Dissertation

SIMOL - A Modeling Language for Simulation and Reconfiguration

Iulia-Dana Nica

Graz, 2013

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: O.Univ.-Prof. Dipl.-Ing. Dr. Gerhard Friedrich

”The most expensive knowledge is less expensive than the lack of knowledge.”

- Grigore Constantin Moisil

Abstract (English)

Simulation and configuration play an important role in industry, from the early development stages, as well as after deployment, as part of system maintenance. The adaption of a system's structure or behavior over time becomes necessary because of changing requirements due to new customer needs, faults in system parts, or changes of technology among others. To assist these needs, many tools are nowadays available. Modeling languages like Matlab/Simulink or Modelica are often used to model the dependencies between the components of physical systems. However these are less suitable for the area of knowledge-based systems.

In our research, we are interested in providing a general methodology, where systems can be modeled and where the models can be used in order to change the systems, such that the new systems fulfill the desired functionality. Therefore, we have introduced SIMOL - an object-oriented language, that can be used for (restricted) simulation and reconfiguration at the same time. The key concept of SIMOL is the definition of basic and hierarchical components, which are used to represent the desired system. The behavior of a component has to be provided as set of equations. Furthermore, in SIMOL, it is possible to assign equations to specific behavior modes and also to model the systems behavior over time. Besides the language, we present here our reconfiguration algorithms, which make use of constraint solving and ideas from model-based diagnosis in order to compute system changes.

The implemented approach was intensively applied to the machine-to-machine communication domain. Machine-to-machine (M2M) communication is of increasing importance in industry due to novel applications, i.e., smart metering or tracking devices in the logistics domain. These applications provoke new requirements for mobile phone networks, which have to be adapted in order to meet these future requirements. Hence, reconfiguration of such networks depending on M2M application scenarios is highly required. In particular, within the *Simulation and Configuration of Mobile Networks with M2M Applications (SIMOA)* project, our objective was to develop a simulation and

reconfiguration environment for smart metering applications in order (1) to ensure that current mobile phone networks are capable of providing enough resources, and (2) to give advice for changing either the smart metering solution or the communication network in case of lack of resources.

Although the presented approach has been developed with focus on smart metering applications, it is not restricted to this domain. Any reconfiguration problem, that can be represented using the underlying modeling language SIMOL, can also be solved using the proposed approach.

Abstract (German)

Simulation und Konfiguration spielen eine wichtige Rolle in der Industrie, angefangen bei den frühen Entwicklungsstadien, sowie nach der Bereitstellung, als Teil der Systemwartung. Die Anpassung der Struktur oder des Verhaltens eines Systems wird im Laufe der Zeit notwendig, unter anderem aufgrund der wechselnden Anforderungen durch neue Kundenbedürfnisse, Fehler in Systemsteilen oder Änderungen der Technik. Um diese Anforderungen zu unterstützen, stehen heute viele Werkzeuge zur Verfügung. Modellierungssprachen wie Matlab/Simulink oder Modelica werden häufig verwendet, um die Abhängigkeiten zwischen den Komponenten der physikalischen Systeme zu modellieren. Diese sind jedoch für den Bereich der wissensbasierten Systeme weniger geeignet.

In unserer Forschung, sind wir daran interessiert, eine allgemeine Methodik zu bieten, mit der Systeme modelliert werden können und wo man die Modelle verwenden kann, um die Systeme zu ändern, so dass die neuen Systeme die gewünschte Funktionalität erfüllen.

Deshalb haben wir SIMOL eingeführt. SIMOL ist eine objektorientierte Sprache, die man zur gleichen Zeit für (eingeschränkte) Simulation und Rekonfiguration verwenden kann. Der Schlüsselbegriff der SIMOL Sprache ist die Definition der basis- und hierarchischen Komponenten, die man verwenden kann, um das gewünschte System zu repräsentieren. Das Verhalten einer Komponente muss als Satz von Gleichungen vorgesehen werden. Weiterhin ist es in SIMOL möglich, Gleichungen bestimmte Verhaltensmodi zuzuweisen und auch das Verhalten der Systeme im Laufe der Zeit zu modellieren. Neben der Sprache präsentieren wir hier unsere Rekonfiguration Algorithmen, die Constraint Solving und Ideen der modellbasierten Diagnose verwenden, um Änderungen am System zu berechnen.

Der implementierte Ansatz wurde intensiv in der Domäne der Machine-to-Machine(M2M) Kommunikation angewendet. M2M Kommunikation ist von zunehmender Bedeutung in der Industrie aufgrund der neuartigen Anwendungen, d.h., Smart Metering oder Tracking-Systeme in der Logistik-

Domäne. Diese Anwendungen provozieren neue Anforderungen für Mobilfunknetze, die angepasst werden müssen, um diese zukünftigen Anforderungen zu erfüllen. Daher wird die Netzrekonfiguration, die von den M2M Anwendungsszenarien abhängig ist, dringend benötigt. Insbesondere im Rahmen des "Simulation and Configuration of Mobile Networks with M2M Applications" (SIMOA) Projekts, war unser Ziel, eine Simulations- und Rekonfigurationsumgebung für Smart-Metering-Anwendungen zu entwickeln, (1) um sicherzustellen, dass die aktuellen Mobilfunknetzen in der Lage sind, genügend Ressourcen aufzubieten und (2) um Ratschläge für die Änderung der Smart-Metering-Lösung oder des Kommunikationsnetzes im Falle des Mangels an Ressourcen zu geben.

Obwohl der Ansatz bis jetzt in Smart-Metering-Anwendungen verwendet wurde, ist er nicht auf diese Domain beschränkt. Jedes Rekonfiguration Problem, das mit der zugrunde liegenden Modellierungssprache SIMOL dargestellt werden kann, kann auch mit dem vorgeschlagenen Ansatz gelöst werden.

Acknowledgments

First, I would like to thank my supervisor, Prof. Franz Wotawa, for all that he taught me during the last three and a half years, for always encouraging me, for all his help and confidence. Without his vision and without our intensive discussions, it would not have been possible for me to complete this work. I also wish to thank the members of my defence committee, Prof. Gerhard Friedrich and Prof. Helic Denis, for taking the time to review my thesis and to supervise the exam. Many thanks go to my industrial collaborators at Kapsch Vienna, for initiating me in the domain of machine-to-machine communication and for providing feedback on my work. I would also like to thank my colleagues at the Institute for Software Technology, especially Prof. Alexander Felfernig, for giving me the chance to write my first book chapter and for his valuable comments. I also express my gratitude to the *Austrian Research Promotion Agency (FFG)*, who funded the BRIDGE research project "Simulation and Configuration of Mobile Networks with M2M Applications" (SIMOA).

I wish to thank my parents, Cristina and Florin, for all their love, guidance, and support throughout the years and for instilling in me the desire to travel the world. I would like to thank my dear grandparents, for their endless affection and for the lovely childhood I had, together with my brother Petruț, whom I also thank for always being honest to me and for proofreading parts of this thesis. I have to thank my sister-in-low Simona, for all her help and for the time she has invested in reading the whole thesis. I would also like to thank Alina, for our long-lasting friendship, and Valentin, for always answering my many questions. Further thanks go to Simona, Cătălin, and Ionela, for their encouragements to finish the dissertation on time, and also to Regina, for her kindness and for proofreading my abstracts.

Last but surely not least, I am truly grateful to my beloved husband Mihai, for his strong support during my PhD, I thank him for being my "supervisor at home", for his patience and understanding.

Above all, I thank God for making all this possible.

Iulia-Dana Nica.
-Graz 2013.

Dedicated to my grandparents

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Contents

List of Figures

List of Tables

Chapter 1

Foreword

1.1. Motivation

From automotive and up to telecommunication industry, configuration and simulation are used for solving complex problems connected to the ever growing number of components, which have to work together. Generally, the purpose of simulating a system before deployment is to evaluate its design and behavior quickly and cost effectively. *Simulink* and *MathModelica* simulation environments are often used to model the dependencies between the components of physical systems and also their continuous or discrete time evolution. Both tools provide powerful capabilities, that facilitate complex, multi-domain simulations, based on the general modeling languages MATLAB and Modelica, respectively. However, these languages are mainly optimized towards simulation and therefore can be hardly used for reconfiguration.

Why would we need to reconfigure/adapt a given system? Reasons for adaptation could be, for instance, necessary corrections due to faults in system parts, changes in user requirements, or changes of technology among others. All activities necessary for increasing the lifetime of a system and retaining its usefulness are summarized under the general term maintenance. When considering the overall cost of systems during the whole lifetime, maintenance accounts for more than 50 percent. Any support provided for maintenance potentially reduces costs or provides improved results while retaining costs at the same level. Thus, the adaptation of technical systems after deployment to ensure the desired system's functionality over time is an important task, that can never be avoided. We will further focus

on system changes due to changes in requirements. For example, consider a cellular network where the base stations are initially configured to ensure current and future needs to some extent. Due to changes in the environment, i.e., new apartment buildings constructed in reach of the base station or an increased use of cellular networks for data communication, the base station or even the local topology of the network has to be adapted. A real life scenario in which reconfiguration is particularly needed is the overloading of electricity mains resulting in a blackout. Although many of the most powerful blackouts in history were caused by natural hazards, there were cases when the electrical grid was blown due to the large number of people using electrical appliances (India Blackout, July 2012, 670 million people affected). Blackouts can last from a few hours to weeks, depending on the cause and the infrastructure of the affected region, but regardless of how long they last, their consequences can be severe especially for modern economy heavily depending on the continuous availability of electricity. Therefore, emergency solutions to adapt the electricity network to the fluctuating needs have to be provided.

This adaption can more or less be classified as a reconfiguration problem, where the current system's structure, behavior, and the new requirements are given as input. Changes in the structure and behavior of the system in order to cope with the changes in the requirements are a solution of the reconfiguration problem. In order to provide a method for computing solutions for a given reconfiguration problem, we need to state the problem in a formal way. Therefore, we require a modeling language for stating systems comprising components and their relationships. In principle, formal languages like first order logic or constraint languages would be sufficient for this purpose, but using such languages usually is not easy and prevents systems based on such languages to be used in practice. Hence, there is a strong need for *easy to learn and use modeling languages*, that are expressive enough to state reconfiguration problems.

This thesis addresses some of the challenges from configuration and reconfiguration domains, considering also the requirements of a state-of-the-art simulation tool. The main contribution to the research of model-based reconfiguration is represented by the implemented modeling language SIMOL - a general, declarative, object-oriented language with discrete time modeling capabilities. To be more accessible for non-AI experts, we decided to adopt for it a Java-like syntax. Furthermore, we present our reconfiguration engine that makes use of constraint solving and ideas from model-based diagnosis in order to compute system changes. The approach has the following advantages: it is flexible and adaptable to various systems and needs, it determines the system's capabilities to handle certain requirements, and suggests system adaptations in cases where the requirements cannot be satisfied.

The work presented in this thesis was mostly conducted within the *Simulation and Configuration of Mobile Networks with M2M Applications (SIMOA)* project, where the objective has been to develop a simulation and reconfiguration environment for smart metering applications in order (1) to ensure that current mobile phone networks are capable of providing enough resources, and (2) to give advice for changing either the smart metering solution or the communication network in cases of lack of resources. We will report on the results obtained within SIMOA project in a dedicated chapter. It is also worth noting that only a flexible approach like model-based configuration is capable of handling the needs and requirements coming from this application domain.

1.2. Problem Statement

Given a system's structure, behavior, and new requirements on the system, we have to provide a method to complete the following tasks:

1. *state the problem in a formal way*, by means of a language capable of :
   - user-friendly modeling, based on constructs with widely accepted terminology,
   - describing the *dynamic behavior* of the system over discrete time,
   - defining physical units for a more realistic view of the running model,
   - natural modeling of the system's structure through partonomy and taxonomy relations and through components arrays,
   - defining *alternative functional modes* for the reconfigurable components/parameters of the system,
   - easy integration with various constraint-based solving engines;
2. *simulate* the constructed model in order to ensure that it can cope with the user's requirements;
3. *automatically deliver a system's reconfiguration*, if the *default simulation* fails. As already discussed, this function is of major importance as it ensures the system's availability even in an degraded mode. Here we make use of the functional modes, encoded in the components's behavior, that were not explicitly modeled in the user configuration, but are part of the components specification (in the knowledge base).

1.3. Contribution

The major contributions of our research include the idea of integrating simulation and configuration specific features and solving techniques in order to cope with a wide variety of systems and needs. Thereby, we have implemented a modeling language -SIMOL- that combines the two different directions. Due to its generality and high-flexibility in modeling a system, we have easily managed to apply SIMOL in different domains as sensor systems, combinational circuits, or mobile networks and with different purposes, from that of systems' simulation and reconfiguration to the one of diagnosing system's failures. Moreover, various optimality criteria can be specified in SIMOL and the integrated solving engine searches for reconfigurations fulfilling the optimality criteria. Due to the combinatorial nature of reconfiguration problems, a straightforward decision was to use a state-of-the-art constraint solver as reconfiguration engine.

The research presented in this thesis is based on and extends the following list of publications:

- *(Re-)configuration of Communication Networks in the Context of M2M Applications* [94]
- *Kapsch: Reconfiguration of Mobile Phone Networks* [96]
- *The Route to Success - A Performance Comparison of Diagnosis Algorithms* [97]
- *Model-based simulation and configuration of mobile phone networks- The SIMOA approach* [95]
- *ConDiag - computing minimal diagnoses using a constraint solver* [92]
- *The SiMoL Modeling Language for Simulation and (Re-)Configuration* [93]
- *Diagnosis-based Reconfiguration using the MINION constraint solver* [91]
- *SiMoL - A Modeling Language for Simulation and (Re-)Configuration* [98]

The remainder of this thesis is organized as follows. In Chapter 2 we provide an introduction in knowledge-based configuration, reconfiguration, and simulation domains and discuss related work. Afterwards, we introduce the SIMOL language (Chapter 3). In Chapter 4 we present our constraint-based approach for model-based simulation and reconfiguration, including the SIMOA tool architecture and the implemented reconfiguration mechanisms. Thereafter, Chapter 5 shows the results obtained when applying our techniques in the domain of mobile phone networks with machine-2-machine communication capabilities. Extensions of the applicability of the introduced language in

solving diagnosis problems are given in Chapter 6. Finally, with Chapter 7 we conclude the thesis, discussing also the limitations of the current approach and the future research directions.

Chapter 2

Preliminaries and Related Research

As preamble to our approach, we introduce in this chapter the most important concepts from the domain of configuration and simulation, with focus on modeling languages requirements (Section 2.2), configuration and simulation techniques (Section 2.3 and Section 2.4, respectively). Additionally, we state the terminology used in the current work (Section 2.1).

2.1. Terminology

In order to be self-contained, we review in this section some brief (informal) definitions and characteristics of the key terms encountered throughout this thesis and of those that our modeling and solving techniques are based on.

Modeling language: A modeling language is an artificial language used to represent a real-world or conceptual system/product, information, or knowledge in a structured way.

Simulation: ”Simulation is the imitation of the operation of a real-world process or system over time.” ([9])

Basically, there are three types of simulation: discrete-event, continuous, and Monte Carlo. Nevertheless, according to [9], in practice, these types are becoming less distinct.

Discrete time simulation: It is based on the notion of *discrete-event* simulation (DES)*, where the model (mathematical and logical) of a physical system has changes at precise points in simulated time ([88]).

Model-based simulation: The existence of a description for the system/product -the model- is assumed. "The purpose of model-based simulation is to evaluate a design quickly and cost effectively, so to be effective, useful, and not a waste of time, the model must accurately represent the actual system." ([140])

Discrete-event simulation model: "A discrete-event simulation model is defined as one in which the state variables change only at those discrete points in time at which events occur." [3]

State: The *state of an object/component* is the enumeration of all *attribute values* of that object/component at a particular instant.(adapted from [3])

Constraint satisfaction problem (CSP): A constraint satisfaction problem (CSP) has a finite set of variables, each with a finite domain, and a set of constraints, defining restrictions over those variables. A *solution* to a given CSP is an assignment of values to variables such that no constraint is violated.

Constraint solver: Typically, constraint solvers get as input an instance of a CSP and search for one ore more/all solutions to the problem. If no solution was found, the constraint solver may provide an explanation why we do not have any solution. Among the new, highly efficient constraint solvers on the market we can mention Gecode ([45, 122]) and JaCoP ([66]).

Knowledge-based configuration: Knowledge-based configuration is one of the most successful applications of expert systems, where the main constituent parts of a *configuration problem* are: (1) a predefined set of components, each component being described through a set of properties, ports (i.e., connection interfaces) and constraints, (2) the description of the desired configuration, and (3) optional criteria for optimizing the solution (adapted from [146]). Informally, a *configuration* (i.e., solution to the configuration problem) will be a set of components and a description of the connections between those components.

Another currently used model-based definition was proposed by [86], where two restrictions on the configuration task are introduced: the f*unctional architecture(s)* for the desired configuration and the *key components per function*.

*Although we do not use the term of *event* in our approach, we define it as "an occurrence that changes the state of the system"([9]).

Throughout the years, miscellaneous methods for solving knowledge-based configuration problems were developed. We further provide a short description of the basic concepts:

- *rule-based configuration*: Rules are used for the representation of both domain and problem solving knowledge. This leads to maintenance problems, as no proper separation between the two types of knowledge can be provided ([81, 126]).
- *structure-based configuration*: The compositional, hierarchical structure of the domain's model guides the configuration process, in which either a top-down method (through decomposition of the artifact) or a bottom-up strategy (through aggregations between components) can be adopted ([50, 52, 18]).
- *resource-based configuration*: Each component in the domain is described by the set of resources it supplies and the set of resources it consumes. "Given a set of open resource requirements (specifying the needs of a customer), the configurator adds components to the configuration until all requests of a resource are satisfied." [29] ([58, 59])
- *constraint-based configuration*: Constraints are used to define restrictions on the system and relationships between components, attributes or between a component and its attributes. Thus, the configuration problem can be naturally mapped to a CSP - also called a *static CSP*, when the number of components to be used in configuration is fix. Extensions to the classical CSP have been implemented, such as dynamic, composite or generative CSP [85, 116, 129], for improving the expressiveness of the knowledge representation and the solving mechanisms.
- *case-based configuration*: Computed configurations are stored for future usage. The reason is that, generally, similar tasks lead to similar solutions. According to [52], case-based technologies have a few disadvantages, as, for instance, they provide only conservative solutions or almost no causal explanations exist for the suggested solutions. Some examples of case-based systems are [62, 112, 37].

Reconfiguration: Generally, the term reconfiguration is used to refer to the situation, when "a configuration problem solver is confronted with an incorrect configuration". We can have two basic hypotheses for this situation: either the configured system does not (longer) provide the original desired function or some new desired requirements are not consistent with the originally configured system([131]). In our work, we focus on the case when the reconfiguration of a system is requested as a result of new requirements on the system. Hence, given the new requirements, the goal of the reconfiguration process is to find a configuration, where the number of changes is minimal.

Diagnosis-based reconfiguration: By exploring the analogy between diagnosis and reconfiguration, we are able to define the reconfiguration problem similar to the diagnosis problem. While diagnosis is the problem of identifying those components, which, when considered abnormal, justify the observed misbehavior, the reconfiguration can be seen as the problem of identifying those components, which, when changed, restore the desired behavior.

2.2. Modeling Languages for Simulation and (Re-)Configuration

System or product modeling represents a significant part both in the simulation and configuration process. Therefore, various languages have been developed over time, aiming at creating an expressive, but also effective model for the given problem. In this section, we review first the general requirements for a modeling paradigm in the context of simulation, and afterwards the desired attributes of languages, supporting the modeling of configuration problems. Moreover, the presented modeling concepts will be briefly exemplified.

In the history of simulation, one of the pioneers was the *Simula* language, specially designed for simulation [20] and considered to be the first object-oriented programming language, as the specific simulation domain's needs provided the framework for many of the features of nowadays object-oriented languages. Among others, it used the concepts of *object, class, subclasses, virtual methods, coroutines, garbage collection* and *discrete event simulation*. Bernard P. Zeigler introduced in the 1970's the DEVS (Discrete Event System Specification) formalism - a formal approach for building models, making use of a hierarchical and modular approach, that was lately integrated with object-oriented programming techniques [150]. This method would allow to build a *model base*, permitting easy reuse of models that have been validated. Later on, Sargent [118] outlined fourteen (14) requirements for a modeling paradigm, which will be reproduced bellow ([3]).

A modeling paradigm in the context of simulation should have the following key features:

1. *general purpose*: to allow modeling of a wide variety of problem types and domains.
2. *theoretical foundation*: "to move the modeling process towards science".
3. *hierarchical capability*: to facilitate modeling of complex systems.
4. *computer architecture independence*: sequential, parallel and distributed implementations should be based on same model.

5. *structured* : for guiding the user in model development.
6. *model reuse*: have a model database for component reuse.
7. *separation of model and experimental frame*: model should be separate from its inputs and outputs.
8. *visual modeling capabilities*: for graphical model construction.
9. *ease of modeling*: "world view(s)[†] should ease the modeling task."
10. *ease of communication* : the conceptual model should easily communicate to other parties.
11. *ease of model validation* : both conceptual and operational validity should be provided.
12. *animation*: "model animation provided without burden to the modeler."
13. *model development environment*: to support the modeling process.
14. *efficient translation to executable form*: model automatically converted to computer code, or if not automated, facilitate programmed model verification.

Major work on the development of ontologies for simulation and modeling has been done over the years and for more information in this direction we refer the interested reader to [101, 138, 124, 13, 38, 75].

Nowadays, a widely used general-purpose language that implemented and extended most of the afore mentioned modeling and simulation features is *Modelica* [42]. Modelica is an equation-based object-oriented language with multi-domain modeling capability. In [42], the authors discuss the advantage of acausal modeling over the causal block oriented modeling in languages like MATLAB/Simulink. In the first case, the declarative modeling style based on equation instead of assignment statements specifies no cause-effect flow, making the Modelica classes more reusable. In the second case, with fixed input-output flow, the model reusability is restricted and, moreover, the physical topology of the real system might be lost.

In the area of configuration, we have also a quite long research history dedicated to the modeling problem, from the early rule-based representation languages to the model-based approaches, where the problem solving knowledge is clearly separated from the domain knowledge.

[†]In this context, world views are defined as conceptual frameworks. [7] introduced four world views : event scheduling, activity scanning, three-phase approach and process interaction.

Considering the visual modeling technique, we can have basically two kinds of modeling languages: *graphical languages* like UML, SysML, feature models (FM) graphical languages - the domain knowledge is represented by means of symbols and diagrams, and *textual languages* like ConTalk [39], ConBAConL [70], or EXPRESS - the domain knowledge is described through predefined keywords. Although graphical languages provide a clear representation of the domain knowledge, the interpretation of the model is not always a trivial task. The textual languages on the other side are more flexible, but sometimes not so easy to learn and thus less suited for model designers with few programming skills.

The Unified Modeling Language(UML) is a general-purpose object-oriented language, widely used in industry, and therefore a good choice for configuration ([31, 35]). As presented in [110], there are three relations in UML particularly interesting for configuration: the *association*(describes a semantical relationship between two components), *composition* (establishes a parent-child relationship between components), and *generalization*(models the inheritance). Furthermore, starting with UML 2.0, stereotypes can be also used in the modeling process for creating a model from an existing one. SysML (Systems Modeling Language) [1] is a UML profile, aimed at representing systems and product architectures. By limiting the number of available diagrams and simplifying the original UML notations, SysML becomes easier to use. Furthermore, units, dimensions, optimization functions and parametrized constraints can be defined. Both languages use the OCL (Object Constraint Language) extension in order to express standardized constraints.

Feature models [12, 11, 19, 74, 63] have been developed with the goal of representing variability properties in software product lines. A feature must have a unique name and can have two possible states: included in a configuration, or excluded from a certain configuration. The idea here is to represent all the possible variants of a configurable product in a single graphical model. A feature model contains two parts: a structural part, that defines the hierarchical relationship between the features and a constraint part, that adds constraints, defining cross-hierarchy restrictions.

From the category of textual languages, EXPRESS [2] is an object-oriented data modeling language, which provides also a graphical variant EXPRESS-G. Models are organized here according to schemas, that allow to group elements within a model. EXPRESS components are defined as entities with specific attributes. Another features of the language are the usage of abstract classes, subtypes and the possibility to declare units. Although the user can define his/her own functions in a procedural manner, the lack of built-in functions and of configuration-specific keywords restrict the usage of the language in the domain. The other two examples of textual languages, CanTalk and ConBaConL are

presented in Section 2.3 within the description of the corresponding configuration systems.

When further considering the techniques to be used in solving the configuration problem, most of the model-based approaches use the powerful mechanism of constraint solving and thus the *constraint-based knowledge representation* ([86, 78, 39, 72]). As already mentioned, constraints can be used to define restrictions on the system and relationships between components, attributes or between a component and its attributes. Thus, the configuration problem can be naturally mapped to a CSP - also called a *static CSP*, when the number of components to be used in configuration is fix. Extensions to the classical CSP have been implemented, such as dynamic, composite or generative CSP [85, 116, 129], for improving the expressiveness of the knowledge representation and the solving mechanisms.

Except the constraint-based solution, logic-based techniques like Answer Set Programming (ASP) and Description Logics (DL) have been also successfully applied to configuration. ASP is a logic-based knowledge representation formalism, that proved to be well suited for compact descriptions of configuration and reconfiguration problems (see, e.g., [125, 41, 137]). ASP programs are represented as a finite set of rules, where each rule has a head and a body and, depending on the rule's structure, we differentiate between integrity constraints, facts, and weight constraints. A detailed presentation of ASP can be found in [44, 77].

Descended from the so-called "structured inheritance networks", Description Logics is another popular knowledge representation formalism in the context of Semantic Web. [34] have analyzed the applicability of DL for configuration knowledge representation and the conclusion was that its usage is limited by some factors: no aggregation functions are allowed, constraints on complex connection structures are not supported and also some specific constraints types are supported in a restricted manner. For further information on DL we refer the reader to [6]. Details about the configuration systems, where DL has been used, can be found in [149, 82, 72, 61].

Based on the upper presented research and on the modeling techniques analysis from [110], we can state the key features of a modeling paradigm in the context of industrial configuration:

1. be *user-friendly*, by means of using a widely accepted terminology.
2. be *textual*: unlike graphical languages, it will facilitate the interpretation and the optimization of the implemented model and will enhance designers productivity. Working with a textual language is often faster than through user interfaces, especially in an industrial environment, where models are complex and contain a big amount of data. Moreover such a language can be integrated in a development environment (Eclipse, Microsoft Visual Studio etc.), which provides

syntax-highlighting, code auto-completion, structured projects etc.

3. be *object-oriented*: this approach has been preferred by many researchers ([64], [114]). It is indeed very suitable for systems modeling, as the analogy between components and objects is obvious.
4. provide a *graphical representation*: this is important to the modeler for creating a clear view of the real model.
5. be *extensible*: various applications can be integrated with a modeling environment.
6. allow *easy structure modeling* my means of *partonomy and taxonomy relations*.
7. *definition of units*: the modeling language must support the definition of specific units of measurement.
8. *default values and ranges*: allow the user to get default configurations quickly.
9. *built-in functions and constraints* available to the modeler: will ease the work with large numbers of components in the model.
10. *soft constraints*: constraints that may be violated if they are overridden by a user selection or indirectly as a consequence of a constraint with higher priority.
11. *structural and functional decomposition*: large systems are usually designed with structural decomposition in mind in order to get flexible and independent solutions for different subproblems.

2.3. Configuration Tools

Over time, the AI community has developed a large variety of configuration tools that fitted the different necessities and goals in each practical area, thus creating a strong foundation for newcomers.

In the following, we will shortly recall a couple of configuration systems, which illustrate the main approaches in the field of knowledge-based configuration, as presented in Section 2.1.

In the early 80s, Digital Equipment Corporation was already using the R1/XCON program in the selling process, by automatically selecting the computer system components based on the customer's requirements. [81] implemented XCON as a production-rule-based system, based on the OPS4 language [40]. OPS4's memory was divided into a *production memory*, that stored the set of

rules(productions), where each rule was defined in the form *if(condition/s)-then(action/s)*, a *working memory*, which held the set of data elements, and a *data base*, containing the description of all the components that can appear in a configuration. In the configuration process, when R1 receives the customer's order, first, it gets the corresponding component descriptions from the *data base*. Then, in a *recognize-act* cycle, it determines, based on the defined rules, the particular type of extension to be added to the current partial configuration. Thus, the actions corresponding to the rules with the fulfilled conditions modify the content of the *working memory* in a repeated manner. From 772 rules in 1980, R1's production memory extended in 1989 to about 11.500 rules, specified in a new version of OPS [115].

Another historical system is SICONFEX[76], which had a similar using purpose as R1, being used for the configuration of SICOMP process computers[142]. The input data was represented by hardware components, software specifications and intended functionality. In this case, the system used many techniques in order to get a better structuring of the domain knowledge: rules, inheritance mechanisms, domain procedures, concept hierarchies, schemes for describing objects and LISP Code.

