Deliverable number:	D3.4
Deliverable title:	Diagnosis and Assessment

Delivery date:	2011/07/31 (extension 2011/09/30)
Submission date:	2011/10/14
Leading beneficiary:	University of Amsterdam (UVA)
Status:	Version 05 (final)
Dissemination level:	PU (public)
Authors:	Wouter Beek and Bert Bredeweg

Project number:	231526
Project acronym:	DynaLearn
Project title:	DynaLearn - Engaging and informed tools for learning conceptual system knowledge
Starting date:	February 1st, 2009
Duration:	36 Months
Call identifier:	FP7-ICT-2007-3
Funding scheme:	Collaborative project (STREP)

Abstract

In this deliverable we will explain the use of consistency-based diagnosis for providing automated assistance in knowledge modeling. We show how the attested technique of consistency-based diagnosis can be used to support active learning. Since QR knowledge construction is an open-ended domain, applying diagnosis in this way poses several problems. We will solve several of those problems by representation or algorithms.

Internal review

David Mioduser, Science and Technology Education Center, Tel Aviv University.

Esther Lozano, Ontology Engineering Group, Universidad Politécnica de Madrid.

Ackmnowledgements

We would like to thank the reviewers for sharpening our minds with their thoughtful comments. We would also like to thank Sander Latour, on whose work the sections on automated testing and automated repair have been loosely based.

Document history

Version	Modification(s)	Date	Author(s)
1	Outline + first text	2011-09-01	Beek, Bredeweg
2	Chapters 3, 4, 5	2011-09-21	Beek
3	Chapters 1 and 2	2011-09-22	Bredeweg
4	First draft for internal review	2011-10-01	Beek
5	Final version after internal review	2011-10-17	Beek, Bredeweg

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Articulating thought in media is a powerful means for humans to develop their understanding of phenomena. Computer-based tools are used to further enhance this process by implementing dedicated media and accompanying interactive mechanisms. A new class of tools is emerging that is logic-based (symbolic, non-numerical) and dedicated to the construction of conceptual models. The focus is on education, and having learners construct models to learn. Particularly, to develop explanations of why systems behave as they do.

Computers are a unique class of cognitive tools because of their interactive nature; they can gather information, reason with it, and communicate to different users in forms that address their specific needs [34]. Particularly, computer-based modeling is considered fundamental to human cognition and scientific inquiry [39, 36]. Modeling helps humans to express and externalize their thinking; visualize and test components of their theories; and make materials more interesting. Modeling environments can thus make a significant contribution to improve scientific understanding.

In these new knowledge construction tools, learners can create a conceptual model of a dynamic system and can simulate the system's behavior over time. Learners can run into situations in which the simulation results are different from what they expected. The simulation is an instrument that helps learners reflect on their knowledge as expressed in their model, and work towards improving it. While constructing 'knowledge in the outside world' (using a representation), learners also 'construct knowledge in their inner world' (in their mind).

In principle, there are two ways to align simulation results with expectations thereof: learners can change their model, or they can change their expectations. Both are relevant for learning, and the two almost always interact. However, a problem occurs when learners want to adjust their model such that the simulation results align with their expectations, but are unable to do so. They get stuck in a model debug-loop without solving the problem. Reasons include having insufficient domain, representation, and simulation knowledge. Other factors are their inability to handle the complexity of the modeling task, and the difficulty of maintaining a consistent reasoning trajectory. We want to address this problem by developing a software tool that engages learners into a Knowledge Construction Dialog, which coaches learners to reason in a coherent manner, and meanwhile supports them in debugging their model such that the simulation results align with their expectations. For this we will employ Consistency-Based Diagnosis [14, 38], which was originally developed for diagnosing electrical circuits.

The goal of Technical Diagnosis is to single out the components that behave faulty and cause a device to behave inappropriately. Once found, the device is repaired by replacing these components by ones that behave correctly. Consistency-based reasoning is an important model-based technique for diagnosing faulty components in devices. The Gen-

eral Diagnostic Engine (GDE) [14] is an implementation of this approach that has been studied to a great extent, with successful applications, e.g. [39]. An important feature of Consistency-Based Diagnosis is that it does not require any knowledge of how faulty components misbehave.

Technical Diagnosis was adjusted for Cognitive Diagnosis [40, 21, 19] (Figure 1.1), the process of inferring a person's cognitive state from his or her performance [35]. In particular, Cognitive Diagnosis can be used to find the faulty inference steps in the reasoning of a learner. In Technical Diagnosis a norm model drives the diagnostic algorithm. Similarly, in Cognitive Diagnosis learners interact with an existing model (created by experts or teachers) and faulty answers are diagnosed using this model as the norm. While developing the approach special care was taking to ensure didactic plausibility of the components involved [20, 5].

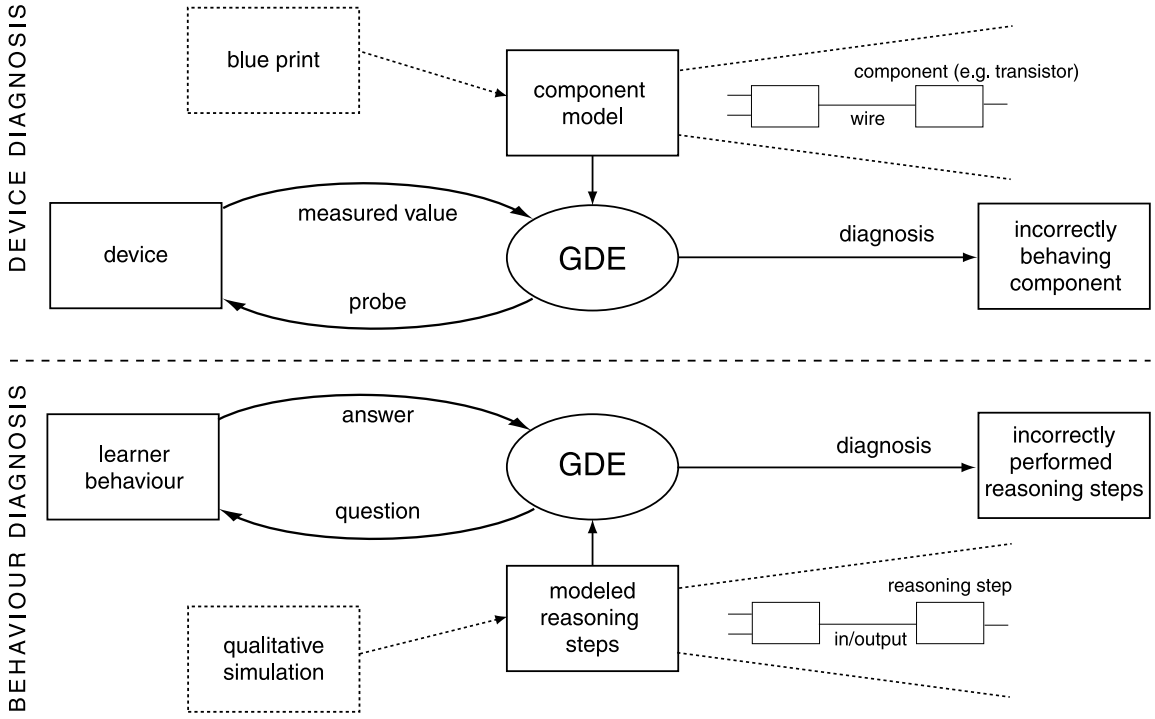

Figure 1.1: Mapping Consistency-Based Diagnosis as used for Technical Diagnosis onto Cognitive Diagnosis (form [21]).

The approach discussed above was implemented as a software tool and proven applicable. However, interacting with an existing (norm) model does not line up well with contemporary theories on 'active learning' (originating from constructivist perspectives on learning, e.g. [9, 8, 43, 37], in which the learners themselves should be creating knowledge and meaning using tools.

With the arrival of knowledge construction tools, active learning can be facilitated, e.g. [22, 27]. But, the question now is: how to automatically generate feedback and modeling support on a model and its simulation results, without having a norm? This is a serious challenge, and the problem we try to address here.

The central research goal is to develop an algorithm that takes discrepancies between the actual and the user-expectation simulation results of a model and identifies the model ingredients accountable for these differences. The following preceding research is taken as input: The qualitative vocabulary for creating and simulating conceptual models as defined in [8]. The 'base model' generation algorithm defined by [21] to automatically transform the simulation results into a Component-Connection Model (CCM) that matches the General

Diagnostic Engine (GDE) as developed for Technical Diagnosis [14], including the underlying Assumption-based Truth Maintenance System [16, 17, 18]. GDE will form the kernel of the diagnostic algorithm.

1.1 Overview

The document is structured as follows. In chapter 2 we discuss our adaptation of existing research performed by [19, 21] to provide modeling assistance though model-based diagnosis. This chapter discusses the transformations necessary when moving from a technical to a cognitive domain, as well as the educational appropriation of our technique.

Chapter 3 discusses how we create the basic representation that we use for diagnosis. This is the Component Connection Model (CCM) that represents the simulation results and uses cognitive components to describe and perform the simulation's reasoning steps. To go beyond the basic representation, we introduce aggregations or higher-level components in chapter 4. These more abstract components are used for pedagogical purposes (shorter interactions and more dedicated feedback), as well as for complexity reasons (reducing calculation costs).

In chapter 5 we detail our implementation of the diagnosis algorithm, including the many additions that allow it to operate in the new setting (explained in chapter 2) in an effective way, thereby making heavy use of the dedicated representations from chapters 3 and 4.

Chapter 6 enumerates research issues and extensions that we have come across while creating modeling assistance based on model-bases diagnosis. We indicate how we think our approach allows these issues to be treated and those extensions to be utilized.

Chapter 2

Theoretical background

Individualized interaction between a learner and a computer-based learning environment seems to require a fundamental solution to the problem of student modeling. But the latter is an outstanding problem in research on Artificial Intelligence and Education (AIED) that remains unsolved because it requires a thorough understanding of how humans learn and solve problems. An individualized interaction thus seems to be a goal that is far beyond the current state of possibilities. Our approach takes an alternative route to establish individualized interaction in a learning environment.

Our approach relates to modern views on communicative interaction [24], in that it considers it fundamentally impossible for a person (an agent) to truly 'know' the kind and amount of knowledge that is embodied by the agent who one interacts with. At the most, one can observe that given certain 'inputs' (e.g. a problem solving exercise) the 'outputs' (intermediate and final solutions) of the other agent are consistent (or inconsistent) with the 'outputs' of one's own thought processes. Moreover, further communication often seems to focus on the discrepancies, particularly in educational settings, until a seemingly consistent set of beliefs is shared by the agents [20].

The notion of consistency-based reasoning has been researched to a great extend in the context of technical diagnosis. It seems only logical to investigate the potential of that approach for cognitive diagnosis as was done by [40]. Following this idea [21] developed an approach in which qualitative simulation and consistency-based diagnosis are employed to represent and analyze the reasoning behavior of a learner when predicting the behavior of a device. We build on that work. Particularly, by showing how the consistency-based approach is central to a communicative interaction between learners and an Interactive Learning Environment, particularly DynaLearn.

2.0.1 Problems with the traditional notion of student models

The idea of a student model has been an important aspect of research on computer-based coaching and teaching systems for a long time [35, 44, 40, 29]. The ultimate goal in this work is a running model, i.e. a piece of executable computer software, which models the cognitive state of a learner. Typically a student model should capture the learner's knowledge state, including a detailed representation of his or her level of expertise (correct knowledge), misconceptions (incorrect knowledge) and knowledge gaps (missing knowledge). The assumption behind the use of a student model is that based on this model tutoring actions can be optimized to address the individual needs of a learner. Providing individualized help or explanation, selecting the best next exercise from the curriculum right on the edge of the learner's capabilities, generating the perfect counter-example to induce insight, and so forth. However, in contrast to what one would expect, there are not so many examples in

the literature on AIED that actually support this assumption. In fact, the reverse seems to be the case. There is growing consensus among AIED researchers that the assumption may not hold. Not because the tutoring actions cannot be handled, but because the notion of a running student model is not straightforward. But before jumping to such conclusions let us examine the idea of a student model in more detail.

One of the early student model types is the overlay model, e.g. [10, 11]. Here the hypothesis is that the knowledge of the learner is a subset of the knowledge possessed by the expert or teacher. This approach works well in situations where the knowledge concerning the subject matter can be transferred 'directly' to the learner. This is true for teaching factual knowledge such as geography or English vocabulary, and for a limited set of procedural domains (e.g., training fixed safety-critical procedures in nuclear power plants). Building the student model requires an index that records which knowledge items possessed by the expert are understood by the learner and which items still have to be learned. Ignoring the issue of forgetting, the overlay approach still has two main problems. In many studies it has been pointed out that the problem solving expertise of novices is significantly different from that of experts, both in storage and how it is applied to solving problems (e.g. [23]). Moreover, the distinction is not always that straightforward. There are different levels of being a novice, some significantly more knowledgeable then others. And even the notion of an expert is ill defined. Acquiring knowledge seems to be a gradual process, during which multiple levels of intermediate expertise exist (e.g. [2]). In fact, it is even debatable whether levels of expertise can be discriminated and organized into discrete categories. But whatever the outcome, an overlay model will be insufficient, because it is unable to record anything in between being a novice and being an expert. This also relates to the second problem, namely that an overlay only stores correct (or missing) knowledge. This ignores the fact the learners make errors, because they may have mis-learned something. The incorrect knowledge learners may have, cannot be stored in an overlay model.

Perturbation models have been proposed to overcome the limitations of the overlay model [30]. This approach views the learner's knowledge not as a subset of the expert's knowledge, but as intersecting with it: part of the learner's knowledge will be correct (i.e. identical to the knowledge of the expert), and part of it will be different. This 'difference' is usually referred to as misconceptions or bugs. Hence, the learner model consists of an overlay of the subject matter, possibly extended with a number of buggy facts or procedures (a bug catalog). One of the key ideas behind the perturbation approach is that these extensions explain the learner's erroneous problem solving behavior. But the enumeration of bugs is not for free. Constructing a catalog of bugs is an extremely laborious task that requires thorough study of learners making errors while solving problems. Moreover, a bug catalog is domain-dependent and has to be reconstructed for each new domain. Finally, it is difficult to prove that all bugs have been captured in a catalog, even for a specific domain. In fact, a bug catalog is probably never complete.

Problems with the bug catalog have led to the approach of generative theories of bugs. Instead of enumerating the bugs by carefully analyzing learner's behavior, the idea is to automatically generate the bugs by pruning the correct knowledge. The REPAIR theory [7] is probably one of the most well known examples of this approach. Generative approaches are based on cognitive theories that explain how misconceptions arise. They present a cognitive architecture of how humans learn which explains the sub-optimal problem solving behavior, such as making errors. In the case of the REPAIR theory the main idea was 'impasse-driven problem solving': an impasse occurs when no further relevant knowledge is available or when conflicting knowledge is applied, resulting in the learner being unable to proceed. Learners will now search for repairs to fill this knowledge gap. Applying the repair to an impasse may lead to a bug. Requiring a cognitive theory is both the beauty and the problem with

the generative approach (and in fact of the student model idea as a whole). Constructing a proper student model, i.e. fully capturing the knowledge state of a learner, indeed requires a mature theory of how humans learn and solve problems. But such a (computationally viable) theory does not exist, and it is not very likely that it will be established soon. See e.g. TIP [31] for an illustrative overview of alternative and competing views on human cognition. We conclude that, for the time being, a encompassing and practical use of the student model approach in a computer-based learning environment is not feasible. For a practical approach to individualizing the interaction with a learner, we have to search for alternative solutions.

2.0.2 Consistency-based diagnosis

A typical goal in technical diagnosis is to single out the components that behave faulty and cause the device as a whole to behave inappropriately. Once found, a device is repaired by replacing these components by ones that behave correctly. Consistency-based reasoning is an important model-based technique for diagnosing faulty components in devices. The General Diagnostic Engine (GDE) [14] is an implementation of this approach that has been studied to a great extent. This approach has also been successful in terms of practical applications, particularly in the automotive industry (on-board diagnosis in cars). In an interesting 'cross-over' paper, Self investigated whether this approach to technical diagnosis could be used as an approach to student modeling [40]. But before discussing his findings, let us take a closer look at the characteristics of consistency-based diagnosis.

Technical diagnosis

Figure 2.1 shows an example of a digital circuit, consisting of 3 multipliers and 2 adders, that behaves faulty. Instead of 12 the output of A_1 measures 10. An inconsistent observation between what is expected and what is observed is called a discrepancy. The expected value is predicted using a model that simulates the correct behavior of the device. The observed value is the outcome that is measured on the real device. A discrepancy tells us that there is a mismatch between how we expect a device to behave from how it actually behaves. The goal of the diagnosis is to find the components (one or more) that 'cause' this discrepancy to occur.

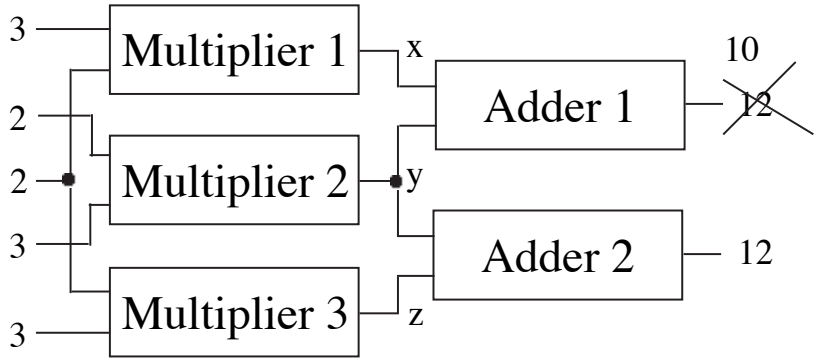

Figure 2.1: Malfunctioning multipliers and adders, a typical GDE example (from [14]).

Although there are many subtle issues involved, the basic consistency-based algorithm is rather straightforward. Following the measurement the diagnostic algorithm is applied to find sets of components that lead to inconsistencies. These sets are called conflict sets. In the example $\{A_1, M_1, M_2\}$ and $\{A_1, M_1, A_2, M_3\}$ are such conflict sets. Generating conflicts starts by taking the observed deviating value (10 at the output of A_1) and working backwards

to the inputs of the device as a whole. If A_1 produces 10, and is working correctly, then by definition it can have a limited set of inputs ($1 + 9$, $2 + 8$, etc.) Assuming that M_1 is also working correctly, then one of the inputs of A_1 has to be 6 (at x), which leaves 4 on the other input of A_1 (at y). Now if we assume that M_2 is also working correctly, it has to produce 4 (at y). But M_2 has 2 and 3 as inputs, so it produces 6 (at y). Here we have an inconsistency, i.e. the assumption that these three components all behave correctly cannot be true. At least one of these components (or more) must be broken. That is why $\{A_1, M_1, M_2\}$ is a conflict set. If we proceed with the search for conflicts, the next thing is to ignore M_2. Instead we assume that A_2 is also working correctly. One of the inputs for this component is already known (4 at y) and also the output is known (i.e. 12), which implies that A_2 must have had 8 as the second input (at z). If it is assumed that M_3 is also working correctly, another conflict occurs. M_3 has 3 and 2 as inputs and hence will produce 6 (at z). That is why $\{A_1, M_1, A_2, M_3\}$ is a conflict set as well. The assumption that all these components behave correctly cannot be true. At least one of them must be faulty. And so forth. Having found all the conflicts, the next step is to find candidates: all sets of components that have nonempty intersections with all conflicts.

