 ) ]),
        parameters([
            temperature( Object1, Temperature1, _, _ ),
            temperature( Object2, Temperature2, _, _ ) ]),
        par_relations([
            greater( Temperature1, Temperature2 ) ]) ]),
    givens([
        parameters([
            heat( Object1, Heat1, _, p ),
            heat( Object2, Heat2, _, p ),
            flow_rate( (Object1, Object2), Flow_rate, _, zp ) ]),
        par_relations([
            equal( Flow_rate, min( Temperature1, Temperature2 ) ),
            inf_neg_by( Heat1, Flow_rate ),
            inf_pos_by( Heat2, Flow_rate ) ]) ]) ).
```

Table 5.18: Process model for heat-flow

5.4.1.5 Transformation Rules

Termination rules that are often used during the behaviour analysis of the refrigerator are those that model changes in inequality relations, in particular: *from equal to greater* and the *from greater to equal*. In table 5.20 an example is given of an equal to greater termination rule. It specifies that when two parameters are equal and change in opposite directions, that they become unequal. This rule also puts continuity constraints on the values of the derivatives of the parameters in the next state.

5.4.1.6 The Behaviour Prediction

Given the input description and the partial behaviour models described above, *GARP* predicts eight SMD's. In figure 5.8 the important aspects of the behaviour description are depicted. In particular, it shows the inequality relations between the different temperatures, the value of the temperature of the cooling area, the status of the agent models for the compressor and throttle valve and the activity of the heat-flow processes.[19] In SMD1 the thermostat senses that the temperature of the cooling area is greater then its

[19]The condensation and evaporation processes are active when the heat-flow processes are active for the condensor and the evaporator.

```
system_structures( compressor_on( Comp ),
    isa([ agent_model ]),
    conditions([
        system_elements([
            instance( Comp, compressor ),
            has_attribute( Comp, from, From ),
            has_attribute( Comp, to, To ) ]),
        parameters([
            pressure( To_Sub, Pressure_to, _, p ) ]),
        system_structures([
            closed_contained_substance( (From, _) ),
            closed_contained_substance( (To, To_Sub) ),
            thermostat_activates( _ ) ]) ]),
    givens([
        parameters([
            compressing_rate( Comp, Comp_rate, _, zp ) ]),
        par_values([
            value( Comp_rate, _, plus, zero ) ]),
        par_relations([
            inf_pos_by( Pressure_to, Comp_rate ) ]) ]) ).
```

Table 5.19: Agent model for the compressor