Other configuration systems include ConBaCon (Constraint-Based Configuration) [70] and CAWICOMS (Customer-Adaptive Web Interface for the Configuration Of products and services with Multiple Suppliers) [36]. ConBaCon treats the special case of reconfiguration, using the conditional propagation of constraint networks and has its own input language - ConBaConL. In [70], the authors present ConBAConL, a "largely declarative specification language", by means of which one can specify the object hierarchy, the context-independent constraints and the context constraints. Furthermore, the constraints are divided into *simple constraints*, *compositional constraints* and *conditional constraints*. Although successfully integrated in industry, the performance problems were observed in case of large products/systems, as a result of the large number of generated constraint variables and associated CE (consistency-ensuring) constraints within the solution. In [69], the author tackles these issues by clustering the ConBaCon model.

The goal of the CAWICOMS project was the development of a "**C**ustomer-**A**daptive **W**eb **I**nterface for the **C**onfiguration **o**f Products and Services with **M**ultiple **S**uppliers",i.e., an interface that enables the communication between various configuration systems. In [36], an application scenario for semantic Web services is presented, choosing as example the domain of telecommunication services. In order to define a common language for representing the properties of configurable products and services, the authors use "a hierarchical approach of related ontologies[14, 49]"[36]. By means of the DAML+OIL language, a flexible product ontology for complex, customizable products can be

modeled, the domain knowledge being consistently represented in XML. For the configuration task, ILOGs domain-independent and Java-based JConfigurator [72] was adopted. JConfigurator implements *Generative Constraints Satisfaction* for solving complex configuration problems. Actually, the configuration task is executed on a distributed architecture, i.e., each involved configurator has only a partial view on the product model. The communication between the configurators occurs by means of XML-based SOAP messaging and Web Services.

LAVA is another successful automated configurator [39], used in the complex domain of telephone switching systems. It makes use of generative constraints and is the successor of COCOS [130], a knowledge-based, domain independent configuration tool. The modeling language is ConTalk, an enhanced version of LCON (the constraint-based representation language used in the COCOS project) that follows the Smalltalk notation. A ConTalk constraint is a statement which describes a relationship between components ports or between the attributes values. During configuration, the inheritance hierarchy of component types is exploited and each time a component is generated, a new constraint object is instantiated for that component and propagated to all its related components, ports and attributes.

A powerful configuration system that combines constraint programming(CP) with a description logic(DL) is the ILOG (J)Configurator [72]. The combined CP-DL language, in which the configuration problem is formulated provides, on the one hand, the constraints, needed in the decision process, and on the other hand, the constructs of the description logic, able to deal with unknown universes. When solving the problem, the constructs of description logic, which are well-suited to model the configuration specific taxonomic and partonomic relations, are mapped on constraints and thus the wide range of constraint solving algorithms may be used.

Products like COSMOS ([58],[51]) and KIKon ([29]) used the resource-based configuration approach, by seeing the components as resources and splitting the system in sub-functionalities (COSMOS) or sealing the obtained configurations in order to store them for future usage (KIKon). In the first case, the knowledge base is divided into resources definition and components description. As mentioned in [51], a great advantage of the used modeling methodology is that, considering resources' longer lifetime (when compared with components' lifetime), the maintenance efforts are drastically reduced. Complex, modular products may be successfully configured in COSMOS under the following conditions: the components can be easily modeled as resources, the design of the system to be configured allows functional decomposition, the most relations are summing relations ([51]).

Originally based on COSMOS, the interactive configuration system KIKon was thought to sup-

port the configuration of telecommunication systems, which "differs from the configuration of other technical systems like cars and airplanes"([29]). One of the reasons presented in [29] is that the communication relation has to be considered in case of telecommunication systems. In terms of graphical interface, the tool offers a domain-dependent GUI, which includes *a dynamic table mechanism* ([127]) used for comparing and selecting components.

Among currently available products on the market, we mention well-known systems like: Tacton configurator, SAP's configurators, camos configurator, Oracle configurator *.

Tacton configurator [103] is based on the research prototype OBELICS [5], that uses logic programming, object-oriented principles, SICSTUS Prolog [108] and its object-oriented extension SICSTUS Objects. In [5], the authors introduce "a less complex and more interactive approach to configuration", named *Interactive Configuration by Selection*(ICS). The motivation for ICS was that, "from the end user's perspective, the goal is not to have a maximally "intelligent" system, but rather one that will help him the most in his day-to-day work", as also stated in [107] ([5]). Therefore, the authors prefer a configuration system which can assist the task of configuration, rather than one that automatically produces configurations from the given requirements, without any further user interaction. Further on, in [103], three types of configuration engines are identified: the configuration engine which assists the user in making a selection by using *restriction tables* and in the end checks only if the user selection is valid, the *constraint solver engine* that can generate all the valid configurations or just the optimal one, and the *constructive search engine*, capable of searching one suitable configuration that satisfies the requirements. The last two types of engines are incorporated in the Tacton configurator: a dynamic constraint solver for capturing the requirements and a constructive search engine for finding a suitable configuration.

In 1998, SAP AG software company was already implementing its third generation of sales configurators - the *SCE (Sales Configuration Engine)*([55]). Focusing on the sales configuration in business processes, "the SCE attempts a clear separation of interface and configurator engine, which lets SAPS customers tune the interface to a particular configuration problem or completely redefine it."([55]) Thus, the SCE architecture comprises three main parts: the *configurator user interface*, the *knowledge base server* (where the basic modeling entities - classes, dependencies, materials, etc.,- are defined), and the *configuration engine*(designed also as a server, so that it can handle multiple configuration requests simultaneously). Concerning the used AI technology, the approach is based on previous AI research ([54, 18, 58, 90]) and makes use of: *constraints* (for expressing the dependen-

*www.tacton.com/en, www.sap.com, www.camos.de, www.oracle.com

cies in a declarative manner), *defaults*(defined by the author in [55] as "a younger brother of a soft constraint"(referring to a default value assignment)), and a *Truth Maintenance System (TMS)*[25] as problem solving module. The TMS was later on replaced by an assumption-based TMS ([21]) capable of handling specification relations - *ATMS**, described in [54]. In [56], two successor configuration systems are presented: the SAP ERP (Enterprise Resource Planning) Variant Configurator, which appeared in 1994 and provides high-level configuration in SAP ERP systems, and SAP CRM IPC (Customer Relationship Management Internet Pricing and Configuration), that is on the market since 2000, with a standalone configuration engine. More recently, the knowledge representation issues in the SAP configuration environment were discussed in [57].

In case of *camos Configurator* [51, 115], the components are represented in a class tree and can be used in both partonomy and taxonomy relations. Moreover, inheritance mechanisms were implemented. During the configuration process, the user can detect the inconsistencies - if they appear - and she/he can manually eliminate conflicts by selecting another adequate component. Heuristics for the automation of the configuration process are not available ([115]). In order to define relations/dependencies between components, two possibilities are provided: one can use either predefined keywords as "can", "must", "cannot", "initialize" to define constraints or one can use procedural dependencies. In the sales area, for increasing the level of acceptance, an unsatisfied constraint could be ignored during the manually driven configuration process, and, eventually, the corresponding components (implied in the unsatisfied constraints) will appear in a list with technical explanations.([51])

As presented in [102], the Oracle Configurator is a selling and configuration product, based on state-of-the-art configuration technology. Using constraint-based reasoning, the system provides valid configurations either automatically, if the user chooses to automatically complete configurations, or stepwise, in an interactive guided session, when real-time feedback about the impact of each user's selection is given. *Configurator Extensions* offer the possibility to extend the Configurator using Java[†]. In the integrated modeling environment, the user is able to create the desired model structure, and afterwards, constraints can be applied to this structure. Advanced, personalized constraints can be defined by means of *Statement Rules*: the user can add, delete or edit the text of a predefined rule.

[†]No further information in this direction was found in the documentation or related papers.

2.4. Simulation Tools

Beside modeling and reconfiguration features, which were the main targets of our research, we also aimed at a tool capable of simulating the constructed model, in the sense of adding dynamic behavior to the static configuration knowledge base. Therefore, in order to get an idea about general requirements and domain specific technologies, existing simulation systems had to be also investigated. The selection briefly presented here is based on Internet searches, papers about state-of-the art mobile networks applications, and the survey [3] (based on Swain's survey [135]).

When referring to general dynamic systems' modeling and simulation, MATLAB/Simulink [136] and Modelica/Dymola [30] are the most famous names. In case of Simulink, the user is capable of modeling the desired system in the graphical interface, based on a large library of standard components, called *blocks*. As part of the standard library, the *S-Function block* allows users to implement their own custom routines, which can include new algorithms not directly supported by Simulink or existing code, written in C, Ada, Fortran or the proprietary MATLAB coding ([60]). (CITE projets)

In the second case, MathModelica problem solving environment [42, 79], currently called Wolfram SystemModeler, integrates Modelica-based modeling and a couple of software products such as Mathematica - for providing features like traditional mathematical notations, numerical and symbolic calculations, functional, procedural, rule-based and graphical programming [144]; Microsoft Visio as diagramming software; and Dymola [26], that serves as simulation engine. (CITE projets)

Further on, if we restrict ourselves to the domain of mobile networks applications, ns-2[80] and OPNET[15] are two of the most popular network simulation packages on the market. The ns-2 simulator uses OTcl - an object oriented version of Tcl(tool command language [104]) - as command and configuration interface, while in OPNET the user creates networking models by using the drag and drop approach. One drawback of using these tools in the domain of M2M applications is the difficulty of simulating new platforms and protocols. In [67], the author mentions that implementing new protocols in the existing tool packages is rather difficult and time consuming. Further more, both ns-2 and OPNET operate at the packet level, which is not the case in our approach. For instance, in case of smart metering requirements, all the meter profiles parameters are given per day and cell.

There is also a multitude of commercial software planning tools, which combine the environment specific constraints with signal propagation. In [47], the implemented WLAN modeling tool defines the environment by describing the floor plan structure and the wall types. This definition will further serve as input for the electromagnetic propagation model, which eventually leads to an estimation of

the maximum achievable throughput in specific sites within the considered building. The optimization module is used to automatically optimize the number of access points and their position to meet site specific demands.

Just like ns-2 and OPNET, many other network simulators use *discrete event simulation*, in which a list of pending "events" is stored and events are processed in a specific order.

In his "Introduction to Discrete Event Simulation", [3] presents a survey on 87 discrete event simulation and modeling tools (SMTs), that were available back in 2007. Of these, thirty-three (33) provided support for academia or were open source, while forty-nine(49) were commercial tools. The author chooses four of these commercial offerings: Arena [113], Extend [65], SIGMA [141] (all three providing academic versions), and Ptolemy [16] for an in depth evaluation and comparison.

The Arena modeling environment is an integrated development environment (IDE), allowing the use of visual, drag and drop modeling by connecting predefined and user defined blocks. Structured to aid in model development, Arena has some animation ability and allows a varying level of model description. Model components can be reused from a library, with a two level hierarchy. However, there are minimal alternatives in model creation beyond the supplied IDE. Furthermore, debugging and verification modes are possible and also tools to allow validation of results are available. Basic statistical tools, such as standard distributions, seeding random numbers, sampling, and distribution fitting are integrated in the IDE. Arena is easy to learn, well documented, and appropriate for experimentation purposes and therefore nowadays hundreds of educational institutions worldwide are using its simulation software. Rockwell Automation's Educational [113] version of Arena provides a valuable teaching tool to universities and colleges by introducing students to the principles of simulation.

Extend [65] is an extensible program with add-ons from several sources, allowing validation of results and also running simulations in debugging or verification mode. Like in Arena, we can visually model a system, using predefined or user defined blocks. Also here the alternatives in creating a model are minimal beyond the supplied IDE. Designed to support model development, Extend has some animation ability and can model continuous, discrete event and discrete rate processes. On the vendor's Web page, different simulation packages are recommended to the user, depending on the type of simulation to be conducted.

Presented by its developers as "the world's fastest and most flexible analytical discrete event simulation software", SIGMA [141] has an internal text editor, where models can be created and edited. Alternatively, the models can be visually constructed from predefined blocks (no user defined blocks are supported). Based on event graphs, SIGMA features the the ability to translate its models into

more readable versions and also some animation ability. Also fairly easy to learn, Sigma is often used as an introductory simulation teaching tool, being very well documented ([120, 73, 119]).

Ptolemy II [16] is a DES/Continuous simulation and modeling tool, with a graphical user interface called Vergil. It is a Java-based component assembly framework, allowing the use of visual modeling with predefined and user defined blocks. Models can be created from assembling previously developed components, or by direct programming them in Java. A text editor is also present as part of the report module. The basic code is in Java and models can be created to operate as Java applets. Common features like debugging, verification and validation of results are available to the user, as well. However, because of its complexity, Ptolemy II is not so easy to learn as the previous simulation tools. The main citation for the Ptolemy Project is [28].

2.5. Conclusions

Although simulation and configuration domains were, to our knowledge, never linked together in order to take advantage of their individual benefits, we think that the idea of combining the two directions in one language can be useful, especially in case of an industrial environment.

As stated in [8], the benefits of using simulation go beyond just providing a look into the future. We will further recall a couple of them:

- choose correctly: through simulation, one can test every aspect of a proposed change without committing resources to its acquisition.
- prepare for change: answering ”what-if” questions is useful for both designing new systems and redesigning existing systems.
- explore possibilities: once a valid simulation model is obtained, one can explore new possibilities (impose new requirements or add new features to the modeled system) without the any expense.
- identify constraints: production bottlenecks or bottlenecks in the functionality of a system (e.g., communication network). By using simulation to perform bottleneck analysis, one can discover for instance the cause of a blackout in an electrical grid, or the reason for a communication delay in a phone network.

On the other side, the application of configuration for solving problems in the engineering domain has also a long tradition. As we have seen throughout this chapter, configuration of technical products from various domains is a well researched field, especially in the direction of solving techniques for the configuration task. The modeling problem was less covered and therefore the languages used to express the configuration knowledge are mostly not ”user-oriented”, but rather ”solving-oriented”. Hence, such languages are not easy and prevent systems based on them to be used in practice. Graphical representations like UML and feature models have been often integrated in constraint- or description logics-based approaches, but there is still a strong need for easy to learn and use modeling languages, that are expressive enough to state configuration problems.

Chapter 3

SIMOL Language

Parts of the content of this chapter have been published in [98, 93, 94].

Considering the desired attributes of a modeling language from Section 2.2 and the future mapping of the model to a CSP, we implemented the SIMOL language as a declarative, object-oriented language with multiple inheritance, which adopts a clear Java-like syntax. The key concept of SIMOL is the definition of basic and hierarchical components, which are used to represent the given system. Each component is designed to have a set of attributes and a specific behavior, provided as set of equations. For defining the attributes, there can be used one of the two primitive data types with Java like semantics: non-negative integer data type, depicted by `int`, and `bool` data type. In a classical object oriented manner, if a component is a subclass of another component, then the equations of the superclass are taken together with the equations of the component in order to specify the component's behavior. In SIMOL, it is also possible to assign equations to specific behavior modes in order to represent potential reconfigurations, as we will see in Chapter 4. Another feature of SIMOL is its ability to model the systems behavior over time. In this case, SIMOL allows for specifying state transition functions. Such a function itself is a set of equations, that connects values of variables between one state and its successor state. SIMOL does not allow to deal with continuous time, i.e., only systems which can be modeled using discrete time can be represented in SIMOL.

In the remainder of this chapter, we define the language starting from the idea discussed also in [147], [53], i.e., separating constructs in a way that facilitates the language analysis. Hence, given a

language specified as a sequence of characters, we are firstly interested in dividing this sequence into basic lexical units, called tokens. Afterwards the syntax of this sequence of tokens has to be checked, by following some production rules, and, finally, the semantics is added.

3.1. Tokens

For SIMOL, the following tokens are defined*:

- *identifiers for any type of component and attribute* follow the convention: they start with a lower or upper case letter followed by zero or more repetitions of digits or lower/ upper case letters (the letters may contain diacritics);

  `id ::= ([a-z]|[A-Z])+([a-z]|[A-Z]|[0-9])*`

- *reserved words and literals*: `and`, `at_least`, `at_most`, `attribute`, `bool`, `component`, `constraints`, `const`, `else`, `exist`, `extends`, `false`, `forall`, `if`, `import`, `int`, `kbase`, `max`, `min`, `.next`, `or`, `package`, `private`, `product`, `transition`, `true`;

- *literals*:

  - *integer literals* can be 0 or start with any non-zero positive digit (from 1 to 9), which can be followed by any repetition of digits from 0 to 9;

    `int_lit ::=0 | ([1-9] ([0-9])* ) 0|([1-9] ([0-9])*)`

  - *boolean literals*

    `bool_lit ::= true | false`

- *separators*: `{`, `}`, `[`, `]`, `;`, `,`, `.`

- *arithmetic and relational operators*: `=`, `>`, `<`, `:`, `<=`, `>=`, `!=`, `+`, `-`, `*`

- *units of measurement*, for a more realistic description: `KB`, `MB`, `GB`, `kbit` `V`, `Hz`, `A`, `%`, `/2min`, `/sec`, `/day`

- *integer valued ranges*, used for restricting the domain of the variables values:
  `in_range ::= { (0|[1-9]([0-9])*) ((.. ([1-9]([0-9])*)) | (, ([1-9]([0-9])*))*) }`

*Note that the formal definition of the SIMOL tokens is based on the representation given in Annex .1.

- *special tokens - comments*:

  ```
  SINGLE_LINE_COMMENT ::= // (~[ \n,\r])* (\n|\r|\r\n)

  FORMAL_COMMENT ::= /** (~[*])* * (* | (~[*,/] (~[*])* *))* /

  MULTI_LINE_COMMENT ::= /* (~[*])* * (* | (~[*,/] (~[*])* *))* /
  ```

All the upper described tokens can be found in the SIMOL programs provided as examples in Section 3.2.

3.2. Syntax

The complete SIMOL syntax is formally described in Annex .2.

A program written in SIMOL contains basically three sections, as depicted in Figure 3.1:

- *a knowledge base declaration section*, which is optional. It is used to organize components belonging to a specific domain or problem (similar to a Java package):

  ```
  kbase GPRSCell;
  ```

- *an import declaration section*, where knowledge bases can be loaded [†]. It is also optional and contains one or more import declaration statements:

  ```
  import UMTSCell.*;
  ```

- *a component definition section*, that is the main constructing unit of a SIMOL program and it is mandatory (Line 2 to 45). Each component definition starts with the keyword **component** followed by the name of the component and with an optional **extends** followed by a comma-separated list of component names. If **extends** is used, we know that the new component has one or more super components from which constraints are inherited.

  In every component definition, we firstly define the component's attributes after the **attribute** keyword. For example, in Line 3 the attributes `midst`, `code set`, and `mRate` are defined. All these attributes are of type integer (**int**). Besides attributes, constraints can be defined. There are two ways of doing this. Either we use the keyword **constraints** directly followed by a block

[†] not supported by the current version of the language

in surrounding parentheses {}, or we use the same keyword **constraints** followed by an name under parentheses () and again a block statement.

The first definition of constraints only allows for specifying a *single component behavior*. The other definition makes use of modes that are needed later on for reconfiguration (see Section 4.2.2). For example, in Lines 11 to 19, three modes for the component `FPC` are defined. The `default` mode sets the value of variable `value` to 1. The `x1` mode restricts the domain of `value`, where the constraint solver can select one value from the range {2..4} when computing reconfigurations. The last mode (`unknown`) does not specify any value.

By convention, an empty component definition section is not allowed, i.e., if the constraints block is missing, we have to declare at least one attribute for the current component. For example, we accept the following definition:

```
component P2PMeter {
    attribute int mdist,codeset,mRate;
    }
```

In the case of *derived components*, the opposite holds: even with no attributes declared, we may state constraints over the inherited attributes. For instance:

```
component P2PMeter {
  attribute int mdist,codeset,mRate;
  constraints {
    mdist = {1..3};
    codeset = {1..4};
  }
}
component D145Meter extends P2PMeter{
  constraints {
    mdist = 2;
    codeset = {2..4};
  }
}
```

More details about the implemented inheritance mechanism will be given in Section 3.3.

In addition, models written in SIMOL might be described over discrete time. For this purpose SIMOL makes use of the **transition** section. Within the transition section the new values of

```
1.  kbase GPRSCell;
2.  component P2PMeter {
3.    attribute int mdist, codeset, mRate, dataAmount;
4.    constraints {
5.      mdist = {1..3};
6.      codeset = {1..4};
7.    }
8.  }
9.  component FPC {
10.   attribute int value;
11.   constraints(default) {
12.     value = 1;
13.   }
14.   constraints(x1) {
15.     value = {2..4};
16.   }
17.   constraints(unknown) {
18.
19.   }
20. }
21. component BTS {
22.   attribute int fpc;
23.   constraints {
24.     FPC fpc1;
25.     fpc = fpc1.value;
26.   }
27. }
28. component Cell {
29.   attribute int neededR, realR, P2PNo;
30.   constraints {
31.     BTS b1;
32.     P2PMeter s[100], p1, p2;
33.     realR=sum([s], mRate)/P2PNo;
34.     realR>=neededR;
35.     exist(at_least(50), P2PMeter, mRate=12 kbit/sec);
36.   }
37.   transition {
38.     forall (P2PMeter) {
39.       if (mdist=1 and codeset=2) {
40.         codeset.next = {2,3} }
41.       if (mdist=3 and codeset=2){
42.         codeset.next = {2,1}; }
43.     }
44.   }
45. }
```

Figure 3.1.: A SIMOL program.

some, but not necessarily all variables, have to be defined. In our example from Figure 3.1, Lines 37 to 43 define the next values of the `codeset` variables for all `P2PMeters`. In order to distinguish the new value of a variable in the successor state, we make use of the keyword **next**.

It is worth noting that in the transition section we can use all statements from the constraints section. For example, let us reduce the number of P2PMeters in our model to four and have the following statements in the constraints section:

```
constraints {
    BTS b1;
    P2PMeter p1,p2,p3,p4;
    p1.mdist=3;
    p2.mdist=3;
    p3.mdist=2;
    p4.mdist=1;
```

We further assume that, initially, the codeset attribute for all the P2PMeters is set to value 2 and, according to the transition function defined for the codeset in Figure 3.1, we can state the followings.

For each P2PMeter instance, we have the following possibilities:

- if the mdist attribute equals 1 and the codeset attribute is, in the current state, equal to 2, then, in the next state, the codeset can remain unchanged or it can be increased to 3;
- if the mdist attribute equals 3 and the codeset attribute is, in the current state, equal to 2, then, in the next state, the codeset can remain unchanged or it can be decreased to 1;
- otherwise, the codeset remains unchanged.

Note that, in this scenario, we assume that only the codesets are changeable; any other attribute of the model is considered to be fix, i.e., it will not change from one state to another.

Hence, a possible representation for the obtained *state machine model* of the cell with four P2P meters is given in Figure 3.2.

Note that the number of all possible states equals $2^{\text{number of varying codesets}}$, but we restricted the number of states internally to a specific value, depending on the necessities of the conducted experiments.

In the constraints section, we may have the following types of statements:

- an *empty statement*: ;,
- a *component instance declaration*, with the possibility of initializing its attributes in {..}:

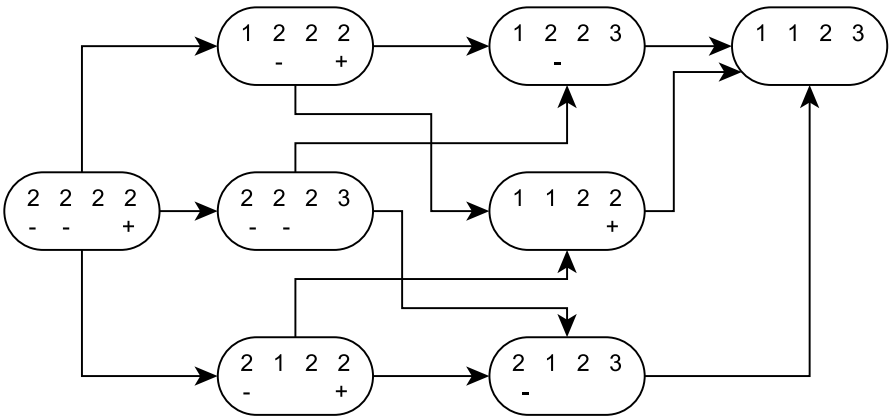

Figure 3.2.: The state machine model for a cell with four P2PMeters. The values inside the circles represent the codeset values (p1.codeset, p2.codeset, p3.codeset, p4.codeset) and the + and - signs suggest the possibilities of increasing and decreasing the codeset in the next state.

```
P2PMeter p1; or P2PMeter p1{mdist=3};
```

Using this kind of statements, we define the subcomponent hierarchy in our model, i.e., the partonomy relations. The cardinality of these relations (i.e., the number of subcomponents which can be connected to a certain component) is always finite. Hence, we consider two situations when declaring a variable: the case in which the variable has a primitive type and the case the variable is a component (an instance). A variable can also hold an array of a specified type.

- an *inrange expression*, as for instance `mdist = {1..3};` of `value = {2,4,5};` This type of expression restricts the domain of variables values either by defining the permitted range or the permitted set.
- an *arithmetic or/and boolean expression*:

```
p1.dataAmount>=p2.dataAmount+p3.dataAmount;
```

Here we treat arithmetic and boolean expression with any number of operands and any number of operators. The implemented operator precedence is the traditional one: firstly, multiplication and division are performed, then addition and subtraction and, lastly,

boolean (relational) operations.

– a *conditional block*, starting with the keyword **if** and optionally followed by an **else**:

```
if(codeset=1){
     mrate=80;
}
```

– a *forall block*:

```
forall (P2PMeter) {
     if (codeset=1) {
       mrate=80;
     }
     if (codeset=2){
       mrate=120;
     }
}
```

– an *exist statement*, for instance:

```
exist(at_least(50), P2PMeter, mRate=12 kbit/sec);
```

with the meaning that at most 50 P2PMeter instances can have the mRate equal to 12 kbit/sec.

– special functions like **sum**, **product**, **min**, and **max**, that allow to state constraints over an array of instances or values:

```
realR=sum([s], mRate)/P2PNo;
```

Hereby we sum the `mRate` attribute of all `P2PMeter` components stored in variable `s`.

```
total= product([s], dataAmount);
```

where the product of the `dataAmount` attribute of all `P2PMeter` components is computed.

```
p2.dataAmount>=min([s,p1], dataAmount);
```

with the meaning that the `dataAmount` attribute of `p2` has to be greater or equal than the minimum `dataAmount`'s value of any `P2PMeter` stored in `s` and `p1`. The **max** function is used in the same manner. Note that the instances in [] do not necessarily have to have the same type, the only requirement is that the components posses the given attribute. For

instance, we can additionally define the component *PLCMeter*, declare an instance $m1$ inside the *Cell* component, and use the *min* function over $s, p1, m1$, as follows:

```
component PLCMeter {
  attribute int mdist,dataAmount;
  constraints {
    mdist = {1..3};
  }
}
component Cell {
  ..
  constraints {
    ..
    PLCMeter m1;
    p2.dataAmount>=min([s,p1,m1], dataAmount);
```

3.3. SIMOL Inheritance

In this section, we make use of a very simple example to discuss SIMOL inheritance. Figure 3.3 depicts a small system comprising 4 components, i.e., a power supply (*PS*), an acceleration sensor (*AS*), a GPS sensor (*GPS*), and a communication device (*CD*). The communication device is used for sending the measured sensor information to a server. The power supply is for providing electricity to the connected components. All these components have a behavior and provide functionality.

For the purpose of specifying functionality, we introduce a function fct that maps a component to a set of attributes, which indicate a certain functionality. For our example, we introduce the attributes **ad**, **gps**, **comm** to state the acceleration sensor functionality, the gps functionality, and the ability for communication, respectively.

$$fct(AS) = \{\mathbf{ad}\}$$
$$fct(GPS) = \{\mathbf{gps}\}$$
$$fct(CD) = \{\mathbf{comm}\}$$

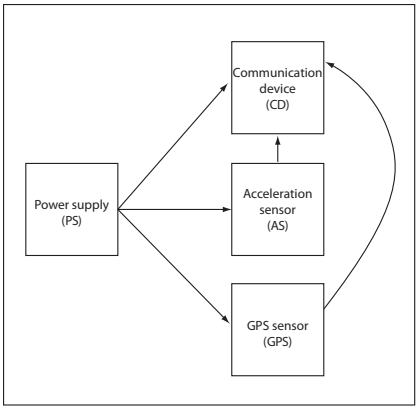

Figure 3.3.: A small sensor system.

Table 3.1.: The component instances for our small sensor system.