Candidates are hypotheses as to why the device as a whole is malfunctioning. Each candidate is a possible explanation for the discrepancies, given the observations that were given to the diagnostic algorithm. In the example $\{A_1\}$ and $\{M_1\}$ are single fault candidates, they occur in all conflict sets. $\{A_2, M_2\}$ is a multiple fault candidate, each conflict contains either A_2 or M_2.

When there are multiple candidates, as is the case in the example, the diagnosis needs additional knowledge in the form of new measurements in order to discriminate between the alternative hypotheses. For example, a new measurement could be conducted at x. If we measure 4 there, then M_1 is the culprit: the faulty component that explains the erroneous behavior of the device as a whole. If on the other hand the measurement shows 6 (at X), then A_1 may be the culprit, but also $\{A_2, M_2\}$ is still an option.

In real applications, multiple cycles of measuring, finding conflicts and generating candidates are needed to find a diagnosis. It has been shown that GDE can easily diagnose large circuits within reasonable time limits [15].

2.0.3 Using fault models

Many modifications and alternative implementations of the consistency-based approach have been developed (for an overview see e.g. [3]). The problem solving engines GDE+ [42] and SHERLOCK [15] implement diagnostic algorithms that use the notion of fault models for diagnosing device behavior. The idea behind this approach is the observation that in a conflict set often components appear that, logically speaking, do explain the faulty behavior of a device, but which in reality can never exist.

The famous example of three light bulbs connected to a power source illustrates this curiosity (figure 2.2). If we include multiple faults, which was one of the main ideas for developing the GDE approach in the first place, then the faulty behavior of the light bulb L_2 (no light) can be accounted for by the battery being broken (does not produce power) combined with the assumption that the other two light bulbs also behave faulty (producing light in the absence of power). Of course our knowledge of electricity tells us that such faults cannot occur. Or better, we know the faulty behavior that a broken light bulb produces, namely: no light at all. Consequently, knowledge of faulty component behavior can be used to reduce the set of possible conflicts.

A few issues can be pointed out with respect to the use of fault models. First, notice that the use of fault models, in the context of technical diagnosis, is not a necessity for finding

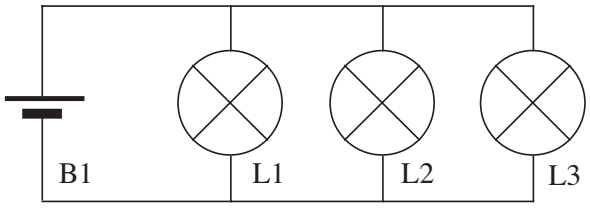

Figure 2.2: Circuit with battery and light bulbs, a typical GDE+ example.

the diagnosis that 'explains' the malfunctioning of a device. Conflict sets that assume behavior that in reality is impossible, will lead to (in reality) impossible candidates and further measurements will remove them from these lists. The main reason for using fault models for technical diagnosis concerns efficiency. But there is a drawback to this argument. One of those arguments is that the typical fault behaviors of a component have to be enumerated before they can be used. This is a difficult and time-consuming task that shows a remarkable resemblance to the problems encountered with the perturbation approach used for the construction of a student model. In technical diagnosis the problem is (1) to determine which components exist and may manifest faulty behavior, and (2) to determine the kinds of faulty behavior for each of these components. The problem is particularly difficult if bridge-faults (short-circuits) and broken wires are also included.

Analogous to technical diagnosis, in cognitive diagnosis (i.e. student modeling) the problem is (1) to determine the set of primitive inferences that a learner uses, and (2) to determine the buggy versions for each of these inferences. Also in the area of technical diagnosis ideas have been presented to automatically generate knowledge about faults on the basis of a model of the correct behavior of a device and its components.

The use of fault models is usually not regarded as a form of consistency-based reasoning, but as abduction. The distinction between these two notions has been the subject of much debate [12]. For the discussion in this article it is important to understand that the procedure for finding the faulty component differs between abduction and consistency. In the case of abduction the fault model is used to infer the 'faulty' output given a 'correct' input. When application of the inference rules, captured in the fault model, actually leads to the observed incorrect output, it is assumed that the fault model 'explains' how the component misbehaves and thus explains the faulty behavior of the device. In the consistency-based approach the incorrect behavior is not derived. Instead, the derived correct behavior is proven to be inconsistent with the observations. Using fault models in a consistency-based approach requires an adaptation of the algorithm in order to include the abduction part.

2.1 Towards consistency-based cognitive diagnosis

As mentioned before, the similarities between student modeling and diagnosis of device behavior are remarkable, both in terms of problems and in terms of proposed solutions. So it seems only logical that Self successfully applied the approach for technical diagnosis to determine the 'cognitive behavior' of learners solving subtraction problems, and thus re-implemented the approach taken with BUGGY, IDEBUGGY and related systems [40].

Self did not use a truly consistency-based approach, but an extension that allowed for including fault models. Despite the successful cross-over made between two seemingly very unrelated areas of Artificial Intelligence research, Self emphasized four problems that he thought were difficult to solve, if not impossible, making practical applications of cognitive diagnosis problematic.

The first problem concerns 'the purpose of diagnosis'. For devices, the purpose of diag-

nosis is to enable faulty components to be mended or replaced. In education, the diagnostic engine should serve didactic goals and hence its purpose is less evident than that of device diagnosis.

The second problem is related to the first and concerns 'the role of diagnosis in education'. The comparison of educational diagnosis with model-based device diagnosis "unfortunately reinforces the outdated view that cognitive diagnosis should focus on knowledge deficiencies. [...] Current ideas about tutoring systems design are moving away from knowledgeable systems re-mediating ignorant students towards environments that support students in managing their learning activities".

The third problem concerns 'defining device models', that is, a model of the device behavior. "We cannot assume that we as diagnosticians, know what 'circuits' are or should be in the head of students. [...] In general, it is hard to see how this problem can be addressed without bringing in almost all the issues of knowledge representation in AI and cognitive science".

The fourth problem concerns 'stability of the diagnostic model'. Learners may self-repair their misconceptions, leaving the tutoring system working on a diagnostic problem that has already become outdated before it has been solved (by the diagnostic software).

All in all, these problems seem rather devastating and one should think twice before even considering to work on educational diagnosis. However, below we will point out that this is not a correct understanding of the problem situation, the solution and its potential. Self revealed a new and very interesting approach for tracking a learner's problem solving behavior, but he killed it off using an outdated paradigm, namely, a rather traditional and outdated view on student modeling. A truly consistency-based approach on the other hand is very much in line with modern ideas on learning.

2.2 Communicative interaction

The term 'communicative interaction' addresses agents. Agents have internal states and intentions [24]. Communicative interaction is triggered by intentions (of at least one of the agents) and is effective if it leads to changes in the internal states of (at least one of) the agents involved. To organize their interaction agents can use 'user information', 'knowledge' and 'assets'. The approach is illustrated in figure 2.3.

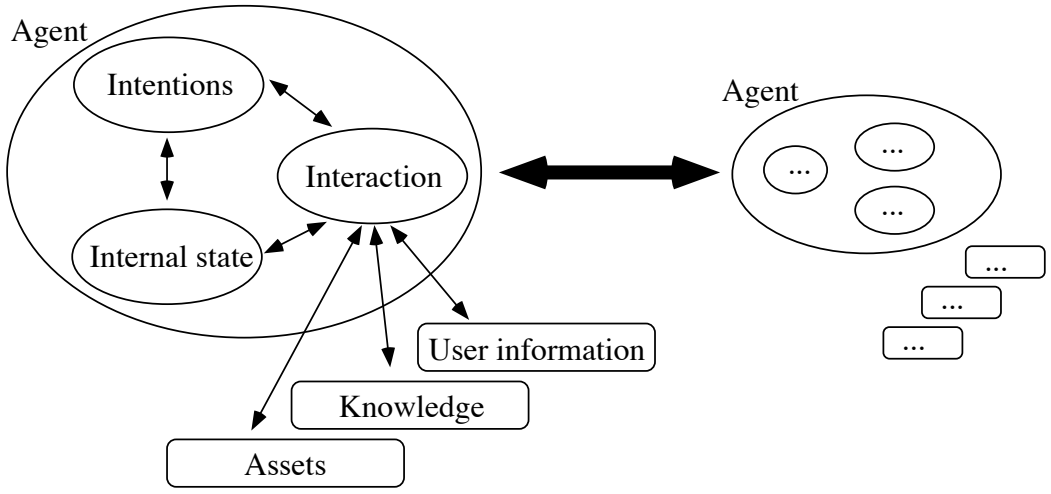

Figure 2.3: Communicative interaction (adapted from [24]).

From an educational perspective, the agents may be a teacher and a learner. For the

former, 'user information' would refer to the 'student model' the teacher has of the learner, 'knowledge' would (among others) refer to the subject matter the teacher wants to teach, and 'assets' would refer to the alternative ways a particular 'message' can be sent to the learner (spoken text, written text, drawing, and so forth).

Following typical models of communication [25], the idea is that agents communicate using channels. Channels relate to modalities. For instance, the channels 'spoken text', 'music', and 'noise' all use the modality 'auditory', which is different from e.g. the modality 'visual' which can be used for channels such as 'written text' and 'pictures'. Putting a message (or transmission) on a certain channel implies the use of assets. Assets relate to languages, which specify the 'lexicon', 'syntax' and 'semantics' available for the transmission of a message (see also figure 2.4).

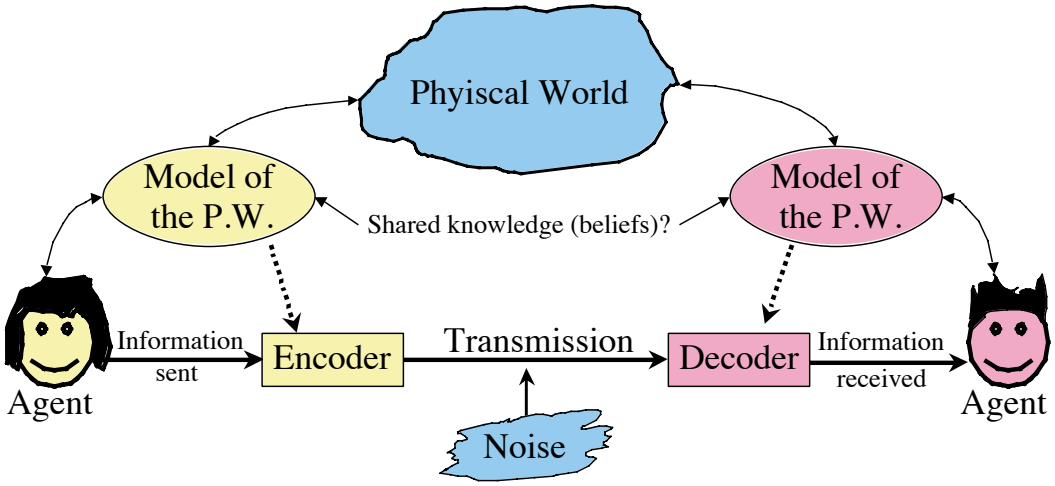

Figure 2.4: Encoding and decoding information.

The overall issue of communication is a complex one and not all aspects can be discussed here. However, one important insight is that communication is not perfect. It may be hampered by 'noise', and moreover, the content of a communicative interaction has to be created (encoded) and interpreted (decoded). The latter is where communicative interaction differs from classic information theory, as it was popular in the 1950's and early 1960's (e.g. [41]). An example of the classic view might be encoding something in Morse code for transmission over a radio frequency. It is assumed that both agents have access to an encoder and a decoder that is effectively perfect. Although the agents may make errors, there is no interpretation going on. Both agents will encode and decode a particular message in Morse code in exactly the same way.

This is not the same for communicative interaction. The encoding and decoding depends on the 'knowledge' an agent has. This knowledge is created while interacting with the 'physical world' and interacting with other agents and, as a result, is by definition different between two agents. As a consequence the encoders and decoders are unique for each agent. Communication is therefore never perfect, but depends on the interpretations of the agents involved. Of course, when agents interact a lot, and work towards consensus, their knowledge may, at least for some topic, become more similar. It is then said that they have a set of 'shared beliefs', which embodies the 'socially defined platonic knowledge' in a particular area [24]. Experts (or teachers) can be said to have such shared beliefs. Learners on the other hand, have to acquire those beliefs if they want to become knowledgeable in an area.

An important consequence of the theory discussed above is that one agent can never fully 'know' what goes on in the head of another agent. Or for that matter, a teacher can never fully 'know' the set of beliefs a particular student holds. So why did we think in

the first place that a computer program, a piece of educational software, would be able to fully capture the knowledge state of a learner? A student model, in the traditional view, is impossible, as it is impossible for the teacher to fully grasp the knowledge state of a learner. But that does not mean that all the ideas related to cognitive diagnosis are useless. On the contrary. When we look at what teachers can do, we see that they give assignments, explanations, ask questions, and so forth. They are engaged in a communicative interaction with their students. Continuously probing a student to find the difficulties that need further instruction.

Now, it may be argued that sometimes teachers go beyond identifying inconsistencies and actually focus on the misconceptions that learners may have. This would correspond to the inclusion of fault models (as discussed above). We may even put effort in trying to automatically derive such fault models. But more important, in the absence of such models the technique will still be applicable.

2.3 Consistency-based cognitive diagnosis for learning by modeling

In order to bear out the distinction between our approach and existing ones, we first discuss the componentization of technical diagnosis. We then discuss the componentization of cognitive diagnosis with a norm model (i.e. the approach followed in [40] and [19]). We then show that cognitive diagnosis with a norm model is, in some respects, still pretty much like technical diagnosis. We then make some fundamental changes to arrive at the componentization of our cognitive diagnosis approach (without a norm model). The rest of this document will discuss the crucial components of this new componetization.

Technical diagnosis

We give a diagrammatic overview of technical diagnosis in figure 2.5. The initial observations, asserted prior to the invocation of the diagnostic algorithm, are sample inputs to be placed on the device under consideration. The initial observation outputs are the values that are measured in the device under the given sample inputs. Besides these initial observation, the diagnostic algorithm also receives the Component Connection Model (CCM) that is the representation of the blueprint of the device for diagnostic purposes. The blueprint is the authoritative description of the device, in the sense that all the behavior that we think of as defining the operation of the device should be unambiguously deducible from this description. Finally, the diagnostic algorithm takes the component definitions or functions that describe the context-independent behavior of the individual (abstractions of) components that constitute the device.

Cognitive diagnosis with a norm model

The approach in cognitive diagnosis entails three replacements of modules in the technical diagnosis diagram. The first one is to replace the device by the learner (not a model of the learner but the real learner). Second, all inputs given to the device / learner become questions posed; all outputs become answers given. This holds for the inputs and outputs that are the initial observations, as well as for the additional probe point observations. The third replacement concerns the blueprint, which becomes the norm model of the system whose behavior is to be learned. Building the Component Connection Model (CCM) involves (1) running the simulation on the norm model, and (2) generating the CCM representation

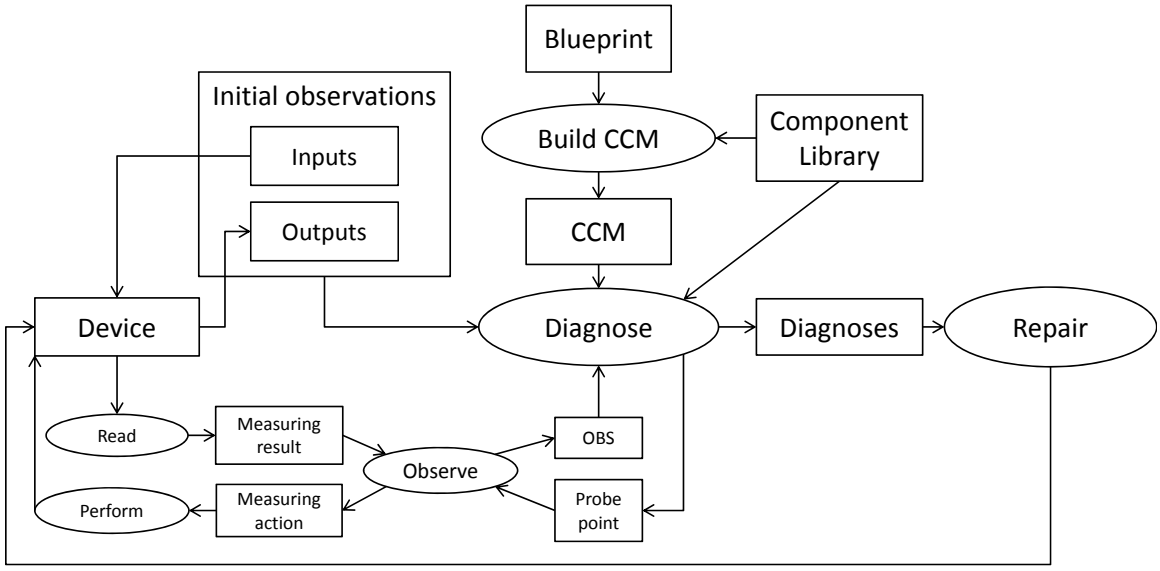

Figure 2.5: Diagrammatic representation of technical diagnosis.

based on the normative simulation results. To fit this change, the component library no longer contains technical components, but functional descriptions of reasoning steps.

In cognitive diagnosis the notion of repair does not have a direct correlate. The diagnoses that are the output of the diagnostic interaction indicate reasoning steps the learner has not yet understood with respect to the correct norm model. This could prompt the generation of further explanations, additional questions, or the determining of a grade. But there is no such thing as an automatic replacement of some piece of faulty knowledge by correct knowledge.

Cognitive diagnosis without a norm model

Starting out with the componetization of our approach, it is important to keep in mind that there is no such thing as a (single) true model. Instead, there is only a drive towards consistency between expectations and simulation results; both of which can be, and typically are, in flux.

Figure 2.6 shows the componetization of our approach. The device is again replaced by the learner, but the initial observations are no longer questions posed and answers given. Since there is no norm model, there can be no such thing as predefined questions which the learner should answer. Instead, the initial observations are all given by the learner. We distinguish two types of initial observation. The first are the *modeling goals*. These are assertions of hypothetical system behavior, specified prior to simulating the model. The second are *expectations*. These are assertions of disparate behavior, that are uttered upon inspecting the simulation results. It is not necessary to have both modeling goals and expectations, as long as at least one initial observation is given the diagnosis can proceed. Having more initial observations helps to make the diagnosis more focused and generally results in less probe questions before finding the culprits. Also, making modeling goals explicit can be an important step in modeling.

As see in figure 2.6, the CCM is connected to the learner (via some intermediary steps). This shows our adherence to the constructivist paradigm in learning. The model that we run diagnosis on (CCM) is based on the simulation results of the learner created model. There are some restriction on the modeling and simulation process to assure that what the

learner creates can actually be diagnosed. Firstly, the modeling environment only allows grammatically correct QR models to be created. In the DynaLearn user interface, the build actions are automatically activated/deactivated based on whether performing a certain build action, given the current model and the current selection of model elements, will amount to a grammatical QR model or not. Secondly, the simulation engine's output should be fully interpretable by the CCM generation process. Thirdly, the important reasoning steps by which the simulation results are deduced should be replicated in the CCM by providing the right component functions inside the component library.