```
rule( termination, [from_equal_to_greater( Par1, Par2 )],
        condition([
            par_relations([
                equal( Par1, Par2 ) ]),
            par_values([
                value( Par1, Q1, Value1, plus ),
                value( Par2, Q2, Value2, min ) ]) ]),
        result([
            par_values([
                value( Par1, Q1, Value1, _ ),
                value( Par2, Q2, Value2, _ ) ]),
            par_relations([
                greater( Par1, Par2 ),
                d_greater_or_equal( Par1, zero ),
                d_smaller_or_equal( Par2, zero ) ]) ]) ).
```

Table 5.20: Termination rule for changing from equal to greater

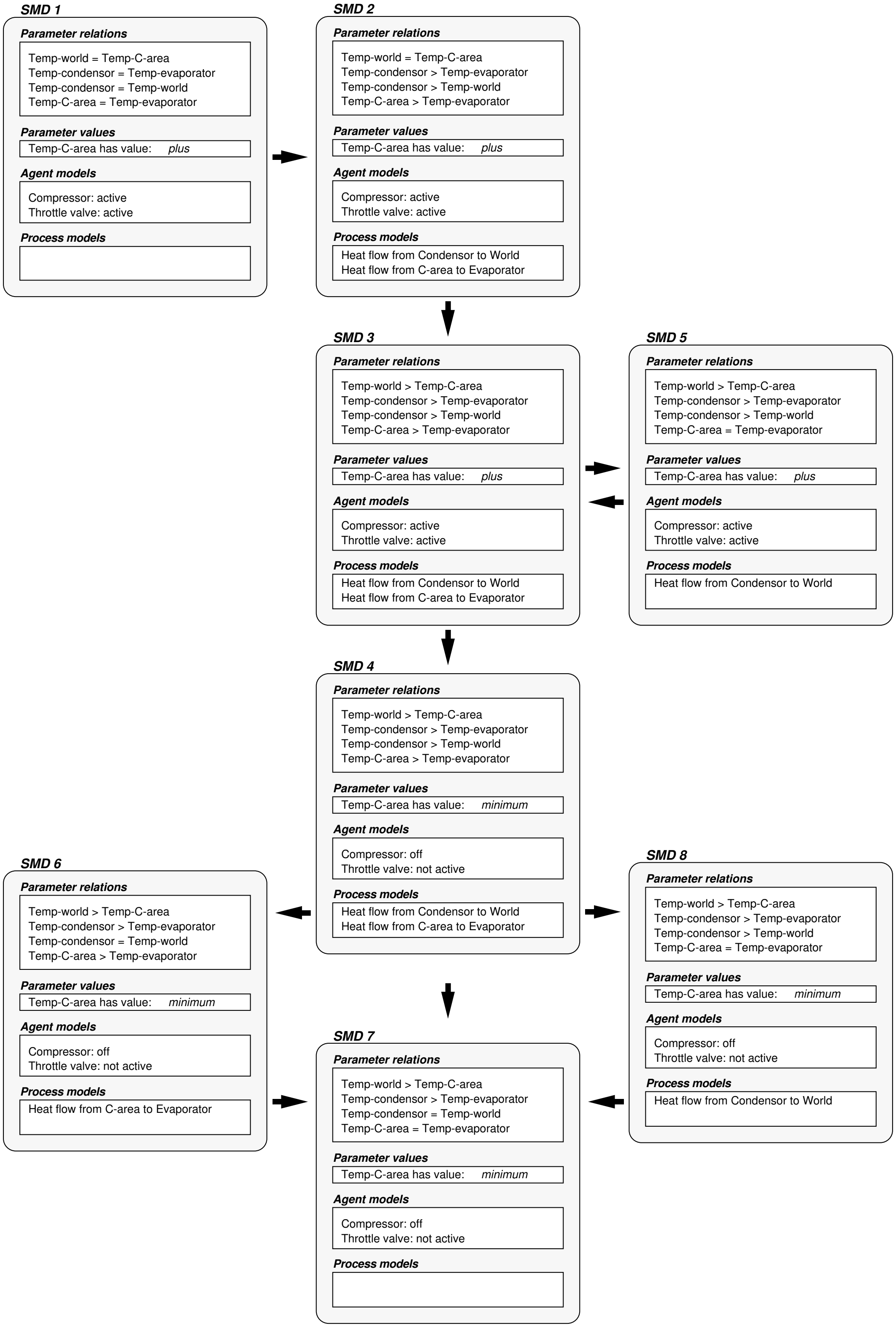

Figure 5.8: System model descriptions predicted by *GARP*

minimum value, which causes the activation of the compressor. The compressor increases the pressure in the condensor. This activates the throttle valve, which causes the pressure in the evaporator to decrease. The changes in the pressures have an effect on behaviour of the temperatures in each of the 'closed containers'. The temperature of the substance in the condensor increases and the temperature in the evaporator decreases. The different directions of change for these temperatures leads to three inequality changes (see figure 5.9). These three inequality changes are merged during the ordering of possible terminations, resulting in one inequality change to SMD2.

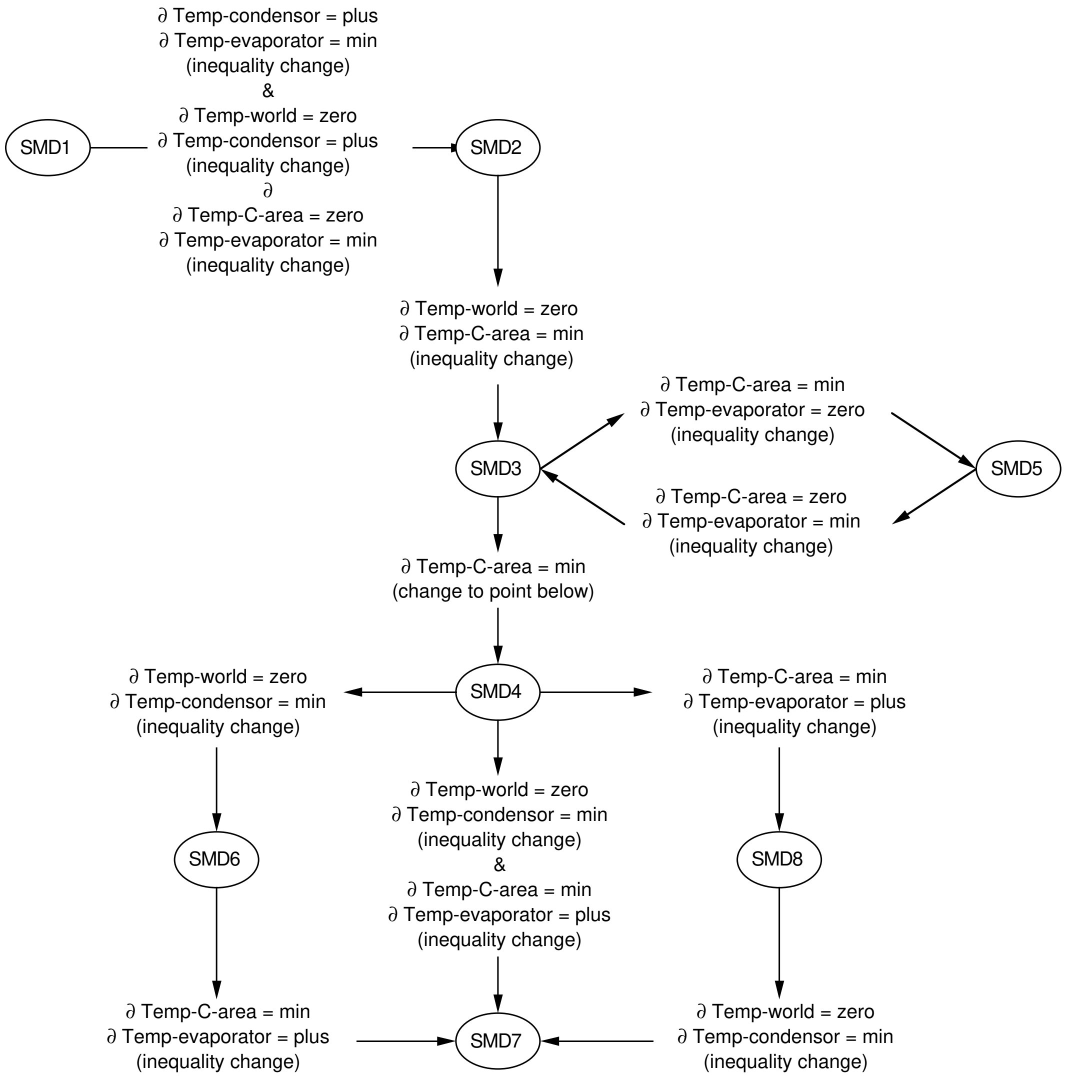

Figure 5.9: Causes of behaviour transitions

In SMD2 the temperature of the substance in the condensor is higher than the temperature of the world and the temperature of the substance in the evaporator is lower than the temperature of the cooling area. This results in two heat-flow processes, one from the condensor to the world and one from the cooling area to the evaporator. The behaviour of the world is modelled such that its temperature does not change, despite the heat-flow from the condensor. The temperature of the cooling area, on the other hand, starts to decrease, which leads to an inequality change between these two temperatures to SMD3.

In SMD3 the temperature of the cooling area is lower than the temperature of the world, but has not yet reached its minimum value. As a result all processes and agent models are still active. The heat-flow from the cooling area to the evaporator may cause the temperatures of these to objects to become equal. This inequality change leads to SMD5. In this state of behaviour the heat-flow between the cooling area and the evaporator is not active. SMD5 immediately changes back to SMD3, because the throttle decreases the temperature of the substance in the evaporator.[20]