Generic Component	**Instance 1**	**Instance 2**
PS	$PS_1 : costs(PS_1) = 10, power(PS_1) = 10$	$PS_2 : costs(PS_2) = 20, power(PS_2) = 15$
AS	$AS_1 : costs(AS_1) = 2, power(AS_1) = 4$	$AS_2 : costs(AS_2) = 20, power(AS_2) = 1$
GPS	$GPS_1 : costs(GPS_1) = 6, power(GPS_1) = 5$	
CD	$CD_1 : costs(CD_1) = 10, power(CD_1) = 10$	$CD_2 : costs(CD_2) = 20, power(CD_2) = 4$

We now specify the additional constraints of the system. The following constraint formally represents the requirement that the power provided by *PS* must be larger or at least equivalent to the sum of the power consumption of the other components:

$$power(PS) \geq power(AS) + power(GPS) + power(CD)$$

Moreover, we state that the device has to provide at least **ad**, **gps**, **comm** functionality.

$$fct(AS) \cup fct(GPS) \cup fct(CD) \supseteq \{\mathbf{ad}, \mathbf{gps}, \mathbf{comm}\}$$

Finally, we have the requirement that the sum of the cost of each part of the device is not allowed to exceed a certain pre-defined maximum cost.

$$cost(PS) + cost(AS) + cost(GPS) + cost(CD) \leq max\_cost$$

In configuration, we are interested in providing specific implementations of the components *PS*, *AS*, *GPS*, and *CD*, such that all requirements are fulfilled and no constraint is violated. Hence, what we do now for our running example, is to introduce specific implementations of the generic components with different costs and power consumptions. Table 3.1 summarizes all the used concrete component implementations.

A valid configuration is now a set of components that fulfills all constraints. For example, when assuming maximum cost of 60, the set $\{PS_1, AS_2, GPS_1, CD_2\}$ is a valid configuration but $\{PS_2, AS_2, GPS_1, CD_2\}$ is not because of violation of the cost constraint.

The ability to extend the functionality and behavior of existing components is of great importance for the taxonomic structure of a configuration domain. In any object-oriented languages, the taxonomy relations are represented through the inheritance mechanism. As we have already mentioned, we designed SIMOL with multiple inheritance. In order to demonstrate the necessity of this feature, let us consider the following scenario. For our afore described small system, we introduce a new requirement that refers to a specific signal modulation which can be accomplished by a new component - a modem (*M*). The modem receives the measured sensor information and transmits the modified signal to the communication device.

The function fct will similarly depict for *M* the modulation-demodulation functionality:

$$fct(M) = \{\mathbf{mdm}\}$$

Now the additional constraints of the system become:

$$power(PS) \geq power(AS) + power(GPS) + power(CD) + power(M)$$

$$fct(AS) \cup fct(GPS) \cup fct(CD) \cup fct(M) \supseteq \{\mathbf{ad}, \mathbf{gps}, \mathbf{comm}, \mathbf{mdm}\}$$

$$cost(PS) + cost(AS) + cost(GPS) + cost(CD) + cost(M) \leq max\_cost$$

The problem appears, if the predefined maximum cost is always exceeded, because of the new added component. In other words, we cannot afford both a modem and a communication device. Therefore, a new component type - a communication device with integrated modem (*MCD*)- will solve the case (under the assumption that $cost(MCD) \leq cost(CD) + cost(M)$). In SIMOL, the *MCD* definition has the following syntax:

```
component MCD extends CD,M {
    constraints{
      power={4,6} W;
      costs={2..30};
    }
}
```

3.4. Semantics

We now specify the semantics of the SIMOL language, where we rely on mathematical equations. In particular, we map every statement to a mathematical equation, and combine these equations for

a component, taking care of component inheritance and component instances. In the first part of the definition of the semantics we ignore discrete time. We discuss this issue later in this section.

For each component C, defined in SIMOL, we have a set of equations $constr_0$, that is defined within the **constraints** { ... } block. Moreover, the component C also receives equations from its super components and the instances used in the component definition. For example, when specifying `BTS b1;` in the constraints section of *Cell*, a new instance of `BTS` is generated. All constraints of `BTS` are added to the constraints of *Cell*. Because of the possibility of having more than one instance of a component, we have to rename the variables used in the constraints of an instance. For this purpose we assume a function *replace* that takes constraints M and a name N and changes all variables x in M to $N.x$. Hence, the set of equations that corresponds to a particular component C is given by the following definition:

$$constr(C) = constr_0(C) \cup constr_I(C) \cup constr_V(C),$$

where $constr_I$ are the constraints inherited from the super components of C

$$constr_I(C) = \bigcup_{C' \in super(C)} constr(C'),$$

$constr_V$ are the constraints coming from the components used in the definition of C (and requiring variable renaming using the function *replace*, that adds a new prefix to the variables used in the components, in order to make them unique)

$$constr_V(C) = \bigcup_{(C',N) \in vd\_inst(C)} replace(constr(C'),N),$$

where $vd\_inst(C)$ are the variables used in the constraints of the instances defined in the component C,

and finally, $constr_0$ are the constraints defined in C directly.

Each constraint within the **constraints** { ... } block contributes to $constr_0(C)$ as follows:

- C_{attr_val} : *attribute-equals-value/s* constraints, formulated with = operator and applied on component attributes together with one single integer/boolean value or with a set of values;

- C_{attr_attr} : *attribute-equals-attr* constraints, formulated with $=$ operator and applied on component attributes;
- C_{num} : *numeric constraints*, formulated with basic relational operators over numeric expressions. Note the usage of the special array functions **sum**,**product**, **max**, **min** within the numeric expressions;
- C_{cond} : *conditional constraints*,

  $if(C_x$ *is satisfied*$)$ C_y *must be satisfied else* C_z *must be satisfied*;
- C_{exist} : *existence constraints*,

  $exist(at\_least(NR)\ |at\_most(NR)|NR,C,ATTR = VALUE)$, with the meaning that at most, at least or exactly NR components of a given type C have $ATTR = VALUE$.
- C_{forall} : *forall constraints*,

  $forall(C)$ C_t *must be satisfied* , with the meaning that for all components of a given type C, the constraints C_t have to be satisfied.

Further on, if we consider also the definition of constraints, which makes use of modes, then each constraint C_{mode} within the **constraints**(*mode*) $\{\ \ldots\ \}$ block contributes to $constr_0(C)$, only if *mode* is active. Hence, we can define this as a *conditional mode constraint* $C_{cond_{mode}}$: $if(mode$ *is active*$)$ C_{mode} *must be satisfied*.

$$constr_0(C) = \bigcup_{mode \in MODES(C)} constr_{mode}(C),$$

where MODES(C) is the set of functional modes, defined for component C, and $constr_{mode}(C)$ is the set of *conditional mode constraints*.

How to handle time?

Within the **transition** section we have constraints that define a relationship between the variables of a state and its successor state. In order to represent states, we introduce an index that is assigned to each variable used in $constr(C)$. This variable represents the state. Hence, what we do is to define constraints that hold in each state $i \in \{0,\ldots,s\}$, where s represents the maximum considered number of states within a reconfiguration model. These constraints are obtained from $constr(C)$, by adding an index i to the variables. We represent these constraints using the function $constr_i(C)$. For example,

if `value = 1` is element of $constr(C)$, then `value_4 = 1` is element of $constr_4(C)$. Such constraints are valid within a state and therefore called *state constraints*.

In addition to state constraints, we require *transition constraints*. The transition constraints can be easily computed from the **transition** section. In principle we make use of the same translation as in the **constraints** block, but also take care of the **next** attribute assigned to variables. If a variable v has such an attribute and we consider state i, we replace v.**next** with v_i+1. Variables v without the **next** attribute are changed to v_i. Hence, we obtain all transition constraints $trans_i(C)$ for state i and component C.

In summary, the constraints for the whole SIMOL program can now be obtained as follows:

$$constr = \bigcup_{i\in\{0,\ldots,s\}} \bigcup_{C} (constr_i(C) \cup trans_i(C))$$

Hence, the set of the obtained equations represents the behavior of the SIMOL program. It is worth noting that we took the semantic definition based on equations from the semantics of Modelica [42]. In contrast to Modelica, the handling of time is different as well as some of the functions that can be used within SIMOL.

3.5. Implementation

For the SIMOL language, we implemented a compiler in Java, which translates SIMOL programs into MINION input files [68]. MINION is an open source constraint solver, that has been optimized for solving large and hard problems. Coming with its own constraint language, MINION outperforms many other constraints toolkits due to its implementation design. The syntax of the input language was thought to map exactly to MINION's internals and therefore no extra variables are internally added and complex constraints are not decomposed into simple ones. This provides complete control over the way problems are implemented in MINION, but, at the same time, requires a deep understanding on how MINION works.

The reason for choosing MINION was thus the very good reasoning performance, but also the easy integration into programs written in Java played a role. Our prior experience with MINION (applied in the automatic debugging) can be found in [148, 100]. For further details in this direction we refer the reader to [99].

In order to obtain the MINION constraints specification, we applied a translation function from the SIMOL program to MINION specific input. The systems modeled in SIMOL are always finite. Hence, the complexity of the *SIMOL2MINION* mapping algorithm is polynomial (O(N)), where N is the size of the SIMOL program. The *SIMOL LOC/ MINION constraints* ratio depends mainly on the type of declared instances - simple instances or vectors of components - and on the complexity of the arithmetic operations. If we consider also the case of *SIMOL models defined over discrete time*, then the number of MINION constraints increases with every new state, i.e., it will be multiplied with the number of states. In what follows, we first present the MINION data types and constraints, which we used in implementing the semantics of SIMOL. Afterwards, we give some concrete examples of systems modeled in SIMOL and further converted into the corresponding MINION specification.

3.5.1. Basic MINION Constructs used in the SIMOL Implementation

In the followings, we discuss the MINION data types and constraints, which were used to map the SIMOL constructs.

Regarding data types, SIMOL provides two primitive ones: *int* and *bool*, and the possibility of declaring instances of already defined components. As previously presented in Section 3.1, *int* variables take non negative integer values, whereas *bool* variables have two values, denoted by $true$ and $false$. From the available MINION variable types, we choose the $DISCRETE$ and the *BOOL* type, respectively. The $DISCRETE$ data type allows a (any) range of integers, which has to be specified in $\{\}$ when declaring a $DISCRETE$ variable, whereas *BOOL* variables are defined over domain $\{0,1\}$. For more details on the MINION variable types, we refer the interested reader to [68]. In our implementation, the lower bound of the range for $DISCRETE$ variables is always 0, while the upper bound can be internally determined from the variable's type, e.g., variable describing academic grading can take only a few values, so, $\{0..20\}$ would be sufficient, whereas a variable describing the number of students attending a course may need hundreds of values ($\{0..500\}$). However, such a strict internal limitation is not preferred, as it may happen that a result of a computation, specified in form of a SIMOL arithmetic expression, to exceed the defined upper limit and, in this case, the MINION solver will not report on the error, but will just print out that we have no solution to the problem.

Considering the SIMOL program given in Figure 3.1, we further describe how SIMOL statements can be mapped to MINION encodings:

- *empty statements*, depicted by `;`, and comments will not be taken into consideration in the

mapping process.

- a *component instance declaration*:

```
P2PMeter p1;
```

will be translated to the following MINION variables declarations:

```
DISCRETE Cell_p1_P2PMeter_mdist {0..1000}
DISCRETE Cell_p1_P2PMeter_codeset {0..1000}
DISCRETE Cell_p1_P2PMeter_mRate {0..1000}
DISCRETE Cell_p1_P2PMeter_dataAmount {0..1000}
```

by linking the name of the instance `p1` to its attributes and the component `Cell`, inside which the instance was declared:

```
component P2PMeter {
     attribute int mdist, codeset, mRate, dataAmount;
     constraints {
       mdist=1..3;
       codeset= 1..4;
     }
}
component Cell{
     constraints {
       ..
     }
}
```

The reason why the SIMOL components are mapped to their attributes, i.e., in our example, to DISCRETE variables, is that there is no support for any object-oriented constructs in MINION.

- an *inrange expression*, as for instance `mdist = {1..3};` or `value = {2,4,5};`
- an *arithmetic or/and boolean expression*:

```
p1.dataAmount>=p2.dataAmount+p3.dataAmount;
```

has the following MINION representation in our implementation:

```
weightedsumgeq([1, 1], [Cell_p2_P2PMeter_dataAmount,
                  Cell_p3_P2PMeter_dataAmount], aux1)
```

```
weightedsumleq([1, 1], [Cell_p2_P2PMeter_dataAmount,
                  Cell_p3_P2PMeter_dataAmount], aux1)
ineq(aux1,Cell_p1_P2PMeter_dataAmount, 0)
```

where the *weightedsumgeq(constantVector, varVector, result)* and the *weightedsumleq(constantVec, varVector, result)* constraints state that *constantVector*varVector >= result* and *constantVector*varVector <= result*, respectively, where constantVector*varVector is the scalar product of constantVector and varVector, as presented in [68]. The *ineq* constraint ensures that *aux1 <= Cell_p1_P2PMeter_dataAmount*.

Note that in our compiler we treat arithmetic and boolean expressions with any number of operands and any number of operators, and therefore sometimes we have to internally divide an expression into two ore more expressions, so that we are able to automate the mapping process. Manually, the upper SIMOL expression could be translated to MINION in a more natural way:

```
weightedsumleq([1, 1], [Cell_p2_P2PMeter_dataAmount,
Cell_p3_P2PMeter_dataAmount], Cell_p1_P2PMeter_dataAmount)
```

In the automated implementation, we have to add an auxiliary variable `aux1` and divide the expression into:

```
aux1=p2.dataAmount+p3.dataAmount
p1.dataAmount>=aux1
```

Given that in MINION we do not have any *sum equals* constraint, we combine the defined MINION scalar product constraints to obtain the equality and, afterwards, we bind the auxiliary variable to the original left operand in an *ineq* constraint.

The MINION representation for the basic SIMOL arithmetic and boolean expressions can be found in Table 3.2.

- a *conditional block*:

```
if(codeset=1){
    mrate=80;
}
```

will become in MINION, for a `P2PMeter` instance p1 defined in the `Cell`:

```
reify(eq(Cell_p1_P2PMeter_codeset,1), ifCond0)
reifyimply(eq(Cell_p1_P2PMeter_mRate,80), ifCond0)
```

Table 3.2.: MINION representation for the basic SIMOL arithmetic and relational expressions.

SIMOL Expression	MINION Encoding
$a<b, a>b, a<=b, a>=b$	$ineq(a,b,-1), ineq(b,a,-1), ineq(a,b,0), ineq(b,a,0)$
$a=b, a!=b$	$eq(a,b), diseq(a,b)$
$a_1+a_2+..+a_n$	$weightedsumgeq([1,1,..1],[a_1,a_2,..,a_n],aux)$
	$weightedsumleq([1,1,..1],[a_1,a_2,..,a_n],aux)$
$a-b$	$weightedsumgeq([1,-1],[a,b],aux)$
	$weightedsumleq([1,-1],[a,b],aux)$
$a*b$	$product(a,b,aux)$
a/b	$div(a,b,aux)$

where $reify(constr, cond)$ ensures that *cond* is set to 1 if and only if *constr* is satisfied, whereas $reifyimply(constr, cond)$ states that *constr* has to be satisfied if *cond* is set to 1 [68].

The optional *else* branch will be similarly mapped to:

```
reifyimply(eq(Cell_p1_P2PMeter_mRate,120), !ifCond0)
```

- a *forall block*:

```
component Cell {
    constraints {
      ..
      P2PMeter s[3];
      forall(P2PMeter) {
        mdist=1;
        if(codeset=1) {
          mRate=8;
        }
      }
    }
}
```

becomes in MINION:

```
eq(Cell_s_P2PMeter_mdist[0],1)
reify(eq(Cell_s_P2PMeter_codeset[0],1), ifCond0)
```

```
reifyimply(eq(Cell_s_P2PMeter_mRate[0],8), ifCond0)
eq(Cell_s_P2PMeter_mdist[1],1)
reify(eq(Cell_s_P2PMeter_codeset[1],1), ifCond1)
reifyimply(eq(Cell_s_P2PMeter_mRate[1],8), ifCond1)
eq(Cell_s_P2PMeter_mdist[2],1)
reify(eq(Cell_s_P2PMeter_codeset[2],1), ifCond2)
reifyimply(eq(Cell_s_P2PMeter_mRate[2],8), ifCond2)
```

Thus, for each component of type `P2PMeter`, the statements inside the `forall` block have to be mapped to MINION constraints.

- an *exist statement*:

  ```
  exist(at_least(2), P2PMeter, mRate=12 kbit/sec);
  ```

  is translated to:

  ```
  occurrencegeq([Cell_s_P2PMeter_mRate,], 12,2)
  ```

  where the MINION constraint $occurrencegeq([Cell\_s\_P2PMeter\_mRate,],12,2)$ states that there are *at least 2* occurrences of the value *12* in the vector $[Cell\_s\_P2PMeter\_mRate,]$ [68]. For the other 2 SIMOL variants of the exist statement, we find in MINION the corresponding constraints as follows:

  ```
  exist(2, P2PMeter, mRate=12 kbit/sec);
  ```

  is translated to:

  ```
  occurrence([Cell_s_P2PMeter_mRate,], 12,2)
  ```

  ```
  exist(at_most(2), P2PMeter, mRate=12 kbit/sec);
  ```

  is translated to:

  ```
  occurrenceleq([Cell_s_P2PMeter_mRate,], 12,2)
  ```

- special functions like **sum**, **product**, **min**, and **max**:
  - `realR=sum([s], mRate)/3;`

    becomes in MINION:

```
sumleq([Cell_s_P2PMeter_mRate], aux1)
sumgeq([Cell_s_P2PMeter_mRate], aux1)
div(aux1, 3, aux2)
eq(Cell_realR, aux2)
```

where *sumleq*, together with *sumgeq* constraint, ensures that the sum of all elements of vector *Cell_s_P2PMeter_mRate* equals aux1.

- `total= product([s], dataAmount);`

will be in MINION, for a five elements vector *s*:

```
..
**VARIABLES**
DISCRETE Cell_s_P2PMeter_mdist[5] {0..1000}
DISCRETE Cell_s_P2PMeter_codeset[5] {0..1000}
DISCRETE Cell_s_P2PMeter_mRate[5] {0..1000}
DISCRETE Cell_s_P2PMeter_dataAmount[5] {0..1000}
DISCRETE aux1 {0..20000}
DISCRETE aux2 {0..20000}
DISCRETE aux3 {0..20000}
DISCRETE aux4 {0..20000}
..
**CONSTRAINTS**
..
product(Cell_s_P2PMeter_dataAmount[0],Cell_s_P2PMeter_dataAmount[1], aux1)
product(aux1,Cell_s_P2PMeter_dataAmount[2], aux2)
product(aux2,Cell_s_P2PMeter_dataAmount[3], aux3)
product(aux3,Cell_s_P2PMeter_dataAmount[4], aux4)
eq(Cell_total,aux4)
..
```

- `p2.dataAmount>=min([s,p1], dataAmount);`

where *s* is a vector instance and *p*1 and *p*2 are simple instances of type *P2PMeter*, is mapped to:

```
min([Cell_s_P2PMeter_dataAmount,Cell_p1_P2PMeter_dataAmount], aux1)
ineq(aux1,Cell_p2_P2PMeter_dataAmount, 0)
```

- `maxim=max([s,p1], dataAmount);`

is converted to:

```
max([Cell_s_P2PMeter_dataAmount,Cell_p1_P2PMeter_dataAmount], aux1)
eq(Cell_maxim,aux1)
```

3.5.2. SIMOL - MINION Mapping. Practical example

In the followings, we present an example of using SIMOL for modeling the **c17** combinational circuit from the ISCAS-85 benchmark suite, graphically depicted in Figure 3.4.

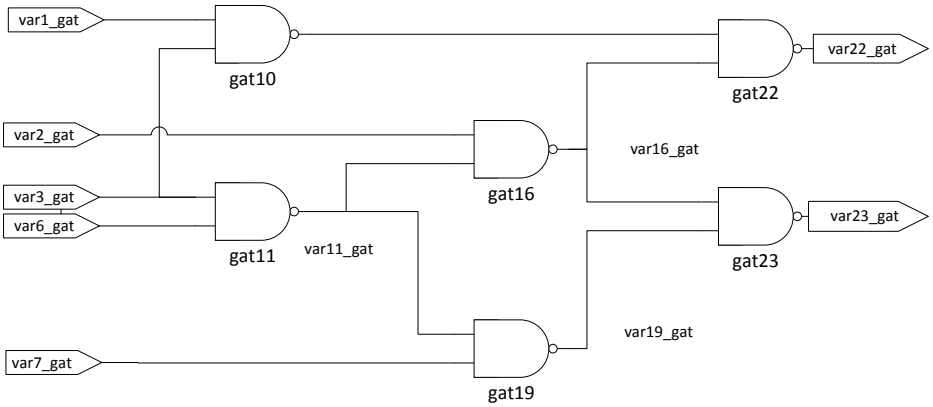

Figure 3.4.: The c17 combinational circuit.

The corresponding SIMOL model is as follows:

```
component Nand_Gate_2 {
    attribute bool out, in1, in2;
    constraints (nab) {
      if (in1 = true and in2 = true)
      {
        out = false;
      }
      else {
        out = true;
```

```
      }
      if (out = false) {
        in1 = true;
        in2 = true;
      }
    }
    constraints (ab) {; }
}

component ISCAS {
    attribute bool var_1gat, var_2gat, var_3gat, var_6gat, var_7gat,
    var_10gat, var_11gat, var_16gat, var_19gat, var_22gat, var_23gat;
    constraints {
      Nand_Gate_2 gat10;
      Nand_Gate_2 gat11;
      Nand_Gate_2 gat16;
      Nand_Gate_2 gat19;
      Nand_Gate_2 gat22;
      Nand_Gate_2 gat23;

      gat10.in1 = var_1gat;
      gat10.in2 = var_3gat;
      gat10.out = var_10gat;
      gat11.in1 = var_3gat;
      gat11.in2 = var_6gat;
      gat11.out = var_11gat;
      gat16.in1 = var_2gat;
      gat16.in2 = var_11gat;
      gat16.out = var_16gat;
      gat19.in1 = var_11gat;
      gat19.in2 = var_7gat;
      gat19.out = var_19gat;
      gat22.in1 = var_10gat;
      gat22.in2 = var_16gat;
      gat22.out = var_22gat;
      gat23.in1 = var_16gat;
      gat23.in2 = var_19gat;
```

```
      gat23.out = var_23gat;
    }
}
```

In order to present the MINION mapping of the upper SIMOL model in a compact manner, we will resume here the MINION variables declarations section. The complete MINION program can be found in Annex .3.

```
MINION 3

**VARIABLES**
BOOL ISCAS_var_1gat
..
BOOL nab_ISCAS_gat10_Nand_Gate_2
..
BOOL ab_ISCAS_gat10_Nand_Gate_2
..
**SEARCH**
..
**CONSTRAINTS**
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_in1,1), ifCond0),
                                    nab_ISCAS_gat10_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_in2,1), ifCond1),
                                     nab_ISCAS_gat10_Nand_Gate_2)
reify(watched-and({eq(ifCond0,1),eq(ifCond1,1)}), ifCond2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_out,0), ifCond2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_out,1), !ifCond2)
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_out,0), ifCond3),
                                    nab_ISCAS_gat10_Nand_Gate_2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_in1,1), ifCond3)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_in2,1), ifCond3)
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_in1,1), ifCond4),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_in2,1), ifCond5),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reify(watched-and({eq(ifCond4,1),eq(ifCond5,1)}), ifCond6)
```

```
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_out,0), ifCond6)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_out,1), !ifCond6)
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_out,0), ifCond7),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_in1,1), ifCond7)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_in2,1), ifCond7)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_in1,1), ifCond8),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_in2,1), ifCond9),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reify(watched-and({eq(ifCond8,1),eq(ifCond9,1)}), ifCond10)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_out,0), ifCond10)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_out,1), !ifCond10)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_out,0), ifCond11),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_in1,1), ifCond11)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_in2,1), ifCond11)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_in1,1), ifCond12),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_in2,1), ifCond13),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reify(watched-and({eq(ifCond12,1),eq(ifCond13,1)}), ifCond14)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_out,0), ifCond14)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_out,1), !ifCond14)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_out,0), ifCond15),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_in1,1), ifCond15)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_in2,1), ifCond15)
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_in1,1), ifCond16),
                                    nab_ISCAS_gat22_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_in2,1), ifCond17),
                                    nab_ISCAS_gat22_Nand_Gate_2)
reify(watched-and({eq(ifCond16,1),eq(ifCond17,1)}), ifCond18)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_out,0), ifCond18)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_out,1), !ifCond18)
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_out,0), ifCond19),
                                    nab_ISCAS_gat22_Nand_Gate_2)
```

```
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_in1,1), ifCond19)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_in2,1), ifCond19)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_in1,1), ifCond20),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_in2,1), ifCond21),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reify(watched-and({eq(ifCond20,1),eq(ifCond21,1)}), ifCond22)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_out,0), ifCond22)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_out,1), !ifCond22)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_out,0), ifCond23),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_in1,1), ifCond23)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_in2,1), ifCond23)
eq(ISCAS_gat10_Nand_Gate_2_in1,ISCAS_var_1gat)
eq(ISCAS_gat10_Nand_Gate_2_in2,ISCAS_var_3gat)
eq(ISCAS_gat10_Nand_Gate_2_out,ISCAS_var_10gat)
eq(ISCAS_gat11_Nand_Gate_2_in1,ISCAS_var_3gat)
eq(ISCAS_gat11_Nand_Gate_2_in2,ISCAS_var_6gat)
eq(ISCAS_gat11_Nand_Gate_2_out,ISCAS_var_11gat)
eq(ISCAS_gat16_Nand_Gate_2_in1,ISCAS_var_2gat)
eq(ISCAS_gat16_Nand_Gate_2_in2,ISCAS_var_11gat)
eq(ISCAS_gat16_Nand_Gate_2_out,ISCAS_var_16gat)
eq(ISCAS_gat19_Nand_Gate_2_in1,ISCAS_var_11gat)
eq(ISCAS_gat19_Nand_Gate_2_in2,ISCAS_var_7gat)
eq(ISCAS_gat19_Nand_Gate_2_out,ISCAS_var_19gat)
eq(ISCAS_gat22_Nand_Gate_2_in1,ISCAS_var_10gat)
eq(ISCAS_gat22_Nand_Gate_2_in2,ISCAS_var_16gat)
eq(ISCAS_gat22_Nand_Gate_2_out,ISCAS_var_22gat)
eq(ISCAS_gat23_Nand_Gate_2_in1,ISCAS_var_16gat)
eq(ISCAS_gat23_Nand_Gate_2_in2,ISCAS_var_19gat)
eq(ISCAS_gat23_Nand_Gate_2_out,ISCAS_var_23gat)
watchsumleq([nab_ISCAS_gat10_Nand_Gate_2,ab_ISCAS_gat10_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat11_Nand_Gate_2,ab_ISCAS_gat11_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat16_Nand_Gate_2,ab_ISCAS_gat16_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat19_Nand_Gate_2,ab_ISCAS_gat19_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat22_Nand_Gate_2,ab_ISCAS_gat22_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat23_Nand_Gate_2,ab_ISCAS_gat23_Nand_Gate_2,], 1)
```

```
watchsumgeq([nab_ISCAS_gat10_Nand_Gate_2,ab_ISCAS_gat10_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat11_Nand_Gate_2,ab_ISCAS_gat11_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat16_Nand_Gate_2,ab_ISCAS_gat16_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat19_Nand_Gate_2,ab_ISCAS_gat19_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat22_Nand_Gate_2,ab_ISCAS_gat22_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat23_Nand_Gate_2,ab_ISCAS_gat23_Nand_Gate_2,], 1)

 **EOF**
```

3.6. Analysis

We now discuss the expressiveness of the language by classifying its capabilities with respect to the framework offered in the chapter on configuration from [114]. In the context of the successful integration of constraint programming in solving a large variety of configuration problems, the author defines several distinguishing constraint models, each corresponding to a specific type of configuration problem. To set up the constraint model, the appropriate variables and constraints are deduced from the given configuration knowledge. The author states that this knowledge may have three different forms: the component catalogs, the component structure and the component constraints.

The catalog knowledge, as defined in [114], is modeled in SIMOL by means of:

- **generic components**: correspondent to the term of *technical types* in [114]),
- **concrete components**: derived (extended) from generic component/s or from other concrete component/s, in our case, and correspondent to the term of *concrete or functional types* in [114]).

Both generic and concrete components have a set of attributes, mapped to variables in the constraint model. Based on this kind of knowledge, we build the catalog constraints ([114]), which are stated over the set of variables and formulated by means of C_{attr_val} and C_{attr_attr} constraints.

The structural knowledge of a SIMOL model is determined by the component instances declared in the current model. In this manner, we generate for our system the set of subcomponents, that are either generic or extended components. The logic behind this mechanism has been previously detailed, when presenting the semantics of the language. The connection ports defined in [114] have no correspondent term in SIMOL yet, but the connection between component instances is possible through C_{attr_attr}

constraints. Also the statement in [114] according to which ”the sets of direct subtypes of two types are mutually disjoint” does not hold in our approach, because we accept multiple inheritance.