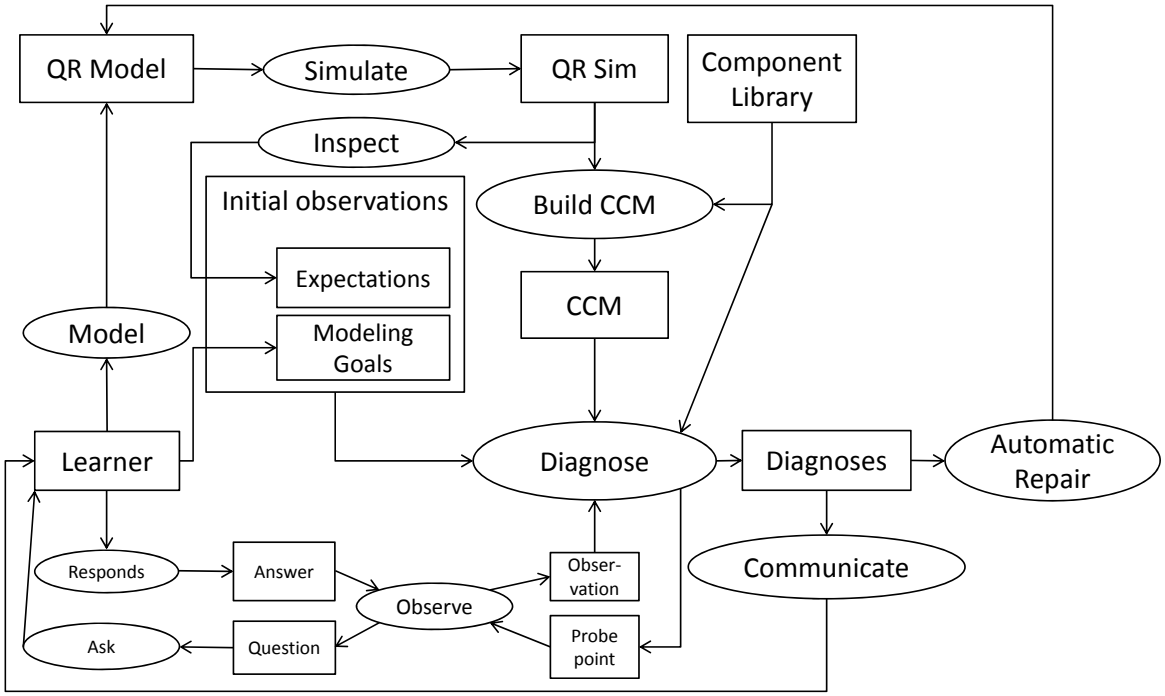

Figure 2.6: Cognitive diagnosis without a norm model.

The primary purpose of our diagnosis is to communicate the culprits to the learner. This is done by identifying those elements inside the build model to which the culprits relate. The learner is then in charge to change either her model or her modeling goals, and run diagnosis again. We think the iterative character of the interaction is characteristic of the model creation process.

In our approach, the notion of repair is added as a secondary result of the diagnosis, replacing the build model equivalents of the culprits with QR elements that make the simulation agree with the modeling goals and/or expectations. This notion of automated repair is an addition to our core diagnosis. It is discussed in section 6.6.

Chapter 3

Component Connection Model generation

This chapter describes the generation of the basic Component Connection Model (CCM) for arbitrary QR simulations that are performed by the Garp simulation engine. Section 3.1 treats of the output of the Garp engine on the basis of which the states and expressions of the CCM are generated, as explained in section 3.2. In section 3.3 we discuss the addition of components to the CCM. Section 3.4 enumerates the various types of cognitive components in the CCM and the respective roles they play. The chapter ends with section 3.5 which explains the algorithms that generate the basic CCM.

3.1 Garp engine output

The Garp engine [4] takes qualitative models as input and returns qualitative simulation results in which the dynamics of the modeled system are characterized over time. For these simulation results several dedicated database formats are used. These formats have been optimized for Garp's reasoning tasks. An API was constructed to be able to extract the required data from the simulation results in an easy way.[1] Since we will explain the generation of the CCM in terms of the Garp simulation elements that it covers, we briefly discuss the various simulation element types that we make use of.

States States are qualitatively distinct situations that the simulated system can be in. States either correspond with a *time point* or with a *temporal interval* of indefinite length. Part of the QR approach is that the temporal extension of interval states is unknown. States consist of information regarding the structure and behavior of the system at that point/interval in time.

Structural information This structural information consists of the agents and entities, including their configuration relations and attributes. Agents and entities may have quantities associated with them, that are also part of the structure of the state.

Behavioral information Besides structural information there is *behavioral information*, including causal relations (direct and indirect ones), correspondences, calculi, and (explicit) inequalities. The behavioral relations are shown in table 3.2.

[1] An alternative representation for Garp simulation results is given by [33]. We do not make use of this representation for the reasons given in section 3.2.

Other elements that are part of a state are identity relations, imported model fragments (for knowledge reuse), quantity spaces (every quantity has one), and assumptions (conditions under which a model fragment applies). Each state in Garp is represented as a so-called System Model Description (SMD). We briefly discuss the various components that constitute Garp SMDs.

Table 3.1: The structural aspects of Garp simulation results.

Name	**Explanation**
Agents	Entities that are outside of the modelled system. Their quantities are called *exogenous* and have *external influences* on the system.
Assumptions	Labels that are used to indicate that certain conditions are presumed to be true, and are used to constrain the system behavior generated by a model.
Configurations	Relations between entities and/or agents.
Entity	Non-changing concepts that constitute the basic components of the system structure.
Quantity	Changeable features of entities and agents. Each has a quantity space that contains the possible values that the quantity can have, called its magnitude. A quantity also has a (first) derivative value. change).
Quantity space	A sequence of alternating interval and point labels that characterize the range of possible values a quantity can have.

Table 3.2: The behavioral aspects of Garp simulation results.

Name	**Explanation**
Correspondences	Directed or undirected relations between qualitative values of quantity spaces belonging to different quantities, modeling the notion of simultaneity. There are correspondences between magnitudes, derivatives, quantity spaces, the latter of which can also occur in reverted form.
Direct influences	Directed relations between two quantities, cause change within a model, i.e. modeling processes. They can be positive or negative.
Indirect influences (proportionalities)	Directed relations between two quantities, propagating the effects of a process. They can be positive or negative.
Inequalities	Ordinal relations between two items. There are 11 ways to use inequalities, depending on the type of the two items related by them. They can have 5 different signs, i.e. $<$, $\leq$, $=$, $\geq$, and $>$.

3.2 Garp-based CCM generation

This section describes the construction of that part of the Component Connection Model (CCM) that is generated based on the simulation results discussed in section 3.1. The other part of the CCM is not based on simulation results and is discussed in section 3.3.

In QR simulations time is the most unwieldy dimension. This is why we split the CCM representation in two gross parts: the atemporal and the temporal.[2] The atemporal part contains the elements that represent knowledge and allow reasoning procedures irrespective of time. For instance, if quantity *Pressure* is proportional to quantity *Height* and quantity *Height* is increasing, then quantity *Pressure* is increasing as well. This derivation holds irrespective of the behavioral states in which it may or may not be manifest.

The temporal CCM part consists of temporal instantiations of atemporal derivations. The temporal part may, for instance, represent that quantities *Height* and *Pressure* are indeed related in the aforementioned way in states 12 and 25, but not in any of the other states (e.g. because the height is not increasing there).

The structural elements are all represented in the atemporal part of the CCM (the gray, blue and purple nodes in figure 3.1). In addition to that, expressions (green nodes) are also only represented in the atemporal part. This includes the expression that describe behavioral relations (e.g. 'Expression 3' in figure 3.1). States and points (yellow and orange nodes respectively) are only represented in the temporal part. The only connections between the two parts are those between expressions and points.

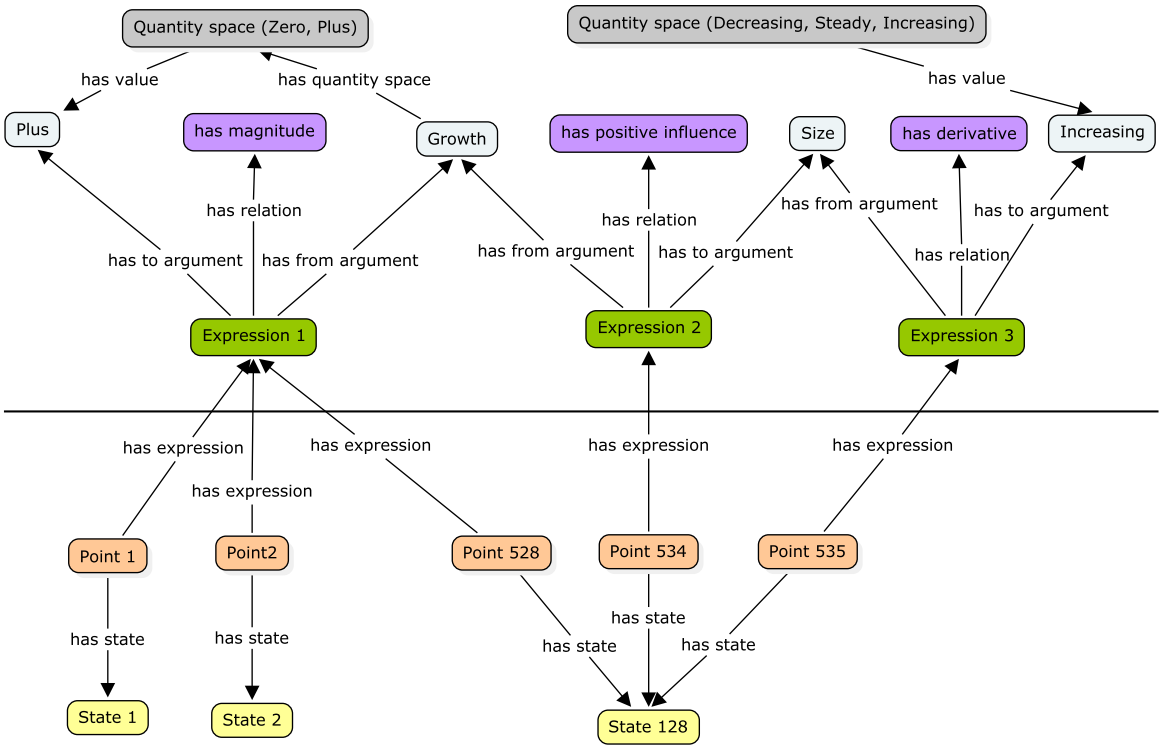

Figure 3.1: The temporal/atemporal representation of the CCM. Above the horizontal line are shown the atemporal aspects, i.e. the expressions and various QR ingredients. Under the horizontal line are shown the temporal aspects, i.e. points and states.

The alternative for the temporal/atemporal distinction that we use, is the replication of all atemporal information in each state in which it is used. This approach has been used in [33], leading to very explicit but also representationally verbose results.[3] Besides

[2]The generative parts of the CCM, i.e. simulation-based or not, is linked but not identical to the conceptual parts of the CCM, i.e. temporal/atemporal.

[3]To give an impression, the verbose version uses 10MB for an average model, while our method uses a few hundred kilobytes instead, including additional diagnostic information such as agents (section 5.2 and ATMS labels (section 5.3. The difference verbosity becomes even bigger (absolute as well as in percentages) when

the downside of a verbose format, our representation is specifically optimized for diagnostic purposes, allowing us to perform often occurring operations in speedy ways.

3.2.1 Relations

Before we discuss the various QR elements that are part of the CCM, we discuss the relations that can exist between QR elements. Relations must belong to a relation definition. This definition specifies the properties that the arguments of the relation should have. Relations can be added to the relation definition hierarchy in a modular way.

All domain-dependent relations, i.e. the relations that occur in our parlance of the QR language, are reified. This allows relations to have explicitly represented properties (like e.g. symmetry, transitivity). Moreover, this allows the relations to be hierarchically related. Finally, reification allows the formulation of non-binary relations (see section 3.2.8). Examples of expression nodes can be seen in figure 3.1, i.e. the green nodes.

Figure 3.2 shows the top relation definitions. As we discuss the various QR elements in the next couple of sections, we will occasionally show extensions of this hierarchy.

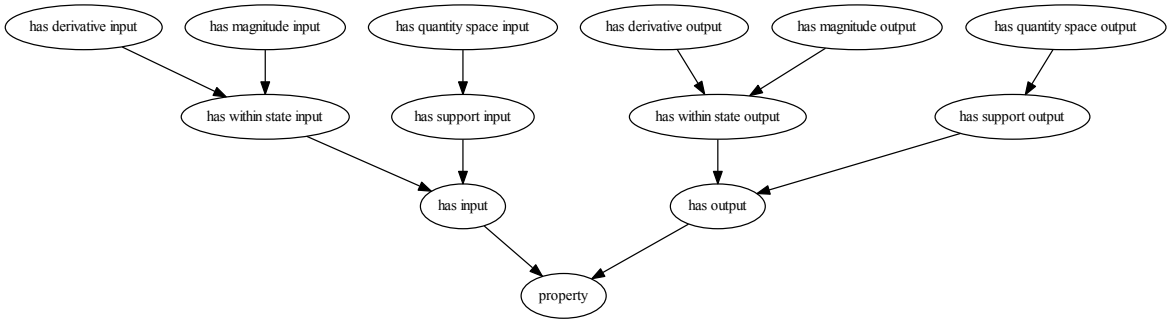

Figure 3.2: The top definitions of the relation definition hierarchy.

3.2.2 Entities & agents

In Qualitative Reasoning, entities and agents are either physical objects or abstract concepts that are themselves stable but whose properties model the dynamics of the system. The difference between entities and agents is that the former are part of the system, whereas agents are acting upon the system from without.

Entities and agents are represented as URIs in the CCM. Since entities and agents represent static aspects of the simulated system, they are represented by singular URIs. They are related to the dynamic aspects of the model via links to their quantities.

A complication in representing entities and agents is that can be multiply instantiated by imported model fragments. An imported model fragment is a model fragment that is incorporated into another model fragment as a condition. Multiple instances of a model fragment can be incorporated into another model fragment as conditions. A single model fragment can import multiple model fragments and any finite number of iterative imports is possible. Entities and agents are therefore identified based on the (nested structure of) model fragments to which they belong in the build model.

3.2.3 Quantities

Qualitative Reasoning is about modeling and simulating the changeable features of entities and agents. These potentially changing features or dynamic aspects of a model are called

the number of states in the simulation increases

quantities. Quantities can take on any finite number of discrete values. The value of a quantity is called its magnitude. Besides its magnitude, a quantity can have a (first-order) derivative.[4]

Quantities are represented as URIs in the CCM. There are relations between entities/agents and their quantities. Table 3.3 shows the properties of quantities. Since quantities represent the dynamic aspects of a simulated system, their values are not (guaranteed to be) persistent over states. Quantities manifest two forms of multiple instantiation: (1) Within a singular model fragment (without imports) different entities and/or agents can have the same quantity.[5] (2) Because of imported model fragments, the same entity/agent may occur multiple times with dito occurrences of its quantities. For each copy of a quantity in the engine results, a copy is created in the CCM as well.

Table 3.3: The properties of quantities.

Class	**Relation**	**Argument**	**Q**[6]	**Description**
	`has_entity`	Value	1	The entity or agent to which the quantity belongs.
	`has_qs`	Quantity Space	1	The quantity space of the quantity.
`has_val`	`has_magn`	Value	*	Links to magnitude values the quantity has in some expression.
	`has_der`	Value	*	Links to derivative values the quantity has in some expression.

3.2.4 Quantity spaces

The magnitude values that a quantity can have are defined in a quantity space. This is a sequence of labels designating discrete values. The derivative value of a quantity belongs to the default derivative quantity space $\langle \text{decreasing}, \text{steady}, \text{increasing} \rangle$ for the derivative of the quantity.

Quantity spaces are represented by URIs. They have relations to their values which are also URIs. The properties of quantity spaces are enumerated in table 3.4. An example of quantity spaces is given in figure 3.3. Different quantities often use the same quantity space. In these cases multiple quantity spaces are created, since the same value need not have the same meaning for two different quantities (e.g. 'average' on population's size is not the same as 'average' on the sun's light intensity).[7] For this reason separate URIs are created for the same value in different quantity spaces. There is no representational difference between quantity spaces used for magnitude and the one used for derivative values.

Table 3.4: The properties of quantity spaces.

Relation	**Argument**	**Q**	**Description**
`has_value`	Value	+	A value of the quantity space.
`has_first_value`	Value	1	The first value of the quantity space.
`has_zero`	Value?	+	The zero point value of the quantity space.

[4]Quantities can also have second-order derivatives, but these have not yet been used by the diagnostic engine.

[5]An example of this is a model with predator and prey populations, both of which having the 'Size' quantity.

[7]Actually, 'average' on a wolf population's size is not necessarily the same as 'average' on another wolf population's size.

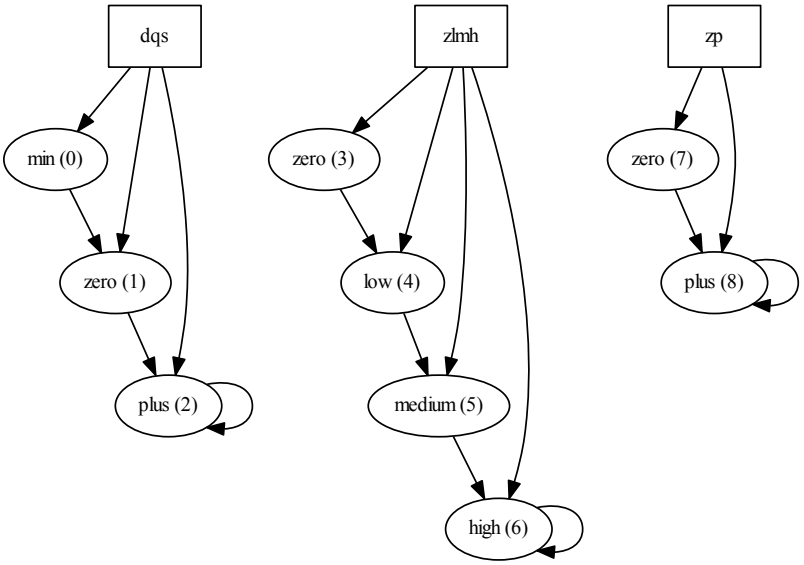

Figure 3.3: Three quantity spaces in the CCM representation. The links between the values are relations to the next value.

3.2.5 Expressions

Expressions are formed according to a recursive grammar. Expressions express binary relations between conceptual modeling elements. Rules for generating expressions are expression definitions that are modularly added to the CCM generation algorithm (section 3.2.8 will treat of expression definitions). Each expression instance asserts a relation between two arguments. The definitions can pose strictures on the relation as well as on the arguments.

Expression are represented by URIs that refer to (1) a 'from' argument, (2) a 'to' arguments, and (3) a relation. Expressions must use a relation that is itself defined in the relation definition hierarchy (see section 3.2.1). The properties of expressions are shown in table 3.8.

In line with the temporal/atemporal distinction discussed earlier, expressions are not temporally annotated (as opposed to points, see section 3.2.6). This means that they characterize QR facts irrespective of the behavioral state(s) in which they hold. This is important, since we often want to perform atemporal reasoning procedures that have time as a combinatorial metric. A simple example of this is checking whether some fact holds somewhere in the simulation, i.e. irrespective of the specific state in which it holds. E.g. "Will the dam break at some point?"[8]

As explained, the realm of expressions is useful for performing atemporal reasoning procedures in efficient space and time, but its use is extended to hybrid (temporal/atemporal) procedures also. An example of such a hybrid method is the adding of components to the CCM, discussed in section 3.3).