Instead of changing to SMD5, SMD3 may change to SMD4. In SMD4 the value of the temperature of the cooling area has reached its minimum value. This value is sensed by the thermostat and causes the compressor to stop working. The activity of the throttle valve is directly related to the pressure increase of the compressor. No pressure increase causes the throttle valve to become inactive. The heat-flows still continue as a result of unequal temperatures. Three terminations are now possible, the temperature of the condensor becomes equal to the temperature of the world (SMD6), the temperature of the cooling area becomes equal to the temperature of the evaporator (SMD8), or both inequality changes happen simultaneously (SMD7). In SMD6 the heat-flow between the condensor and the world disappears and in SMD8 the heat-flow between the cooling area and the evaporator disappears. Eventually both SMD6 and SMD8 change to SMD7 because the remaining heat-flow causes the unequal temperatures to become equal.

SMD7 is the final state of behaviour in this behaviour description. There is no change possible from this SMD to any other SMD, because there is no heat-flow possible from the world into the cooling area. Introducing a heat-path between these two system elements, would facilitate such a heat-flow and lead to an increase in the temperature of the cooling area.

5.4.2 Model of Heart Diseases

This section describes a model of the heart diseases *angina pectoris* and *myocardial infarction* which has been developed and implemented in *GARP*. A myocardial infarction happens when the blood supply to the heart is blocked immediately. No blood supply results in a lack of oxygen, that manifests itself as pain and, most of the time, the dying of some muscle-tissue, causing a severe malfunctioning of the heart. Angina pectoris, on the other hand, is caused by a relative shortage of blood as a result of a decrease in the available flow area of the artery (see figure 5.10). Angina pectoris typically manifest itself when the heart muscle has high activity and as a result the amount of blood supplied to the heart is less than the amount of blood needed by the heart. Pain in combination with a high level of activity are therefore its symptoms.

5.4.2.1 System Elements

The isa-hierarchy of system elements is shown in figure 5.11. The *body* is modelled as a container that can supply an infinite amount of blood. The *artery* is modelled as a 'fluid

[20]The changes from SMD1, SMD2 and SMD5 to other SMD's are all immediate because they change from being equal at a point to being unequal in intervals.