Finally, the configuration constraints are divided into compatibility constraints, requirement constraints and resource constraint. The first ones specify which value combinations are legal for the attributes given in the model and they are modeled in SIMOL through C_{attr_val} and C_{attr_attr} constraints. The requirement constraints describe a relation between two component attributes ([114]), which is best depicted by combining C_{cond} with C_{attr_val} or C_{attr_attr}. Moreover, the resource constraints on numerical attributes were intensively addressed throughout this paper.

Consequently, we find the expressive power of the language sufficient for modeling the discussed configuration knowledge forms. As also stated in [114], the configuration problem complexity may vary from very simple option selection problems to complex cases, but they all appear as combinations of the specified knowledge forms.

Chapter 4

SIMOA Approach

Parts of the content of this chapter have been published in [95, 91, 96, 94].

Integrating simulation and configuration specific features and solving techniques, in order to cope with a wide variety of systems and needs, was the main idea that led us to SIMOL - a language able to formalize the behavior for simulating a dynamic system and the knowledge necessary for carrying out configuration and, later on, optimization. In order to save investments, we decided not only to come up with the general modeling language SIMOL, but also to develop a tool and its underlying framework that is as general, and thus flexible, as possible.

In the following, we detail our approach. In Section 4.1, we present the basic SIMOA system architecture. Afterwards, in Section 4.2, we discuss the analogy between the reconfiguration and the diagnosis problem and, based on this, we present the SIMOA approach to reconfiguration.

4.1. SIMOA Architecture

The suggested SIMOA system architecture comprises three modules that are connected and interchange information, as depicted in Figure 4.1.

The central input to the tool is, besides observations, i.e., case specific knowledge, the domain knowledge. Both case specific knowledge and domain knowledge can be formalized using the SIMOL

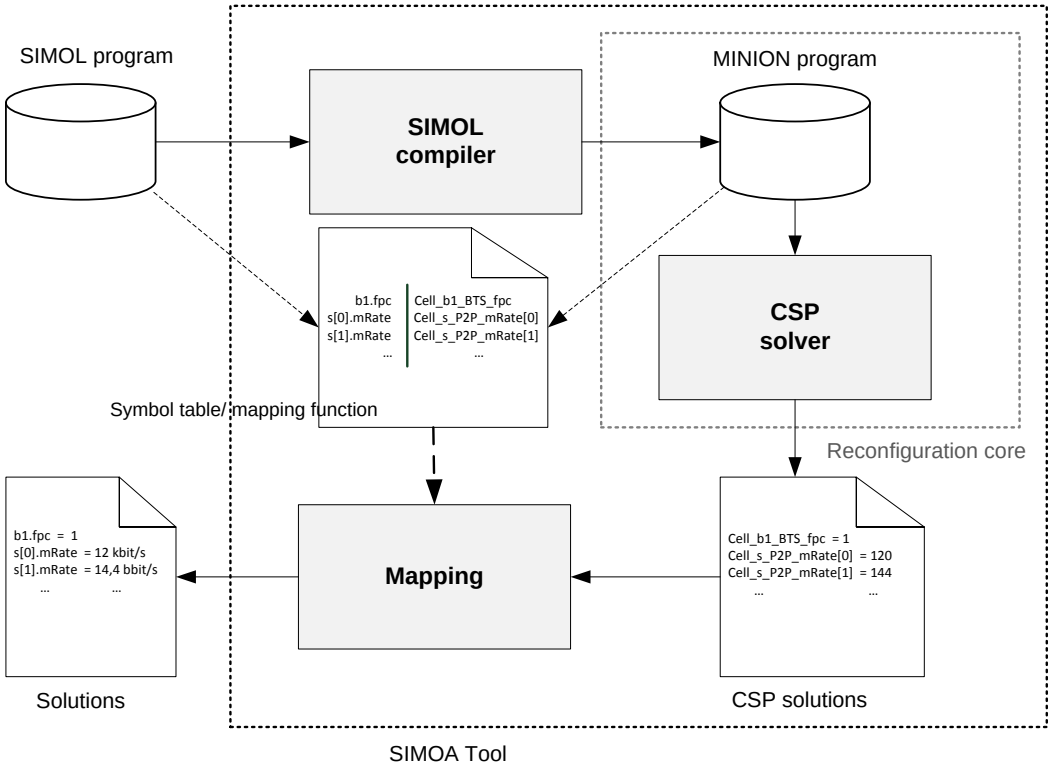

Figure 4.1.: The SIMOA architecture.

language. We have already discussed SIMOL in Chapter 3 of this thesis and therefore omit any information regarding the language. The program written in SIMOL is taken as input of the first module, i.e., the *SIMOL compiler*, which maps the program to a set of constraints. Moreover, the compiler generates a symbol table, that allows for mapping the variables used in the SIMOL program to the variables used in the constraint representation. This is necessary later on to map back the computed solutions.

The second module is the *CSP solver*. For more information on constraints and constraint solving we refer the interested reader to Rina Dechter's book [24]. As we have already mentioned, in our work we are using the MINION constraint solver [68]. Therefore, the SIMOL compiler generates a MINION program, which is used by the CSP solver for computing solutions. A solution to a

CSP problem is an assignment of values to variables, such that no constraint is violated. Hence, the result of the second module is basically nothing else than a list of variable assignments. This list of variable assignments is taken by the third module - the *Mapping module*. This module takes the solutions coming from the constraint solver and changes the variable names to the names used in the original SIMOL program. For this purpose, the generated symbol table is used. In this context, the reconfiguration core is represented by the CSP representation (MINION program) and the constraint solver, i.e., the constraint solver is responsible for determining reconfigurations. The CSP solutions in Figure 4.1 are mapped to the actual reconfigurations, depicted by *Solutions* in Figure 4.1.

Due to this architecture, the obtained tool is general and can be used to solve various tasks. The only requirement is that the domain and the case specific knowledge are specified using SIMOL. Chapter 5 presents in more detail the domain of mobile networks with machine to machine (M2M) communication, for which smart metering applications based on the SIMOA approach were implemented. In the remainder of this chapter, we present the reconfiguration mechanism used in the SIMOA approach.

4.2. Reconfiguration Mechanism

Regarding the underlying reconfiguration mechanism, two algorithms have been implemented up to now: one which considers the possibility to dynamically activate or inactivate specific components, and another algorithm that makes use of components' non-default functional modes.

4.2.1. MCDiag Algorithm

In this section, we discuss a simple diagnosis algorithm that allows for computing minimal cardinality diagnoses using a constraint solver. In our case, we use the MINION constraint solver [68]. The reason for relying on a constraint solver rather than a more sophisticated diagnosis engine, like Reiter's hitting set approach [111], or De Kleer and William's GDE [23], is the required expressiveness of the modeling language, where at least boolean and integer domains have to be handled appropriately. Of course, it is also possible to combine a horn-clause based reasoning system, like the ATMS [21], with a general problem solver, that has been tailored for an application domain. Moreover, in this scenario we obtain conflicts, which are used to compute the diagnoses. However, we are more interested in computing diagnoses directly from the model without the need of additional hitting-set computations.

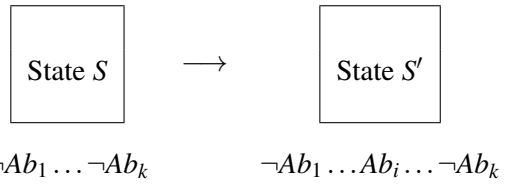

Figure 4.2.: Reconfiguration as a function changing the state of a system.

By exploring the analogy between diagnosis and reconfiguration, we are able to define the reconfiguration problem similar to the diagnosis problem, and therefore apply diagnosis algorithms for reconfiguration purposes.

Note that we restrict ourself to reconfigurations not changing the structure of the system, but parameters and other changeable functions that can be assigned to parts of the system. With this restriction, reconfiguration can be seen as a method for identifying changeable parts that enable a system transition, such that the resulting system fulfills the stated requirements. Moreover, the restriction ensures that all information needed is available and formalized. Figure 4.2 shows reconfiguration in a graphical way, where system states are represented using *Ab* predicates, stating whether a component (part) i of a system behaves as originally designed ($\neg Ab_i$), or it should be changed (Ab_i).

Formally, the reconfiguration problem can be stated as follows:

Definition 1 (Reconfiguration problem 1) *A tuple* $(SD, REQ, COMP)$ *represents a reconfiguration problem where SD denotes the configuration knowledge, i.e., the system structure and the provided functionality as well as general system constraints, REQ denotes the desired new system requirements, and COMP is a set of system parts that can be changed.*

This definition is almost equivalent to the definition of the diagnosis problem accordingly to Reiter [111] and indeed the relationship between diagnosis and (re-)configuration has been already pointed out (e.g., [131, 132]). The slight difference is that in this work we purely rely on consistency-based reasoning, ignoring fault models, which seems to be sufficient for our purpose.

A solution to the reconfiguration problem can be easily formalized using the definition of diagnosis from Reiter [111].

Definition 2 (Reconfiguration 1) *Given a reconfiguration problem* $(SD, REQ, COMP)$. *A subset*

$\Gamma \subseteq COMP$ *is a valid reconfiguration iff* $SD \cup REQ \cup \{\neg Ab_C | C \in COMP \setminus \Gamma\} \cup \{Ab_C | C \in COMP\}$ *is satisfiable.*

A valid reconfiguration Γ is minimal iff no valid reconfiguration $\Gamma' \subset \Gamma$ exist. A valid reconfiguration Γ is minimal with respect to cardinality iff no valid reconfiguration Γ' with $|\Gamma'| < |\Gamma|$ exist. In practice we are more interested in smaller reconfigurations, i.e., reconfiguration where we have to change only small parts of the original system. Therefore, throughout this section, we assume *minimal cardinality reconfiguration* when using the term minimal reconfiguration.

Due to the similarities of the definition of reconfiguration and diagnosis, we are able to use diagnosis algorithms directly for computing valid reconfigurations.

The **MCDiag**$(SD, OBS, COMP)$ diagnosis algorithm (Alg. 1) comprises basically two steps. In the first step, the MINION model is derived using the model *SD* and observations *OBS*, written in another formal language like SIMOL. The second step comprises the computation of all the diagnoses of cardinality *i*, where *i* ranges from 0 to the maximum number of available components $|COMP|$. The MINION solver is called restricting the solutions to a specific cardinality, which is performed by adding constraints regarding the total number of abnormal components. We use for this purpose the MINION constraints `sumleq(Ab,i)` and `sumgeq(Ab,i)`, which together specify that the sum of abnormal components should be equal to *i*. If a non-empty set of diagnoses is found for a specific *i*, the algorithm terminates and returns the diagnoses as solutions. Because of this construction, the algorithm guarantees to terminate and always computes minimal cardinality diagnoses.

Regarding the practical results of the upper presented diagnosis algorithm on solving reconfiguration problems, we present two scenarios from the domain described in Chapter 5. Note that further empirical results, showing that the algorithm can also be used for solving the diagnosis problem within a reasonable amount of time, are given in Chapter 6.

For the sake of clarity, we will restrict our modeling to a single cell of the network. Also the types and the number of components, that we may have in a cell are reduced, so that the reader could follow the process of reconfiguration. Let us assume that the cell serves a given number of smart meters, which communicate via a given number of channels and there are new desired requirements, that must be fulfilled.

In the first scenario, we consider the possibility to inactivate specific smart meters, if the constraints stated over their attributes are no longer satisfied. Consequently, the *Ab* predicate states whether the meter *i* is active, i.e., it behaves as originally designed ($\neg Ab_i$), or should be changed/ inactivated (Ab_i).

Algorithm 1 MCDiag$(SD, OBS, COMP)$

Input:A model *SD*, observations *OBS*, and a set of components *COMP*
Output: Minimal cardinality diagnoses

1: Generate the MINION model *MM* from *SD* and *OBS*
2: **for** $i = 0$ **to** $|COMP|$ **do**
3: Call **MINION** $\left(MM \cup \left\{ \begin{array}{l} \texttt{sumleq(Ab, i)} \\ \texttt{sumgeq(Ab, i)} \end{array} \right\} \right)$ and store the result in Δ_S.
4: **if** $\Delta_S \neq \emptyset$ **then**
5: **return** Δ_S
6: **end if**
7: **end for**
8: **return** $\emptyset$

That means, each constraint which contains the meter i will be connected with the $(\neg Ab_i)$ predicate. For instance:

$$\texttt{eq(smSet1\_SmartMeter\_dataAmount[0], 20)}$$

becomes the disjunction:

$$\texttt{watched} - \texttt{or}(\{\texttt{eq(ab[0], 1), eq(smSet1\_SmartMeter\_dataAmount[0], 20)}\})$$

where watched-or$(\{c1, ..., cn\})$ ensures that at least one of the constraints $c1, ..., cn$ is satisfied.

The new watched-or constraint states that, if the original MINION constraint cannot be satisfied, i.e., in this particular case, the dataAmount transmitted by $smSet1[0]$ is not equal to 20, then the meter $smSet1[0]$ must be inactive, i.e., the $eq(ab[0], 1)$ must be satisfied.

The MINION model, generated from the SIMOL description (depicted in Figure 4.3), - for the case with 51 smart meters - contains 270 constraints, stated over 14 variables, from which 3 are vectors with 20, 30 and 51 elements respectively and taking BOOL or DISCRETE values. The running time for computing the minimal cardinality reconfiguration (3, in our case) is 234 ms, but the obtained time strongly depends on the constrains types and may be improved by restricting the attributes of

```
component MyCell extends Cell {
     attribute int total;
     constraints {
       P2PNo=21;
       SmartMeter smSet1[20];
       SmartMeter smSet2[30];
       SmartMeter s1;
       forall(SmartMeter) {
         dataAmount=20 KB/day ;
       }
     }
}
```

Figure 4.3.: Partial SIMOL description of a cell with 51 *SmartMeter* components.

the inactive meters to default ”non-influencing” values, instead of identifying them as inactive and ignoring them afterwards (as if the meters would not exist).

In the second scenario, we assume the meters should be all active (not changeable), but the channels may be active, permitting data communication, or inactive (blocked for transmission). We also make the assumption that at least one channel is available for transmitting data, whereas the rest of the channels must be activated, if a constraint in the MINION model is violated. For instance, we will have:

```
sumleq([smSet1_SmartMeter_dataAmount, smSet2_SmartMeter_dataAmount,
                    s1_SmartMeter_dataAmount], aux1)
```

replaced by:

```
                watched-or({eq(ab[0],1), eq(ab[1],1),
sumleq([smSet1_SmartMeter_dataAmount, smSet2_SmartMeter_dataAmount,
                    s1_SmartMeter_dataAmount], aux1)})
```

and the following constraints are to be added in the constraint section of the MINION input file:

```
                reifyimply(eq(ab[0],1),ab[1])
          reify(ineq(BaseCapacity2, current_dataAmount,-1),ab[0])
```

```
reify(ineq(BaseCapacity3, current_dataAmount,-1),ab[1])
```

The disjunction `watched-or({eq(ab[0],1), eq(ab[1],1),`C_i`})` states that if the constraint C_i is not satisfied, then at least one of the two elements in the *ab* vector should be *true*, meaning that either one or two more channels should be activated.

The MINION model, generated for the case with 300 smart meters - contains 690 constraints and the running time for computing the minimal cardinality reconfiguration (1, in this case) is 218 ms. For 51 meters, the solution is computed in less than 30 ms.

4.2.2. RECONFIG Algorithm

While the afore described reconfiguration approach does only consider the possibility to activate/inactivate components, now, we assume that, for each component of the system, we know how to reconfigure the component. Here we borrow the idea coming from Model-Based Diagnosis (MBD) [111, 23] and introduce *modes for components*. Hence, every component has at least one mode. We assume the $default$ mode to be the standard mode of a component, and all other modes to be potential reconfiguration of this component. For simplicity, we introduce a function $modes : COMP \mapsto MODES$ mapping components from $COMP$ to their $MODES$. At least $default$ has to be element of $modes(c)$ for all components $c \in COMP$.

As discussed in Chapter 3, SIMOL allows for specifying models of systems comprising components and their modes. For example, in Lines 2–27 of our SIMOL program from Figure 3.1, the components `P2PMeter`, `FPC` and `BTS` were defined. `P2PMeter` and `BTS` have only one mode (i.e., the $default$ mode), whereas for `FPC`, 3 modes ($default$, $x1$, and *unknown*) are defined. Besides the structure and behavior of the system, the new system requirements were also defined in Lines 28–45 of the program from Figure 3.1.

Definition 3 (Reconfiguration problem 2) *A reconfiguration problem is a tuple* $(KB, COMP, MODES)$ *where* $KB = SD \cup REQ$ *is the knowledge base comprising the model of the system SD and the requirements REQ, COMP is a set of system components, and MODES is the set of functional modes for the elements of COMP.*

As we have already seen, all the information regarding the reconfiguration problem can be obtained from SIMOL programs. The program from Figure 3.1 allows us to derive the knowledge base KB,

which is the set of equations *constr* representing the semantics of the SIMOL program (for more details, see Section 3.4), the set of components $COMP = \{\texttt{P2PMeter}, \texttt{FPC}, \texttt{BTS}, \texttt{Cell}\}$, and the set of modes $MODES = \{default, x1, unknown\}$.

A solution of the reconfiguration problem is an assignment of modes to each component, such that the knowledge base, together with this assignments, is satisfiable.

Definition 4 (Mode assignment) *Given a set of components COMP and a set of functional modes MODES. A mode assignment M is a function* $M : COMP \mapsto MODES$ *mapping each component to one of its modes, i.e., for all* $c \in COMP : M(c) \in modes(c)$.

Having now all ingredients, we are able to formally state a valid reconfiguration as follows:

Definition 5 (Reconfiguration 2) *Given a reconfiguration problem* $(KB, COMP, MODES)$. *A mode assignment M is a valid reconfiguration iff* $KB \cup \{M(c) | c \in COMP\}$ *is satisfiable.*

In reconfiguration, we are interested in finding mode assignments that do not imply too many changes. Hence, we can use the number of required system changes to indicate the optimality of a reconfiguration. The number of changes necessary in a mode assignment is the number of used modes that are not equivalent to the *default* mode.

Definition 6 (Number of changes) *Given a reconfiguration M for a reconfiguration problem* $(KB, COMP, MODES)$. *The number of changes (NOC) of M is equivalent to the number of modes in M deviating from the default modes, i.e.,* $NOC(M) = |\{M(c) | c \in COMP \wedge M \neq default\}|$.

We say that a reconfiguration M is optimal with respect to its NOC, if it is minimal, i.e., there exists no other reconfiguration M' with $NOC(M') < NOC(M)$.

After stating the underlying definitions, we introduce an algorithm for reconfiguration, that is based on **ConDiag**, a diagnosis algorithm, presented in Section 6.3. Computing reconfigurations in our context is nothing else than searching for minimal mode assignments, i.e., mode assignments that are as close to the original assignments as possible. When assuming that small changes lead to a satisfiable knowledge base, it would be good to start search considering small deviations of mode assignments from the *default* mode first. The number of changes can be increased, if no solution is found. Therefore, an iterative algorithm seems to be sufficient.

Algorithm 2 RECONFIG$(KB, COMP, MODES, n)$

Input: A reconfiguration problem $(KB, COMP, MODES)$ and the maximum *NOC* n
Output: All minimal reconfigurations (up to the predefined cardinality n)

1: **for** $i = 0$ **to** n **do**
2: $CM = \{|\{M(c)|c \in COMP \wedge M \neq default\}| = i\} \cup KB$
3: $S = \mathcal{P}(\textbf{CSolver}(CM))$
4: **if** $S \neq \emptyset$ **then**
5: **return** S
6: **end if**
7: **end for**
8: **return** $\emptyset$

Algorithm 2 **RECONFIG** takes a reconfiguration problem and a maximum number of changes and computes all minimal reconfigurations. Algorithm 2 is an iterative algorithm that starts with no changes of modes and continues to search, if necessary, up to the predefined value n. The termination criteria before reaching n is given in Line 4, where a non-empty solution obtained from the satisfiability check is returned as result. In case no solution is found, the empty set is returned (Line 8). The **CSolver** is a constraint solver taking a set of constraints *CM* and is expected to return a set of mode assignments, if a satisfiable solution can be found. Otherwise, the empty set is returned, indicating that no reconfiguration of the given size is possible. The function $\mathcal{P}$ is assumed to map the output of the solver to a set of solutions.

In the SIMOA prototype implementation, we use MINION for this purpose, but every other constraint solver would also be sufficient, providing that it is capable of handling the constraints stored in *CM*. Line 2 of Algorithm 2 adds a new constraint to the model, stating that we are interested in finding solutions, that comprise exactly i modes that are not equivalent to $default$.

Algorithm 2 obviously terminates, assuming that **CSolver** terminates. The complexity is of $O(n \cdot k)$, where k is the complexity of **CSolver**. In the worst case, searching for solutions for a finite constraint satisfaction problem is exponential in the size of the problem. Therefore, **RECONFIG** is also exponential in the worst case.

However, in practice solutions can be found in a much faster way. See, for example, the results

reported in Section 6.3 and Section 6.4, where search for solutions up to a size of 3 is within seconds even for constraint satisfaction problems comprising up to 3,800 constraints. Although these results are for diagnosis (**ConDiag**), they also can be applied to reconfiguration, because of the similarity of the algorithms.

Further empirical results will be provided in Section 5.3.3, together with an analysis of possible MINION encodings for the functional modes defined in SIMOL.

4.2.3. Conclusion

As already mentioned, the relationship between diagnosis and reconfiguration has been pointed out by many authors (e.g., [17, 131, 132, 109]). However, unlike [17, 109], we do not interleave diagnosis and reconfiguration in our approach (the constraint solver determines reconfigurations directly, i.e., without using any previously computed diagnoses). In [131, 132], the authors introduce the use of model-based diagnosis for parameter reconfiguration, that is based on consistency-based diagnosis together with fault models. Note that there is a strong relationship between our approach to reconfiguration and the work done by Stumptner and Wotawa. The slight difference to their work is that our diagnosis algorithms for reconfiguration compute potential reconfiguration directly, instead of computing hitting sets of conflicts.

Regarding the published paper on model-based diagnosis, we refer the reader to Chapter 5, where we briefly introduce other algorithms for diagnosis and describe how our approach deviates.

Chapter 5

Case Study

Parts of the content of this chapter have been published in [96, 95, 94].

5.1. Motivation

The machine-to-machine (M2M) communication market is expected to grow in the coming years and to become the largest customer of mobile phone network operators. The application scenarios of M2M communication are many-faceted. Currently, the field of main application of M2M communication is *smart metering*, i.e., providing a direct link between power suppliers and electricity meters at home, and on tracking of vehicles and goods in the transportation industry. In this chapter, we focus on the first scenario (linking electricity meters at home).

These smart metering application scenarios do not only provide advantages for companies, but also additional benefits for the customers. For example, smart metering solutions have the potential to avoid blackouts by limiting power consumption far before a blackout might occur. Another example is CO_2 emission reduction in the electricity sector. More and more people produce and consume energy independently and feed unspent energy into the network. One hope in this context is to reach greater energy efficiency through short meter-reading intervals, as these allow the customers to control their energy consumption throughout the day. Energy network operators are then able to collect consumption data more frequently and to provide this data to end consumers in a transparent way. To complete all these tasks, intelligent communication solutions are needed, which are able to handle increased

data volumes over the wireless networks and especially over mobile GPRS/UMTS networks. Originally, communication infrastructures were not designed to fulfill the requirements of the mentioned M2M communication scenarios.

The *Kapsch Smart Energy Management System* for metering and building automation offers a future-proof, open, and manufacturer-independent solution, which fits the needs of all the groups involved: from end consumers, building owners and facility managers to network operators and energy suppliers. In this context, functionalities that support a detailed analysis of alternative smart metering scenarios are highly requested. In order to support such functionalities, simulations of various parts of a mobile network such as cell, backbone, and data storage have to be conducted. The major purpose of such a simulation is to check whether a network configuration can support a set of given user requirements. In order to simulate the dynamics of a mobile network, we treat each part of the mobile network, which is to be simulated, as a state machine with a predefined number of states. For example, in one particular state a component might be active whereas in another one this is not the case. Another example is a network configuration that may change from state to state in order to handle different communication requirements.

If a simulation fails (the requirements can not be fulfilled by the mobile network), a working *reconfiguration* of the given system has to be delivered. In particular, adding too many M2M devices to a cell configuration might lead to an unacceptable reduction of service availability, which can only be handled by changing the default value of the parameters in the cell (for instance, by increasing the number of serving frequencies) or by extending the communication network via adding new mobile phone cells . Note that when adding a new mobile phone cell to a network, in fact, a new base station is added to the system and the transfer data is redistributed to the cells. A cell configuration contains one base station, a number of metering devices, and a base load value.

The major contributions of this chapter are the following: (1) we provide important insights into the principles of knowledge-based reconfiguration in the domain of M2M communication networks. (2) we discuss major business cases that are related to the application of (re-)configuration technologies in the context of mobile phone networks and M2M applications.

This chapter is divided as follows. In Section 5.2 we discuss in more detail the major requirements regarding the support of M2M communication scenarios. Afterwards, we introduce two application scenarios of the SIMOA approach (Section 5.3). In Section 5.4 we discuss relevant business cases that are related to the application of M2M SIMOA technologies. With Section 5.5 we conclude this chapter.

5.2. Domain Requirements

Unlike today's mobile networks, the future GPRS (General Packet Radio Service) networks used in Automatic Meter Management (AMM) communication have to support the following requirements:

- The GPRS networks have to be highly reliable and extensible
- There must be a large IP address pool available
- There is a required data amount specified for AMM communication
- The amount of data communicated via a GPRS network varies depending on the AMM readout frequency and the used communication protocols, such as the DLMS/COSEM or the IEC61107 protocol. Note that the AMM readout frequency refers to the meter reading interval, i.e., for instance, an electricity meter can record the power consumption every 15 minutes.

These requirements are of great importance for the configuration process. The user should be able to add a large number of devices in the network and the device type is either already defined in the knowledge base or a new type can be additionally defined. The last two requirements limit the space of possible reconfigurations by means of constraints stated over the data amount transferred in the network.

Figure 5.1 illustrates a simplified cell from a smart grid, taking into consideration the GPRS data communication on the one side and the electric power transmission from the power generation station to smart meters, on the other side (depicted in gray). A smart grid is a power transmission network, which makes use of supplier and consumer information and communication infrastructure to improve the overall network performance in terms of efficiency, reliability, and sustainability in an automated way. Note that SIMOA simulation and (re-)configuration focuses on GPRS data communication settings.

As depicted in Figure 5.1, IP-enabled meters (Point-to-Point (P2P) individually addressable smart meters), that are connected directly to the WAN (Wide Area Network) using GPRS, communicate with the AMM application, without requiring data concentrators in the field. A data concentrator is a device that is connected with several smart meters via a wire. In this case all connected smart meters communicate to the AMM application over the data concentrator. Data concentrators are used, for example, in buildings with a large number of apartments where the costs for the additional wiring are low. The meters connected to data concentrators are called PLC (Power Line Carrier) meters.

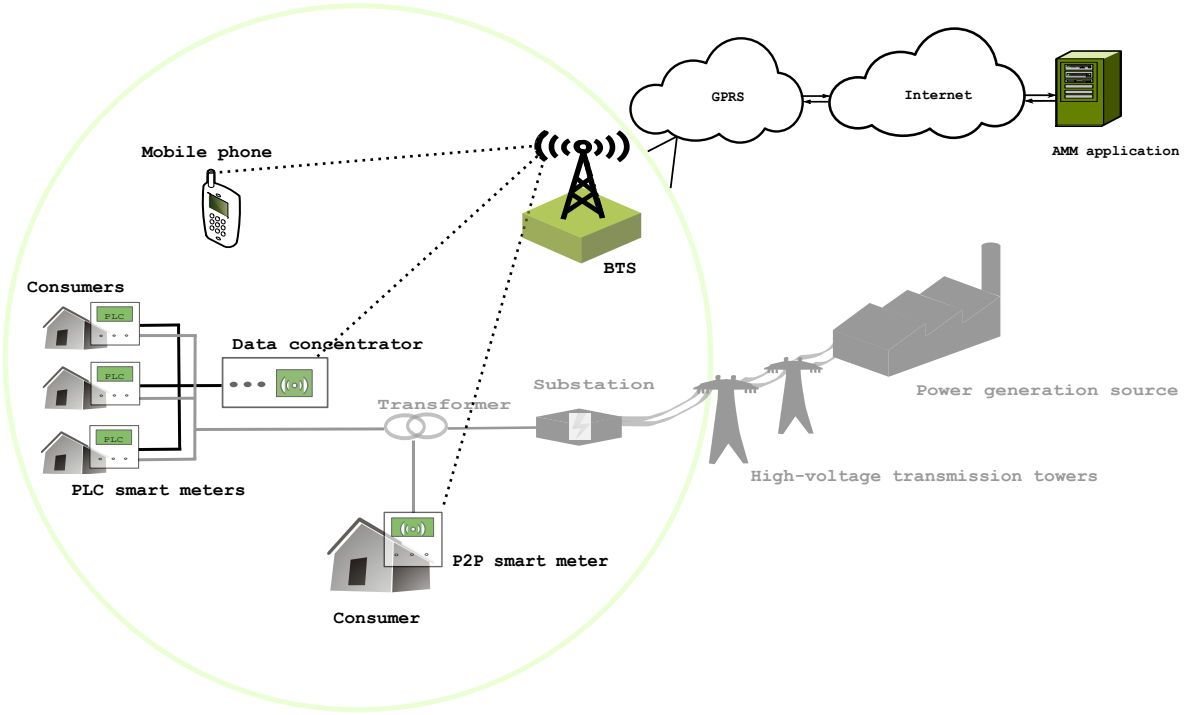

Figure 5.1.: Schematic representation of a *Smart Grid Cell* containing a base transceiver station (BTS), one P2P smart meter, 3 PLC smart meters, connected to one Data concentrator, and a Mobile phone. The base station facilitates at cell level the data transfer from the P2P meter and data concentrator to the central meter management office, called AMM application.