3.2.6 Points

As explained in the previous section, expressions are not temporally annotated. The simulation results of QR models crucially involve the time segments at which expressions express true facts. For this we use points. A point is the use of an expression within a context. The term 'instantiation' stands for an occurrence that adheres to an explicit definition, i.e.

[8]From this it is clear that we cannot do consistency checks or applications of traditional entailment on the level of expressions, because of the possibility (and likely occurrence) of contradictions.

Table 3.5: The properties of expression instances.

Class	Relation	Argument	Q	Description
has_arg	has_from_arg	Resource	1	The from argument of the expression.
	has_to_arg	Resource	1	The to argument of the expression.
	has_rel	Relation	1	The relation that the expression asserts between the from and to arguments (in that order). The expression relations are children of some (grand)child of the expression relation definition in the relation hierarchy displayed in figure 3.2.
	has_point	Point	*	An arbitrary number of relations to points that are instances of this expression some state. An expression need not have any points. Some expressions cannot have points (i.e. those expressing inherently generic knowledge).

by instantiating all the variables in that definition with constant terms. An example of this are the expression definitions and instances. A point does not instantiate an expression in this sense (since an expression is itself an instance). Instead a point replicates an expression instance within a temporal context. In this sense it adds a temporal argument to the expression and fills it with a constant term (i.e. a state instance). This latter notion of replication within a temporal context (see section 3.2.7 for an explication of 'temporal contexts').

A point is represented by a URI that is related to an expression and a space. Points are the first type of elements we discuss that belong to the temporal layer of the CCM. They are related to spaces. The space is the time at which the point's expression is supposed to be true. Table 3.6 shows the properties of points.

Table 3.6: The properties of points.

Class	Relation	Argument	Q	Description
	has_expr	Expression	1	The expression that is contextualized by this point.
has_comp	has_incomp	Component	*	A component that has this point connected to one of its output ports.
	has_outcomp	Component	*	A component that has this point connected to one of its input or support ports.
	has_space	Space	1	The space of this point. The point belongs to this space. The expression of this point is contextualized in this space.

3.2.7 Spaces

As mentioned in the description of points, they relate to spaces. Spaces are collections of points and components (see section 3.3.1). In the foregoing we have talked about points as uses of expressions within a context. (Spaces are what was designated by the inexact term 'context'.)

Some spaces have temporal designation, these spaces are called states. A state collects

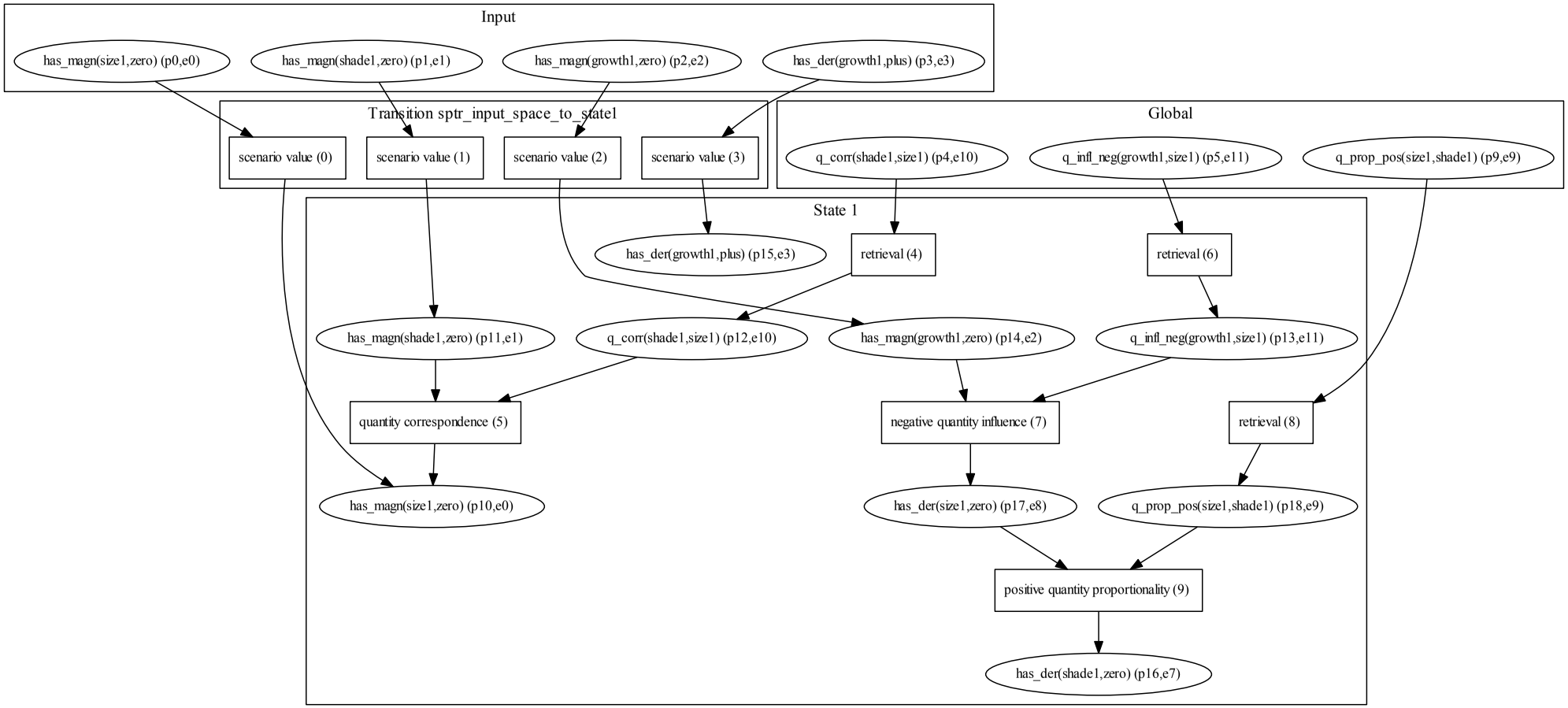

Figure 3.4: An example showing several spaces. The ellipses are points, the squares are components. The big squares, encompassing points and components are spaces. There are space called ‘Input’ and ‘Global’. There is a state space (i.e. ‘State 1’), and there is a transition space from ‘Input’ to ‘State 1’.

points and components that represent a behavioral state with a certain duration.[9] A single state contains at most one point for every expressions. The same expression can be related to by multiple points, provided they belong to different states.

Between each pair of consecutive state spaces, there exists a space of another type: a transition space. A transition designates that it is possible for a state space to follow directly from another state space. The possibility modality points to the existence of ambiguous behavior, i.e. multiple transition pairs with the same first argument can exist.

In addition to state and transition spaces, there are two specific spaces: the input space and the generic space. The input space contains points for all the expressions that are specified in the scenario on which the QR simulation has been run.[10] [11] States that are the first ones to be generated by the simulator within a behavior path are called the initial states of a simulation. Between the input space and any initial states, input space transitions are added.

The generic space collects all points that do not fit in either a state, transition, or input space. In other words, all expressions that have no point in any of the foregoing spaces are related to a single point in this generic space.

Conceptually speaking, the generic space contains the facts that are atemporal. We nevertheless include the generic space into the temporal layer of the CCM, since it contains all and only points related to expression that are not temporal. In this sense it is negatively defined by the temporal aspect. (If you know the temporally related expressions, then you also know the generic ones. There are no transition spaces between the generic space and any other space.

Spaces play an important role in reducing computational complexity. By annotating functions with spaces, the search space is vastly reduced by only having to consider points and components in that very space.

The use of spaces is contrasted to the use of expression (see section 3.2.5). While expressions provide an efficient representation for performing atemporal functions, spaces provide the right representation for efficiently calculating temporal functions.

Spaces are all represented by URIs. There is no temporal representation except from the state spaces.

Table 3.7: The properties of spaces.

Class	**Relation**	**Argument**	**Q**	**Description**
`has_element`	`has_component`	Component	*	A component that belongs to this state.
	`has_point`	Point	*	A point that belongs to this state.
`has_space`	`has_from_space`	Space	*	A space that links directly to this space.
	`has_to_space`	Space	*	A space that is directly linked from this space.

[9] It is part of the QR paradigm that the exact temporal duration is abstracted away. There are only two durations distinguished: point and interval durations.

[10] Scenarios are used to initiate a simulation. The simulation generates the states and transitions based on this initial state.

[11] As is clear from the use of expressions in the context of the input space, not all points are temporally annotated in the strict sense of the word. However, the input space can be conceived of as the temporal state before the simulation began.

Table 3.8: The properties of states.

Class	Relation	Arg	Q	Description
	has_duration	Bool	1	When true, this state has a duration, meaning it takes some amount of time. When false, this state has no duration but takes an infinitesimally small amount of time. An example of a state without duration is when temperature increases from below to about zero degrees Celsius. Then the behavioral state in which the temperature is 0 degrees has no duration.
has_state	has_from_state	State	*	A state that transitions directly into this state. This means that the state is temporally prior to this state.
	has_to_state	State	*	A state that that state transitions into directly. This means that this state is temporally prior to it.

3.2.8 Expression definitions

In section 3.2.5 it was said that all expressions in the CCM are instances of expression definitions. These define strictures on the arguments and relationship of an expression.

By combining two expression definitions with non-contradictory strictures, new expression definitions can be formed. New expression definitions can also be formed by adding additional strictures to an existing expression definition.

By making the expression definition hierarchy dynamically extensible, we can ensure that any expression can be added to the CCM.[12]

In figure 3.5 we shown the top definitions of the expression definition hierarchy. We explain a few of these expression definitions to illustrate the fundamental characteristics of the expression definition hierarchy.

Value expressions are relations between a value and a quantity. This definition is further refined by requiring the value to be a magnitude or derivative respectively. In this way the *derivative* and *magnitude* expression definitions are extensions of the *value* expression definition. The *correspondence* expression definition has parents *support* and *within state*, taking over both of their strictures.

The *within state* expression type states that an expression must occur within a simulated sate of behavior. This opposes the *transition* expression type which states that an expression must occur in a transition from one behavioral state into another. It is not possible to define an expression type that is the child of both the *within state* and the *transition* expression types.

Not all expression definitions can be instantiated; some have the purpose of ordering other expression definitions. Examples of this are the *within state* and *transition* expression definitions.

All expression definitions are created as separate files that adhere to a certain syntax.[13] Presently, the expression definitions we use, utilize the properties in table 3.9.

[12]Tertiary expressions, and ones with higher arity, are formed by reification, as in [13]. This is an extension of the binary relations that we already use, since all QR relations are reified (see section 3.2.5).

[13]The definitions are included into the hierarchy only by virtue of being included in a specific directory.

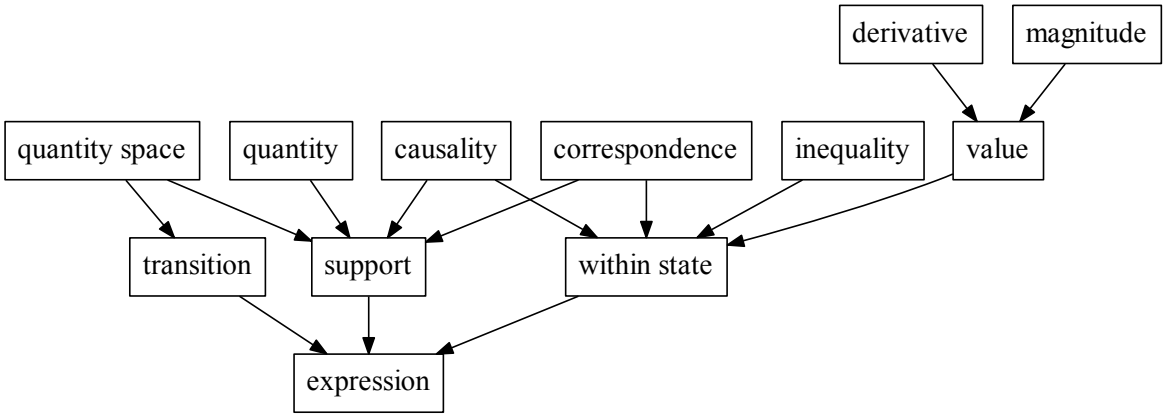

Figure 3.5: Expression definitions: top of hierarchy

Table 3.9: The properties of expression definitions. These are dynamically extended by adding expression definitions that make use of additional properties.

Relation	Argument	Q	Description
`subClassOf`	Expression Definition	+	Specifies a parent expression definition, taking over all of its strictures.
`is_directed`	Bool	1	States whether the relation is directed (directional) or undirected (bidirectional). The default is undirected.
`has_changeable_prop`	{`has_from_arg`, `has_rel`, `has_to_arg`}	1	Specifies that aspect of the expression that can be changed by a user's expectation (see section 5.5).
`is_instantiable`	Bool	1	Whether the expression definition can have direct expression instances, or is only intended to order other expression definitions.

3.3 CCM generation: fitting components

This section takes the partial CCM as generated based on the Garp simulation results, and complements it by fitting in components. The fitting of components only depends on the partial CCM and on the component definitions that have been loaded, but it does not use the simulation results (directly).

3.3.1 Components

Components are functions that can be fitted into the collection of points and spaces that are the result of the prior CCM generation phase (section 3.2). The parameters of a component function are called ports. Instantiating parameter variables is visualized by connecting points to ports. Since, in general, the order of arguments is relevant for the outcome of a function, the order in which the points are connected to a component's ports is relevant as well.

Components are necessarily related to spaces, as are points. In other words, components

are inherently temporal.indextemporal aspects Component functions are always related to the time at which they are supposed to be calculated.

Components are represented by URIs. Table 3.12 enumerates the properties of component instances.

Table 3.10: The properties of component instances.

Relation	Argument	Q	Description
has_point_spec	Component Point Specification	+	For every port that this component has, a specification of its point is given. Relations to the various points would not represent which point is which (i.e. the order in which the parameters are fed to the function). Defining a sequence of ports would be a solution, but we would have to keep track of which port had which restrictions. Component point specifications contain a link to the point, but also to the component-to-point relation that defines the role the point plays for this component. See section 3.3.2 for the full treatment of these restrictions.
has_space	Space	1	The space of this component.

3.3.2 Component to point links

The links from components to points have specific meanings. The links from components to points are differentiated into hierarchically ordered links that signify the meaning of the point with respect to the component that links to it. Since the same point can obviously play disparate roles with respect to different components, it is necessary to indicate the meaning of the point with respect to a component, i.e. by the link from the latter to the former.[14]

The types of relations between a component and its points depends on the component definition. Component definitions give point specifications. These specifications are carried over to component instances, and consist of three relations, displayed in table 3.11.

First, the point that connects to the component must use an expression that is of a certain type. This is the way in which the component definition and expression definition hierarchies are intertwined. Adding a new component can either involve the new use of an existing expression definition (other order; other arity), or it can require the addition of new expression definitions.

Second, the relation between the component and the point is of a specific type. There are two reasons for having typed component-to-point relations. The relations allow the component functions to refer to points that cannot be lifted out without referring to the component itself.[15] Additionally, by hierarchically ordering the component-to-point relations, many operations on the CCM can be performed easily and efficiently (e.g. retrieving all inputs for a given component involves retrieving all relations that are (grand)child of `has_input_point`).

Third, a piece of code is supplied for every point. We do not only want to define the component function using the points as parameters, but we also want to define a partial function

[14] An example of the same point playing different roles for two different components is a derivative value assertion, which may be the output of an influence component but also the input to a proportionality component.

[15] For instance, a component may have two magnitude value inputs, requiring the value of one of them to be subtracted from the other. The order in which the input points are fed into the component function cannot be distinguished by looking at the two points independent of the component.

for each component/point pair individually. The component function is build out of these (partial) component/point functions. This is useful since it allows some component/point functions to be defined in a parent component definition, while filling in the component/point functions for other points in a child component definition.

Table 3.11: The properties of component-to-point specifications.

Relation	Argument	Q	Description
has_expr_def	Expression Definition	1	Specifies that the point that is connected to the component should have an expression that is of the given definition.
has_comp_rel	Component Point Relation	1	Specifies the relation from component to point. The relations are drawn from the component relation hierarchy.
has_code	Code	1	A functional description that can be parsed by the component fitting algorithm. These placement functions are used to determine when and where a component is added to the CCM, see section 3.5.

The properties in table 3.11 are used by a component to refer to its points. The other way round, points refer to components with either the `has\_outcomponent` or `has\_incomponent` relation. These relations are annotated by the `has\_component\_delta` property. This tells the diagnostic engine that the link goes to a component that encompasses a certain number of atomic components. See section 4.4 for the use of this component delta property.

3.3.3 Component definitions

Components are fitted into the collection of points and spaces based on component definitions. As with expression definitions, we allow the set of component definitions to be extended by adding separate files that adhere to some specification language.

There are four types of strictures a component definition may gave:

- In- and output expressions
- Spaces
- Support expression
- behavior functions

We now discuss each of these in turn.

In- and output expression

A component is defined to have one or more input ports and one or more output ports. Although the distinction between in- and output is irrelevant from the point of view of diagnosis, it does indicate the direction of deduction according to common sense interpretation.[16] For each port strictures on the allowed expressions are given. A component fits into the CCM if there are expressions in the CCM that adhere to these port-related strictures. Observe that although components are connected to points, their fitting criteria are defined and calculated with respect to expression (thus abstraction away the temporal aspect).

[16]The distinction between in- and output ports thus plays a role in giving causal explanations, see section 6.7.

The strictures put on ports refer to expression definitions drawn from the hierarchy described in section 3.2.8. The specification language specifies for each component function which expressions (for points) are allowed as its parameters.

Spaces

Further strictures can be placed on the spaces of the points that connect to the ports of a component. In addition to that, the (temporal) distance between these spaces can be specified to answer arbitrary strictures. This effectively amounts to providing temporal annotations for components (adding a temporal argument to component functions), allowing components to be defined for behavior that e.g. has not changed for N consecutive states.[17]

As a matter of fact, being able to define component definitions that conjugate multiple spaces is an important reason to not extract the components from the Garp simulation results directly. QR deductions that Garp makes always amount to the calculation of a future state of the modeled system based on the facts that characterize one of the directly preceding states. The component definition specification language we defined allows to transcend this traditional restriction of using the directly preceding state to establish the facts that describe a new state.

Support expression

Every component definition has exactly one support expression. The support expression encodes a piece of declarative knowledge necessary to perform a certain derivation.

Component function

The component function is a mapping from the expressions of the points connected to a component to $\{0, 1\}$. When the component function outputs 1 the component describes the relations between the parameter expressions correctly.

The strictures must be explicit enough to ensure the exclusion of unintended ambiguity caused by adding multiple components in between the same collection of points. Section 4.1 will cover intended CCM ambiguity in the sense of aggregation. In aggregation multiple components are placed in between the same expressions, in order to provide alternative calculations of the same behavior.

It is important that components be, functionally speaking, context-independent. Only in this way can a compositional circuit behavior be given. This means that the component function can only refer to its own in- and outputs, as well as to its own definition.

The component definitions have to be defined context-independently. This assures that every generated CCM will have computable behavior.