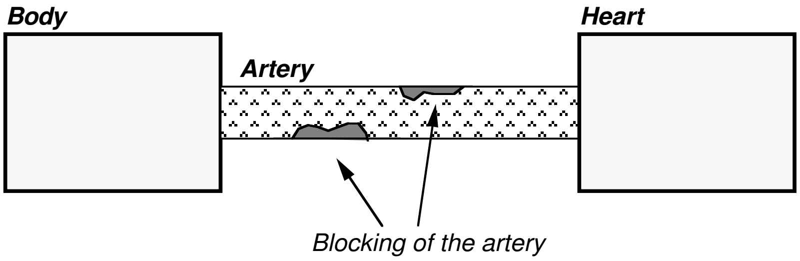

Figure 5.10: A model of heart diseases

path' that transports blood from the body into the heart. The *heart* is modelled as a muscle that, depending on its activity, needs an amount of blood. There are three components: a *fat increaser*, a *fluid path decreaser* and a *muscle activity increaser*. They are used in the model as actors that can influence specific features of the heart configuration (see also section 5.4.2.3).

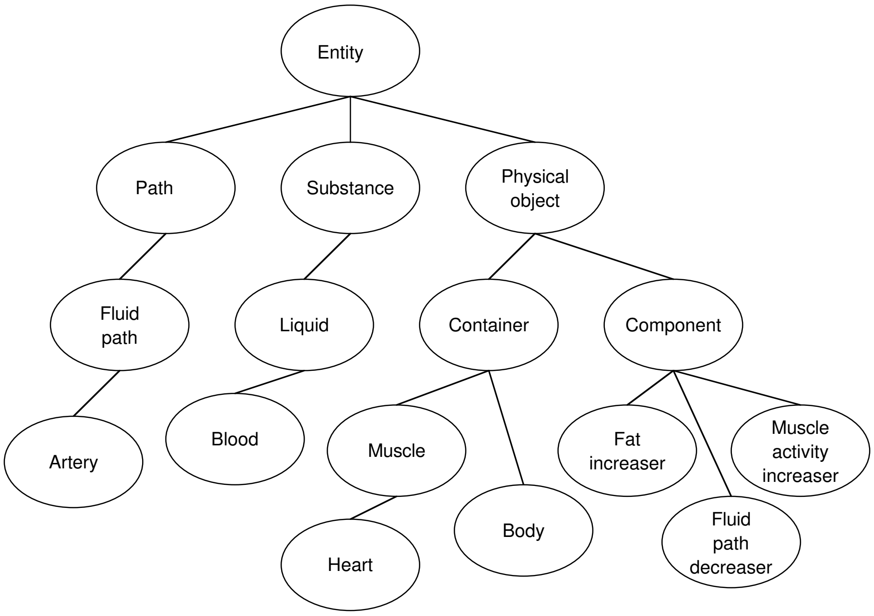

Figure 5.11: System element hierarchy for heart diseases

5.4.2.2 Input System, Parameters, Values and Relations

A part of the input system is used for instantiating the structural description of the heart configuration. Instances have to be created for the heart itself, the body, the blood in the heart and in the body, and the artery that connects the body with the heart. In addition, there are two components specified in the input system which will trigger agent models and as such enforce changes in the heart configuration. The 'fluid path decreaser' causes the flow area of the artery to decrease. The 'muscle activity increase' causes the activity of the heart to increase. Together these two components will cause malfunctioning of the heart.

```
smd( input_system( 'Heart with artery decr. and muscle activity incr.' ),
        system_elements([
            instance( Body, body ),
            instance( Heart, heart ),
            instance( Artery, artery ),
            has_attribute( Artery, from, Body ),
            has_attribute( Artery, to, Heart ),
            instance( Blood1, blood ),
            instance( Blood2, blood ),
            has_attribute( Body, contains, Blood1 ),
            has_attribute( Heart, contains, Blood2 ),
            instance( _, fluid_path_decreaser ),
            instance( _, muscle_activity_increaser ) ]),
        parameters([
            flow_area( Artery, Flow_Area, _, zlnh ),
            use_of_O2( Heart, Use_of_O2, _, zlnh ),
            amount( Blood1, Amount, _, zlnh ),
            amount_of_fat( Blood1, Fat, _, nh_max ),
            amount_of_lumps( Blood1, Lumps, _, nh_max ),
            pressure( Blood1, Press_Bl1, _, p ),
            pressure( Blood2, Press_Bl2, _, p ) ]),
        par_values([
            value( Flow_Area, _, normal( Flow_Area ), _ ),
            value( Use_of_O2, _, normal( Use_of_O2 ), _ ),
            value( Amount, _, normal( Amount ), _ ),
            value( Fat, _, normal( Fat ), _ ),
            value( Lumps, _, normal( Lumps ), _ ) ]),
        par_relations([
            greater( Press_Bl1, Press_Bl2 ) ]),
        system_structures([]) ).
```

Table 5.21: Input system for reasoning about the heart diseases

In order not to generate all possible states of behaviour belonging to this configuration of system elements, we can add additional information. In this input system we included a pressure difference between the blood of the heart and the blood of the body:

$$greater(Press\_Bl1, Press\_Bl2)$$

This causes a blood flow from the body to the heart (and not the other way around). Also some parameters have been given initial values. They all refer to the *normal* value that the parameter should have.