The usage of P2P meters becomes inevitable when the connection distance between a meter and a data concentrator is too large (more than 300 meters). In the simulation process we have to take into account the following variants of M2M terminals in a specific cell: (1) only P2P meters (2) only data concentrators with PLC meters, and (3) both P2P meters and data concentrators with PLC meters.

In Figure 5.1, the Smart Grid (GPRS) cell comprises the following parts:

- Base Transceiver Station (BTS): facilitates the wireless communication between user equipment (mobile phones, smart meters, etc.) and the network. The BTS has the equipment for encrypting and decrypting communications, spectrum filtering tools, antennas, etc. Typically a BTS has several transceivers (TRXs) which allow it to serve several different frequencies and allocates a

number of time slots for the M2M communication in the cell.

- P2P meters and Data Concentrators: these are described by configurable attributes that represent fixed or variable data amounts (derived from the chosen transfer protocols), network topology (determined by the distance between BTS and data concentrators/P2P meters), and - in the case of data concentrators - the number of PCL meters connected to them.
- GPRS base load represented by the Mobile phone in Figure 5.1 and which represents all the non-smart-metering traffic in the cell.

It is worth noting that the structure of mobile phone networks varies and that communication requirements depend on the M2M application scenario. A typical AMM scenario is, for instance, the *transfer of the daily load profiles* from the meters to the AMM application within a defined time period. This profile contains the amounts of power consumption recorded every 15 minutes. In this case, the traffic from the cell to the AMM application is investigated. Given, for instance, a GPRS cell with M2M communication capabilities comprising, among others, P2P meters, data concentrators, and mobile phones, we are interested in checking if the defined cell can handle the transfer of the load profiles or not. If the transfer is not possible (because there are insufficient time slots allocated for M2M communication in the cell or the transfer rate is too low due to the given channels encodings), a reconfiguration of the cell is required. This task comprises the identification of the inconsistent requirement/s in the default configuration as, for instance, the assignment of an insufficient number of time slots for M2M communication, and changing the values of the corresponding attributes such that the new configuration is consistent. In another scenario, a large number of *smart metering devices has to be simultaneously shut down* by the power supply company in order to avoid a blackout. In this case a pre-specified maximum delay (e.g., 2 minutes) between issuing the command and receiving it on side of the smart metering device must not be exceeded.

Hence, a solution for showing that a communication network fulfills the requirements of M2M application scenarios and taking counter measures in case of detected shortcomings has to be flexible and adaptable.

5.3. Applying the SIMOA Approach

From the application domain we are able to derive the requirements for a tool that provides the possibility of executing changes to the given mobile network, by taking into account future usage scenarios.

The future usage scenarios are determined by the communication needs of M2M applications (see, for example, the two scenarios discussed in the previous section). In the following, we discuss tool requirements with regard to the knowledge representation language, simulation features, reconfiguration, and optimization functionalities.

- *Language.* The application domain requires information on the structure and behavior of the network as well as on usage scenarios. Because of the huge variety of networks and usage scenarios, a modeling language has to be provided for formalizing the information. This language has to allow for stating the structure of a system and the behavior of its components. Temporal aspects, like changes in network access over time, have to be captured as well. Moreover, in case of optimization, the language has to provide means for stating optimality criteria.
- *Simulation.* The first step to evaluate whether an available mobile network is capable of handling all requests imposed by a future M2M application is to simulate the network behavior using the communication requirements of the M2M application. Communication requirements might be maximum delays of replies of remote machines on messages send or the required amount of data to be send at the same time. If the simulation returns that the network cannot handle certain requirements of the M2M application, someone is interested in finding a reconfiguration of the network, that allows for fulfilling the requirements. The simulation itself should be based on the introduced modeling language. Although there are a lot of languages for simulation used in industry, they are less appropriate for (re-)configuration purposes, because they cannot handle under-constrained problems, where a variety of solutions is possible.
- *Reconfiguration.* Changing an existing network for fulfilling communication scenarios imposed by M2M applications can be done on several levels. Certain parameters of mobile phone cells can be changed in order to provide more channels. New cells can be added to the network, thus changing the network topology. In our application all such possible changes should be reported. The same modeling language that is used for simulation should also be used for providing configuration information.
- *Optimization.* Since there is very likely more than one reconfiguration for handling future communication requirements, an optimal solution would be of great interest. Depending on who the user is, different optimality criteria are to be fulfilled. For instance, a mobile operator would prefer the reconfiguration which offers the best quality of service, whereas a customer would pick the solution with least costs for communication. Therefore, a requirement is that various optimality criteria can be specified and that it is possible to search for reconfigurations

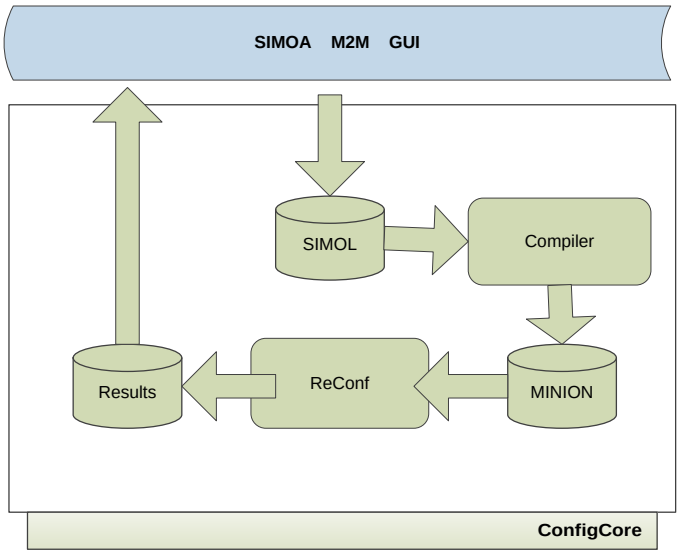

Figure 5.2.: The M2M SIMOA architecture.

fulfilling the optimality criteria.

The SIMOA approach, presented in Chapter 4, is capable of fulfilling all the mentioned requirements. In particular, SIMOL allows for specifying the structure and behavior of the mobile networks, as well as configuration information and requirements. Furthermore, the language can be used by non-AI experts, as it has a simple, Java-like syntax.

Based on the architecture presented in Section 4.1, the resulting M2M SIMOA system architecture, depicted in Figure 5.2, comprises at the highest level two parts: a graphical user interface (*SIMOA M2M GUI*) and the configuration core (*ConfigCore*). The latter is general and can be used in various applications, whereas the other is application specific and has to be tailored accordingly to the requirements. The configuration core itself comprises a compiler that translates models written in SIMOL into MINION constraints. The MINION program is used in the reconfiguration engine *ReConf* together with the MINION constraint solver to compute valid reconfigurations, which are given back as *Results*.

The interface between the graphical user interface and the configuration core (*ConfigCore*) is represented by the SIMOL program and the results obtained from *ReConf*. The SIMOL program comprises

the information necessary to specify the system to be reconfigured and the given pre-specified requirements. The reconfiguration result is basically a set of possible component modes, that are necessary to fulfill the requirements together with the computed variables values. The presentation of these results to the user has to be implemented in the user interface and is application specific.

5.3.1. The Domain-specific Tools

This section briefly describes two of the implemented smart metering applications, together with their specially tailored M2M user interfaces, that hide the domain knowledge from the user. The purpose of these applications is to show that a given set of smart meters can be used in a region, while still fulfilling all requirements.

The first SIMOA M2M GUI is depicted in Figure 5.3 and has been implemented in Java using the Google Web Toolkit*. The tool is available for all partners of the project via the web interface. Here, in order to simulate a specific configuration of a network's cell, the following steps will be conducted:

1. the user introduces the desired values for the meter profiles and cell parameters,
2. a SIMOL model is generated, based on the user's values and on a specific pattern file,
3. the SIMOL model is mapped to the MINION representation, and afterwards the solver checks, if the resulting CSP has any solution,
4. finally, the user gets two possible results, depending on the MINION outcome: the *simulation failed* (CSP with no solution found) or the *simulation was successful* (the user gets the transfer rate distribution over time (i.e., over the states) and the parameters of the simulation, as depicted in the lower part of Figure 5.3)

Note that, in this case, the simulation is limited to one cell with any number of P2P meters.

In Figure 5.4, the current user interface of the SIMOA M2M application is given. This graphical user interface enables the user to specify a smart metering application by placing the meters as well as the cells. These provide access to the mobile phone network, among other components at the appropriate positions. Moreover, the user might specify additional parameters for components. In case of a reconfiguration, the GUI generates a SIMOL program that makes use of the components, their behavior and additional parameters. It is also worth noting that also the positions of the components

*see `code.google.com/webtoolkit/`

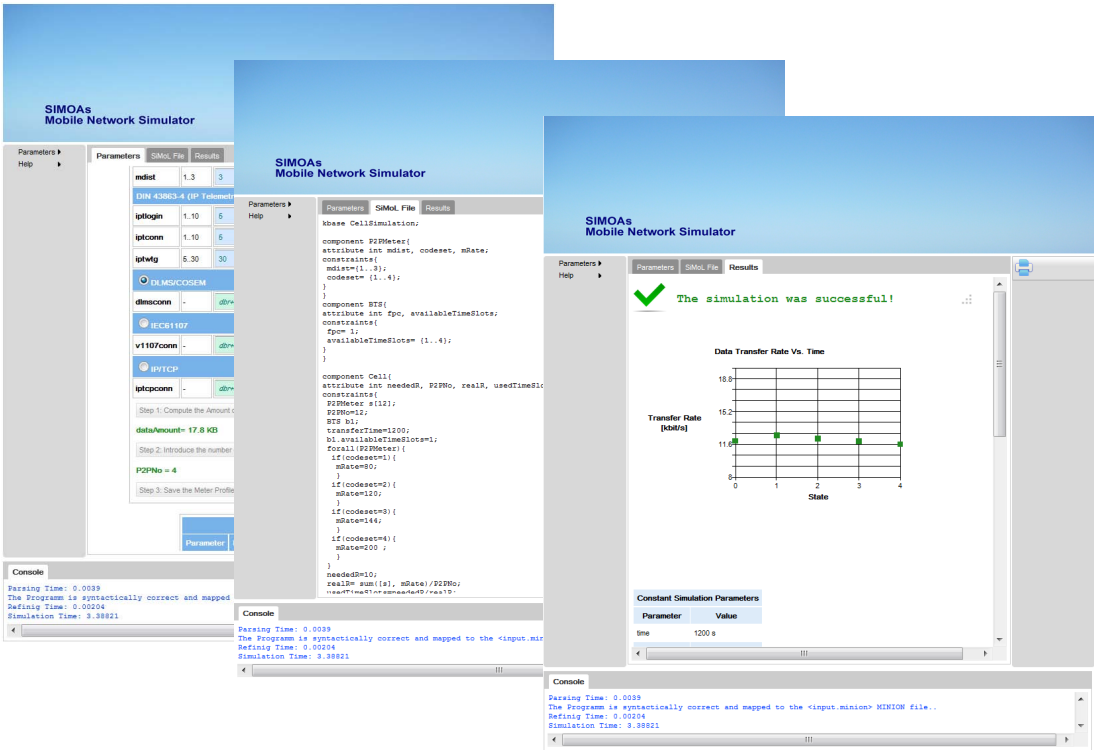

Figure 5.3.: (1) SIMOA M2M Web GUI.

in the map are used. For example, when specifying a base load for the mobile phone network, the concrete assignment to cells is done considering the distance between the base load and the cell. If a base load is not within reach, there is no effect. If a base load might influence two or more cells, the load is assigned to each cell accordingly to their distance, e.g., closer cells will have a larger percentage of the communication base load than cells that are more far away. Besides the map view capture, in the middle and lower part of Figure 5.4, we can find the integrated editor containing the generated *SIMOL file* and the provided *Explanations* for the obtained configuration.

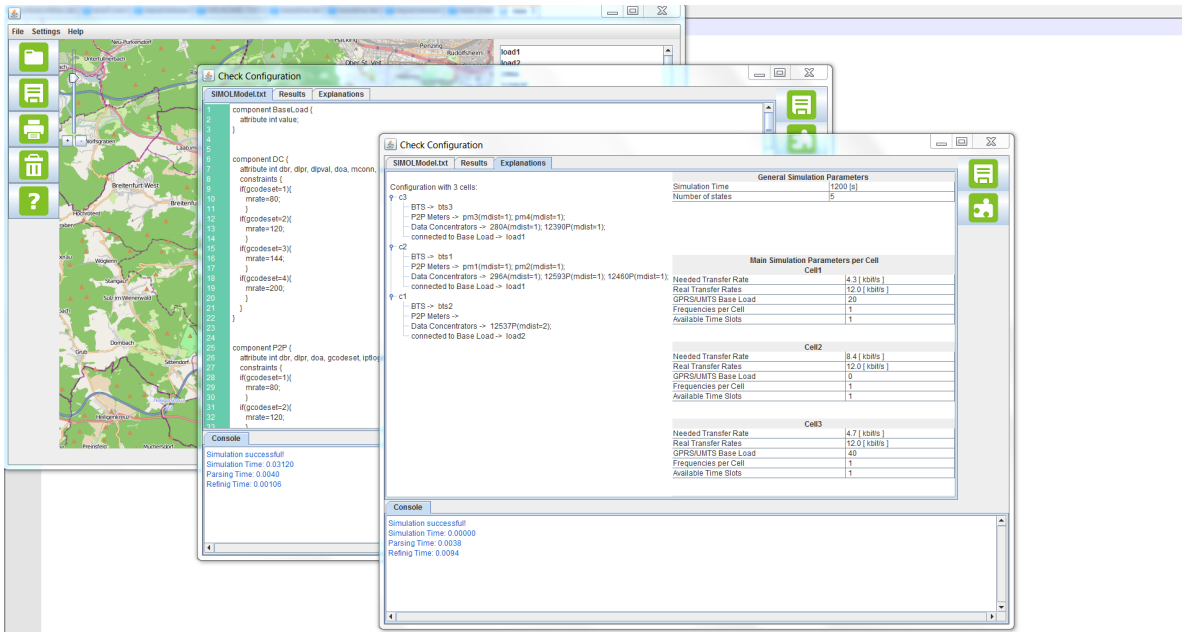

Figure 5.4.: (2) SIMOA M2M Map GUI.

5.3.2. Running Example without Reconfiguration Mechanism

In this section, we present in more detail the simulation process for a simplified cell with M2M communication capability, comprising one base transceiver station and five P2P meters meters. As illustrated in the web interface in Figure 5.3), both P2P meters and BTS present specific configurable parameters, based on which the simulator must generate the appropriate output, corresponding to the desired use case. In the following, we present one of the simulated use cases.

A typical scenario is the transfer of the daily load profiles from the meters to the central meter-management office. It means that only the up-link traffic of the cell is investigated. Assuming that the meters can only send this information from 12 a.m. to 7 a.m., the required transfer time is seven hours, that is 25,200 seconds. We also assume that, during this time period, the data transfer is uniformly distributed.

Consider the input parameters depicted in Table 5.1. The SIMOA simulator computes as outcome the resource utilization, i.e., the number of the used time slots, and the average, minimum and maximum data transfer speed in $kbit/s$ per cell (see Table 5.3).

The M2M SIMOA tool simulates the cell over a number of states. Each state corresponds to a

Table 5.1.: Input Parameters

Parameter Name	**Variable/Value**
Total number of P2P meters	$z = 5$
Number of P2P meters with CS1	v
Number of P2P meters with CS2	w
Number of P2P meters with CS3	x
Number of P2P meters with CS4	y
Data Amount per P2P Meter	$D_{Meter} = 20kB$
Required Transfer Time T	$00:00-07:00$

different code set distribution, where a code set is the channel encoding that influences the amount of data to be transferred within a second. See Table 5.2 for possible channel encodings. In order to simulate the dynamics of a real mobile network, we allow the cell passing from one state to another. Note that the model and the transition function between the states are defined in the SIMOL program, representing the system.

For every state, the tool computes the real transfer rate per cell R_{real} according to the following formula

$$R_{real} = (v * CS1 + w * CS2 + x * CS3 + y * CS4)/z,$$

where *CS1..CS4* refer to the channel encodings, given in Table 5.2. Consequently, we obtain the maximum data transfer speed in state $S0$ and the minimum data transfer speed in state $S1$ (see Table 5.3) .

Note that the needed data transfer speed R_{needed} differs from the upper mentioned real transfer rate and is computed using the following formula:

$$R_{needed} = z * D_{Meter}/T$$

In our example, the average (needed) data transfer speed R_{needed} becomes:

$$R_{needed} = 5 * 163840Bit/25200s = 32,51Bit/s$$

Table 5.2.: Channel Encoding

GPRS Coding Scheme	**kbit/s**
CS1	8 kbit/s
CS2	12 kbit/s
CS3	14,4 kbit/s
CS4	20 kbit/s

Table 5.3.: Real Transfer Rate per Cell

State: (v,w,x,y)	**Real Transfer Rate per Cell** R_{real}
S0:(0,5,0,0)	12 kbit/s
S1:(2,3,0,0)	10,44 kbit/s
S2:(2,2,1,0)	10,88 kbit/s
S3:(2,1,2,0)	11,36 kbit/s

It is easy to observe that, in each of the generated states from Table 5.3, R_{real} is much greater than the upper computed R_{needed}, which means that the requested data transfer is possible for all the codeset distributions.

But let us consider now a scenario in which the transfer would not be possible.

If we assume that the time duration is T=1 minute, then the needed data transfer speed R_{needed} becomes:

$$R_{needed} = 5 * 163840Bit/60s = 13,33kBit/s$$

Another important parameter that one must take into consideration at this point is the number of available GPRS time slots in the cell $TS_{available}$.

We should also mention that the number of the used time slots, depicted by TS_{needed}, is derived from the needed data transfer speed R_{needed} divided by the real transfer rate R_{real}, i.e.:

$$TS_{needed} = R_{needed}/R_{real}$$

Assuming that our BTS allocates $TS_{available}$=1 time slots for the M2M communication in the cell, the constraint that must be satisfied is:

$$TS_{needed} <= TS_{available}$$

In other words, if $R_{needed} > R_{real}$ and we have just one time slot allocated for the data transfer, the simulation will fail.

Experimental results

In order to prove the feasibility of the proposed approach, we conducted some experiments in the application domain using the model of a mobile network cell that communicates with a varying number of P2P meters. The number of P2P meters varied from 5 to 1,000, leading to quite large constraint systems, comprising up to 36,000 constraints and 16,000 variables. Table 5.4 shows the obtained running times when restricting to searching for 10 solutions at maximum and 4 temporal states. For the experiments we made use of a notebook with Intel(R) Core(TM) i7 CPU 1.73 GHz and 4 GB of RAM running under Windows 7.

The obtained results indicate that the approach is feasible for the application domain. Especially when considering that in most cases no more than 1000 P2P meters should be connected to one cell. Moreover, usually the simulation should only clarify whether there is a result for the intended configuration. Hence, finding a small number of solutions is acceptable in the M2M domain. In order to give an estimation on the impact of searching for more solutions, we also conducted an experiment where we restricted the variable domain to [0..1,000] and used Minion to search for a maximum of 60 solutions. Figure 5.5 shows the measured running times in comparison to searching for 10 solutions. From this figure, we see an increase of the running time. However, the results are still in an acceptable range and we are able to proof feasibility with respect to our application domain.

In future experiments we will use more sophisticated models from other domains in order to explore the influence of the models for solving constraints using SIMOL as modeling language. Moreover, there are many ways of representing SIMOL as a set of constraints, which might also influence the overall running time. Exploring different mappings from SIMOL to Minion has to be done in future research.

Table 5.4.: Simulation results obtained for different integer variable domains. The solution number is limited to 10, by making use of the MINION command-line switch $-sollimit10$ when running the MINION Solver. The given running time is a rounded average value over 3 runs.

SIMOL Program		**MINION File**		**Simulation Time [sec] for different domains**				
P2P Meters No.	**LOC**	Constraints No.	Variables No.	[0..100]	[0..200]	[0..500]	[0..1,000]	[0..10,000]
5	46	215	97	0.015	0.031	0.046	0.047	0.078
10	46	406	178	0.031	0.046	0.046	0.062	0.078
100	46	3,507	1,620	0.320	0.320	0.350	0.420	0.700
500	46	19,015	8,020	1.960	2.070	2.260	2.400	na
1,000	46	36,010	16,011	4.570	4.630	4.800	5.780	na

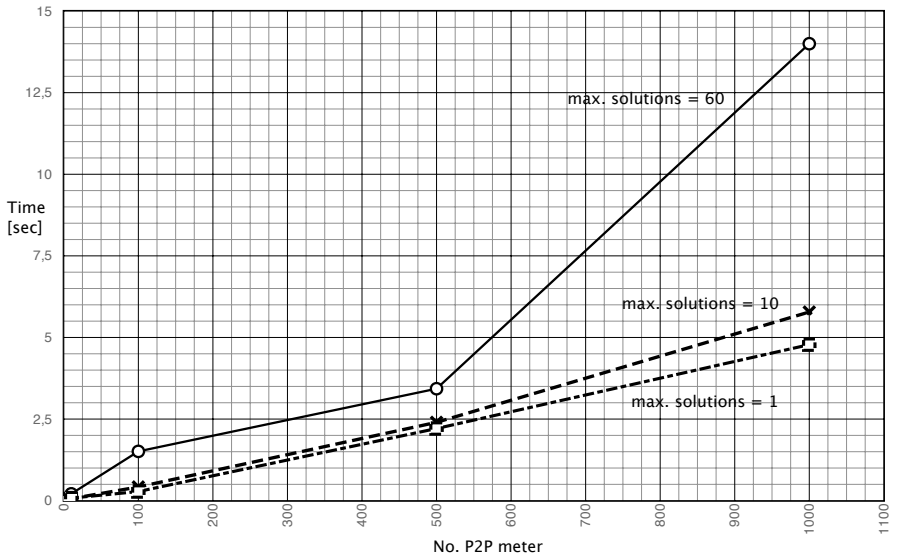

Figure 5.5.: Comparing running times for constraint solving for a specific maximum number of solutions when the variable domain is fixed to [0..1,000].

5.3.3. Running Example with Reconfiguration Mechanism

In this section, we discuss first the implemented GPRS Cell Model in more detail, starting from its UML Diagram and analyzing it. Afterwards, in Section 5.3.3, we report on first empirical results obtained for the reconfiguration algorithm introduced in Section 4.2.2 , when using a SIMOL model

of our M2M application domain.

Figure 5.6 presents the UML diagram for a `GPRS cell` used in the SIMOL model and thus within our current implementation. As one can see, a cell owns one `BTS` component, a limited number of `P2P Meter` - and `Data Concentrator` components, and a `Base Load` component. Moreover, the `GPRS Cell` has one attribute that refers to the possible GPRS/UMTS base load (`gbase`), i.e., the non smart metering related data transfer, which is in fact defined in the `value` attribute of the `Base Load` component. A `P2P Meter` component type is described by the distance between the P2P meter and the BTS (`mdist`), the used codeset (`codeset`), i.e., the channel encoding that influences the amount of data to be transferred within one second, and by the various parameters corresponding to the protocol suites used to retrieve data from a smart device (`dbr` billing reads, `iptlogin` M2M gateway logins, etc.). Many of these attributes are to be found in the definition of the `Data Concentrator` type, but mostly with values depending on the number of PLC meters connected to the concentrator. Both `PLC Meter` and `P2P Meter` types are derived from the generic type `Smart Meter`. A `BTS` component has two attributes: the number of frequencies (`fpc`) and the number of time slots (`availableTimeSlots`), which the BTS allocates for M2M communication in a cell. Note that, unlike the other attributes depicted in Figure 5.6, these two can be automatically reconfigured in case of a failed simulation. In order to do that, two ”fake” component types `FPC` (Frequencies Per Cell) and `ATS` (Available Time Slots) are introduced in the model (see Figure 5.6). They are ”attributes-seen-as-components” types and, when linked to the actual attributes `fpc` and `availableTimeSlots` they are able to reconfigure the given system. Note that in the SIMOL program from Figure 3.1 (used to discuss the syntax in Chapter 3), we can find the component `FPC` (Lines 9–20), where the `value` attribute of the `FPC` component has to be equal to 1 by default or in the set $\{2..4\}$ - in mode `x1`. In the `unknown` mode, no constraint is defined as the behavior of some `FPC` instances may become unimportant in a certain reconfiguration. For instance, when thinking of the possibility to deactivate smart metering devices, there is no contribution of inactive devices to the overall system behavior. Hence, in this case no constraints can be defined for the mode representing an inactive component.

The complete SIMOL program corresponding to the model defined in Figure 5.6 can be found in Annex .3.

Experimental results

In this section we report on first empirical results obtained using a SIMOL model of our application domain, i.e., smart metering. The SIMOL source code has 95 lines of code. When compiling the

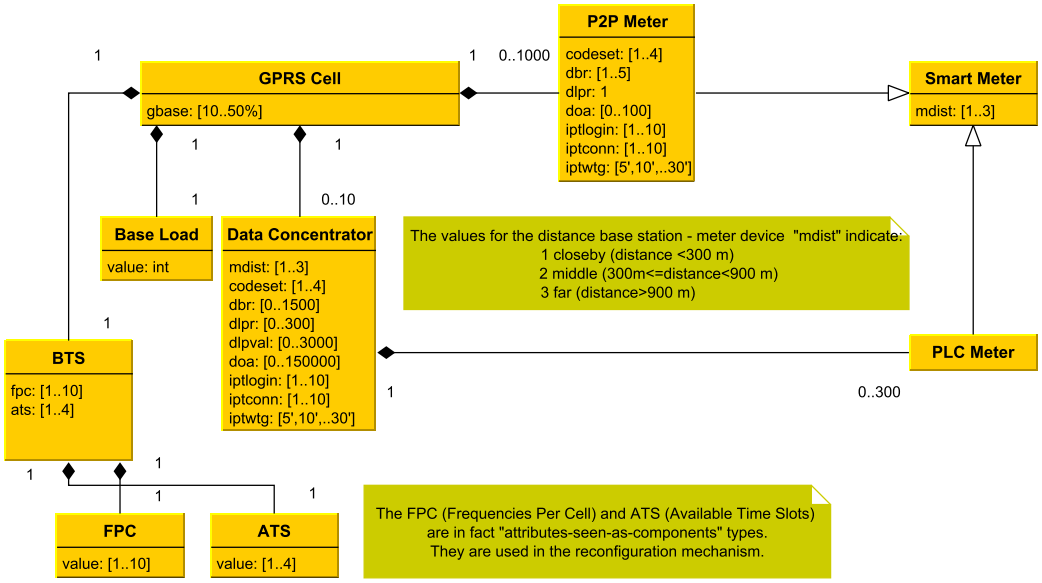

Figure 5.6.: UML Diagram for the GPRS Cell Model.

SIMOL program to its MINION representation, considering at the maximum 5 states, we obtain up to 2,387 variables and 7,320 constraints depending on the the number of smart meters considered. In principle, there are many possibilities of mapping SIMOL to MINION and also for computing solutions for a given maximum number of changes *NOC*. In the following we discuss the encoding of SIMOL modes within MINION and show that the choice of certain MINION parameters influences the computation of reconfigurations substantially.

The mode encoding in MINION is rather straightforward. In principle a mode of a component can be either active or inactive. Therefore, we map each mode $mode_x$ to a Boolean variable in MINION, which is 1 (true) if the corresponding component is in mode $mode_x$, or 0 (false) otherwise. In order to compute a solution for a particular *NOC* we have somehow to maximize the number of $default$ modes in the solution. In the first version of our implementation we used the *MAXIMISING* option of MINION for this purpose. In addition, we decided to control the way the solver searches for a solution also by directly specifying the instantiation order for the MINION variables representing a mode. Hence we used the MINION variable ordering (*VARORDER*) as well as the corresponding value ordering (*VALORDER*) with the following settings: for all the $default$ mode variables their values should be searched in descending order, whereas for the other mode variables the searching

should be done in ascending order. The intuition behind is to prefer solutions with more *default* modes to be true over the other solutions.