3.3.4 Component definition hierarchy

Component definitions are added by adding an file written in a certain specification language. Whenever a component definition is imported, the availability of the expression definitions and component relations it uses is automatically verified.

Some component definitions are not intended to have component instances, but have the purpose of ordering the component definition hierarchy. Examples of these are the *influence* and *inequality* component definitions. Their strictures are combined in e.g. the *inequality influence* component, which does allow instantiations.

[17] The temporal argument of component definitions is also used for setting arbitrary space distances in the definition of aggregate components, see section 4.4.

Table 3.12: The properties that have been defined for component definitions.

Relation	Argument	Q	Description
has_inspace_def	Space	1	The type of space that the input points should belong to, drawn from the spaces hierarchy.
has_outspace_def	Space	1	The type of space that the input points should belong to, drawn from the spaces hierarchy.
instantiable	Bool	1	Whether the expression definition can have direct expression instances, or is only intended to order other expression definitions.
is_directed	Bool	1	Whether the intended meaning of the component involved is directed or not. This is not the same as the directions in which behavior rules can be defined for use in the diagnostic engine. A component can be directed (e.g. a quantity termination between successive states), whereas its diagnostic reasoning rules can be applied both ways (e.g. calculating the value of a quantity in the previous state based on its value in the current state and a diagnostic behavior rule).
requires	Predicate identifier[18]	+	Poses a predicate requirement. The CCM generation engine must know this predicate in order to be able to calculate the fitting criteria for the component definition. If a predicate is marked as required for a component definition, then the name of that predicate can be used in the functions of the code segments of component point specifications (see section 3.3.2).

3.4 Cognitive components

All cognitive components represent some cognitive operation. For educational purposes we want to distinguish between different kinds operation, in order to be able to give more directed feedback. Figure 3.6 shows a very small CCM in which we can identify these different types of components. We will discuss each of them in turn.

Retrieval components

Performing a QR inference involves two types of knowledge: conceptual and procedural. The conceptual knowledge is generic, e.g. that the quantity space of the quantity in figure 3.6 is $\{\text{Zero}, \text{Plus}\}$. The procedural knowledge is specific for every application inference, e.g. that the quantity in figure 3.6 changes its magnitude in the transition from state 1 to state 2.

The component that provides the generic knowledge input is called a retrieval component, since the correct inference of this components corresponds to the cognitive act of retrieving a piece of knowledge from memory [19].

Scenario input components

In traditional diagnosis, some observations are given for a system. The task of diagnosis is to find the components that cannot be working correctly given these observations. Observations cannot be retracted later. In our situation we do not want to exclude the possibility that some

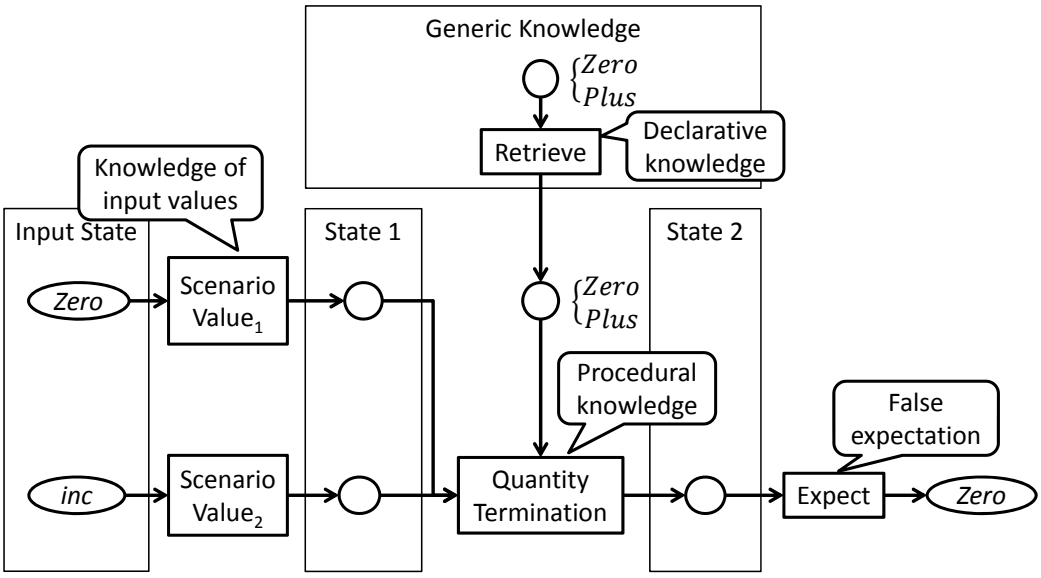

Figure 3.6: A simple example of a CCM, only modeling a single quantity termination. The quantity termination component makes a quantity change value from state 1 to state 2. The value changes from *Zero* to *Plus* because the derivative of the quantity is *Increasing* in state 1. The quantity has value *Plus* in state 2 because the quantity space of the quantity is $\langle \text{Zero}, \text{Plus} \rangle$.

of the initial observations a learner provided are wrong. Based on a diagnostic interaction a learner may choose to make changes to the model fragments in the QR model, but sometimes learner needs to make changes in the scenario on which he ran the simulation. Mistakes in setting the initial values are diagnosed by adding components in from of each scenario input. These components are called 'scenario value components'. They are added in the transitions between the input space and each initial state.

Expectation components

As is the case with scenario inputs, expectations are another example of observations that a learner should be able to retract. For each expectation an expectation component is added to the CCM, see the rightmost component in figure 3.6.

3.4.1 Submissive components

From [19] we took over the notion of competing influences. In QR there are ample situations in which processes either cancel each other out or where the one process overrules the other. In these cases, the suppressed processes are modeled by *submissive components*. These are components that fit between points that would allow the non-submissive variant to be place, if only one of the points would have has a different value. It is important that we define the placement criteria of the submissive components without referring to the actual presence of their active or overruling equivalents, since doing so would violate the criterion of context-independentness.

Figure 3.7 gives an example of a submissive component. The modeled system is a vat with liquid flowing in (Fl_{in}) and out (Fl_{out}) at the same time. The derivative of the volume of liquid contained in the vat (δV) depends on which flow (in or out) is the biggest. The inequality information in $Fl_{in} < Fl_{out}$ tells us that there is more water flowing out (negative

influence on δV) than in (submissive positive influence on δV), resulting in a decreasing volume of liquid.

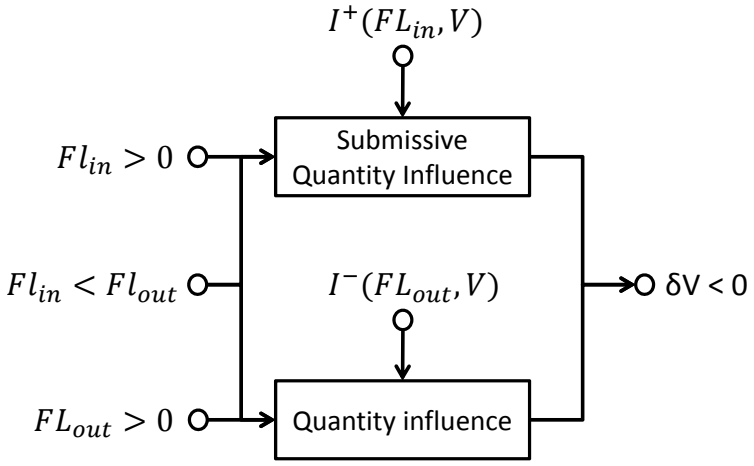

Figure 3.7: An active and a submissive component in determining the derivative of the volume of liquid in a tap that both an in- and an outflow of liquid.

3.5 Component fitting algorithm

The creation of the Component Connection Model (CCM) consists of the following steps:

1. Assert the definitions into the CCM. This includes the component, relation, and expression definition hierarchies.
2. Assert the states that we generated by the Garp simulation engine.
3. Assert the transitions between states.
4. Assert the input and generic space.
5. Assert the transitions between the input space and each initial state.
6. For the input space (first) and every state (thereafter), assert:
   (a) Retrieve an expression from the Garp simulation engine.
   (b) Make sure the expression already exists. If it does not, then assert the relation between argument as indicated by the Garp simulation results (the engine API is used for this).
   (c) Any argument that does not already exist is asserted. These are entities, agents, quantities, quantity spaces, and values (in that order).
   (d) Assert a point for the expression, and link it to the expression as well as to the given space.
7. Assert all support expressions in the generic space.
8. Fit all components (within-state and transition ones; aggregated and non-aggregated ones).

As the list of CCM construction tasks shows, modeling elements are created when they are needed. This makes generating partial CCMs very fast, since this only requires the elements that we are interested in to be asserted. The alternative, first asserting all entities,

then asserting all quantities, etc. would not have these properties. We have put a lot of effort in allowing partial CCMs to be generated, since we would like to use dynamic CCM generation in the future to speed up diagnosis. Using scope (section 5.2.1) we should be able to extend the CCM as diagnosis proceeds. Alternatively, it may be possible to reassert parts of the CCM without having to recalculate the whole model.

We add the components based on the support expressions in the generic space. For every support expression we retrieve the corresponding component definition. (Since every component definition has to define a support expression, this link is always available.) The requirements for the component definition are retrieved and it is checked whether the required predicates are currently loaded. (If a predicate that is required by a component definition is missing, then components of that type cannot be asserted.)

The component function of the definition is then retrieved. This consists of (a) predicates that are loaded as requirements, (b) directs links that exist in the CCM, (c) references to the ports of the component (via which points are accessed), and (d) two static references: to the support expression and to the component definition. The support expression and component definition are the two Archimedian points for each component function, since all sequences of function applications are eventually applied to one of these referents.

The component function is used to find the possible insertion point for components, algorithm 1. The algorithm returns all combinations of expressions that satisfy the component specification. This process is very efficient, since it is performed atemporally, having complexity 3.1 instead of the expected $O(|\mathcal{P}|^{ar(c)}$. E_i are the expressions that are instances of the hierarchically ordered expression definition that the component definition c specifies for the i-th expression (i.e. ED_i). Because of the hierarchical ordering of the expressions, E_i is only a small subset of E.

$$\prod_{i=1}^{ar(c)} (|E_i| \cdot \log_b |ED_i| + |\{p|e \in E_i \wedge f^{ep}(e) = p\}|) \tag{3.1}$$

For each of the found expression collections, any accompanying points are found. Expressions refer to points directly (see complexity metric 3.1).

The points must adhere to additional space restrictions (also specified in the component definition, see section 3.3.3). Some expressions may have no accompanying points that satisfy these space conditions. Other expressions may evaluate into multiple collections of points. Because there can be many states, the fitting of components would have been a computational hassle without the temporal/atemporal distinction. Component functions are evaluated according to algorithm 2.

3.6 Database infrastructure & underlying representation

Since diagnosis for multiple faults is inherently exponential in the number of environments, it is extremely important to make all the other functions as efficient as possible. Since few systems have been specifically designed for cognitive diagnosis, it is difficult to benchmark the performance of our system.[19]

All representations are stored as binary relations. This is a very old principle in the area of logical database programming, going back to at least [32]. This practice is of course echoed in the recent upheaval of Semantic Web techniques. In order to appear modern, our approach adheres to the W3C standards [28, 1, 6].

[19] Computational details of the system used in [40] are not available. The STARlight system ([19, 21]) that our implementation supersedes has been significantly improved upon complexity-wise in almost every respect.

Algorithm 1 The algorithm for fitting components to the CCM, based on the support expressions.

Require: The set of support expressions E, points P, components C, predicates $\mathcal{R}$, and spaces $\mathcal{S}$
Require: The function from support expressions to component definitions $f^{s2c} : \mathcal{E} \to \mathcal{P}(\mathcal{C})$.
Require: The function from component definitions to required predicates $f^{c2r} : \mathcal{C} \to \mathcal{P}(\mathcal{R})$.
Require: The function from component definition to the number of points $ar : \mathcal{C} \to \mathrm{N}$.
Require: The function from expression vectors to point vectors $f^{e2p} : \mathcal{E} \times \mathcal{C} \to \mathcal{P}(P^{ar(c)})$ (where c in the output is the component from the second input).
Require: The function from component definitions to the delta space requirement $f^{\delta} : \mathcal{C} \to \mathrm{N}$.
Require: The function from points to spaces $f^{p2s} : \mathcal{P} \to \mathcal{S}$.

for all $e \in \mathcal{E}$ **do**
  $C^e \leftarrow f^{s2c}(e)$
  for all $c \in C^e$ **do**
    $R^c \leftarrow f^{c2r}(c)$
    for all $r \in R$ **do**
      $Loaded(r)$
    end for
    $\vec{e} \leftarrow WalkTree(e, c)$
    $P \leftarrow f^{e2p}(\vec{e}, c)$
    for all $\vec{p} \in P$ **do**
      $s^{in} \leftarrow f^{p2s}(p_1)$, the input space for the component.
      $s^{out} \leftarrow f^{p2s}(p_{ar(c)})$, the output space for the component.
      $\delta \leftarrow f^{\delta}(c)$
      if $\Delta(s^{in}, s^{out}) = \delta$ **then**
        $NewComponent(c, s^{in}, s^{out}, e)$
      end if
    end for
  end for
end for

For the database infrastructure we use SWI Prolog's RDF storage [45]. This is a relatively recent storage implementation that provides huge efficiency improvements when compared to traditional tuple storage databases. Since binary relations or triples are fully indexed, the performance of querying the data is constant with regard to the size of the triple set. (The default Prolog storage indexes tuples on the first argument solely.) There is a penalty due to occasional re-indexing. Since in diagnosis the number of reads (multiple reads per environment) hugely outweighs the number of writes (occasional assertion of expectations and probes, not proportional to the number of environments), this is a very good fit for our diagnostic system.[20] We agree with [46] that an efficient triple store can be used as an internal data model (besides being a handy exchange model also).

Because of the use of hierarchical representations, most operations are very efficient. For instance, when we know the definition of a relation, expression, or component, the search space is already significantly reduced, i.e. $\log_b(|Def|)$ (the base b of the logarithm is not constant since the branching factor of the hierarchies varies due to the dynamic

[20] In addition to being much faster than traditional logical programming stores, the SWI Prolog RDF triple store is also much faster than other triple stores (SWI Prolog outperforms the famous Jena system by factor 7 to 10, [45]).

Algorithm 2 $WalkTree(e, c)$ The algorithm for evaluating component functions.

1: $R \leftarrow f(cd)$, the set of component-to-point relations.
2: **if** $e =$ support **then**
3: **return** Support expression se and empty store.
4: **else if** $e =$ component definition **then**
5: **return** Component definition cd and empty store.
6: **else if** $Atomic(e)$ **then**
7: Return e and empty store.
8: **else if** $Complex(e)$ **then**
9: $e = o(args)$, operator o and arguments $args$.
10: Run $WalkTree(args)$, returning $args\_results$ and $args\_store$.
11: $ExecuteO(o, arg\_results, o\_result, o\_store)$.
12: Returns $args\_store$, o_store.
13: **end if**

Algorithm 3 $ExecuteO(o, arg\_results, o\_result, o\_store)$ The algorithm for executing operations in the component function, used by $WalkTree$.

1: **if** $o =$ store **then**
2: $o\_result \leftarrow arg\_result_2$
3: $o\_store \leftarrow value(\text{KEY} : arg\_results_1, \text{VALUE} : arg\_results_1)$.
4: **else if** $o \in R$ **then**
5: o_result, $o\_store \leftarrow o(arg\_results)$.
6: **else**
7: $o\_result' \leftarrow$ the predicate resource that is reached by following the link named o in the CCM starting from subject $args\_results$.
8: o_result, $o\_store \leftarrow WalkTree(o\_result'1)$.
9: **end if**

addition/removal of definitions).

In the future we would like to benchmark our system against those of others that work on a domain with a strong temporal dimension.

Chapter 4

Component Connection Model aggregation

The basic representation of the CCM has been extensively treated in chapter 3. But the CCM has only been discussed at the most basic or atomic level up till now. This chapter treats of the 'molecular' or aggregated components in the CCM representation.

There are two important reasons for defining aggregate components, a technical and a pedagogical one. The technical reason why we need aggregate components is to temper the complexity of the diagnostic operation. Since this is a task that is exponential in the number of components, abstracting multiple components away into a single bigger one has important computational merits.

The pedagogical reason why we need aggregate components, is that we want to be able to communicate mistakes that a learner makes at the right level of abstraction. Although the most extensive diagnosis always finds the smallest sets of faulty components at the atomic level, we may not always be interested in the low-level verdict. Because a pedagogical interaction is always a trade-off between the usefulness of the tool, i.e. giving the right culprits, and the investment required from the learner, i.e. the number of question that need to be answered, we want the diagnosis to be able to communicate higher-level mistakes without having to exhaust the diagnosis. In this way the system should be able to give usable results without requiring too many questions to be answered.

We allow arbitrary aggregations to be defined in our component definition specification language. An aggregate component is defined in the same way as a non-aggregate one, i.e. restrictions are put on the expressions of the connecting points and the spaces in which they reside. The only difference is that we put different restrictions on the spaces of the points. We can define arbitrary distances between the in- and output points of aggregate components. Moreover, we can use the value for this space delta inside the component function, thus making calculations based on the temporal parameter (see section 4.4).

4.1 Aggregation representation

In [19, 21] the aggregations were separate CCMs. This means a lot of information is duplicated. Also, aggregation were divided into distinct levels. This meant that when a single component was unpacked, from level L to level $L-1$, the information in multiple CCMs (i.e. the one at level L and the one at level $L-1$) has to be kept in sync.

In our implementation of aggregation, we assert the aggregate components directly into the CCM. This means that atomic and molecular components are represented in the same way. The diagnosis engine can choose between an atomic and a molecular component because

the links between points and components are annotated with a component delta value, which indicates how many atomic components are encompasses in a component. Based on this number, the diagnostic engine can choose which component to visit from a certain point.

4.2 Uninformed versus expectation-driven aggregation

Because of our reimagining of the task of diagnosis, allowing learners to expression expectations regarding the simulation results, the use of aggregations has changed as well. Since expectations can be expressed with respect to any point in the CCM, uninformed aggregations may not always be useful where abstraction is most crucial, i.e. near the expectation points.

If we look at the aggregation example in figure 4.1, we see that the aggregation becomes unusable when an expectation regarding Q_3 is expressed.

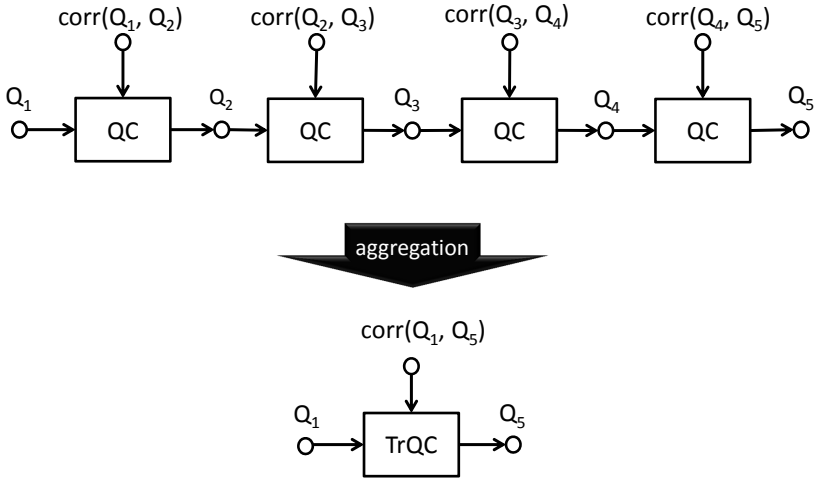

Figure 4.1: Example of uninformed aggregation.