As mentioned before, only the parameters that we want to give values or additional constraints must be defined in the input system explicitly. All the other parameters are created by *GARP* during the reasoning process, as it finds the partial models that apply to the input system. In total *GARP* can use 18 parameters for predicting the behaviour of the model described here (see table 5.22). Each parameter models a specific property

Property of system element	Parameter name	Quantity space
Blood in the body	Amount	zlnh
	Amount of O2	zlnh
	Amount of fat	nh_max
	Amount of lumps	nh_max
	Pressure	p
Blood in the heart	Amount	zlnh
	Amount of O2	zlnh
	Amount of fat	nh_max
	Amount of lumps	nh_max
	Pressure	p
Artery	Flow area	zlnh
	Flow rate	zp
	Substance flow	zlnh
	Disposition rate	zp
Heart	Use of O2	zlnh
Fluid path decreaser	Decreasing rate	zp
Muscle activity increaser	Increasing rate	zp
Fat increaser	Fattening rate	zp
Body	Dummy	zp

Table 5.22: Parameters for reasoning about heart diseases

of the heart configuration that is relevant for the heart diseases we want to reason about:

- *Amount* refers to the amount of blood.[21].
- *Amount of O2* refers to the amount oxygen in the blood.
- *Amount of Fat* indicates the amount of fat in the blood
- *Amount of Lumps* the amount of 'solid' blood present in the blood.
- *Pressure* the pressure of the blood.
- *Flow area* the area in the artery available for blood flow.
- *Flow rate* the speed at which blood flows through the artery.
- *Substance flow* the flow of blood through the artery.
- *Disposition rate* the speed at which the artery becomes smaller.
- *Use of O2* the amount of oxygen used by a muscle (in this model the heart).
- *Decreasing rate* the speed at which the 'fluid path decreaser' narrows the flow area. This parameter differs from the disposition rate because it is introduced by a component (and not by a process).

[21] Contained by the heart or the body

- *Increasing rate* the speed at which the 'muscle activity increaser' increases the use of oxygen.

- *Fattening rate* the speed at which the 'fat increaser' increases the amount of fat in the blood.

- *Dummy* is used in the body to make sure that the amount of blood in the body stays constant.

To give the parameters a specific value, a number of quantity spaces must be defined. The most complicated quantity space distinguishes four values. Starting at point *zero* it moves on to an interval called *low*, a point called *normal* and an interval called *high*:

$$quantity\_space(zlnh, X, [point(zero), low, point(normal(X)), high]).$$

This quantity space typically represents the intuitive notion that some parameters, such as the *substance flow* through the artery, can have values zero, low, normal or high.[22] The second quantity space starts at *normal* then moves on to some positive interval called *high* and stops at a maximum called *max*:

$$quantity\_space(nh\_max, X, [point(normal(X)), high, point(max(X))]).$$

This quantity space is typical for parameters such as the *amount of fat* in the blood, namely the amount is normal, high, or has reached its maximum. The third quantity space starts at zero and has only one positive interval:

$$quantity\_space(zp, \_, [point(zero), plus]).$$

This quantity space is used for modelling the values of parameters that are either zero or plus, for example, the flow rate of the blood when it flows through the artery. Finally, there is a quantity space that has only one positive interval:

$$quantity\_space(p, \_, [plus]).$$

This is used for parameters that do not reach any significant values within this model, as for example the pressure.

5.4.2.3 Static Models

The hierarchy of partial behaviour models is depicted in figure 4.10. In this section we describe the most important models from that hierarchy in more detail. The partial model for *blood* describes the properties of blood in general (see table 5.23). It introduces four parameters: the amount of blood, the amount of oxygen in the blood, the amount of fat in the blood, and the amount of lumps in the blood. The four relations between the parameters amount of blood and amount of oxygen are used to synchronise the values and the derivates of these two parameters. They specify that the amount of oxygen in the blood is directly related to the amount of blood. The *greater* relation is used to define

```
system_structures( blood( Blood ),
    isa([ single_description_model ]),
    conditions([
        system_elements([
            instance( Blood, blood ) ]) ]),
    givens([
        parameters([
            amount( Blood, Amount_of_Blood, _, zlnh ),
            amount_of_O2( Blood, Amount_of_O2, _, zlnh ),
            amount_of_fat( Blood, _, _, nh_max ),
            amount_of_lumps( Blood, _, _, nh_max ) ]),
        par_relations([
            equal( normal( Amount_of_O2 ), normal( Amount_of_Blood ) ),
            q_correspondence( Amount_of_O2, Amount_of_Blood ),
            equal( Amount_of_O2, Amount_of_Blood ),
            prop_pos( Amount_of_O2, Amount_of_Blood ),
            greater( Amount_of_Blood, zero ) ]) ])).
```

Table 5.23: Partial behaviour model for blood

that the amount of blood is at least greater than zero. In other words: there is an amount of blood.