For the experiments we made use of a notebook with Intel(R) Core(TM) i7 CPU 1.73 GHz and 4 GB of RAM running under Windows 7. We obtained the results presented in the upper diagram of Figure 5.7 for models containing a rather small number of `P2PMeter`s ranging from 7 to 50. It is worth noting that when using the *MAXIMISING* function the measured running times exceeded 300 seconds for more than 100 meters (which is unacceptable in some situations). Hence, we decided to use only the *VARORDER* and *VALORDER* and ignore the *MAXIMISING* function. From the results depicted in the bottom diagram of Figure 5.7 we see a substantial improvement in the measured running time. Note that the obtained results without *MAXIMISING* were always correct.

From the diagram at the bottom of Figure 5.7 we can extract two observations. First, when checking only that a given system fulfills the requirements, the running time, even in case of 100 `P2PMeter`s, is within seconds. Second, even for a NOC of size 6 the reconfiguration time never exceeds 25 seconds. Since, for the application domain these running time results are sufficient and the proposed approach is feasible.

5.4. Business Cases

For (re-)configuration in the context of M2M applications there are many business opportunities supported using the presented SIMOA approach. We discuss three of the business cases where configuration technology can be effectively used in the context of mobile phone networks and M2M applications.

- For *M2M application providers* there is the opportunity to clarify whether their requirements regarding the mobile phone network capabilities (e.g., bandwidth or available communication slots) can be fulfilled. Moreover, they are able to specify certain future scenarios to predict and ensure availability of their services. Hence, M2M application providers gain the advantage of being able to show their customers that the services they offer can be carried out (also in the future) and this is the key differentiator to the competitors.
- *Mobile network providers* can exploit (re-)configuration technologies to adapt and change network parameters and the topology to fit the need of M2M application providers. Via optimization, the least expensive solution or the one that maximizes revenue can be computed. Hence,

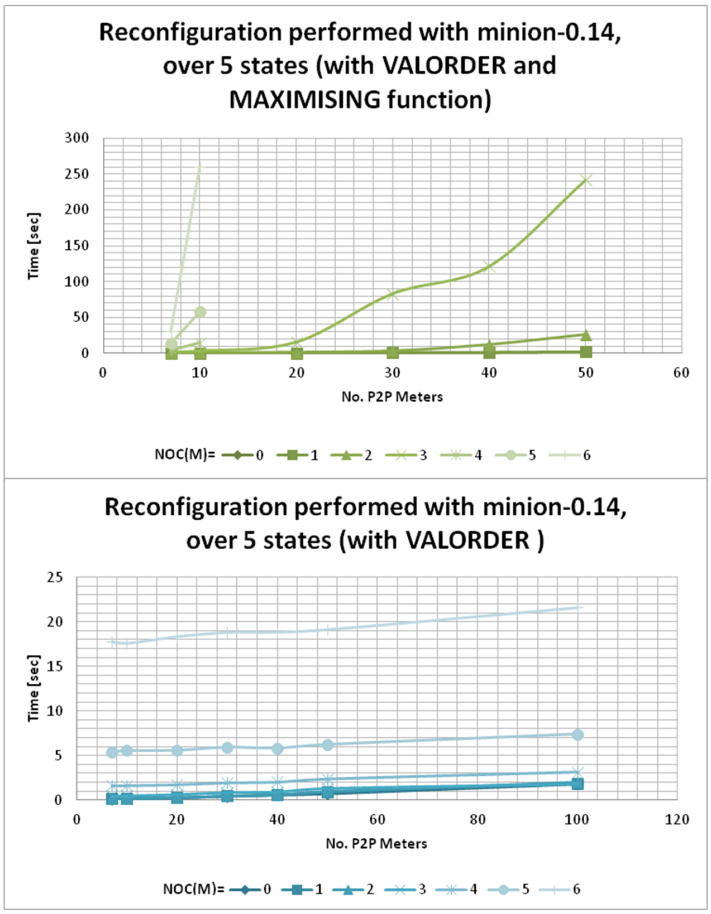

Figure 5.7.: Comparing running times for reconfiguration when the Integer variable domain is fixed to [0..20,000] and the number of solutions is limited to 1.

mobile operators gain the advantage of being able to react almost immediately to requirements demanded by their customers when using reconfiguration technology based on models. Moreover, when using model-based approaches, adaptations caused by topological changes or technology changes are rather straightforward. Costs of adaptations for parts of the models are lower compared to developing a new configurator from scratch for each configuration problem.

- There is also a business opportunity for *network equipment vendors*. The SIMOA tools allow for computing reconfigurations of networks in order to meet future requirements. Hence, they can actively promote products and services to network service providers, which are based on

well founded results obtained using system models and requirements. Because of the flexibility of modeling, SIMOA is not restricted to one customer and can be easily adapted to handle different networks, network components, and services.

In addition to these business cases there are two advantages of model-based configuration worth mentioning. One is the fact that the obtained results (used either to verify whether a system meets given requirements or to compute reconfigurations) are well founded because they are based on system models. Another advantage is that system changes can be easily incorporated into the system models. Hence, there is no need to change the underlying simulation and configuration engine. Moreover, changes are typically local at the level of components and do not influence larger parts of the model. Although the initial modeling of an application might be demanding and more expensive, the overall costs of model-based systems considering the costs of necessary maintenance activities are usually smaller compared to very specific solutions, e.g., using rule-based systems methodologies.

5.5. Conclusions

Because of the still increasing amount of data that is transported using the currently available mobile networks' infrastructure, there is a growing need for actively controlling the infrastructure in order to prevent delays or breakdowns in the worst case. The requirements of the application domain indicate a general language that allows for modeling technical systems comprising components which are interconnected. Moreover, the language should be able to formalize the behavior for simulating the system and the knowledge necessary for carrying out configuration and later on optimization. A language that is restricted to one M2M scenario is likely not capable of handling different new scenarios at the same time. In order to save investments we decided not only to come up with the general modeling language SIMOL, but also to develop a tool and its underlying framework that is as general as possible.

Our focus was to investigate and solve problems in wireless networks under the assumption of a future user behavior and growing M2M applications. In particular, we are interested in reconfiguring a mobile network in case of bottlenecks caused by requirements coming from future M2M applications. The result of the suggestions for reconfiguration coming from the tool might be changes in the structure or the parameters of the mobile network. It is worth noting that only a flexible approach, like model-based configuration, is capable of handling the needs and requirements coming from the application domain.

In summary, today's configuration technology offers a lot o benefits for the M2M application domain, i.e.: (1) providing evidence that a certain application scenario can be carried out in a current mobile phone network given the system model and the requirements, (2) suggesting recommendations for changing the system's parameters or structure in order to meet requirements, and (3) being flexible enough for adaptation to different usage scenarios. The latter is of particular interest in order to save investments and to cope with the fast-paced M2M application domain.

Chapter 6

Extensions

Parts of the content of this chapter have been published in [91, 92, 97].

Explanations for unexpected or even faulty system behavior are a most welcome asset when faced with such behavior during system development or maintenance. The task of diagnosing a system, that is, identifying root causes for encountered errors, got a lot of attention in the AI community, so that in the last three decades many different AI-based approaches tackling this issue have been emerging. Therefore, as extension to our work on reconfiguration, we discuss here the implemented diagnosis algorithms and also address the questions: (1) which algorithm to prefer in a certain situation and (2) whether publicly available general reasoning engines can be used for an efficient diagnosis.

First we introduce in Section 6.1 several general definitions and discuss related work, including available algorithms. Afterwards, we refer to our diagnosis algorithm, introduced in Section 4.2.1, by presenting some empirical results obtained from experiments using the well-known ISCAS-85 benchmark suite (Section 6.2).

Another algorithm, meant to compute this time all minimal diagnoses up to a given size, is **ConDiag**. Described in Section 6.3, **ConDiag** tightly couples constraint solving with diagnosis in order to compute minimal diagnosis up to a predefined cardinality. The basic idea behind **ConDiag** is to use the constraint solver directly to compute diagnoses of a given size. In order to do so we represent the predicate stating the correct behavior of a component as variables *nab* and let the constraint solver search for a valid setting of the *nab*s. Moreover, we are able to compute minimal diagnoses up to a

given size by searching from smaller diagnoses to larger ones and adding information about the previously smaller computed diagnoses during the process. Hence, there is no further need for minimizing results and computing conflicts. First empirical results indicate that the running time of **ConDiag** is comparable to specialized diagnosis algorithms that makes it a valuable alternative.

Finally, in Section 6.4, we compare two classes of diagnosis algorithms. One class exploits conflicts in their search, i.e., sets of system components whose correct behavior contradicts given observations. The other class ignores conflicts and derives diagnoses from observations and the underlying model directly. In our study we used different reasoning engines ranging from an optimized horn-clause theorem prover to general SAT and constraint solvers.

6.1. Preliminaries and Related Work

[111] formalizes a model-based diagnosis approach based on a system description's consistency with actually observed behavior: A system description *SD* defines the nominal behavior of a set of interacting components $c \in COMP$ via sentences $\neg AB(c) \Rightarrow NominalBehavior(c_i)$, where $AB(c)$ is an assumption whether a component c's status is *abnormal* or not and *NominalBehavior* defines the system's correct behavior (Reiter uses first order logic). As Reiter includes no knowledge about faulty behavior, his approach is considered to implement a weak fault model. Given some actual observations *OBS*, a system is considered to be at fault iff $SD \cup OBS \cup \{\neg AB(c) | c \in COMP\}$ is inconsistent.

Definition 7 (Diagnosis 1) *A diagnosis* $\Delta \subseteq COMP$ *is a subset-minimal set such that* $SD \cup OBS \cup \{\neg AB(c) | c \in COMP \setminus \Delta\}$ *is consistent.*

Reiter proposed to derive diagnoses explaining the inconsistent observations via conflicts. His reasoning is based on the fact that for his definitions, the set of diagnoses is equal to the the set of minimal hitting sets of the set of (not necessarily minimal) conflicts, i.e., if such a set includes at least all minimal conflicts.

Definition 8 (Conflict) *A set* $C \subseteq COMP$ *is a conflict if and only if* $SD \cup OBS \cup \{\neg AB(c) | c \in C\}$ *is inconsistent. If no proper subset of* C *is a conflict,* C *is a minimal conflict.*

Reiter's complete algorithm maintains and prunes a tree that encodes the search pattern and intermediate results in order to achieve a structured and complete exploration of the diagnosis search

space that is exponential in the number of components. [48] proposed an improved version, HS-DAG, using a directed acyclic graph and addressing minor but serious flaws in Reiter's formulations. [145] suggests with HST a variant of Reiter's idea that tries to avoid building nodes that would be pruned anyway.

[23] used an assumption-based truth maintenance system (ATMS) [21] to deduce the set of conflicts for an observation, where their General Diagnosis Engine (GDE) then derives via hitting set computations the desired diagnoses. If re-framing the diagnosis problem into an optimal constraint satisfaction problem, the conflict-directed A* algorithm [143] generates the diagnoses incrementally in best-first order and, additionally, uses conflicts to focus the search.

[128] introduced the switching diagnostic engine SDE that interleaves the search for diagnoses and conflicts, exploiting the dual relation between diagnoses and conflicts via minimal hitting sets also in the reverse direction. Also interested in minimal conflicts, [71] introduces a preference-controlled algorithm, based on a divide-and-conquer search strategy. Starting from the idea that previous conflict detection algorithms have not exploited the basic structural properties of constraint-based recommendation problems, [121] came up with another algorithm for an efficient identification of minimal conflict sets, based on a table representation of the input and inspired by HS-DAG. While [87] aims at computing the minimal diagnoses directly via non-minimal ones, conflicts are still computed and used to prune the search space.

[43] proposed to directly manipulate logic models when searching for a diagnosis, without computing conflicts. Via the notion of satisfiability, one can easily encode an MBD problem. That is, introducing the corresponding variables $AB(c)$ and connecting them to nominal behavior as above, one searches in a single query for a solution (or all, depending on the engine) up to some problem bound k limiting the diagnosis cardinality. Starting with 1, k is incrementally increased when no solution is found (anymore). As we search for all solutions in our scope (to be complete), we have to add each diagnosis Δ found as a blocking clause (in the form of $\neg\Delta$ when considering Δ as conjunction of its elements) to the problem, in order to exclude itself and its supersets from further search. This way, the subset-minimality of derived diagnoses is ensured, and incrementally raising bound k enables us to derive all diagnoses up to some desired cardinality. Essential, however, are a reasoning model and an engine that allow us to limit k in the search.

[33] proposed to use MaxSAT in this respect, and implemented with MERIDIAN a corresponding approach. Via Odd-Even Mergesort (OEMS) networks [10], or Cardinality Networks [4] this requirement can be easily encoded in the Boolean domain to be directly attached to a model (obviously one

could describe these networks also with constraints). An example approach in this direction is [83], which uses constraints as intermediate format that are compiled into a Boolean satisfaction problem. In this category we can place also [92] **ConDiag** (see Section 6.3), our algorithm capable of obtaining diagnoses directly from a constraint description, using a general purpose constraint solver as reasoning engine.

There is yet another category of diagnosis algorithms computing diagnoses directly from the model without deriving hitting sets of conflicts. All these algorithms have in common that they are based on tree-structured models. [32] and later [133] described algorithms that exploit tree-structured constraint systems. [117] generalized these algorithms. [134] discuss the coupling of decomposition methods for constraint satisfaction problems with tree-structured diagnosis algorithms, in order to make those compatible with non-tree-structured models.

6.2. MCDiag Algorithm for Diagnosis

It is worth noting that the algorithm presented in Section 4.2.1 cannot only be used for reconfiguration purposes. Moreover, someone might be interested in comparing the performance of such an algorithm with more traditional diagnosis algorithms. Although, an in-depth comparison has been left for future research, we have performed some initial experiments using the well-known ISCAS 85 benchmark suite*. We converted all circuits to the MINION input format and computed minimal cardinality diagnoses for one test case. Note that in the experiments all minimal cardinality diagnoses are single fault diagnoses. The results have been obtained, using a certain input for the circuit and changing one output. The initial results are depicted in Figure 6.1.

When comparing the obtained results with other published diagnosis results, e.g., [106], we see that the diagnosis time for the smaller circuits is similar. But for larger circuits **MCDiag** requires more time. This comparison is of course not significant and further experiments are needed. However, the results indicate that the direct computation of diagnosis using a constraint solver handling a more expressive input language is feasible especially when the number of involved constraints is small enough. It is worth noting that in our application domain the number of constraints is limited and does not exceed several hundreds.

*Available at `http://www.cbl.ncsu.edu/benchmarks/`

Table 6.1.: Diagnosis results obtained when using MINION for diagnosing ISCAS 85 circuits. Single faults only. The experiments were carried out on a MacBook Pro 2.53 GHz Intel Core 2 Duo, 4 GB DDR3 RAM, under the OS X 10.6.7 operating system. The given running time is a rounded average value over 3 runs.

Circuit	Gates	No. inputs	No. outputs	Constraints without observations	No. Diagnoses	Running time [ms]
c17	6	5	2	12	1	<1
c432	160	36	7	320	38	45
c499	202	41	32	404	2	90
c880	383	60	26	766	19	490
c1355	546	41	32	1,092	5	2,400
c1908	880	33	25	1,760	5	9,065
c2670	1,193	233	140	2,386	1	23,390
c3540	1,559	50	22	3,118	15	60,700
c5315	2,307	178	123	4,614	1	150,670
c6288	2,406	32	32	4,812	109	211,840
c7552	3,512	207	108	7,024	117	189,495

6.3. ConDiag Algorithm

In order to be self-contained we briefly recall the basic definitions of model-based diagnosis [111] and adapt them whenever necessary to fit our purpose. We start with the definition of a model based on constraints. A constraint model comprises a set of variables, each associated with a domain, and a set of constraints. Each constraint has a corresponding set of used variables, i.e., its scope, and a definition of a relation between the variables from the scope. In the case of diagnosis, we are interested in locating components that are responsible for a deviation between the observed and the derived values for variables. Hence, we have to have means for representing correctness assumptions. For each component j we introduce a variable nab_j from the boolean domain $\mathbb{B} = \{0,1\}$ (where 0 represent false, and 1 true) and assume that the constraints representing the behavior B_j of j are of the form $nab_j = 1 \rightarrow B_j$. For describing the constraints, we further assume a language allowing to state mathematical equations over certain domains comprising at least the Boolean domain , with operators $\wedge$ (And), $\vee$ (Or), $\neg$ (Not), $\rightarrow$ (Implies).

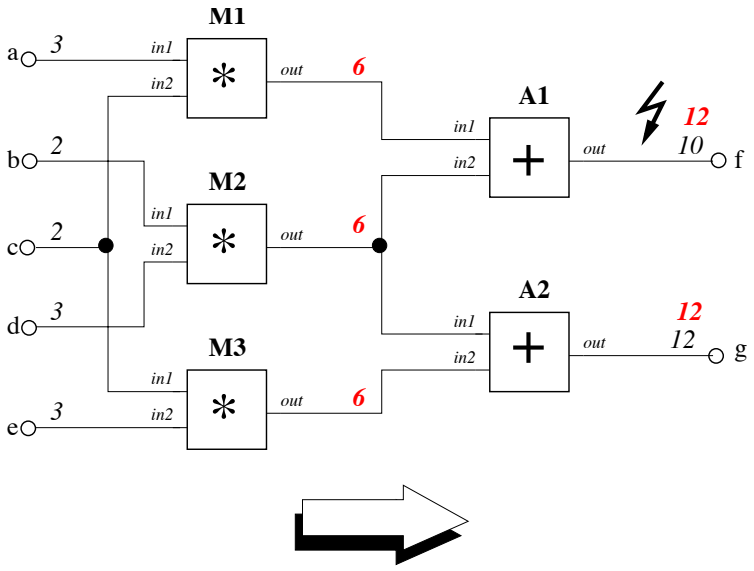

Figure 6.1.: The D74 circuit including observations and assuming all components to behave correctly.

We formalize the constraint model similar to a constraint satisfaction problem [24].

Definition 9 (Constraint model) *A constraint model is a tuple* $(VAR, DOM, CON, COMP)$ *where VAR is a set of variables, DOM is a set of domains, CON is a set of constraints, and COMP is a*

set of components. The boolean domain $\mathbb{B}$ *has to be element of DOM. VAR comprises at least a variable* nab_j *with domain* $\mathbb{B}$ *for each* $j \in COMP$.

For the (well-known) D74 circuit from Figure 6.1 the constraint model comprises the variables $VAR = \{a,b,c,d,e,f,g,x,y,z,nab_{M1},nab_{M2},nab_{M3},nab_{A1},nab_{A2}\}$ where x,y,z represent the internal variables, i.e., the outputs of components $M1$ to $M3$. The domain of $a,b,\ldots,y,z$ would be $\mathbb{N}$, hence $DOM = \{\mathbb{B},\mathbb{N}\}$. The behavior of the multiplication component $M1$ can be described using the constraint $nab_{M1} = 1 \rightarrow (a \cdot c = x)$. The behavior of the other components can be described in a similar way. For D74 we have the set $COMP = \{M1,M2,M3,A1,A2\}$.

Given a constraint model we are interested in finding a solution. For this purpose we first define an instantiation, which is an assignment of a value from the respective domain to each variable defined in the constraint model. A solution of a constraint model is an instantiation that does not contradict any constraint. For example, assuming $a = 2,c = 3,x = 5,nab_{M1} = 1$ violates (or is in contradiction to) the constraint $nab_{M1} = 1 \rightarrow (a \cdot c = x)$.

Definition 10 (Constraint solution) *Given a constraint model* $(VAR,DOM,CON,COMP)$. *An instantiation is a solution for the given constraint model, if no constraint is violated.*

For example, the instantiation $a = 2,b = 2,c = 3,d = 3,e = 2,x = 6,y = 6,z = 6,f = 12,g = 12,nab_{M1} = 1,nab_{M2} = 1,nab_{M3} = 1,nab_{A1} = 1,nab_{A2} = 1$ is a solution of the D74 constraint model.

Because of the definition of constraint solutions a simple algorithm would only set variable values and check whether each constraint does not violate the instantiation. There are of course better algorithms for computing all solutions. We refer the interested reader to [24] for more details on constraint solving and algorithms. From here on we assume an algorithm CSolver that given a set of constraints returns $\perp$ if no solution exists, or a set of solutions, otherwise.

The next step before defining diagnosis is to state the diagnosis problem. For this purpose we need a constraint model and observations. Observations themselves are constraints over some variables stating observed values or relationships. For the D74 example from Fig. 6.1, $a = 2,b = 2,c = 3,d = 3,e = 2,f = 10,g = 12$ are the constraints representing the observed values.

Definition 11 (Diagnosis problem) *Given a constraint model* $M = (VAR,DOM,CON,COMP)$ *and a set of observations OBS. The tuple* (M,OBS) *states a diagnosis problem where the goal is to identify those components of M that are the root cause for obtaining OBS.*

In the vein of [111], we define diagnoses to be components that when assumed to behave faulty, lead to a valid solution. What we have to do is to generate a set of constraints that represent the model of the system, the observations, and the underlying assumptions, and to call a constraint solver CSolver to check whether there are any solutions.

Definition 12 (Diagnosis 2) *Given a diagnosis problem* (M, OBS) *with* $M = (VAR, DOM, CON, COMP)$. *A subset* Δ *of* $COMP$ *is a diagnosis if and only if* $\mathsf{CSolver}(SD \cup OBS \cup \{nab_j = 1 | j \in COMP \setminus \Delta\}) \cup \{nab_j = 0 | j \in \Delta\}) \neq \bot$

A diagnosis Δ is minimal if there is no set $\Delta' \subset \Delta$ that is itself a diagnosis for the same diagnosis problem. For the D74 example $\{M1\}$ is a minimal diagnosis but $\{M2, M3, A2\}$ is not because $\{M2, M3\}$ is already a diagnosis.

Before introducing the algorithm, we briefly describe **ConDiag** using the D74 circuit depicted in Figure 6.1. When assuming all components to behave correctly, we see that a value of 10 is computed for output f, which is in contradiction to the expected value of 12. Hence, we are interested in finding a root cause, i.e., a set of components that when assumed to be faulty explains the observed behavior. What we can do in the first place is to provide a diagnosis problem which comprises the constraint model and the constraint representation of the observations. For the D74 we have the following constraints M_{D74}:

$$\left\{ \begin{array}{l} nab_{M1} = 1 \rightarrow (a \cdot c = x), \\ nab_{M2} = 1 \rightarrow (b \cdot d = y), \\ nab_{M3} = 1 \rightarrow (c \cdot e = z), \\ nab_{A1} = 1 \rightarrow (x + y = f), \\ nab_{A2} = 1 \rightarrow (y + z = g), \end{array} \right\} \cup \left\{ \begin{array}{l} a = 2, \\ b = 2, \\ c = 3, \\ d = 3, \\ e = 2, \\ f = 10, \\ g = 12 \end{array} \right\}$$

A classical model-based diagnosis algorithm takes these constraints and tries to find a subset of the set of components that when assumed to be incorrect, eliminates the contradiction. In case of a constraint model we would set nab_j to 1 if a component works as expected, and to 0, otherwise. This can be done again by representing these value assignments as constraints. The consistency check can be performed using a constraint solver. Such a procedure seems not to utilize the constraint solver

as much as possible. Instead of only calling the constraint solver for checking consistency, we use it directly for computing diagnoses of a particular size.

What we want to do is to call CSolver(M_{D74}), which tries to find a solution for the given constraint model. A solution in constraint solving is a variable assignment, i.e., an instantiation, where all constraints are fulfilled. If there is a solution, we know that it also assigns a value for each variable nab_j. Since a diagnosis is also such an assignment, the constraint solver returns directly diagnoses. In order to compute single or double faults, the only thing to do is to state this again using constraints. For single faults we only have to say that the sum of all nab_j should be the number of all components minus 1. For double faults it is minus 2, and so on.

Let us assume that for the D74 we are interested in single faults. Hence, we call the constraint solver again but this time using $M_{D74} \cup \{nab_{M1} + nab_{M2} + nab_{M3} + nab_{A1} + nab_{A2} = 4\}$. The CSolver call would return either nab_{M1} or nab_{A1} to be 0 and the other *nab* variables to be set to 1. If after computing single faults we are interested in double faults, we are able to call again CSolver, but this time adding $nab_{M1} + nab_{M2} + nab_{M3} + nab_{A1} + nab_{A2} = 3$. In this case we would obtain all double diagnoses but unfortunately also all supersets of single diagnoses, which are obviously per definition also diagnoses. The reason here is that the constraint solver would not have any information regarding minimality. Consequently, we have to provide this information and again we can use this by stating that we are not interested in diagnoses that we already calculated. For our example we add the constraint $\neg(nab_{M1} = 0 \vee nab_{A1} = 0)$, which formalizes that we are not interested in solutions that either include $M1$ or $A1$ anymore.

We are able to generalize this principle. Let $DS = \{\Delta_1, \ldots, \Delta_k\}$ be all diagnoses already obtained. Each diagnosis Δ_i is a subset of $COMP$ but can be interpreted as stating all components to be faulty. Let us assume a function $\mathcal{C}$ that maps a diagnosis to its constraint interpretation, i.e., $\mathcal{C}(\Delta) = \bigwedge_{c \in \Delta}(nab_c = 0)$. Then we are able to define also a constraint interpretation $\mathcal{C}$ for DS, which is nothing else than a disjunction of the constraint interpretation of every diagnosis, i.e., $\mathcal{C}(DS) = \mathcal{C}(\Delta_1) \vee \ldots \vee \mathcal{C}(\Delta_k)$. Finally, we only have to negate $\mathcal{C}(DS)$ and add this constraint to the constrain model in order to prevent CSolver from computing supersets of already computed diagnoses.

We have discussed all ingredients of **ConDiag**, i.e., restricting the computation of diagnoses to a particular size, and avoiding supersets of already computed diagnoses. Algorithm 3 gives the algorithmic description of **ConDiag**. The constrain model M and the maximum size of diagnoses n to be computed have to be provided as input. **ConDiag** computes a set of minimal diagnoses of size less equal to n. In the algorithm we use a function $\mathcal{P}$ that extracts the components j where nab_j is 0 from

Algorithm 3 *ConDiag*(M,n)

Input:A constraint model M and the desired diagnosis cardinality n
Output: All minimal diagnoses up to the predefined cardinality n

1: Let DS be $\{\}$
2: **for** $i = 0$ **to** n **do**
3: $\quad CM = M \cup \left\{ \sum_{j=0}^{|COMP|} nab_j = |COMP| - i \right\}$
4: $\quad S = \mathcal{P}(\textbf{CSolver}(CM))$
5: $\quad$ **if** i is 0 **and** S is $\{\{\}\}$ **then**
6: $\quad\quad$ **return** S
7: $\quad$ **end if**
8: $\quad$ Let DS be $DS \cup S$.
9: $\quad M = M \cup \{\neg(C(S))\}$
10: **end for**
11: **return** DS

each solution. Hence, after applying $\mathcal{P}$ only the diagnoses remain.

The algorithm starts by setting the already found diagnoses to the empty set. Afterwards, there is a loop starting from 0 to the maximum given size of diagnoses. Within the loop we first construct the constraint model to be used when calling the constraint solver CSolver. The result of this call is mapped to the diagnoses. In case there is no fault, i.e., when there is no contradiction occurring when setting all *nab* variables to 1, we are ready and return the empty set as the only diagnosis. Otherwise, we add the diagnoses to the set of diagnoses (Step 8). In Step 9 the algorithm adds information of the already computed diagnoses for the next iteration. Obviously, **ConDiag** terminates and computes all minimal diagnoses.

In order to evaluate the performance of **ConDiag**, we performed some initial experiments using the well-known ISCAS 85 benchmark suite.

Our evaluation is based on a Java implementation of **ConDiag**. For the implementation we chose a state of the art constraint solver - the MINION solver [46]. Because the currently interface between **ConDiag** and the MINION solver works via calling MINION using a Shell and extracting the solutions again from the MINION printouts at the Shell, there is an extra running time overhead.

Unfortunately, MINION does currently not offer a direct interface to Java.

The input language of the solver provides many alternatives for modeling the circuit. This can lead to significant differences in terms of solving times. Implicitly, the time needed by the solver to compute the solution for the constraint problem has a great impact on the performance of the introduced algorithm. Hence the solving time depends not only on the execution time of the algorithm, but also on how the constraints propagate.

We consider circuits containing *AND*, *OR*, *NAND*, *NOR*, *XOR*, *XNOR*, *NOT* and *BUFFER* gates, and define the model via the following MINION constraints: the minimality and maximality constraints $\texttt{min}(\textit{vector}, \textit{value})$ and $\texttt{max}(\textit{vector}, \textit{value})$, the equality and dis-equality constraints $\texttt{eq}(\textit{var1}, \textit{var2})$ and $\texttt{diseq}(\textit{var1}, \textit{var2})$ and the two forms of reification provided by the language, i.e., $\texttt{reify}(\textit{constraint}, \textit{value}_{\textit{bool}})$ and $\texttt{reifyimply}(\textit{constraint}, \textit{value}_{\textit{bool}})$. For details on the solver specific constraints please refer to MINION's documentation[†].