The solution for this is to perform aggregations conditional on expectations. In figure 4.2 we see the alternative aggregation that takes the expectation regarding Q_3 into account.

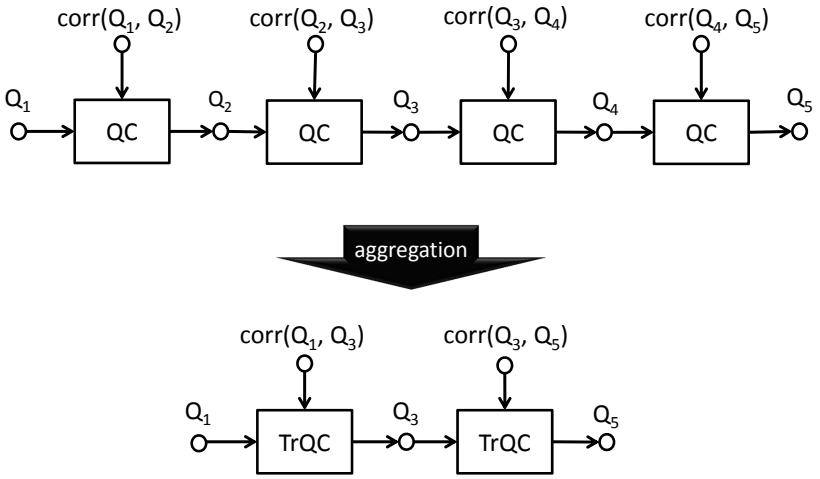

Figure 4.2: Example of expectation centric aggregation, for an expectation regarding Q_3.

4.3 Atemporal / within-state aggregation

We use the same within-state aggregations as in [19], which we replicated in component definitions specified in our dedicated component definition language (section 3.3.3). Since we allow arbitrary aggregation components to be defined, it is very easy for us to experiment with different types of aggregation components, opening the possibility for comparison studies. All aggregations that are defined in this section are due to [19]. Appendix .1 sections contains examples of new aggregation types that we have introduced. the most important thing to keep in mind is that we have now created an extensible architecture that allows any type aggregation to be defined. We think that affinity with the diagnostic use case will bring us closer towards the set of viable / usable aggregation types.

4.4 Temporal / between-state aggregation

Besides the existing atemporal or within-state aggregations of section 4.3, we also allow a new type of aggregation that does take time into account. Using the space delta property of component definitions (see section 3.3.3) it is possible to transgress spaces using a component. When a component transgressing spaces, this means that given the number of spaces that is traversed, the diagnostic engine is able to calculate the value of one unknown port under the assumption that all the other ports are known.

An example of temporal or between-state aggregation is given in figure 4.4. In this example there are two quantity terminations: the magnitude of a quantity changing from 'Zero' to 'Low' to 'Medium' under an increasing derivative. The bottom shows the same situation in aggregated form, where the two terminations have been replaced by a single one. Observe the temporal annotation in the support knowledge, i.e. $t = 2$.

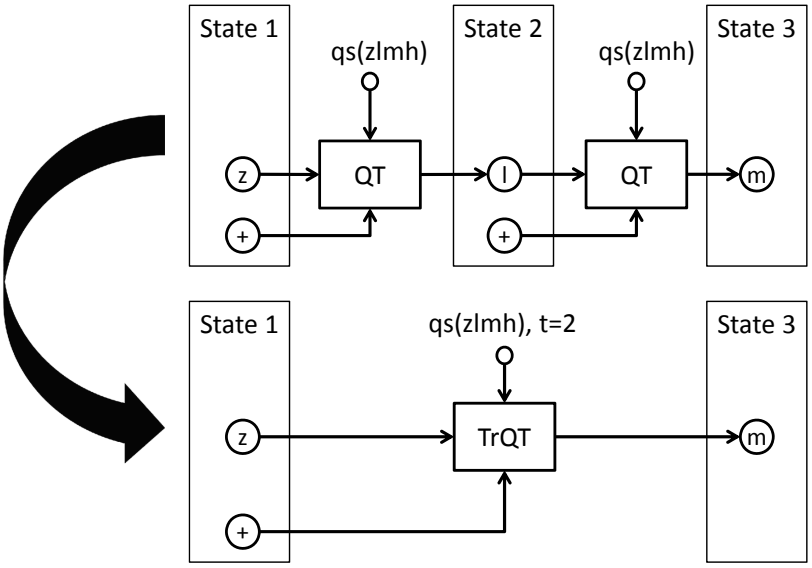

Figure 4.3: Temporal aggregation of quantity termination behavior. The top picture shows the non-aggregated situation. The bottom picture shows the aggregated situation.

Chapter 5

Diagnostic Component

In this chapter we explain how diagnosis works in our implementation of cognitive diagnosis. We make heavy use of the CCM representation in chapter 3 and the aggregations discussed in chapter 4. The core of the diagnostic algorithm is in line with [14]. The basic principles behind the ATMS are in line with [26]. However, we have made additions and alterations to (1) adapt the diagnosis to a new domain, (2) work with a learner's expectations, and (3) make the whole process more efficient.

Section 5.1 discusses the way in which expectations are expressed by the learner, as well as the way in which these are given to the diagnostic engine. Section 5.2 introduces the notion of an agent, which plays a role in our monitoring of a learner's consistency during diagnostic interactions. Our novel implementation of the ATMS, i.e. inside the CCM representation that is used by the diagnosis algorithm and not as a separate component, is detailed in section 5.3. We then discus the diagnosis algorithm itself, both its initialization, in section 5.4, and its main loop, in section 5.5. This is followed by important related processes like probe selection, section 5.5.1, and identification of end criteria, section 5.6.

5.1 Expressing expectations

Expressing expectations is an important component of the way in which our system provides modeling assistance. We want to put as few constraints on the expectations that a learning can express, while at the same time those expressions should be related to the task at hand. This is solved by allowing the learner to express any expectation that can be build using the vocabulary that is currently present in his QR model and the grammar for QR expressions.

We have two atomic expression schema's:

1. Quantity Q has [magnitude | derivative] V at state S.
2. In state S, the [derivative | magnitude] of quantity Q is [smaller than | equal to | greater than] the [derivative | magnitude] of quantity Q_2.

Observe that more complex expectations can be expressed in terms of these two schema's and the ordinary connectives. Negative expectations are formed by a disjunction of positive statements, etc. For example "Quantity Q never reaches its maximum value" would be expressed as "Quantity Q is *Zero*, *Small*, or *Medium* in states S_1 though S_8".

Expectations can be formed in a visual window that automatically enumerates the various entities, quantities, states, and relations in the current simulation. Figures 5.1 and 5.2 show screenshots of this window. Various expectations can be formulated and 'stalled'. After expressing any number of expectations, the diagnosis can be run.[1]

[1]The current way of expressing multiple or more abstract expectations is a bit dreary, i.e. stalling the

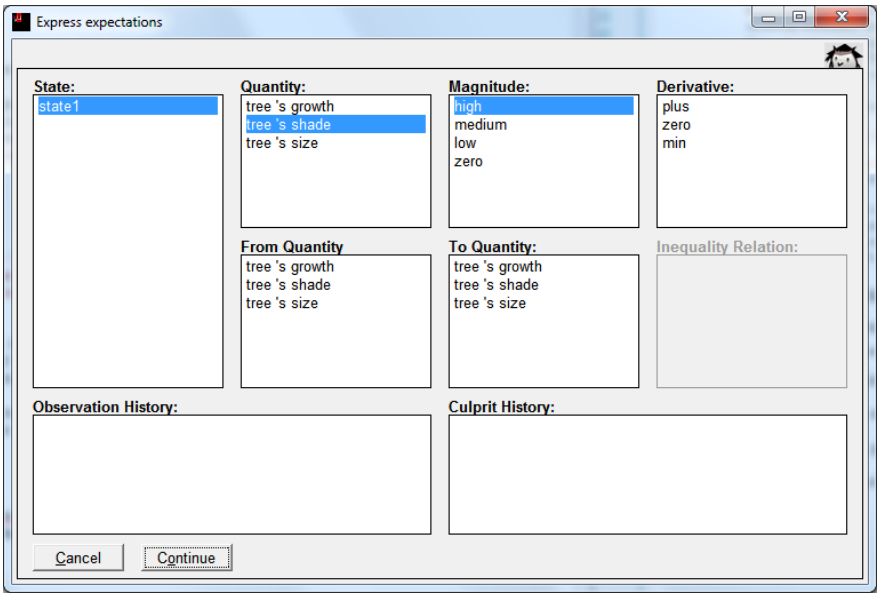

Figure 5.1: Expressing a magnitude expectation.

5.2 Agency: Representing the learner

Cognitive diagnosis does not only consist of a QR model, a simulation, a CCM and an ATMS (and interactions between them). The whole diagnostic effort is situated within a broader context in which the learner is interacting with the system. We added a learner agent to the CCM, as well as agent Garp, who relates to the simulation results. Because a learner is involved, notions of interest/consideration and (consistency of) belief play a role. We discuss both in turn.

5.2.1 Considering: scope and relevance

Oftentimes, getting to understand the dynamics of a complex system involves focusing on subparts of that system. A good teacher knows how to explain these subparts of a complex system one by one, making sure a learner understands each subpart in isolation, and only then shifts to explaining the whole system. In QR there are roughly two options for focusing on subparts, related to the build model and to the simulation results respectively.

One way to focus on particular subparts of a dynamic system is by altering the scenario to trigger less model fragments. Oftentimes, modeling a complex system involves having several scenarios, each adding more and more complexity (in terms of utilized model fragments). The build environment of DynaLearn already allows for this. The other way to focus on subparts of a dynamic system is by constraining the time segments that we look at. Suppose a simulation has a single linear behavior path of five states. It then makes sense to first focus on the changes that occur in the first state. Once the learner understands these, the focus is either replaced or enlarged to other states.[2]

In our case we do not want to explain a normative model to a learner. Still, it is relevant to think of the notion of scope when it comes to diagnosing a model. If the simulation has many states and/or much ambiguous behavior, it may be better to look for culprits in

various atomic expressions. In the future we want to extend or replace the current way of giving the expectations with a more versatile interface that uses representations close to natural language and that translates these higher-level expressions to their atomic equivalents.

[2]This approach was taken by [19].

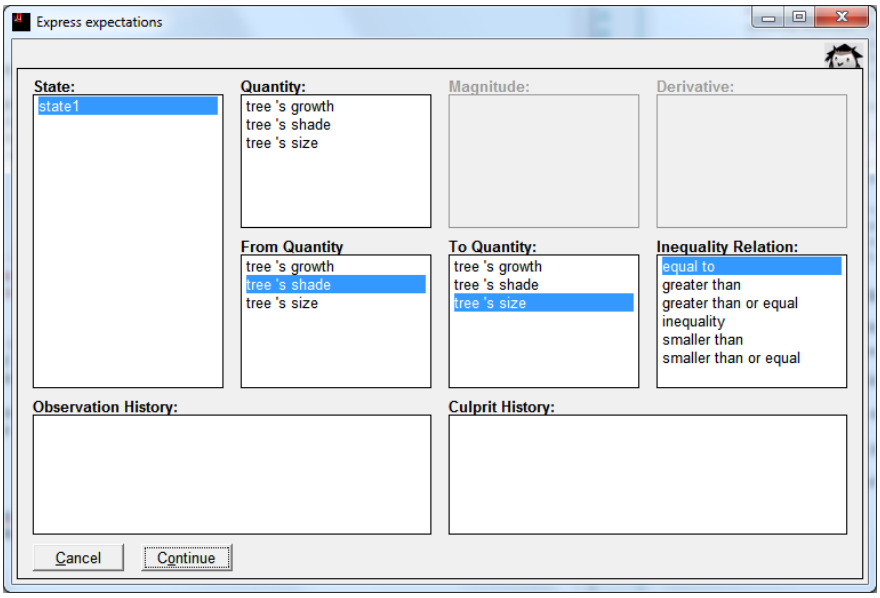

Figure 5.2: Expressing an inequality expectation.

a subset of states. We do this by identifying those states in which expressions have been expressed, and find all states that are within a certain distance of those states. We then only run diagnosis on those states. What this effectively amounts to is performing diagnosis with additional observations.

The points and components that lie within the scope are marked as considered by the learner agent. The observations are those points in the CCM that lie at the outskirts of the sets of considered points and components. From this it follows that what a learner agent considers is always the same as what the Garp agent believes (i.e. overlay model).

5.2.2 Belief: observations and consistency

The expectations a learner expresses, as well as its answers to probe questions, need not agree with the simulation results (for retrieving interesting diagnoses this need to be different). This means that contradictory expressions will be instantiated to different points that are related to the same component port. In order to distinguish the difference, these expectations and probe results are related to the agent who believes them (at this moment either the learner or the Garp agent).

A big problem during cognitive diagnosis is the occurrence of inconsistency on the part of the learner. Inconsistency is not always bad, since it may be induced by self-repair: during the diagnostic interaction a learner may suddenly grasp something she earlier did not understand. This causes her later expectations to no longer be in line with earlier ones she gave. Our system cannot distinguish between inconsistency induced by stupidity and inconsistency induced by self-repair. What our system can do is signal when expectations cause a contradiction, by checking for the consistency of each agent's belief contents, and end the diagnosis there.[3]

[3]We have not yet surveyed the full gamut of possibilities that the agent infrastructure allows. Applications of epistemic logic could prove to be relevant at some point.

5.3 Truth-maintenance

Our TMS is in line with [26]. The biggest distinction is that we do not have a separate TMS representation, but we update the labels of points and components directly asserted inside the CCM.

The labels describe the environments, i.e. sets of components, under which points are the case and components are functioning correctly.[4]

The advantage of integrating CCM and TMS, apart from avoiding the duplication of knowledge, is that it becomes computationally trivial for the diagnosis algorithm to detect conflicts. We add conflict nodes in the CCM that relate conflicting points that can be derived under certain environment assumptions. Since conflicting points are automatically related because of their being related to the same component ports, conflict nodes are added in constant time. The important characteristic of the procedure is that, given any environment, the conflicts that occur are given in constant time as well.[5]

5.4 Diagnosis initialization

Before each round of diagnosis the diagnosis needs to initialize. The scope is determined by establishing which states, points, and components are considered by the learner agent. This scope also plays a role in establishing the relevant components and points.

Irrelevant components and points are networks of connected points and components that either lie not within the scope, or that do not contain an expression point (see algorithm 4).

Algorithm 4 $Scope(P_{exp}, d)$, establishes which components and points are relevant for a learner.

Require: The expectation points, P_{exp}.
Require: The diagnostic depth d.
Require: The function from points to spaces, $f^{p2s} : P \to S$.
Require: The function from spaces to components in that space, $f^{s2c} : S \to \mathcal{P}(C)$.
Require: The function from points to directly connected components, $f^{p2c} : P \to \mathcal{P}(C)$.
Require: The function from components to directly connected points, $f^{c2p} : C \to \mathcal{P}(P)$.
1: $S_{exp} \leftarrow \{f^{p2s}(p_{exp}) | p_{exp} \in P_{exp}\}$
2: $S \leftarrow$ all spaces in vicinity d of S_{exp}.
3: $C \leftarrow \bigcup_{s_i \in S} \left(f^{s2c}(s_i)\right)$.
4: **for all** $p \in P_{exp}$ **do**
5: $\quad C_{rel} \leftarrow f^{p2c}(p) \cap C$.
6: $\quad$ **for all** $c \in C_{rel}$ **do**
7: $\quad\quad$ Agent a considers component c.
8: $\quad$ **end for**
9: $\quad P_{rel} \leftarrow \bigcup_{c \in C_{rel}} f^{c2p}(c)$.
10: $\quad$ **for all** $p \in P_{rel}$ **do**
11: $\quad\quad$ Agent a considers point p.
12: $\quad$ **end for**
13: **end for**

Based on the expectations and observations, all irrelevant parts are deconsidered. This is done by only including points and component that are traversed while moving from expectations to observations. All points and components that are not traversed are not part

[4] A point is the case if its expression is the case at the point's state.

[5] Since the temporal complexity of many existing implementations is unknown, we cannot be sure whether this is an improvement. We do significantly improve on [19] however, which was $O(|Points|^2)$.

of the agent's considerations. It is important to note that these elements are not removed, they are still in the CCM, but they do not have the 'consider' relationship to the active learner agent (section 5.2). These elements are therefore not excluded from possible future diagnostic iterations in which they may be considered (e.g. because of added expectations).

Based on the considered components C_{con} and considered points P_{con}, the points in set 5.1 are added as premises to the ATMS. For each considered component, an assumption is added to the ATMS.

$$\{p \in P_{con} | \neg \exists c \in C_{con}(f^{c2p}(c) = p)\} \tag{5.1}$$

Using scope in this way can be show to influence the results of the diagnosis. A simple example is given in figure 5.3. There are three quantities: Q_1 influencing Q_2, influencing Q_3. All quantities have quantity space $\langle \text{Zero}, \text{Plus} \rangle$. The grey points in state 1 are the observations that the diagnosis starts out with.[6] The grey point in state 4 is the expectation that Q_3 is 'Zero' in state 4. Calculating the thing through, it is apparent that Q_3 is in fact 'Plus' in state 4. The deviating values in the figure are calculated for assumptions $C - \{QI_3\}$ (with C the set of displayed components; QI_3 is distinguished by a dotted border). Given the behavior for the quantity influence component, it is clear that C is a conflict set.

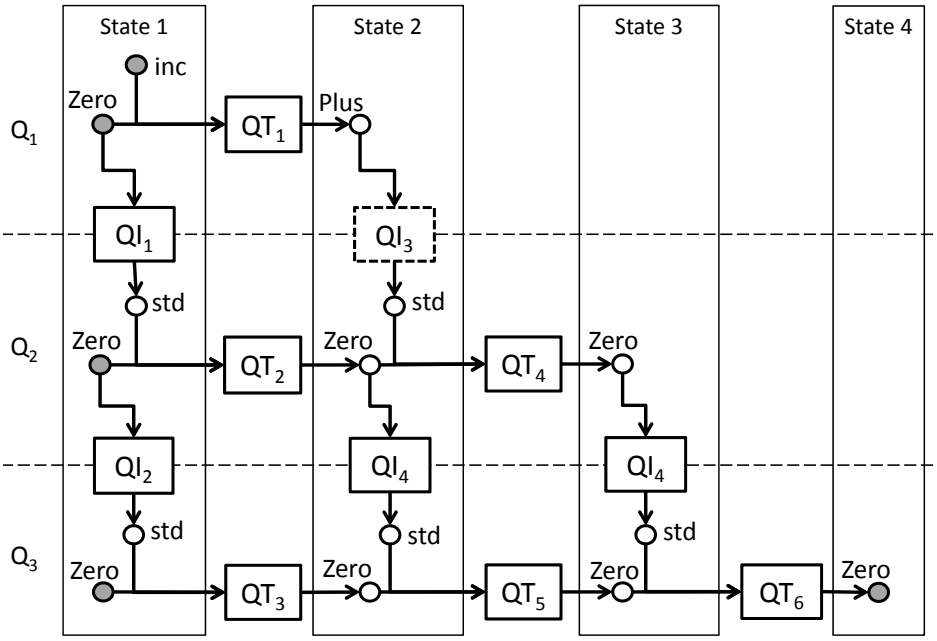

Figure 5.3: Diagnosis results without scoping, for environment $C - \{QI_3\}$.