The quality of the blood is defined by three subtypes of the blood single description model namely: normal, clotted, and fattened blood. The latter is listed in table 5.24. Blood is classified as fattened when the amount of fat is greater than normal. Clotted

```
system_structures( fattened_blood( Blood ),
    isa([ blood( Blood ) ]),
    conditions([
        parameters([
            amount_of_fat( Blood, Amount_of_Fat, _, nh_max ) ]),
        par_relations([
            greater( Amount_of_Fat, normal( Amount_of_Fat ) ) ]) ]),
    givens([])).
```

Table 5.24: Partial behaviour model for fattened blood

blood implies that the amount of lumps in the blood is greater than normal. Blood is classified as normal when both the amount of fat and the amount of lumps are equal or

[22] Note that from a quantitative point of view normal is different for every human being, but from a qualitative point of view it is the same for everyone. Normal refers to 'normal' behaviour of a human being.

less than normal.

The artery is modelled as a liquid path (see table 5.25). The givens of the partial model for a fluid path defines three parameters: flow area, flow rate, and substance flow. In this model we assume that the flow rate is always positive and does not change. In other words, the flow rate through the artery is constant. As a result of this the substance flow through the artery is completely determined by the flow area. Notice that, there is no substance flow when there is no flow area and that each value of the flow area corresponds with a value of the substance flow. Only the value normal is equal for both quantity spaces.

```
system_structures( fluid_path( Fluid_path ),
    isa([ single_description_model ]),
    conditions([
        system_elements([
            instance( Fluid_path, fluid_path ) ]) ]),
    givens([
        parameters([
            flow_area( Fluid_path, Flow_area, _, zlnh ),
            flow_rate( Fluid_path, _, _, zp ),
            substance_flow( Fluid_path, Subst_flow, _, zlnh ) ]),
        par_relations([
            prop_pos( Subst_flow, Flow_area ),
            dir_v_correspondence( Subst_flow, zero, Flow_area, zero ),
            equal( normal( Subst_flow ), normal( Flow_area ) ),
            q_correspondence( Subst_flow, Flow_area ) ]) ])).
```

Table 5.25: Partial behaviour model for fluid path

There are six subtypes of the fluid path model. A fluid path is *closed* when the flow area is zero, *narrowed* when the area is low, *normal* when the area is normal, and *enlarged* when the flow area is high. A fluid path is *clogged* when the flow area is low and the amount of lumps has reached its maximum. It refers to the situation in which a specific lump blocks the small area available for liquid flow. Finally, a fluid path is *turbulent* when there is a liquid flow process going on in the fluid path and the flow area is low. Turbulence is a precondition for the increase in the amount of lumps in the blood.

The body is regarded as a container, containing blood, and capable of endless blood supply. The behaviour model for the muscle is a subtype of a contained liquid. The properties of a muscle are described by three relevant parameters: amount of oxygen, amount of blood, and the use of oxygen by the muscle. The use of oxygen has a negative influence on the amount of oxygen and the amount of blood in the muscle. The muscle has four states of activity. It can be a 'dead' muscle in which case the oxygen use is zero, it can be a low active muscle meaning that the use of oxygen is low, it can be a normal active muscle, meaning that the use of oxygen is normal, and it can be a high active muscle which means that the use of oxygen is high. The latter is listed in table 5.26

```
system_structures( high_active_muscle( Muscle ),
    isa([ muscle( Muscle ) ]),
    conditions([
        parameters([
            use_of_O2( Muscle, Use_of_O2, _, zlnh ) ]),
        par_values([
            value( Use_of_O2, _, high, _ ) ]) ]),
    givens([])).
```

Table 5.26: Partial behaviour model for muscle with high activity

In order to relate the parameter values of the body, the artery, and the heart, we use the notion of a *heart configuration*. In this composition model the quantity spaces of the parameters describing these system elements are related to each other. One of the important connections made here is:

$$q\_correspondence(Amount\_of\_Blood, Substance\_Flow)$$

This correspondence defines that the amount of blood in the heart is equal to the amount of blood flowing through the artery.

The heart itself is seen as a specific subtype of the muscle model. There are five models describing the state of the heart. They all depend on the activity of the muscle and the amount of oxygen available for that activity. Only two of these models reflect 'normal' states of behaviour, namely when the muscle is normal or high active and the amount of oxygen in the heart is sufficient for that type of muscle activity. The three other models all reflect pathophysiological states: pain in heart (short of oxygen and a low active muscle), angina pectoris (short of blood and a high active muscle) (see table 5.27), and heart infarct or myocardial infarction (short of blood due to a clogged artery).

5.4.2.4 Process and Agent Models

The liquid flow process facilitates a blood flow through the artery when there is a pressure difference and some area for blood to flow through (similar to the liquid flow process show in table 5.8).

The clotting of blood is an indirect process in the sense that liquid flow is conditional for a turbulent fluid path (see table 5.28). When the artery grows smaller the turbulence increases (turbulent fluid path). This turbulence results in an increase in the amount of lumps.