Considering Reiter's diagnosis formulations and the various gate types, with Table 6.2 offering the nominal behavior for Boolean gates, we add for each gate j a constraint $\texttt{reifyimply}(\textit{NominalBehavior}, !AB[j])$ with AB a vector defining each gate j's status abnormal ($AB[j] = 1$) or not ($AB[j] = 0$). While observations are encoded straightforward via $\texttt{eq}(\textit{variable}, \textit{value})$, the desired diagnosis cardinality is defined via $\texttt{sumleq}(AB, i)$ and $\texttt{sumgeq}(AB, i)$, requiring the sum of abnormal components to be equal to i. Furthermore, at the end of each loop iteration, the constraint model is extended with blocking constraints for diagnoses obtained in the current iteration:

$$\texttt{watched-or}(\underbrace{(\{\texttt{eq}(AB[x],0),..,\texttt{eq}(AB[z],0)\})}_{i \text{ equality constraints}},$$

if the current size of diagnosis is $i > 1$, or with $\texttt{eq}(AB[x],0))$, otherwise.

Table 6.3 and Table 6.4 show the obtained solutions for the ISCAS 85 circuits where we used three different input-output vectors for the experiments. We measured the running times when computing all minimal diagnoses up to cardinality 3. The given running time $\mathbf{T_{avg}}$ is the average value over the 3 runs. In addition the tables also give the minimum and maximum number of obtained diagnosis solutions $\mathbf{NoS_{min}}$ and $\mathbf{NoS_{max}}$ respectively again for the 3 runs. All the experiments were performed on a notebook with Intel(R) Core(TM) i7 CPU 1.73 GHz and 4 GB of RAM running under Windows 7. Note also that we do not provide any solutions in the tables if the solving time exceeded 600

[†] http://minion.sourceforge.net/htmlhelp/index.html

Table 6.2.: MINION models for various gate types.

Gate	Model	MINION encoding
AND	$out = \min(in_1, in_2)$	`min([in1,in2],out)`
OR	$out = \max(in_1, in_2)$	`max([in1,in2],out)`
NAND	$\neg out = \min(in_1, in_2)$	`min([in1,in2],!out)`
NOR	$\neg out = \max(in_1, in_2)$	`max([in1,in2],!out)`
XOR	$out = 1 \Leftrightarrow in_1 \neq in_2$	`reify(diseq(in1,in2), out)`
XNOR	$\neg out = 1 \Leftrightarrow in_1 \neq in_2$	`reify(diseq(in1,in2), !out)`
NOT	$out \neq in$	`diseq(in,out)`
BUF	$out = in$	`eq(in,out)`

seconds at least for one test case. In this case even successful remaining test cases are not considered in the table. Such cases where the running time is exceeded are marked with *n.a.* in the table.

We highlight the importance of the coding by presenting two sets of results, which were obtained for the same circuits and the same set of observations, but with two different implementations of the constraint model. Note that the number of constraints used to model the system is the same in both situations. In this case, the minimality constraints

$$min(vector, value)$$

for instance coupled with the reification constraints

$$reifyimply(constraint, value_{bool})$$

get a much better propagation than

$$watched - or(constraint_1, ..., constraint_n)$$

and

$$table(vector, constraintsextension)$$

constraints. Hence, it seems that using tables for representing the behavior of gates slows down the whole constraint solving process at least for MINION. For more details about the MINION constraints we refer the interested reader to the MINION manual[‡]. In order to show the time differences, have for instance a look at the running time for gate $c6288$. When computing the double fault diagnosis, the

[‡] `http://minion.sourceforge.net/manual.html`

Table 6.3.: Diagnosis results obtained when running **ConDiag** on ISCAS 85 constraint models with the less efficient MINION coding. We are interested in the running times for computing all minimal diagnoses of size 1, 2, and 3. For each circuit and diagnosis cardinality, we present the average solving time over 3 runs $\mathbf{T_{avg}}$ and the minimum and maximum number of solutions $\mathbf{NoS_{min}}$, $\mathbf{NoS_{max}}$.

Circuit	Single faults			Double faults			Triple faults		
	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$
c17	0.0052	2	2	0.0104	1	1	0.0052	0	0
c432	0.2340	11	38	0.3432	0	72	0.5668	0	18
c499	0.3328	2	2	1.1856	28	28	15.8527	523	523
c880	0.7020	0	0	2.4284	0	0	26.4159	0	0
c1355	2.7612	0	5	86.5129	190	190	n.a.	n.a.	n.a.
c1908	9.2352	5	5	109.2785	0	0	n.a.	n.a.	n.a.
c2670	20.8417	1	1	21.1745	1	3	n.a.	n.a.	n.a.
c3540	37.2582	15	2	n.a.	0	n.a.	n.a.	0	n.a.
c5315	138.3620	1	68	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
c6288	38.2825	0	1	109.5279	0	109	n.a.	0	n.a.
c7752	184.6576	10	117	n.a.	0	n.a.	n.a.	n.a.	n.a.

solver needs 109 seconds to execute using the model with the table constraints, whereas the improved model needs a hundred times less, which means in this case that the solution can be computed in 1 second (see Tables 6.3 and 6.4).

Table 6.4.: Diagnosis results obtained when running **ConDiag** on ISCAS 85 constraint models with more efficient MINION coding. We are interested in the running times for computing all minimal diagnoses of size 1, 2, and 3. For each circuit and diagnosis cardinality, we present the average solving time over 3 runs $\mathbf{T_{avg}}$ and the minimum and maximum number of solutions $\mathbf{NoS_{min}}$, $\mathbf{NoS_{max}}$.

Circuit	Single faults			Double faults			Triple faults		
	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$	$\mathbf{T_{avg}}$ [s]	$\mathbf{NoS_{min}}$	$\mathbf{NoS_{max}}$
c17	0.0104	2	2	0.0000	1	1	0.0052	0	0
c432	0.0364	11	38	0.0468	0	72	0.1404	0	18
c499	0.0468	2	2	0.1508	28	28	2.3088	523	523
c880	0.0624	0	0	0.1352	0	0	1.4352	0	0
c1355	0.1352	0	5	7.2800	25	190	n.a.	n.a.	n.a.
c1908	0.3172	5	5	5.1480	0	0	534.4276	0	0
c2670	0.5252	1	1	0.5304	1	3	47.9858	1	18
c3540	0.4472	2	15	108.7845	0	274	0.234	0	n.a.
c5315	1.7628	1	68	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
c6288	0.2808	0	1	1.0348	0	109	n.a.	0	n.a.
c7552	2.2152	10	117	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.

6.4. Run-Time Performance Trends - A Comparison of Diagnosis Algorithms

With available computation resources growing continuously, AI-based diagnosis approaches are gaining in attraction for more and more practical purposes. This leaves us with the question of which approach to implement for a certain project. Some approaches, e.g. [111, 48, 23, 128], exploit the fact that diagnoses are related to conflicts in the assumption that the system components operate correctly, as deduced from actually observed system behavior and a system model. Others, e.g. [43, 33, 83, 92], derive a consistent assumption on the system's components' status directly, without considering conflicts. Common to all is the use of some reasoning engine/ theorem prover to derive hypotheses or verify their validity (and optionally to derive corresponding conflicts).

In this section, we are going to investigate run-time performance trends for various instances of these two general approaches. For our tests, we used a publicly available diagnosis engine, and implemented selected algorithms for both variants on top of different reasoning engines utilized for the verification of hypotheses deduced during computation. That is, our different setups use a horn-clause theorem prover developed specifically for diagnosis purposes, two general purpose satisfiability solvers, as well as a general purpose constraint solver. The details of the individual setups and their engines are discussed in Section 6.4.1.

6.4.1. Selected Diagnosis Algorithms

Our six setups compare several conflict-driven search algorithms with approaches based on a more direct reasoning, varying the underlying reasoning engines. As we are interested in the performance one can achieve with off-the-shelf tools, we use corresponding publicly available reasoning engines. Also note that, on purpose, we do not apply any pre- or post-processing steps as used in other publications for speedups, e.g. [22, 83]. To be fair, these pre-processing steps would have to be included in every setup, so the effects should be comparable, and our aim is not on the efficiency of such optimizations but the efficiency of the two general concepts. In practice, one certainly will explore model-optimizations suitable for a specific project and the corresponding problem domain, our scope is, however, the general diagnosis approach performance behind search concepts.

Besides **ConDiag**, we selected two other algorithms for computing diagnosis directly:(1) using a MaxSAT problem and (2) by introducing cardinality constraints in a pure SAT search approach.

The setup based on the **ConDiag** algorithm will be further depicted as *Direct-MS*$_{CS}$. MERIDIAN's approach [33] was implemented for our setup *Direct-MS*$_{SAT}$, while our second direct SAT setup *Direct-CN*$_{SAT}$ is based on cardinality constraints (networks) as proposed in, for example, [27, 83]. For more details regarding the direct SAT setups we refer the reader to [97].

The conflict-based setups peruse diagnosis algorithms that compute the subset of conflicts needed on-the-fly, which is an attractive feature for practical applications. There, one often restricts the maximum cardinality in order to keep the search space (and in turn the set of required conflicts) as small as possible. In our analysis, we included two conflict-driven setups using a SAT solver and *HS-DAG*$_{HC}$ using the publicly available diagnosis engine *JDiagengine* [§]. The first SAT based setup, *HS-DAG*$_{SAT}$, is based on Greiner et al's improved version of Reiter's idea, whereas the second, *HST*$_{SAT}$, implements the Wotawa's variant HST of Reiter's idea that tries to avoid constructing nodes that would be pruned by HS-DAG. In case of *HS-DAG*$_{HC}$, *JDiagengine* engine implements a conflict-driven search via HS-DAG. In contrast to the corresponding setup *HS-DAG*$_{SAT}$, it however exploits a Horn-clause reasoning engine [84] instead of a general SAT-solver. [106] described the diagnosis engine and initial results in more detail. It is worth noting that their approach is similar to the one in [89]. For more details regarding the direct conflict-driven setups we refer the reader to [97].

6.4.2. Empirical Results

Like [123], we used ten combinational circuits from the well-known ISCAS85 benchmark suite[¶] for our tests, run on an Apple MacPro4,1 (early 2009) computer featuring an Intel Xeon W3520 quad-core processor, 16 GiB of RAM and running an up-to-date version of OS X 10.8. The algorithms were implemented in Java 1.6 (*HS-DAG*$_{HC}$ and *Direct-MS*$_{CS}$) and CPython 2.7 (*HST*$_{SAT}$, *Direct-MS*$_{SAT}$ and *Direct-CN*$_{SAT}$), while for the theorem provers we used MINION 0.15, Yices 1.0.29 and SCryptoMinisat (29/01/2012) based on CryptoMiniSat 2.5.1.

It is worth noting that, due to the underlying complexity, the ISCAS85 circuits provide a profound base for experiments, so that, e.g., [139] perused the ISCAS85 structure to develop a more general benchmark suite for diagnosis. In Table 6.5, we list circuit details, including their function ranging from Arithmetic Logic Units (ALU)s, via interrupt controllers, adders/comparators and multipliers to single-error correction/double-error detection (SEC/DED) circuits.

[§] http://www.ist.tugraz.at/modremas/downloads.html
[¶] http://www.cbl.ncsu.edu:16080/benchmarks/ISCAS85/

Table 6.5.: ISCAS85 circuit statistics plus the number of variables/literals (#V/#L) and constraints/clauses (#Co/#Cl) for each depicted algorithm's models (excluding blocking constrains/clauses).

		HS-DAG$_{HC}$		**HS-DAG$_{SAT}$**		**HST$_{SAT}$**		**Direct-CN$_{SAT}$**		**Direct-MS$_{SAT}$**		**Direct-MS$_{CS}$**	
Circuit	Function	#L	#Cl	#V	#Co	#V	#Co	#V	#Cl	#V	#Co	#V	#Co
c432	27-ch. interrupt controller	5391	1497	356	321	356	321	990	1509	356	321	197	205
c499	32-bit SEC circuit	8350	2262	445	405	445	405	1247	1991	445	488	244	277
c880	8-bit ALU	10293	3099	826	767	826	767	2358	3495	826	797	444	471
c1355	32-bit SEC circuit	14758	4398	1133	1093	1133	1093	3311	4951	1133	1097	588	621
c1908	16-bit SEC/DED circuit	21555	6345	1793	1761	1793	1761	5307	7708	1793	1765	914	940
c2670	12-bit ALU and controller	30021	9144	2619	2387	2619	2387	7391	10723	2619	2388	1427	1492
c3540	8-bit ALU	41653	12277	3388	3339	3388	3339	10064	14693	3388	3369	1720	1743
c5315	9-bit ALU	62098	18251	4792	4615	4792	4615	14020	20835	4792	4615	2486	2610
c6288	16-bit multiplier	65008	19328	4864	4833	4864	4833	14522	21768	4864	4917	2449	2482
c7752	32-bit adder/comparator	86791	25977	7231	7025	7231	7025	21273	31034	7231	7031	3720	3828

Table 6.6.: Diagnosis samples solved (out of 100).

Direct-MS$_{CS}$	100	68	27
Direct-CN$_{SAT}$	100	89	64
Direct-MS$_{SAT}$	100	65	35
HS-DAG$_{HC}$	87	0	0
HS-DAG$_{SAT}$	100	86	57
HST$_{SAT}$	100	85	54

Into each circuit, we randomly injected ten single-, double- and triple-faults, aggregating test suites `TS1`, `TS2` and `TS3` respectively. We injected faults by changing a gate's Boolean function such that at least one circuit output flipped. When modifying the second and third gate, we allowed only previously unaffected outputs to flip, in order to prevent faults from masking each other. Injected faults were verified to be indeed a minimal diagnosis.

The measured run-time represents the total, user-experienced diagnosis time, including communication with the solvers. For all algorithms we established a run-time limit of 200 seconds. For *HS-DAG$_{HC}$* we further set a limit of 100 diagnoses (whose computation always exceeded 200 seconds) as a precaution. In our tables and figures, we do not include any results for timed-out samples,

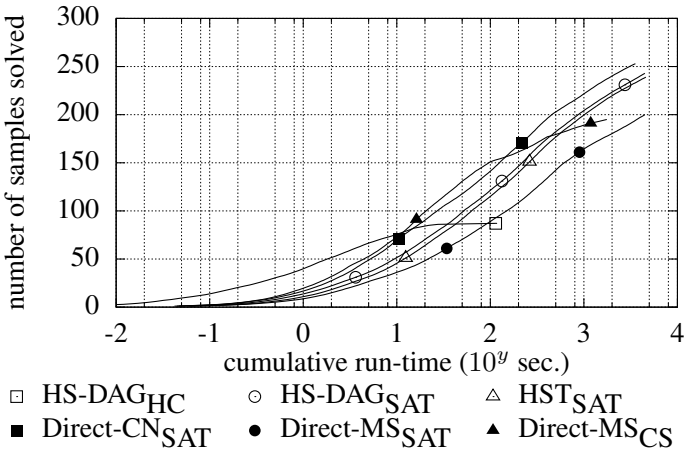

Figure 6.2.: Number of diagnosis samples solved over time.

so that accordingly, for example, in Table 6.7 reporting statistical data, average values are missing. Whenever available we report, however, the minimal and median values.

In Table 6.7 we present from top to bottom the run-times for computing single-fault diagnoses for `TS1`, up to double-fault ones for `TS2` and diagnoses up to a size of three for `TS3`. For clarity, we put the best values per category and circuit in bold face. For Figure 6.2, we ordered all samples from `TS1` to `TS3` according to their run-time, and report the amount of samples solved for growing cumulative time. The amount of completed samples per test suite and setup is given in Table 6.6.

Specifically in the context that we did not exploit model-optimization concepts, like cones of influence, Table 6.7 suggests that any algorithm offers attractive performance, when taking into account single-fault diagnoses for these circuits. *HS-DAG$_{HC}$* and *Direct-MS$_{CS}$* match each other for the crown, where like all other approaches, the latter has the disadvantage of relying on an external reasoning engine. The first could also, even for `TS1` only, complete fewer samples (see Table 6.6), so that *Direct-MS$_{CS}$* becomes even more interesting despite *HS-DAG$_{HC}$*'s advantage for small samples, as evident also in Figure 6.2.

The other conflict driven setups *HS-DAG$_{SAT}$* and *HST$_{SAT}$* performed similarly for all test suites (see Table 6.7) with a slight advantage for the first (see Figure 6.2), which could also solve slightly more samples (see Table 6.6). It is beaten, however, specifically in the completion rate, by the best performing direct SAT setup.

Between the two direct SAT setups *Direct-CN$_{SAT}$* and *Direct-MS$_{SAT}$*, the latter is superior (best seen

Table 6.7.: Run-time in seconds for computing single-, up to double-, and up to triple-fault diagnoses for TS1, TS2 and TS3 respectively (top to bottom).

	HS-DAG$_{HC}$				HS-DAG$_{SAT}$				HST$_{SAT}$				Direct-CN$_{SAT}$				Direct-MS$_{SAT}$				Direct-MS$_{CS}$			
Circuit	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED
c432	**0.003**	0.212	**0.049**	**0.016**	0.047	0.543	0.213	0.130	0.059	0.631	0.254	0.149	0.045	0.100	0.062	0.050	0.062	1.202	0.402	0.194	0.042	**0.086**	0.051	0.047
c499	**0.005**	**0.031**	**0.015**	**0.007**	0.057	1.172	0.501	0.071	0.079	1.381	0.603	0.105	0.059	0.662	0.285	0.151	0.122	3.533	1.410	0.165	0.047	0.098	0.066	0.054
c880	**0.005**	**0.053**	**0.022**	**0.013**	0.099	1.911	0.673	0.584	0.140	2.169	0.791	0.677	0.081	0.403	0.208	0.200	0.201	4.540	1.591	1.405	0.049	0.099	0.072	0.075
c1355	**0.053**	**0.341**	**0.121**	**0.067**	0.269	9.631	1.266	0.302	0.324	10.60	1.421	0.354	0.232	1.905	0.415	0.245	0.587	24.24	8.087	1.931	0.103	0.677	0.218	0.106
c1908	**0.018**	86.42	9.007	0.557	0.236	4.326	2.068	2.471	0.310	4.795	2.369	2.862	0.424	2.050	1.015	0.989	0.459	68.48	12.37	6.724	0.187	**0.227**	**0.209**	**0.211**
c2670	**0.013**	2.663	0.380	**0.063**	0.340	6.000	3.105	3.251	0.415	6.329	3.384	3.678	0.298	3.225	1.609	1.683	0.544	13.19	6.990	7.689	0.081	**0.345**	**0.237**	0.231
c3540	**0.047**			**0.255**	1.331	6.473	4.105	4.308	1.650	7.306	4.728	4.878	1.651	5.081	2.883	2.728	2.789	14.91	9.021	9.409	0.230	**0.604**	**0.475**	0.495
c5315	**0.040**	5.613	0.905	**0.204**	0.484	21.80	5.821	2.549	0.668	24.98	7.038	3.086	0.767	12.01	4.526	3.135	0.813	51.77	12.82	4.717	0.138	**1.341**	**0.843**	0.860
c6288	**0.151**			47.82	3.561	25.14	15.22	15.38	4.257	29.90	17.73	17.53	3.725	12.32	8.084	7.961	6.829	164.8	85.64	97.75	0.294	**1.260**	**0.953**	**1.166**
c7752	**0.177**	17.06	3.465	**0.702**	1.897	42.34	11.07	3.782	2.256	47.50	12.68	4.439	5.310	31.34	11.32	6.652	2.518	102.8	24.40	6.043	1.735	**3.081**	**2.116**	1.906
c432					0.077	1.253	0.434	0.346	0.108	1.463	0.503	0.397	0.071	**0.250**	**0.123**	**0.111**	0.089	3.778	1.185	0.937	**0.064**	0.297	0.197	0.215
c499					**0.088**	2.359	0.380	0.130	0.119	2.754	0.459	0.179	0.094	0.499	**0.145**	**0.103**	0.147	9.802	1.319	0.302	0.326	**0.367**	0.344	0.340
c880					0.536	16.68	5.060	2.537	0.605	18.92	5.837	2.981	0.278	3.518	**1.182**	**0.672**	1.220	55.33	16.39	8.300	**0.179**	**3.539**	2.016	2.687
c1355					1.128	5.541	2.496	1.189	1.322	6.285	2.878	1.454	**0.583**	**1.584**	**0.901**	**0.620**	6.229	45.13	21.14	18.39	10.27	12.10	10.57	10.39
c1908					**0.600**	110.4	25.86	15.18	0.744	135.0	30.80	17.25	0.827	**42.00**	**9.483**	**5.215**	2.004			73.59	0.884	131.5	44.07	42.42
c2670					0.498	171.0	54.08	24.75	0.581			28.25	0.918	**78.49**	**24.68**	**10.76**	0.874			149.2	**0.277**			52.22
c3540					8.370	151.1	61.84	38.60	9.531	178.9	72.07	43.10	**5.678**	**81.42**	**33.21**	**19.91**	29.93				37.94			
c5315					0.618			21.31	0.782			23.32	2.782	**169.0**	**34.50**	**16.39**	0.704			53.21	**0.335**			
c6288					47.65				52.59				32.42								**16.50**			
c7752					18.72				21.16				20.83								**0.176**			
c432					0.181	9.056	2.068	1.218	0.236	11.97	2.636	1.492	**0.102**	**1.404**	**0.356**	**0.243**	0.355	84.16	11.68	4.621	0.320	6.315	1.553	0.710
c499					0.131	6.145	1.085	0.262	0.221	7.708	1.377	0.387	**0.120**	**1.130**	**0.275**	**0.139**	0.202	67.47	8.884	0.822	12.19	14.56	13.13	12.88
c880					0.581			15.89	0.705			18.77	**0.355**	**103.7**	**19.53**	**3.429**	1.127			76.68	0.365			8.907
c1355					5.314			16.16	7.035			19.15	**1.834**	**71.54**	**11.37**	**4.810**	105.8							
c1908					7.677				9.041				**3.185**			**67.78**	90.96							
c2670					**1.372**				1.596				1.610			**83.13**	3.640							
c3540					41.36				46.70				**21.28**											
c5315					12.81				13.92				**10.45**				25.96							
c6288																								
c7752					5.165				5.898				10.05				7.247				**2.062**			

in Figure 6.2 and Table 6.6). Due to internal tests with a *Direct-CN$_{SAT}$* version computing a single solution per query, we can state that a large (but varying) part of the advantage comes from the fewer amount of theorem prover calls, besides the use of cardinality networks.

In the comparison between *Direct-MS$_{CS}$* using constraints and the best direct SAT setup *Direct-CN$_{SAT}$*, we see some advantages for the first in TS1, which changes with higher maximum cardinalities for TS2 and TS3. Presumably due to better performance for larger samples, *Direct-CN$_{SAT}$* completes more samples within the given time limit (see Figure 6.2 and Table 6.6). While this might suggest better scalability for the SAT-based setup, we do assume this to be very domain-dependent. That is, for models with larger domains, this might actually be different.

Chapter 7

Conclusions and Future Work

In this thesis, we have presented our research work, conducted within an industrial project, where the main objective was to develop a simulation and reconfiguration environment for smart metering applications. In particular, we were interested in reconfiguring a mobile network in case of bottlenecks caused by requirements coming from future M2M applications. In order to handle simulation and reconfiguration based on the same underlying information and to cope with the fast-paced M2M application domain, we have adopted a flexible, model-based approach - the *SIMOA approach*.

For this purpose, we have designed *SIMOL* - a general, object-oriented modeling language, capable of specifying the structure and dynamic behavior of systems, as well as the (re-)configuration information and requirements. Currently, the language allows to state constraints dealing with Boolean and Integer values and also arrays can be used for modeling. The limitation regarding the mapping of SIMOL Integer attributes to $DISCRETE$ variables in MINION was discussed in Section 3.5.1. In our current implementation, the lower bound of the range for $DISCRETE$ variables is always 0, while the upper bound can be internally determined from the variable's type. However, such a strict internal limitation is not preferred, as it may happen that a result of a computation, specified in form of a SIMOL arithmetic expression, will exceed the defined upper limit and, in this case, MINION will not report on the error. Extensions in the direction of handling floats or strings can be implemented, but require to change the underlying reasoning engine. In addition, models written in SIMOL might be described over discrete time.

As we have already seen, there are many languages and packages used for simulation in industry.

However, these languages are mainly optimized towards simulation and therefore can be hardly used for reconfiguration. In particular, such languages do not allow under-constrained models, which are necessary for our reconfiguration purpose, when searching for appropriate modes that do not contradict the given requirements, while ensuring that the desired functionality can be fulfilled.

Note that we can find some similarities between Modelica and SIMOL, as both languages try to unify object-oriented modeling concepts, but there are also many differences including the tight integration of component modes in case of SIMOL and the handling of discrete time. While in Modelica both continuous and discrete-event modeling of physical systems is possible, in SIMOL, we have decided to model only discrete time via states. As a limitation of our approach we mention here the predefined number of states, over which the SIMOL model is simulated. In our experiments, we have mostly used from five up to ten states, irrespective of the number of varying attributes . However, the maximum number of states within a reconfiguration model depends on the number of varying components' attributes in the model and for complex systems this could go up to several hundreds or even more.

The general need of easy to learn and use modeling languages that are expressive enough to state configuration problems was detected by many researchers and also tackled in the present thesis. Therefore, the general expressiveness of SIMOL was analyzed and demonstrated by means of modeling different systems, like, for instance, the combinational circuits from the well-known ISCAS85 benchmark suite.

Besides SIMOL, we have discussed the implemented reconfiguration mechanism, based on two diagnosis algorithms, the first - **MCDiag** - aimed at computing minimal cardinality diagnosis and a second one - **ConDiag** - which guarantees to compute minimal diagnosis up to a predefined cardinality. Note that the reconfiguration algorithms derive solutions directly from the constraints, i.e., equations coming from SIMOL. This distinguishes our approach from other similar approaches where search for valid configurations is often based on conflicts and conflict resolution. First empirical results indicated that computation is sufficiently fast and that the results are within expectations. In the future we plan to further evaluate this approach.

Moreover, although, up to now, the SIMOA approach to reconfiguration has been applied to the M2M communication domain, it is not restricted to this domain. Any reconfiguration problem that can be represented using the underlying modeling language SIMOL can also be solved using the proposed SIMOA approach. Within the developed SIMOA prototype, SIMOL is converted in MINION constraints, but this does not restrict the approach since changing constraint solvers is still possible,

only the conversion of SIMOL has to be adapted.

Another important aspect is the novelty of applying AI configuration techniques to the machine-to-machine communication domain. Although configuration of technical products from various domains is a well researched field, its application to the M2M domain that requires M2M specific components and the used communication infrastructure is, to our knowledge, new. Moreover, in contrast to previous work in configuration, we have models that capture the temporal aspects of a real system.

As extension to our work on simulation and reconfiguration, we provided an in depth run-time comparison of **ConDiag** (using the MINION constraint solver as underlying engine) against other five setups using publicly available reasoning engines. Although the setup based on **ConDiag** was, on average, slightly outperformed by the SAT setup based on cardinality constraints networks, we expect a comparison to be domain-dependent. That is, for model domains more complex than the Boolean one of our circuits, future research will have to identify corresponding trends. Note that no further simplifications and optimizations has been used for speeding up the constraint solver. Hence, future research includes optimizing the used MINION models. Of course, also the evolvement of SAT and constraint solvers will influence this choice. In this sense, another interesting task would be the implementation of **ConDiag** with the currently top-performing constraint solver Gecode [45, 122].

List of Theorems and Definitions

List of Definitions

Bibliography

[1] 2001. SysML. OMG Systems Modeling Language (OMG SysML) v1.0 Specification. (Cited on page 20.)

[2] 2004. EXPRESS. 10303-11 ISO - Part 11: The EXPRESS language reference manual. (Cited on page 20.)

[3] ALBRECHT, M. C. 2010. Introduction to Discrete Event Simulation. [online; accessed June 2013]. (Cited on pages 16, 18, 27, and 28.)

[4] ASÍN, R., NIEUWENHUIS, R., OLIVERAS, A., AND RODRÍGUEZ-CARBONELL, E. 2009. Cardinality Networks and Their Applications. *Theory and Applications of Satisfiability Testing – SAT 2009 5584*, 167–180. (Cited on page 93.)

[5] AXLING, T. AND HARIDI, S. 1994. A Tool For Developing Interactive Configuration Applications. (Cited on page 25.)

[6] BAADER, F., CALVANESE, D., MCGUINNESS, D. L., NARDI, D., AND PATEL-SCHNEIDER, P. F., Eds. 2003. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA. (Cited on page 21.)

[7] BALCI, O. 1988. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In *Proceedings of the 20th conference on Winter simulation*. WSC '88. ACM, New York, NY, USA, 287–295. (Cited on page 19.)