When we use diagnosis depth $d = 3$, then the scope consists of state 4 (the space in which the expectation is expressed), the transitions to that space ($d = 1$), all states directly preceding state 4 ($d = 2$), and the transitions to those states ($d = 3$). Because of the scope we now have different observations, though the expectation is the same. This is displayed in figure 5.4. Under this scope the diagnosis will still give results, e.g. environment $\{QT_4, QT_5, QI_4, QT_6\}$ is now a conflict set. But it is not possible to find a conflict that includes the influence between Q_1 and Q_2.

The scope is useful to the extent that is focuses a learner's diagnostic activity. The assumption is that discrepancy that appear close to the expressed expectation are more rel-

[6] Additional elements like the scenario components and scenario space, the transition spaces, the support knowledge components and generic space, are ignored here.

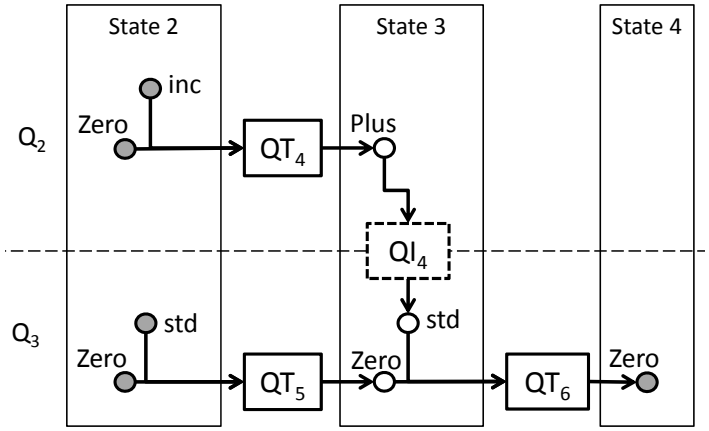

Figure 5.4: Diagnosis results with scoping, for environment $\{QT_4, QT_5, QT_6\}$.

evant than discrepancies that occur at a greater distance. What out diagnosis should not do it neglect conflicts. This is assured to not occur, by gradually increasing the diagnosis depth parameter when no conflicts are found between scoped simulation results and a learner's expectations.

5.5 Diagnosis algorithm

The main diagnosis loop, shown in algorithm 5, starts with generating the next environment based on the previous one (line 3), starting with the empty environment (line 1). Since we start with the empty environment, $NextEnv$ gradually increases the environments, and we are only interested in minimum conflict sets, we know that a big portion of the exponentially big environment space will never have to be traversed.

The details for function $CalcVector(env, c)$ (line 5 in algorithm 5) are given in algorithm 6. This function gives the points for all ports of component c that have known values under the assumption that $env - \{c\}$ are correct. It also gives the single port unk whose expression is not known under these assumptions. As is seen in line 5 of algorithm 6, we only consider cases in which a single port is unknown.[7] As line 12 shows, we only need to consider cases in which $|P_i| \leq 1$, since we know that $env - \{c\}$ is consistent. (If $env - \{c\}$ were inconsistent, then env would not have been generated in the first place.)

Going back to explaining the main diagnosis algorithm 5, we see that rule 6 returns the property of the unknown expression (determined by $CalcVector$) that is changeable. Each expression has exactly one changeable property. For instance, in expression "The magnitude of Q is v," only the magnitude value v can change. This means that the behavior rules for diagnosis cannot suggest a different quantity Q. In the expression "Q_1 influences Q_2," only the relation 'influences' can be changed, but not the quantities Q_1 and Q_2.[8]

The algorithm now checks whether a value can be calculated for the unknown point that $CalcVector$ returned (line 7). If the expression does not already exist, it is automatically added to the CCM (line 10). If there does not exist a point for this expression, then that is created as well (line 13).[9] This point is appropriately connected to the component.

When a new point conflicts with existing points, then relations between these points

[7] It is possible to extend the number of unknown ports in principle, but in QR very few behavior functions exist that could at once determine two unknowns.

[8] This is due to the structural persistence assumption, see section 6.5.

[9] The space of the point is derived from the component definition.

Algorithm 5 *Diagnose*(C), the main diagnosis.

Require: The set of components C.
Require: The function that gives the environment following the given environment, $NextEnv : \mathcal{P}(\mathcal{P}(C)) \rightarrow \mathcal{P}(\mathcal{P}(C))$.
Require: The function that gives tuples of known and unknown points, $CalcVector$.
Require: The function that calculates the value of an unknown point based on the values of the known points, $Behavior$.
Require: The function from component assumptions to conflicts, f^{ca2c}.

```
1:  env' ← ∅
2:  repeat
3:    env ← NextEnv(env')
4:    for all c ∈ env do
5:      while CalcVector(env, c) ≠ void do
6:        ⟨⟨c, rel, s⟩, p⃗⟩ ← CalcVector(env, c)
7:        prop ← f^{r2p}(rel), the property of the expression that is changeable.
8:        v_prop ← Behavior(c, p⃗)
9:        if There is no expression e with value v_prop for property prop then
10:         Add e to CCM.
11:       end if
12:       if There is no point p for expression e in state s then
13:         Add p to CCM.
14:       end if
15:       Update p's label with env.
16:     end while
17:   end for
18:   if f^{ca2c}(env) ≠ ∅ then
19:     Add env to the conflict set.
20:   end if
21:   env' ← env
22: until env = C
```

and a newly created conflict node are also added. Because conflicting points have the same expression definition and are connected to the same component, the search for such conflicting points is of low complexity (section 5.3). Since we are only interested in minimal conflicts, environments that do not give conflicts are not stored, ensuring that the number of environments that have to be asserted is relatively small.

The calculated expressions and points are added to the CCM if they are not present already (due to the dynamic extensibility of the CCM, described in section 3.6). Newly added points are not believed by any agent, but their are true under the environment condition. Existing points (which can but need not be believed by any agent) have their labels updated with the environment that allowed these points to be calculated. This happens in the CCM directly (section 5.3).

For every value that is calculated by the main diagnosis algorithm, a TMS justification is added as well. The justification has antecedent-relations to the component and the known points. The justification has a consequent-relation to the unknown point. If the value was not yet computed in another way, the antecedent of the justification and the label are the same. If the value was already derived in another way, then the existing label is updated based on the justification's antecedent.

The current environment is checked for conflicts (lines 18-19). We simply check whether

Algorithm 6 $CalcVector(env, c)$, calculates the point vector for the given component and environment.

Require: The function from component to component-to-state relations, $f^{c2r} : C \rightarrow \mathcal{P}(Rel)$.
Require: The function from component assumptions to points whose expressions are true under those assumptions, $f^{ca2p} : \mathcal{P}(C) \rightarrow P$.
Require: The function from components to directly connected points, $f^{c2p} : C \rightarrow P$.
Require: The function from components and component-to-point relations to states in which the related to points should be places, $f^{r2s} : C \times Rel \rightarrow S$.

1: **for all** $rel \in f^{c2r}(c)$ **do**
2: $P_{rel} \leftarrow f^{ca2p}(env - \{c\}) \cap f^{c2p}(c, rel)$
3: $s \leftarrow f^{r2s}(c, rel)$
4: **if** $P_{rel} = \emptyset$ **then**
5: **if** unknown $= 1$ **then**
6: **return** void
7: **else**
8: unknown $\leftarrow$ unknown $+ 1$
9: $unk \leftarrow \langle c, rel, s \rangle$
10: Store unk.
11: **end if**
12: **else if** $P_{rel} = \{p_{rel}\}$ **then**
13: Store p_{rel}
14: **end if**
15: **end for**
16: $\vec{p} \leftarrow p_{rel_1}, \ldots, p_{rel_{ar(c)}}$
17: **if** unknown $= 0$ **then**
18: **return** void
19: **else**
20: **return** $\langle unk, \vec{p} \rangle$
21: **end if**

the environment has been stored as URI and what conflicts are directly related to it. Based on the conflicts, the the candidate set are created. This is done by set covering, resulting in the sets that have a nonempty intersection with all the conflict sets. If there are multiple candidates, then candidate discrimination is run to find the most informative probe points in order to discriminate between these candidates.

5.5.1 Probe point selection

The process for determining the probe point is the result of finding a balance between finding the optimal probe point (i.e. the most informative one) and performing the calculation in acceptable time. The optimal entropy-based calculation from [26] is computationally unfeasible. We therefore make do with a relatively simple alternative.

Given that $COMP$ denotes the set of components and $CAND$ the set of candidates, we define the weight function $w : COMP \cup CAND \rightarrow \mathcal{R}$. The weight of a component C is defined in terms the weight of a candidate CC, see equation 5.2.

$$w(C) = \sum_{CC \in CAND} \frac{C \in CC}{w(CC)} \tag{5.2}$$

In equation 5.3, the weight of a candidate is defined in terms of the number of retrieval components $RETR$, the number of aggregate component types $AGGR\_TYPE$, and the number of primitive component types $PRIM\_TYPE$ (excluding aggregate types).

$$w(CC) = .7 \times RETR + .5 \times AGGR\_TYPE + PRIM\_TYPE \tag{5.3}$$

5.6 End criteria

In diagnosis there is the formal end criterion, which is reached when no additional probe points can contribute to the identification of additional diagnoses (and the scope has already been broadened to include all states). The assurance that all faults will be found that can be identified given the knowledge available, was an important theoretical characteristic of [14].

But in the practical appropriation for educational purposes, it is not always necessary to run the diagnosis to an end. We need not always find *all* discrepancies between simulation and expectations. An example of this was the scoping discussing in section 5.2.1. But there are other ways in which a premature result would be preferred to a long-winded interaction.

If a components occurs in every diagnosis up till now, then we know for sure that this component explains a discrepancy. We call these component that occur in every diagnosis *early culprits*. They are determined after each iteration of the main diagnosis algorithm. We display these early culprits inside the UI window that is used for filling in the expectations ans probe points in a non-obtrusive way. clicking on any of these early culprits points the learner to the corresponding aspects of the build model.

Since the learner can cancel a diagnostic interaction at any point, she can choose to end the diagnosis prematurely after an early culprit helped her identify an important inconsistency. Alternatively, the learner can choose to provide the diagnosis with more information in the form of probe points (question & answer interactions) to figure out the additional culprits as well.

Chapter 6

Research topics & future work

We have build a system that provides modeling-assistance using consistency-based diagnosis. We think we have grouped long-existing and well-attested techniques in an exiting new way. In thinking up this new approach and implementing the resulting system we have come across several problems that need to be cleared before we can appropriate our methodology in a satisfying manner. In this chapter we will briefly describe the big issues that we are working on right now, and that should lead to interesting results in the near future.

6.1 Automatic test suite

Evaluating cognitive diagnosis in classroom situations or by experts are both costly, and the testing procedure has little systematic rigor (i.e. the types of mistakes that are made can be peculiar to the domain or topic at hand, and experts may not consider all the mistakes that students could possibly make). We have therefore created an automatic testing procedure for diagnosis. It is therefore a valuable driver for development and debugging purposes.

The automatic test suite take a given model $\mathcal{M}$ as starting point, and performs all possible model alternations. For this the rules that are defined for the automatic repair component are used, see section 6.6. The test difficulty is the number of alteration rules that is performed on the model. The altered model $\mathcal{M}'$ is simulated $\mathcal{S}'$ and compared to the simulation results of the original model $\mathcal{S}$. Expressions that are true in $\mathcal{S}$ but false in $\mathcal{S}'$ are given as expectations for a diagnostic interaction.

The diagnosis proceeds in the usual way and asks for additional knowledge by probing. The answers to these probe questions are the values in the original simulation $\mathcal{S}$. The automated test is successful if the culprit the diagnosis found points to the model ingredients that were altered between $\mathcal{M}$ and $\mathcal{M}'$.

The problematic part of automatic testing is that small changed between models $\mathcal{M}$ and $\mathcal{M}'$ can induce big changes in the simulations $\mathcal{S}$ and $\mathcal{S}'$. This is currently solved by writing a mapping algorithm, but more advanced versions of this are needed in order to assure that two simulations always match so as to ensure an optimal diagnostic interaction.

6.2 Open issue: persistence over time

In the current implementation we run each diagnostic cycle in isolation. It is up to the learner to process the culprits (or their equivalents in the build model), make changes to the QR model, and run simulation (and potentially diagnosis) again. This system could be extended by a module that makes use of patterns of previous interactions. For instance, if a

learner gets the same types of culprits in multiple diagnoses, the module could register this and provide feedback accordingly.

6.3 Open issue: communicating culprits

The diagnosis gives the problematic components, i.e. the culprits. These components belong to the Component Connection Model that was build based on the simulation results of the QR modeling environment. However, the learner is creating a QR model in the build environment of the DynaLearn ILE. The culprit components are not (always) evidently translatable to QR elements in this build model. There are several reasons for this.

First of all, the simulation results are temporally distributed, while the build model only characterizes the starting state of a system. This means that inherently temporal components, e.g. magnitude terminations, do not have corresponding model elements.

Secondly, culprits often consist of multiple component. Provided the translation to QR elements in the build model is unproblematic, the question is how to communicate these model elements to the learner (and whether to communicate all of them or just a wisely chosen subset). Aggregate components relate to this problem, since they by definition represent multiple QR elements.

Thirdly, because model fragments can be (multiple times) imported into another model fragment, it is not always evident in which model fragment a QR element that relates to a culprit component should be located. E.g. it could be that the QR element is wrong in the original model fragment where it was introduced, but it could also be the case that the QR element is only wring in an imported model fragment (where it conflicts with other aspects of the importing model fragment).

6.4 Open issue: extensions to the QR component hierarchy

We would like to add some QR elements to the CCM for the first time, and use diagnosis to find inconsistencies in simulations containing the. These QR elements are most prominently value correspondences and various calculi relations. The problem is that these QR elements have very weak backward reasoning rules, i.e. given the values of all but one port it is still not possible to establish the value of the single unknown point. Suppose two quantities have quantity space $\langle \text{Min}, \text{Zero}, \text{Plus} \rangle$, the former having magnitude 'Min' and the latter magnitude 'Plus'. The relation $+(\text{Min}, \text{Plus}) = X$ can then have three different values for X.

Under some circumstances we are able to determine the value of a port unambiguously for these components. However, in order to calculate these values we need knowledge that does not directly relate to the component in question. This effectively amounts to violating the contextual independence assumption of component behavior definitions.

6.5 Open issue: the structural persistence assumption

At present, our diagnosis system allows expectations and observations to be given in which values and relations differ from the expressions that the Garp simulation delivered. However, we have not yet allowed the structure of the model to be altered.

Suppose there should be a positive proportionality from quantity Q_1 to quantity Q_2, but there is a negative proportionality from the former to the latter. Both the sign and the directionality of the relation may be corrected by diagnosis. However, if there is no relation

between Q_1 and Q_2 whatsoever, then the CCM will contain no component whatsoever between points having expressions regarding Q_1 and Q_2's derivative values. The diagnosis will then never suggest there should be a 'different' component between Q_1 and Q_2 (since there is no component between them in existence to begin with).

This problem is due to the assumption that all discrepancies are explainable in terms of behavioral aspects solely. However, we would like to give feedback to the learner regarding structural changes he or she should bring to his/her model as well.

6.5.1 Open issue: expressing expectations on non-existing states

An instance of the problem related to the structural persistence assumption, is that learners can has expectations regarding a state that does not exist in the Garp simulation results. E.g. if a learner expects the simulation to have 5 states, and there is only 1 state simulated, then it unclear how the learner should express expectations about the absent fifth state.

6.6 Extensions: Automatic repair

We have developed a repair method for automatically improving a given model $\mathcal{M}$ based on expectations Exp the modeler has. Finding the correct repair is searching the space of all possible models, which are collections of FOL sentences. Transitions between models are established by applying model alteration rules 6.1. These alteration rules can be instantiated by a constant vector $\vec{d}$ from the current model. Instantiated rules $r(\vec{d})$ that meet the criteria stated in their condition, are executed by removing and adding FOL rules, see equation 6.2.[1] Applying an instantiated rule to a model always results in a model.

$$r = \lambda\vec{x}.\begin{cases} \phi(\vec{x}) \text{ conditions} \\ \langle A^+, A^- \rangle \text{ actions} \end{cases} \tag{6.1}$$

$$r(\vec{d})(\mathcal{M}) = (\mathcal{M} - A^-) \cup A^+ \tag{6.2}$$

Given the pair of model and expectations, there can obviously be multiple repairs, i.e. sequences of instantiated model alteration rules. Goal nodes are models $\mathcal{M}$ such that $Simulate(\mathcal{M}) \vdash Exp$. The search algorithm is really very simple, the problem is to find the good heuristic. In an educational context it is important to provide the user with the improvement that is the most useful given the current state of his model. This is where the diagnosis plays a crucial role; its results are used for the aposteriori weights in the heuristic function. The aposteriory weights are provided as expert knowledge regarding the error-proneness of edit rules (e.g. learners swap arguments or signs quite often).

6.7 Extensions: Causal explanation as CCM verbalization

The CCM we build for running the diagnosis on is a very strong representation. During the construction of the CCM generation procedure, we tried to be nice and added explanatory labels to all the created nodes.[2] In order to make recursion run its course, the annotations of node representing complex concepts were generated based on the annotations of nodes representing simple or less complex concepts. For instance, the annotation for an expression depends on the annotations for the atoms that constitute the expression.

We did the same for components, adding easy to understand descriptions of the kind of deduction they perform, based on the in- and outputs that they have. To our surprise,

[1] We made it easy to add new rules, allowing arbitrary conditions and actions to be formulated.

[2] RDFS provides default annotation capabilities for annotating nodes with human-readable information.

this representation works remarkably well in order to explain qualitative reasoning. If the learner wants to know why some expression holds true in the simulation results, we identify the node in the CCM corresponding to that expression. Then we run backward through the CCM to explain the way in which the expression was derived. Because all the annotations are generated automatically, using a very simple CFG gives acceptable natural language explanations.

Explanations go one step deep, i.e. explaining one component or deduction. Because of the existence of aggregate components, it is possible to give an explanation in abstract terms. The learner can then choose to either receive an explanation that goes one step deeper, or an explanation of the high-level deduction, in case the component is an aggregate component. In this latter case the lower-level components are explained.

Chapter 7

Conclusion

In this deliverable we have detailed our use of consistency-based diagnosis for providing automated assistance in knowledge modeling. This attested technique can be used to support active learning. Since QR knowledge construction is an open-ended domain, this has posed several new challenges.

The results of a QR simulation engine have to be translated to a representation that can be used for diagnosis. The reasoning steps that have to be performed in order to make the necessary derivations, have been defined in context independent terms. The temporal aspect, which plays an important role in QR, was specifically addressed.

The definition of various component types (scenario values, expectations, descriptive, procedural), allows different types of reasoning to be distinguished. It also allows specifically tailored support to be delivered to the learner. Aggregation reduces the number of probe points, thereby shortening the interaction cycle. At the same time aggregation makes sure support is given at the right level of abstraction.