Depending on where we want to start a simulation, there will be a need for one or more agents that act upon specific system elements in the heart configuration. In this model we have defined three such agents, namely a 'fat increaser', a 'fluid path decreaser', and a 'muscle activity increaser'.[23] 'The fat increaser' can be used to increase the amount of

[23]The input system shown in table 5.21 specifies only the 'muscle activity increaser' and the 'fluid path decreaser'.

```
system_structures( angina_pectoris_heart( Heart ),
    isa([ muscle( Heart ) ]),
    conditions([
        parameters([
            amount_of_O2( Blood, Amount_of_O2, _, zlnh ),
            use_of_O2( Heart, Use_of_O2, _, zlnh ) ]),
        par_relations([
            greater( Use_of_O2, normal( Use_of_O2 ) ),
            greater( Amount_of_O2, zero ),
            greater( Use_of_O2, Amount_of_O2 ) ]),
        system_structures([
            heart_configuration( (_, _, Heart) ),
            contained_liquid( (Heart, Blood) ) ]) ]),
    givens([])).
```

Table 5.27: Partial behaviour model for angina pectoris

```
system_structures( clotting( Blood ),
    isa([ process_model ]),
    conditions([
        system_elements([
            has_attribute( Fluid_Path, from, Container ) ]),
        parameters([
            amount_of_lumps( Blood, Amount_of_Lumps, _, nh_max ) ]),
        par_relations([
            smaller( Amount_of_Lumps, max( Amount_of_Lumps ) ) ]),
        system_structures([
            turbulent_fluid_path( Fluid_Path ),
            contained_liquid( (Container, Blood ) ),
            blood_view( Blood ) ]) ]),
    givens([
        parameters([
            flow_rate( Fluid_Path, Flow_Rate, _, zp ) ]),
        par_relations([
            inf_pos_by( Amount_of_Lumps, Flow_Rate ) ]) ])).
```

Table 5.28: Partial behaviour model for clotting of blood

fat of the blood in the body. 'The muscle activity increaser' can be used to increase the oxygen use by the muscle. Finally, 'the fluid path decreaser' can be used to decrease the flow area of the artery (see table 5.29). Agent models are in fact shortcuts for a number of complex processes that we do not want to consider in detail. If for example we are not interested in what happens to the fattening rate (etc), and only want to simulate the effects of the fluid path growing smaller, then we can use the fluid path decreaser and ignore what happens to the amount of fat in the blood.

```
system_structures( fluid_path_decreaser( Decreaser ),
    isa( [ agent_model ] ),
    conditions([
        system_elements([
            instance( Decreaser, fluid_path_decreaser ) ]),
        parameters([
            flow_area( Fluid_path, Flow_area, _, zlnh ) ]),
        par_relations([
            greater( Flow_area, zero ) ]),
        system_structures([
            fluid_path_view( Fluid_path ) ]) ]),
    givens([
        parameters([
            decreasing_rate( Decreaser, Rate, _, zp ) ]),
        par_values([
            value( Rate, _, plus, zero ) ]),
        par_relations([
            inf_neg_by( Flow_area, Rate ) ]) ])).
```

Table 5.29: Partial behaviour model for 'fluid path decreaser'

5.4.2.5 The behaviour prediction

Given the input description and partial behaviour models described above, *GARP* predicts seven SMD's. The important aspects of this behaviour prediction are shown in figure 5.12. In SMD1 the heart is in normal condition, but because the 'fluid path decreaser' and the 'muscle activity increaser' are active the flow area decreases and the use of oxygen increases. The resulting terminations cannot be ordered and therefore lead to three new SMD's. In SMD2 the flow area is decreased (low) and therefore insufficient blood can flow to the heart. As a result the heart is state of pain. In SMD4 the use of oxygen is higher than normal. The artery provides insufficient blood, because of the 'fluid path decreaser'. As a result, the heart is in state of angina pectoris. SMD3 includes both the changes that lead to SMD2 and SMD4. The muscle activity is high and the artery provides insufficient blood, leading to an angina pectoris state for the heart. There three different ways to get to SMD5: via SMD2, via SMD3, or via SMD6. In SMD5 the area for liquid is low, the use of oxygen high, and the amount of available oxygen low. Moreover the blood starts

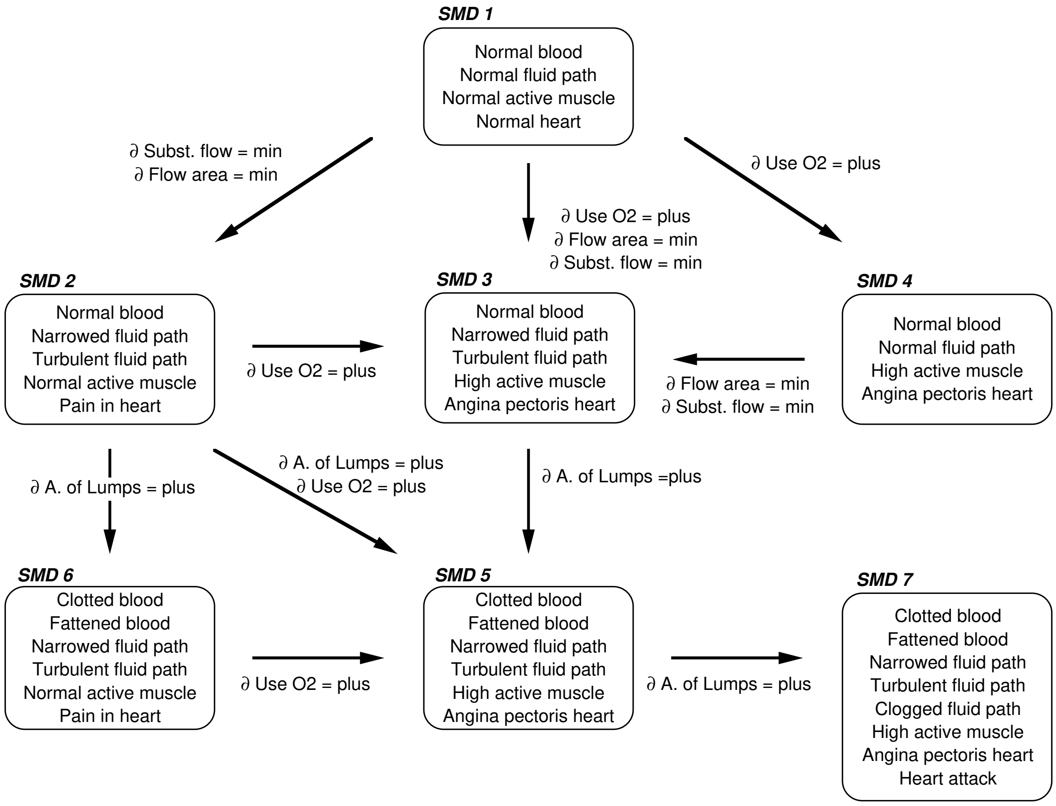

Figure 5.12: System model descriptions predicted by *GARP*

clotting, which may lead to a heart attack. The latter is the transition to SMD7.

5.5 Concluding Remarks

The design model provides a detailed description of an implemented system (*GARP*) that performs state of the art qualitative reasoning. *GARP* cannot only simulate the reasoning processes from the traditional approaches, but implements a new approach to qualitative reasoning that is more general than its predecessors.

Two basic constraints guided the design process:

- the design model must reflect a structure preserving transformation from the model of expertise, i.e. a *type oriented mapping*, and
- the artifact must be easy accessible for prototyping prediction models.

The type oriented mapping is particular relevant for knowledge level reflection, because the type-to-type relation between the design model and the model of expertise provide the basis for accessing the different types of knowledge described in the model of expertise and implemented in *GARP* (see also chapter 7). The usability of *GARP* for prototyping makes it better suited to be used as an operationalised interpretation model that is part of a model driven methodology for knowledge based systems development.

Both constraints had a direct effect on the functional decomposition of the conceptual model. The type oriented mapping required that the design elements had a rich semantics, capable of representing the different knowledge items from the conceptual model as closely as possible. The accessibility constraint resulted in additional functions imposed upon the final artifact. In particular, the functions: *modification* of the domain knowledge, *visualisation* of different aspects of the reasoning process, and the *assessment* of solvability, resulted from this constraint.

Some important properties of the algorithms used for realising the problem solving potential of the knowledge sources can be pointed out. Firstly, the algorithm for specification implements an efficient depth-first search, that not only tests whether conditions can be inferred from the knowledge present in the system model description (SMD), but that in addition uses an assumption mechanism for finding all partial models that are consistent with that knowledge.

Secondly, the computation of parameter relations, which is part of the specification, realises an advanced method for coping with problems related to transitivity reasoning. The qualitative calculus, as originally proposed in [57; 93], exhibits spurious behaviour, especially with regard to conservation of quantities such as energy, flow, and force. We reduced the generation of spurious behaviours by applying reasoning about inequalities and arithmetic summations. Our method combines the axioms for reasoning about transitivity and arithmetic summations and thereby avoids the problems associated with the approaches of [120; 73]. Furthermore, it enables the specification of quantity spaces containing any number of values.

Thirdly, in *GARP* the control on the algorithms is organised such that all the states of behaviour are found that apply to, or follow from, a certain input system. When the input system specifies only a configuration of system elements, this leads to generating all possible states of behaviour (total envisionment). When additional parameter values and parameter relations are added to this input system, then the behaviour prediction will be more directed, resulting in a more specific trace of behaviour (attainable envisionment).

The control on the algorithms is also accessible by the user. The user can decide to give control to the prediction engine (resulting in full autonomous execution of the task as described above), or the user can decide to control the prediction by him- or herself and manipulate the prediction engine such that a required trace of behaviour is derived.