A diagnosis algorithm has been designed and implemented that does not require a normative or correct model, but instead takes a learner's expectations and/or modeling goals into account. Conflict recognition was implemented efficiently by integrating CCM and TMS representations. A mechanism for checking the consistency of the learner, during the diagnostic interaction, was given to deal with the notion of self-repair. Scoping and early culprits were used to focus the diagnostic interaction and allow the completeness of diagnosing multiple faults to be combined with the ability to keep the interactions with the learner short and efficient.

Because of its open data format, the diagnostic component can be easily integrated into existing Integrated Learning Environments (ILEs). Integration firstly requires the (automated) translation from the output of the ILE to the CCM data format, using the component and expression definitions that describe the ILE's domain. Secondly, it requires an interface for expressing expectation and giving answers to probe questions. This was done for the DynaLearn ILE, building on an improved version of the attested Garp reasoning engine.

Chapter 8

Appendices

.1 Overview of within-state aggregations

This appendix enumerated the within-state or atemporal aggregations that were already defined in [19] and that we took over in our new implementation.

Full correspondence

Two quantities that are different in the world may be exactly the same in a model. This is due to modeling abstractions make. For instance, when gravitational force is abstracted away, mass and weight are 'the same', in the sense of fully corresponding. This means that these two quantities always, i.e. in every possible behavior state, have both the same magnitude and derivative values. This is enforced by an undirected correspondence relation (on the full quantity spaces) as well as a bidirectional proportionality relation between the quantities.

The algorithm for finding fully corresponding quantities:

---

1. Find corresponding quantities:
   (a) Find all pairs of fully corresponding quantities.
   (b) Join the pairs of fully corresponding quantities into sets (of arbitrary size) of fully corresponding quantities.
2. Merge corresponding quantities:
   (a) Find affected expressions (i.e. those that mention at least one of the fully corresponding quantities).
   (b) Replace the fully corresponding quantities by aggregate quantities inside these affected expressions.
   (c) Merge all points that are about the fully corresponding quantities and that have equal expressions. Points with equal expressions have been created by the previous step, e.g. A and B are now [A,B] with the same expressions. Connect all the components' in- and outputs as well.
3. Create decompositions for identical and circular components (w.r.t. the new aggregated quantities).

Transitive aggregation

Atomic components with support expressions that have transitive relations can be aggregated into transitive components. The components that can be transitively aggregated are given in table 1.

Table 1: Transitive aggregations.

Atomic components	**Molecular component**
(Quantity Correspondence)+	Transitive Quantity Correspondence
(Quantity Proportionality)+	Transitive Quantity Proportionality
(Inequality Correspondence)+	Transitive Inequality Correspondence
(Inequality Proportionality)+	Transitive Inequality Proportionality

Predecessor aggregation

Predecessor aggregations are aggregations based on an existing aggregation component and a component whose output is directly connected to the input of the existing aggregation component. Table 2 shows the predecessor aggregations.

Table 2: Predecessor aggregations.

Atomic components	**Molecular component**
(Quantity Correspondence, Quantity Influence)	Combined quantity influences
(Transitive Quantity Correspondence, Quantity Influence)	Combined quantity influence
(Value Determination, Quantity Influence)	Combined quantity influence
(Inequality Correspondence, Inequality Influence)	Combined inequality influence
(Transitive Inequality Correspondence, Inequality Influence)	Combined inequality influence
(Inequality Correspondence, Value Determination)	Combined value determination
(Transitive Inequality Correspondence, Value Determination)	Combined value determination
(Inequality Proportionality, Value Determination)	Combined value determination
(Transitive Inequality Correspondence, Value Determination)	Combined value determination
(Inequality Correspondence, Derivative Determination)	Combined derivative determination
(Transitive Inequality Correspondence, Derivative Determination)	Combined derivative determination
(Inequality Proportionality, Derivative Determination)	Combined derivative determination
(Transitive Inequality Correspondence, Derivative Determination)	Combined derivative determination

Successor aggregation

Successor aggregations are aggregations based on an existing aggregation component and a component whose input is directly connected to the output of the existing aggregation component. Table 3 shows the successor aggregations.

Table 3: Successor aggregations.

Atomic components	Molecular component
(Combined Quantity Influence, Quantity Proportionality)	Combined Quantity Influence
(Combined Quantity Influence, Transitive Quantity Proportionality)	Combined Quantity Influence
(Combined Inequality Influence, Inequality Proportionality)	Combined Inequality Influence
(Combined Inequality Influence, Transitive Inequality Proportionality)	Combined Inequality Influence
(Combined Value Determination, Quantity Correspondence)	Combined Value Determination
(Combined Value Determination, Transitive Quantity Correspondence)	Combined Value Determination
(Combined Derivative Determination, Quantity Proportionality)	Combined Derivative Determination
(Combined Derivative Determination, Transitive Quantity Proportionality)	Combined Derivative Determination

.2 Example of a component definition: quantity magnitude termination

This appendix shows the definition of the magnitude termination component. It is a child of the termination component definition. It's condition refers to its various points. These all have their own code (bottom of the code excerpt). It has quantity space expressions as support knowledge. The state delta `linear(positive_natural_number)` indicates that this component can describe quantity changes between any two linearly linked states. This definition can be instantiated.

```
<rdf:Description rdf:about="&component;magnitude_termination">
  <rdfs:subClassOf rdf:resource="&component;termination"/>
  <rdfs:label xml:lang="en">magnitude termination</rdfs:label>
  <component:has_support rdf:resource="&expression;quantity_space"/>
  <component:has_condition rdf:datatype="&xsd;code">
    and(
      neq(has_code(has_derivative_input(comp), 0)),
      neq(has_code(has_magnitude_input(comp), has_magnitude_output(comp)))
    )
  </component:has_condition>
  <component:has_space_delta rdf:datatype="&xsd;code">
    linear(positive_natural_number)
  </component:has_space_delta>
  <component:can_instantiate rdf:datatype="&xsd;boolean">
    true
  </component:can_instantiate>
  <component:has_magnitude_input rdf:resource="&cpr;mt_magnitude_input"/>
  <component:has_derivative_input rdf:resource="&cpr;mt_derivative_input"/>
  <component:has_magnitude_output rdf:resource="&cpr;mt_magnitude_output"/>
```

```
</rdf:Description>

<rdf:Description rdf:about="&cpr;mt_magnitude_input">
  <cpr:has_expression_definition rdf:resource="&expression;magnitude"/>
  <cpr:has_relation rdf:resource="&component;has_magnitude_input"/>
  <cpr:has_code rdf:datatype="&xsd;code">
    has_to_argument(store(
        has_magnitude_input,
        has_magnitude_expression(has_from_argument(supp))
    ))
  </cpr:has_code>
</rdf:Description>

<rdf:Description rdf:about="&cpr;mt_magnitude_output">
  <cpr:has_expression_definition rdf:resource="&expression;magnitude"/>
  <cpr:has_relation rdf:resource="&component;has_magnitude_output"/>
  <cpr:has_code rdf:datatype="&xsd;code">
    has_to_argument(store(
        has_magnitude_output,
        has_magnitude_expression(has_to_argument(supp))
    ))
  </cpr:has_code>
</rdf:Description>

ETC.
```

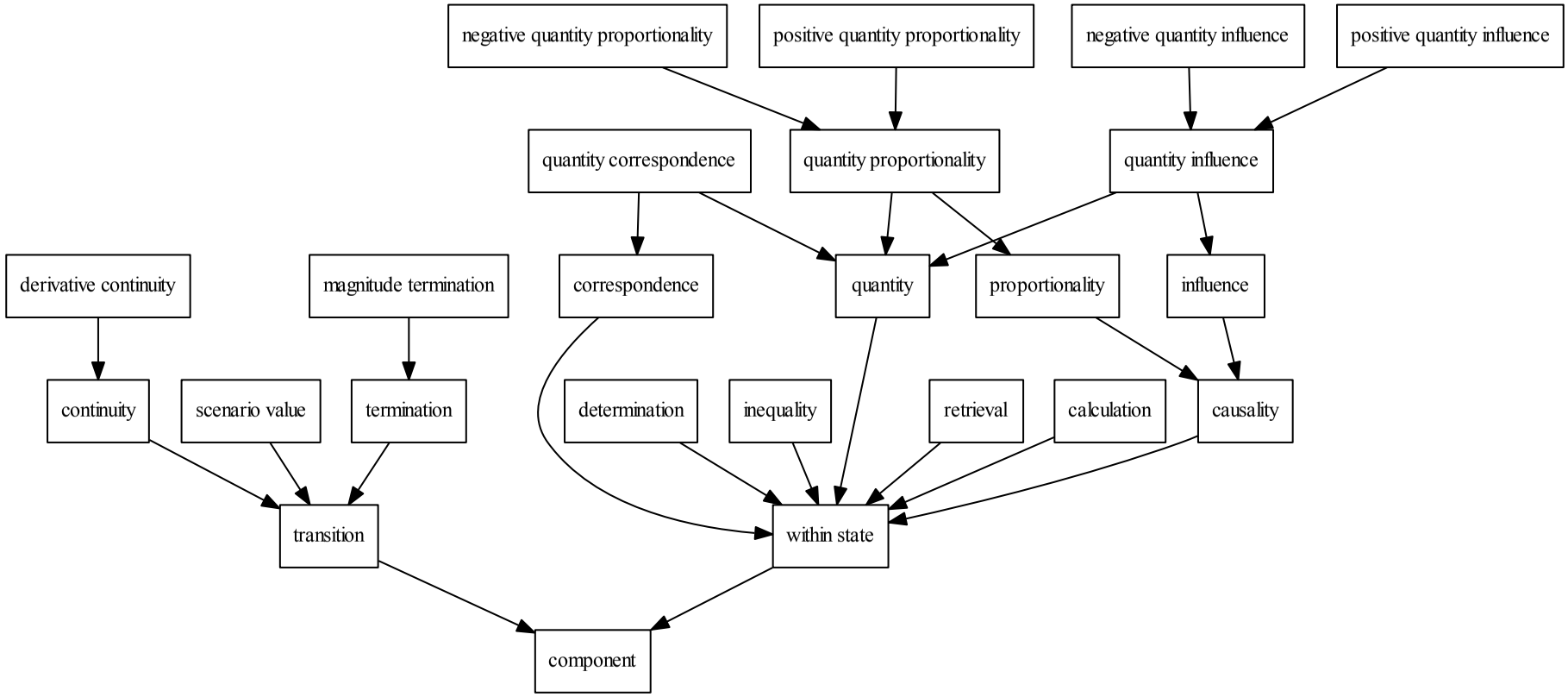

Figure 1: Component definition hierarchy.

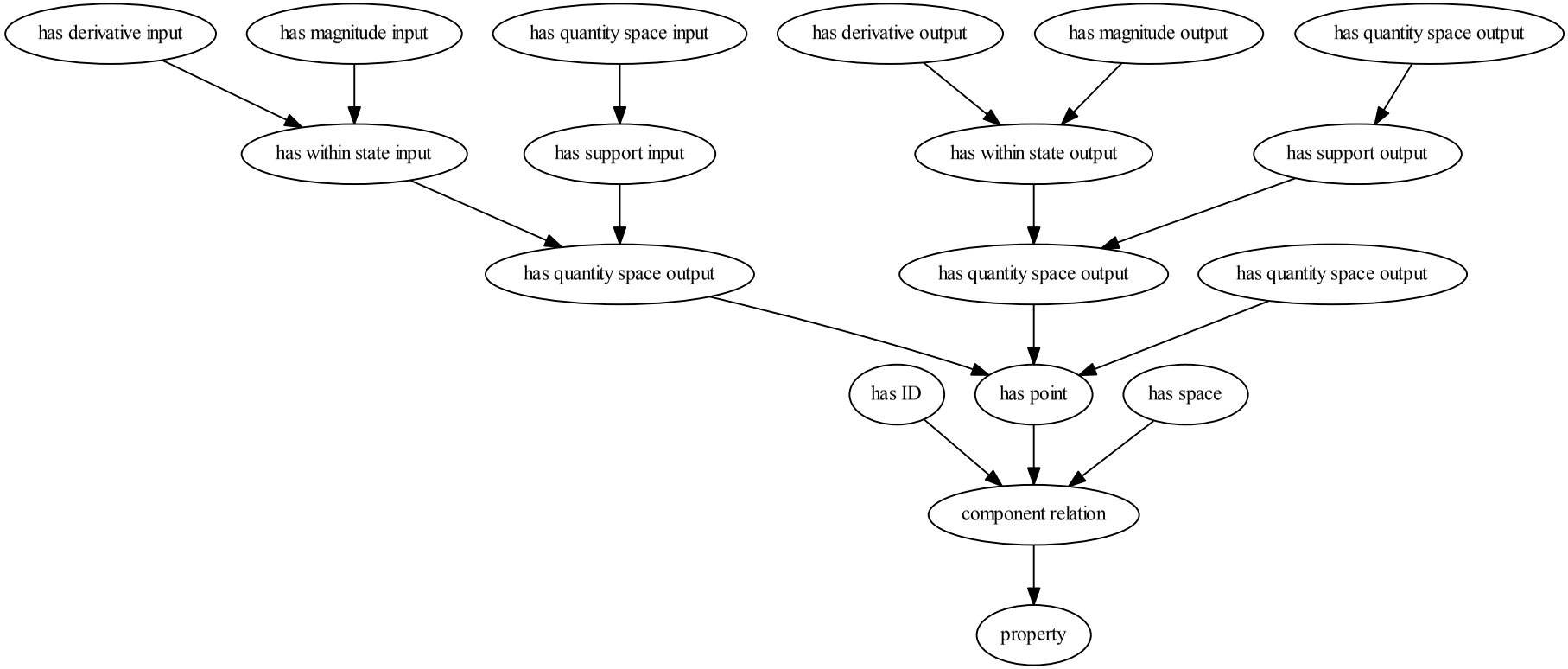

Figure 2: The component-to-point relation definitions. This is part of the relation definition hierarchy (see section 3.2.1). The hierarchy can be dynamically extended by adding new component point relations in the description of some component definition.

Bibliography

[1] Xml schema part 2: Datatypes, May 2001.

[2] John R. Anderson and Christian J. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum, 1998.

[3] Bert Bredeweg. *Expertise in Qualitative Prediction of Behaviour*. PhD thesis, University of Amsterdam, 1992.

[4] Bert Bredeweg, Floris Linnebank, Anders Bouwer, and Jochem Liem. Garp3: Workbench for qualitative modelling and simulation. *Ecological Informatics*, 4(5-6):263–281, November-December 2009.

[5] Bert Bredeweg and C. Schut. Cognitive plausibility of a conceptual framework for modeling problem solving expertise. In K. J. Hammond and D. Gentner, editors, *Proceedings of the 13th Conference of Cognitive Science Society*, pages 473–479. Lawrence Erlbaum, August 1991.

[6] D. Brickley and R. V. Guha. Resource description framework (rdf) schema specification 1.0, March 2000.

[7] J. S. Brown and Kurt van Lehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

[8] J. Bruner. *Toward a Theory of Instruction*. Harvard University Press, 1966.

[9] J. Bruner. *The Culture of Education*. Harvard University Press, 1996.

[10] J. R. Carbonell. Ai in cai: An artificial intelligence approach to computer-assisted instruction. *IEEE Trans. Man-Machine Systems*, 11(4):190–202, 1970.

[11] B. Carr and I. P. Goldstein. Overlays: A theory of modeling for computer aided instruction. AI Memo 406, AI Laboratory, Massachusetts Institute of Technology, 1977.

[12] L. Console and P. Torasso. Integrating models of the correct behaviour into abductive diagnosis. In *Proceedings of the ECAI Conference 1990*, pages 160–166, 1990.

[13] Donald Davidson. The logical form of action sentences. In *The Logic of Decision and Action*. University of Pittsburgh Press, 1967.

[14] Johan de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[15] Johan de Kleer and B. C. Williams. Diagnosis with behavioral modes. In *Proceedings of the 11th. IJCAI conference*, pages 1324–1330, 1989.

[16] Johan H. de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[17] Johan H. de Kleer. Extending the atms. *Artificial Intelligence*, 28:163–196, 1986.

[18] Johan H. de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28:197–224, 1986.

[19] Kees de Koning. *Model-Based Reasoning about Learner Behaviour*. PhD thesis, University of Amsterdam, 1997.

[20] Kees de Koning and Bert bredeweg. Qualitative reasoning in tutoring interactions. *Interactive Learning Environments*, 5:65–80, 1998.

[21] Kees de Koning, Bert Bredeweg, Joost Breuker, and Bob Wielinga. Model-based reasoning about learner behaviour. *Artificial Intelligence*, 117(2):173–229, 2000.

[22] DynaLearn. EC 7th FP project, no. 231526.

[23] R. Elio and P. B. Sharf. Modeling novice-to-expert shifts in problem-solving and knowledge organization. *Cognitive Science*, 14:579–639, 1990.

[24] M. Elsom-Cook. *Principles of Interactive Multimedia*. McGraw-Hill, 2001.

[25] J. Fiske. *Introduction to Communication Studies*. Routledge, 2nd edition, 1990.

[26] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[27] Teachable Agents Group.

[28] Patrick Hayes. Rdf semantics, February 2004.

[29] P. Holt, S. Dubs, M. Jones, and J. Greer. *Student Modelling: The key to Individualized Knowledge-based Instruction*, chapter The state of student modelling. Springer-Verlag, 1993.

[30] L. W. Johnson and E. Soloway. *Artificial Intelligence and Instruction: Applications and Methods*, chapter PROUST: An Automatic Debugger for Pascal Programs, pages 49–67. Addison-Wesley, 1987.

[31] G. Kearsley. Explorations in learning & instruction: The theory into practice database. tip, 1994-2001.

[32] R. Kowalski. *Logic for Problem Solving*. North Holland, 1979.

[33] Jochem Liem, Wouter Beek, Floris Linnebank, and Bert bredeweg. Api for data and knowledge exchange. Deliverable 3.2., UvA, 2010.

[34] Chee-Kit Looi, Gord McCalla, Bert Bredeweg, and Joost Breuker, editors. *Artificial Intelligence in Education: Supporting Learning through Intelligent and Socially Informed Technology*, volume 1258. IOS Press, 2005.

[35] S. Ohlsson. Some principles of intelligent tutoring. *Instructional Science*, 14:293–326, 1986.

[36] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980.

[37] J. Piaget. *The Science of Education and the Psychology of the Child*. Grossman, 1970.

[38] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.

[39] C. V. Schwarz and B. Y. White. Metamodeling knowledge: Developing students understanding of scientific modeling. *Cognition and Instruction*, 23(2):165–205, 2005.

[40] John Self. Model-based cognitive diagnosis. *UserModeling and User-Adapted Interaction*, 3:89–106, 1993.

[41] C. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[42] Peter Struss and O. Dressler. Physical negation: Integrating fault models into the general diagnostic engine. In *Proceedings of the 11th IJCAI Conference*, pages 1318–1323, 1989.

[43] L. S. Vygotsky. *Mind in Society*. Harvard University Press, 1978.

[44] E. Wenger. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Morgan Kaufmann, 1987.

[45] Jan Wielemaker. *Logic Programming for Knowledge-Intensive Interactive Applications*. PhD thesis, University of Amsterdam, 2009.

[46] Jan Wielemaker, G. Schreiber, and Bob Wielinga. Using triples forimplementation: The triple20 ontology-manipulation tool. In *ISWC*, pages 773–785. Springer Verlag, 2005.

e-mail: Info@DynaLearn.eu
website: www.DynaLearn.eu