[8] BANKS, J. 1999. Introduction to simulation. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 1*. WSC '99. ACM, New York, NY, USA, 7–13. (Cited on page 29.)

[9] BANKS, J. AND CARSON, II, J. S. 1986. Introduction to discrete-event simulation. In *Proceedings of the 18th conference on Winter simulation*. WSC '86. ACM, New York, NY, USA, 17–23. (Cited on pages 15 and 16.)

[10] BATCHER, K. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314. (Cited on page 93.)

[11] BATORY, D. 2005. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*. SPLC'05. Springer-Verlag, Berlin, Heidelberg, 7–20. (Cited on page 20.)

[12] BENAVIDES, D., SEGURA, S., AND RUIZ-CORTÉS, A. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst. 35,* 6 (Sept.), 615–636. (Cited on page 20.)

[13] BENJAMIN, P., PATKI, M., AND MAYER, R. 2006. Using ontologies for simulation modeling. In *Proceedings of the 38th conference on Winter simulation*. WSC '06. Winter Simulation Conference, 1151–1159. (Cited on page 19.)

[14] CHANDRASEKARAN, B., JOSEPHSON, J. R., AND BENJAMINS, V. R. 1999. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems 14,* 1 (Jan.), 20–26. (Cited on page 23.)

[15] CHEN, M. 2004. OPNET Network Simulation. Press of Tsinghua University. (Cited on page 27.)

[16] CHESS. 2002-2013. Ptolemy. [online; accessed June 2013]. (Cited on pages 28 and 29.)

[17] CROW, J. AND RUSHBY, J. 1991. Model-based reconfiguration: Toward an integration with diagnosis. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. The MIT Press, 836–841. (Cited on page 69.)

[18] CUNIS, R., GÜNTER, A., SYSKA, I., PETERS, H., AND BODE, H. 1989. Plakon - an approach to domain-independent construction. In *IEA/AIE (2)*. 866–874. (Cited on pages 17 and 25.)

[19] CZARNECKI, K., HELSEN, S., AND EISENECKER, U. 2005. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*. 2005. (Cited on page 20.)

[20] DAHL, O.-J. AND NYGAARD, K. 1966. SIMULA: an ALGOL-based simulation language. *Commun. ACM 9,* 9 (Sept.), 671–678. (Cited on page 18.)

[21] DE KLEER, J. 1986. An assumption-based TMS. *Artificial Intelligence 28*, 127–162. (Cited on pages 26, 61, and 93.)

[22] DE KLEER, J. 2011. Hitting set algorithms for model-based diagnosis. In *22nd Int. Workshop on the Principles of Diagnosis*. 100–105. (Cited on page 104.)

[23] DE KLEER, J. AND WILLIAMS, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence 32,* 1, 97–130. (Cited on pages 61, 66, 93, and 104.)

[24] DECHTER, R. 2003. *Constraint Processing*. Morgan Kaufmann. (Cited on pages 60, 96, and 97.)

[25] DOYLE, J. 1979. A Truth Maintenance System. *Artif. Intell. 12,* 3, 231–272. (Cited on page 26.)

[26] DYNASIM. Dymola. [online; accessed June 2013]. (Cited on page 27.)

[27] EÉN, N. AND SÖRENSSON, N. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation 2,* 3-4, 1–25. (Cited on page 105.)

[28] EKER, J., JANNECK, J., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., SACHS, S., XIONG, Y., AND NEUENDORFFER, S. 2003. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE 91,* 1, 127–144. (Cited on page 29.)

[29] EMDE, W., BEILKEN, C., BORDING, J., ORTH, W., PETERSEN, U., RAHMER, J., SPENKE, M., VOSS, A., WROBEL, S., AND BIRLINGHOVEN, S. 1996. Configuration of Telecommunication Systems in KIKon. (Cited on pages 17, 24, and 25.)

[30] ET AL., M. A. Modelica. [online; accessed June 2010]. (Cited on page 27.)

[31] FALKNER, A., FEINERER, I., SALZER, G., AND SCHENNER, G. 2008. Solving practical configuration problems using UML. In *Proceedings of ECAI Workshop on Configuration Systems*. (Cited on page 20.)

[32] FATTAH, Y. E. AND DECHTER, R. 1995. Diagnosing tree-decomposable circuits. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*. 1742 – 1748. (Cited on page 94.)

[33] FELDMAN, A., PROVAN, G., DE KLEER, J., ROBERT, S., AND VAN GEMUND, A. 2010. Solving model-based diagnosis problems with Max-SAT solvers and vice versa. In *Proceedings of the 21th International Workshop on Principles of Diagnosis (DX-10).* (Cited on pages 93, 104, and 105.)

[34] FELFERNIG, A., FRIEDRICH, G., JANNACH, D., STUMPTNER, M., AND ZANKER, M. 2003. Configuration knowledge representations for Semantic Web applications. *Artif. Intell. Eng. Des. Anal. Manuf. 17,* 1 (Jan.), 31–50. (Cited on page 21.)

[35] FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND ZANKER, M. 2002a. Configuration knowledge representation using UML/OCL. In *LNCS*. (Cited on page 20.)

[36] FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND ZANKER, M. 2002b. Semantic Configuration Web Services in the CAWICOMS Project. In *ISWC '02 Proceedings of the First International Semantic Web Conference on The Semantic Web*. 192–205. (Cited on page 23.)

[37] FENG-JUN, H., SHUI-YING, C., ZHEN, X., AND PING, S. 2010. Extended case-based reasoning for intelligent system configuration. In *Proceedings of the 2010 Second International Conference on Computer Modeling and Simulation - Volume 03*. ICCMS '10. IEEE Computer Society, Washington, DC, USA, 307–309. (Cited on page 17.)

[38] FISHWICK, P. A. AND MILLER, J. A. 2004. Ontologies for modeling and simulation: issues and approaches. In *Proceedings of the 36th conference on Winter simulation*. WSC '04. Winter Simulation Conference, 259–264. (Cited on page 19.)

[39] FLEISCHANDERL, G., FRIEDRICH, G. E., HASELBÖCK, A., SCHREINER, H., AND STUMPTNER, M. 1998. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems 13,* 4, 59–68. (Cited on pages 20, 21, and 24.)

[40] FORGY, C. L. AND MCDERMOTT, J. 1977. OPS, A Domain-Independent Production System Language. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*. MIT, 933–939. (Cited on page 22.)

[41] FRIEDRICH, G., RYABOKON, A., FALKNER, A. A., HASELBCK, A., SCHENNER, G., AND SCHREINER, H. 2011. (Re)configuration based on model generation. In *LoCoCo*, C. Drescher, I. Lynce, and R. Treinen, Eds. EPTCS, vol. 65. 26–35. (Cited on page 21.)

[42] FRITZSON, P. AND BUNUS, P. 2002. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings 35th Annual Simulation Symposium*. 365–380. (Cited on pages 19, 27, and 44.)

[43] FRÖHLICH, P. AND NEJDL, W. 1997. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*. Nagoya, Japan. (Cited on pages 93 and 104.)

[44] GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. A User's Guide to gringo, clasp, clingo, and iclingo. (Cited on page 21.)

[45] GECODE. 2005. Gecode toolkit. [online; accessed June 2013]. (Cited on pages 16 and 111.)

[46] GENT, I. P., JEFFERSON, C., AND MIGUEL, I. 2006. Minion: A fast, scalable, constraint solver. *17th European Conference on Artificial Intelligence ECAI-06*. (Cited on page 100.)

[47] GIBNEY, A. M., KLEPAL, M., AND PESCH, D. 2010. A wireless local area network modeling tool for scalable indoor access point placement optimization. In *Proceedings of the 2010 Spring Simulation Multiconference*. SpringSim '10. Society for Computer Simulation International, San Diego, CA, USA, 163:1–163:8. (Cited on page 27.)

[48] GREINER, R., SMITH, B. A., AND WILKERSON, R. W. 1989. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence 41,* 1, 79–88. (Cited on pages 93 and 104.)

[49] GRUBER, T. R. 1993. A translation approach to portable ontology specifications. *KNOWLEDGE ACQUISITION 5*, 199–220. (Cited on page 23.)

[50] GÜNTER, A. 1992. *Flexible Kontrolle in Expertensystemen zur Planung und Konfigurierung in technischen Domänen*. DISKI, vol. 3. Infix Verlag, St. Augustin, Germany. (Cited on page 17.)

[51] GÜNTER, A., KREUZ, I., AND KÜHN, C. 1999. Kommerzielle Software-Werkzeuge für die Konfigurierung von technischen Systemen. In *Proceedings of KI*. 61–65. (Cited on pages 24 and 26.)

[52] GÜNTER, A. AND KÜHN, C. 1999. Knowledge-based configuration: Survey and future directions. In *XPS*. 47–66. (Cited on page 17.)

[53] GUNTER, C. A. 1992. *Semantics of programming languages: structures and techniques*. MIT Press, Cambridge, MA, USA. (Cited on page 31.)

[54] HAAG, A. 1995. The ATMS* - an assumption based problem solving architecture utilizing specialization relations. Ph.D. thesis. (Cited on pages 25 and 26.)

[55] HAAG, A. 1998. Sales configuration in business processes. *Intelligent Systems and their Applications, IEEE 13,* 4, 78–85. (Cited on pages 25 and 26.)

[56] HAAG, A. 2010. Experiences with Product Configuration? http://www.minet.uni-jena.de/dbis/lehre/ss2010/konfsem/. (Cited on page 26.)

[57] HAAG, A. AND RIEMANN, S. 2011. Product configuration as decision support: The declarative paradigm in practice. *AI EDAM 25*, 131–142. (Cited on page 26.)

[58] HEINRICH, M. AND JUNGST, E. 1991. A resource-based paradigm for the configuring of technical systems from modular components. In *Artificial Intelligence Applications, 1991. Proceedings., Seventh IEEE Conference on*. Vol. i. 257–264. (Cited on pages 17, 24, and 25.)

[59] HEINRICH, M. AND JÜNGST, E. W. 1991. Ressourcen-getriebene Konfigurierung modularer Technischer Systeme. In *Prozeßrechnersysteme*. 59–68. (Cited on page 17.)

[60] HENSON, W. 2005. Real time Control and Custom Components in the Matlab Environment. Tech. rep. (Cited on page 27.)

[61] HOTZ, L., WOLTER, K., AND KREBS, T. 2006. *Configuration in Industrial Product Families: The ConIPF Methodology*. IOS Press, Inc. (Cited on page 21.)

[62] HUA, K., SMITH, I., AND FALTINGS, B. 1994. Integrated case-based building design. In *Selected papers from the First European Workshop on Topics in Case-Based Reasoning*. EWCBR '93. Springer-Verlag, London, UK, UK, 436–445. (Cited on page 17.)

[63] HUBAUX, A. 2012. Feature-based Configuration: Collaborative, Dependable, and Controlled. Ph.D. thesis, University of Namur, Belgium. (Cited on page 20.)

[64] HVAM, L., MORTENSEN, N. H., AND RIIS, J. 2008. Product customization, volume xii. In *volume XII*. (Cited on page 22.)

[65] IMAGINE THAT, I. 2002-2013. Extend. [online; accessed June 2013]. (Cited on page 28.)

[66] JACOP. 2001. JaCoP constraint solver. [online; accessed June 2013]. (Cited on page 16.)

[67] JAWHAR, I. 2009. A flexible object-oriented design of an event-driven wireless network simulator. In *Proceedings of the 2009 conference on Wireless Telecommunications Symposium*. WTS'09. IEEE Press, Piscataway, NJ, USA, 80–86. (Cited on page 27.)

[68] JEFFERSON, C., KOTTHOFF, L., MOORE, N., NIGHTINGALE, P., PETRIE, K. E., AND RENDL, A. 2012. The Minion Manual, Minion Version 0.15. http://minion.sourceforge.net/. (Cited on pages 44, 45, 47, 48, 49, 60, and 61.)

[69] JOHN, U. 2000. Solving large configuration problems efficiently by clustering the ConBaCon model. In *Proceedings of the 13th international conference on Industrial and engineering applications of artificial intelligence and expert systems: Intelligent problem solving: methodologies and approaches*. Springer-Verlag New York, Inc. (Cited on page 23.)

[70] JOHN, U. AND GESKE, U. 1999. Reconfiguration of Technical Products Using ConBaCon. In *Proceedings of WS on Configuration at AAAI99*. Orlando. (Cited on pages 20 and 23.)

[71] JUNKER, U. 2004. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical intelligence*. AAAI'04. AAAI Press, 167–172. (Cited on page 93.)

[72] JUNKER, U. AND MAILHARRO, D. 2003. The logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In *Proceedings of IJCAI-03 Configuration WS*. 13–20. (Cited on pages 21 and 24.)

[73] JURAN, D. C. AND SCHRUBEN, L. W. 2004. Using worker personality and demographic information to improve system performance prediction. *Journal of Operations Management 22,* 4, 355 – 367. (Cited on page 29.)

[74] KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute. November. (Cited on page 20.)

[75] LACY, L. AND GERBER, W. 2004. Potential modeling and simulation applications of the web ontology language - OWL. In *Proceedings of the 36th conference on Winter simulation*. WSC '04. Winter Simulation Conference, 265–270. (Cited on page 19.)

[76] LEHMANN, E., ENDERS, R., HAUGENEDER, H., HUNZE, R., JOHNSON, C., SCHMID, L., AND STRUSS, P. 1985. SICONFEX - ein Expertensystem für die Konfigurierung eines Betriebssystems. In *Proceedings of GI Jahrestagung'1985*. 792–805. (Cited on page 23.)

[77] LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic 7,* 3 (July), 499–562. (Cited on page 21.)

[78] MAILHARRO, D. 1998. A classification and constraint-based framework for configuration. *Artif. Intell. Eng. Des. Anal. Manuf. 12,* 4 (Sept.), 383–397. (Cited on page 21.)

[79] MATHCORE. 2009. MathModelica/Wolfram SystemModeler. [online; accessed June 2013]. (Cited on page 27.)

[80] MCCANNE, S. AND FLOYD, S. ns Network Simulator. http://www.isi.edu/nsnam/ns/. (Cited on page 27.)

[81] MCDERMOTT, J. 1980. R1: An Expert in the Computer Systems Domain. In *Proceedings of First National Conference on Artificial Intelligence (AAAI-80)*. Stanford University, Stanford, California. (Cited on pages 17 and 22.)

[82] MCGUINNESS, D. L. AND WRIGHT, J. R. 1998. An industrial strength description logics-based configurator platform. *IEEE Intelligent Systems 13,* 4 (July), 69–77. (Cited on page 21.)

[83] METODI, A., STERN, R., KALECH, M., AND CODISH, M. 2012. Compiling model-based diagnosis to boolean satisfaction. In *26th AAAI Conference on Artificial Intelligence*. (Cited on pages 94, 104, and 105.)

[84] MINOUX, M. 1988. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters 29*, 1–12. (Cited on page 105.)

[85] MITTAL, S. AND FALKENHAINER, B. 1990. Dynamic constraint satisfaction problems. In *AAAI*. 25–32. (Cited on pages 17 and 21.)

[86] MITTAL, S. AND FRAYMAN, F. 1989. Towards a generic model of configuraton tasks. In *IJCAI*. 1395–1401. (Cited on pages 16 and 21.)

[87] MOZETIC, I. 1992. A polynomial-time algorithm for model-based diagnosis. In *European Conference on Artificial Intelligence (ECAI)*. 729–733. (Cited on page 93.)

[88] NANCE, R. E. 1993. A history of discrete event simulation programming languages. In *The second ACM SIGPLAN conference on History of programming languages*. HOPL-II. ACM, New York, NY, USA, 149–175. (Cited on page 16.)

[89] NAYAK, P. P. AND WILLIAMS, B. C. 1997. Fast context switching in real-time propositional reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. AAAI Press. (Cited on page 105.)

[90] NEUMANN, B. 1990. *Ein Ansatz zur wissensbasierten Auftragsprüfung für technische Anlagen des Breitengeschäfts*. (Cited on page 25.)

[91] NICA, I. AND WOTAWA, F. 2011a. Diagnosis-based reconfiguration using the MINION constraint solver. In *Proceedings of the 22nd International Workshop on Principles of Diagnosis (DX 2011)*. 225–232. (Cited on pages 12, 59, and 91.)

[92] NICA, I. AND WOTAWA, F. 2012a. ConDiag – computing minimal diagnoses using a constraint

solver. In *Proceedings of the 23rd International Workshop on Principles of Diagnosis*. (Cited on pages 12, 91, 94, and 104.)

[93] NICA, I. AND WOTAWA, F. 2012b. The SIMOL modeling language for Simulation and (Re-)configuration. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*. SOFSEM'12. Springer-Verlag, Berlin, Heidelberg, 661–672. (Cited on pages 12 and 31.)

[94] NICA, I. AND WOTAWA, F. 2013. (Re-)configuration of Communication Networks in the Context of M2M Applications. Tech. rep., Institute of Software Technology, Graz University of Technology. Submitted for Publication. (Cited on pages 12, 31, 59, and 71.)

[95] NICA, I., WOTAWA, F., OCHENBAUER, R., SCHOBER, C., HOFBAUER, H., AND BOLTEK, S. 2012. Model-based simulation and configuration of mobile phone networks – the SIMOA approach. In *Proceedings of the ECAI 2012 Workshop on Artificial Intelligence for Telecommunications and Sensor Networks*. 12–17. (Cited on pages 12, 59, and 71.)

[96] NICA, I., WOTAWA, F., OCHENBAUER, R., SCHOBER, C., HOFBAUER, H., AND BOLTEK, S. 2013. Kapsch: Reconfiguration of mobile phone networks. In *Configuration systems: Technologies and Business Cases*. Elsevier ConfSys12. Submitted for Publication. (Cited on pages 12, 59, and 71.)

[97] NICA, I. D., PILL, I., QUARITSCH, T., AND WOTAWA, F. 2013. The Route to Success – A Performance Comparison of Diagnosis Algorithms. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*. (Cited on pages 12, 91, and 105.)

[98] NICA, I. D. AND WOTAWA, F. 2011b. SiMoL - A Modeling Language for Simulation and (Re-)Configuration. In *Workshop on Configuration*. 40–43. (Cited on pages 12 and 31.)

[99] NICA, M. 2010. On the Use of Constraints in Automated Program Debugging - From Foundations to Empirical Results. Ph.D. thesis, Graz University of Technology, Institute for Software Technology, Graz, Austria. (Cited on page 44.)

[100] NICA, M., MORARU, I., AND WOTAWA, F. 2009. Representing Program Debugging as Constraint Satisfaction Problem. In *21st Nordic Workshop on Programming Theory*. (Cited on page 44.)

[101] OKHMATOVSKAIA, A., BUCKERIDGE, D. L., SHABAN-NEJAD, A., SUTCLIFFE, A., FINÈS, P., KOPEC, J. A., AND WOLFSON, M. C. 2012. SimPHO: an ontology for simulation model-

ing of population health. In *Proceedings of the Winter Simulation Conference*. WSC '12. Winter Simulation Conference, 79:1–79:12. (Cited on page 19.)

[102] ORACLE. 2010. ORACLE CONFIGURATOR. [online; accessed June 2013; ORACLE Data Sheet]. (Cited on page 26.)

[103] ORSVÄRN, K. AND AXLING, T. 1999. The Tacton View of Configuration Tasks and Engines. In *AAAI99 Workshop on Configuration, the 16th National Conference on Artificial Intelligence*. 127–130. (Cited on page 25.)

[104] OUSTERHOUT, J. K. 1989. Tcl: An Embeddable Command Language. Tech. rep., Berkeley, CA, USA. (Cited on page 27.)

[105] PATTIS, R. E. 1994. Teaching EBNF first in CS 1. In *Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*. SIGCSE '94. ACM, New York, NY, USA, 300–303. (Cited on page 129.)

[106] PEISCHL, B. AND WOTAWA, F. 2003. Computing Diagnoses Efficiently: A Fast Theorem Prover For Propositional Horn Clause Theories. In *Proceedings of the 14th International Workshop on Principles of Diagnosis*. 175–180. (Cited on pages 94 and 105.)

[107] PFEIFER, R. AND RADEMAKERS, P. 1991. Situated Adaptive Design: Toward a New Methodology for Knowledge Systems Development. In *Verteilte Künstliche Intelligenz und kooperatives Arbeiten*, W. Brauer and D. Hernández, Eds. Informatik-Fachberichte, vol. 291. Springer Berlin Heidelberg, 53–64. (Cited on page 25.)

[108] PROLOG, S. 2005. SICStus Prolog 3.12.2. https://www.sics.se/sicstus/docs/3.12.2/html/sicstus/. (Cited on page 25.)

[109] PROVAN, G. AND CHEN, Y.-L. 1999. Model-based diagnosis and control reconfiguration for discrete event systems: an integrated approach. In *Proceedings of the 38th IEEE Conference on Decision and Control*. Vol. 2. 1762–1768. (Cited on page 69.)

[110] QUÉVA, M., PROBST, C., AND VIKKELSØE, P. 2009. Industrial requirements for interactive product configurators. In *Proceedings of the IJCAI*. Vol. 9. (Cited on pages 20 and 21.)

[111] REITER, R. 1987. A Theory of Diagnosis from First Principles. *Artificial Intelligence 32,* 1, 57–95. (Cited on pages 61, 62, 66, 92, 96, 98, and 104.)

[112] RIVARD, H. AND FENVES, S. J. 2000. SEED-Config: A case-based reasoning system for

conceptual building design. *Artif. Intell. Eng. Des. Anal. Manuf. 14,* 5 (Nov.), 415–430. (Cited on page 17.)

[113] ROCKWELL AUTOMATION, I. 2013. Arena. [online; accessed June 2013]. (Cited on page 28.)

[114] ROSSI, F., VAN BEEK, P., AND WALSH, T. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA. (Cited on pages 22, 56, and 57.)

[115] RUNTE, W. 2006. YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung. M.S. thesis, Universtät Bremen, Germany. (Cited on pages 23 and 26.)

[116] SABIN, D. AND FREUDER, E. C. 1996. Configuration as composite constraint satisfaction. In *in Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop*. AAAI Press, 153–161. (Cited on pages 17 and 21.)

[117] SACHENBACHER, M. AND WILLIAMS, B. C. 2004. Diagnosis as semiring-based constraint optimization. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*. Valencia, Spain, 873–877. (Cited on page 94.)

[118] SARGENT, R. G., KANG, K., AND GOLDSMAN, D. 1992. An investigation of finite-sample behavior of confidence interval estimators. *Oper. Res. 40,* 5 (Sept.), 898–913. (Cited on page 18.)

[119] SAVAGE, E. L., SCHRUBEN, L. W., AND YÜCESAN, E. 2005. On the generality of event-graph models. *INFORMS J. on Computing 17,* 1 (Jan.), 3–9. (Cited on page 29.)

[120] SCHRUBEN, L. W. 1995. *Graphical Simulation Modeling and Analysis: Using SIGMA for Windows*, 1st ed. Course Technology Press, Boston, MA, United States. (Cited on page 29.)

[121] SCHUBERT, M., FELFERNIG, A., AND MANDL, M. 2010. Fastxplain: Conflict detection for constraint-based recommendation problems. In *Trends in Applied Intelligent Systems*. Lecture Notes in Computer Science, vol. 6096. Springer Berlin Heidelberg, 621–630. (Cited on page 93.)

[122] SCHULTE, C. AND TACK, G. 2013. View-based propagator derivation. *Constraints 18,* 1, 75–107. (Cited on pages 16 and 111.)

[123] SIDDIQI, S. 2011. Computing minimum-cardinality diagnoses by model relaxation. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*. IJCAI'11. AAAI Press, 1087–1092. (Cited on page 105.)

[124] SILVER, G. A., HASSAN, O. A.-H., AND MILLER, J. A. 2007. From domain ontologies to modeling ontologies to executable simulation models. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*. WSC '07. IEEE Press, Piscataway, NJ, USA, 1108–1117. (Cited on page 19.)

[125] SOININEN, T., NIEMEL, I., TIIHONEN, J., AND SULONEN, R. 2001. Representing configuration knowledge with weight constraint rules. (Cited on page 21.)

[126] SOLOWAY, E., BACHANT, J., AND JENSEN, K. 1987. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base. In *AAAI*. 824–829. (Cited on page 17.)

[127] SPENKE, M., BEILKEN, C., AND BERLAGE, T. 1996. FOCUS: The Interactive Table for Product Comparison and Selection. In *Proceedings of the UIST 96 Ninth Annual Symposium on User Interface Software and Technology*. Press, 41–50. (Cited on page 25.)

[128] STERN, R., KALECH, M., FELDMAN, A., AND PROVAN, G. 2012. Exploring the duality in conflict-directed model-based diagnosis. In *26th AAAI Conference on Artificial Intelligence*. (Cited on pages 93 and 104.)

[129] STUMPTNER, M., FRIEDRICH, G., AND HASELBÖCK, A. 1998. Generative constraint-based configuration of large technical systems. *AI EDAM 12,* 4, 307–320. (Cited on pages 17 and 21.)

[130] STUMPTNER, M., HASELBÖCK, A., AND FRIEDRICH, G. 1994. COCOS - a tool for constraint-based, dynamic configuration. In *Proceedings of the 10th IEEE Conference on AI Applications (CAIA)*. San Antonio. (Cited on page 24.)

[131] STUMPTNER, M. AND WOTAWA, F. 1998. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*. Lisbon, Portugal. (Cited on pages 17, 62, 69, and 126.)

[132] STUMPTNER, M. AND WOTAWA, F. 1999. Reconfiguration using model-based diagnosis. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*. Loch Awe, Scotland. Newer version of [131]. (Cited on pages 62 and 69.)

[133] STUMPTNER, M. AND WOTAWA, F. 2001. Diagnosing tree-structured systems. *Artificial Intelligence 127,* 1, 1–29. (Cited on page 94.)

[134] STUMPTNER, M. AND WOTAWA, F. 2003. Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*. Acapulco, Mexico, 388–393. (Cited on page 94.)

[135] SWAIN, J. J. 2003. Simulation reloaded: Simulation software survey. (Cited on page 27.)

[136] THE MATHWORKS, I. SIMULINK. [online; accessed June 2010]. (Cited on page 27.)

[137] TIIHONEN, J., HEISKALA, M., ANDERSON, A., AND SOININEN, T. 2013. Wecotin - a practical logic-based sales configurator. *AI Commun. 26,* 1, 99–131. (Cited on page 21.)

[138] TURNITSA, C., PADILLA, J. J., AND TOLK, A. 2010. Ontology for modeling and simulation. In *Proceedings of the Winter Simulation Conference*. WSC '10. Winter Simulation Conference, 643–651. (Cited on page 19.)

[139] WANG, J. AND PROVAN, G. 2008. A benchmark diagnostic model generation system. *IEEE Transactions on Systems, Man, and Cybernetics A*. (Cited on page 105.)

[140] WEST, B. 2010. Model-based motion system simulation and design. [online; accessed June 2013]. (Cited on page 16.)

[141] WIKI, S. 2010. Sigma. [online; accessed June 2013]. (Cited on page 28.)

[142] WIKIPEDIA. 2013. SICOMP. [online; accessed June 2013]. (Cited on page 23.)

[143] WILLIAMS, B. C. AND RAGNO, R. J. 2007. Conflict-directed A* and its role in model-based embedded systems. *Discrete Appl. Math.*. (Cited on page 93.)

[144] WOLFRAM, S. 2003. *The Mathematica Book, Fifth Edition*. Wolfram Media/Cambridge University Press. (Cited on page 27.)

[145] WOTAWA, F. 2001. A Variant of Reiter's Hitting-Set Algorithm. *Information Processing Letters 79,* 1, 45–51. (Cited on page 93.)

[146] WOTAWA, F. 2003. Skriptum zur Lehrveranstaltung 'WISSENSVERARBEITUNG'. Tech. rep., Technische Universität Graz. (Cited on page 16.)

[147] WOTAWA, F. 2007. Software-Paradigmen 2007. Skriptum zur Lehrveranstaltung. (Cited on page 31.)

[148] WOTAWA, F., NICA, M., AND MORARU, I. 2012. Automated debugging based on a constraint model of the program and a test case. *Journal of Logic and Algebraic Programming*. (Cited on page 44.)

[149] WRIGHT, J. R., WEIXELBAUM, E. S., VESONDER, G. T., BROWN, K. E., PALMER, S. R., BERMAN, J. I., AND MOORE, H. H. 1993. A Knowledge-Based Configurator that Supports Sales,

Engineering, and Manufacturing at AT&T Network Systems. In *AI Magazine*. 183–193. (Cited on page 21.)

[150] ZEIGLER, B. P. 1968. On the feedback complexity of automata. Ph.D. thesis, University of Michigan, Ann Arbor, MI, USA. AAI6912283. (Cited on page 18.)

.1. A Variant of EBNF Notation

For the sake of completeness, first, we briefly introduce some notions related to the EBNF representation and, afterwards, mention the particularities of the EBNF variant, chosen to describe the syntax of SIMOL in this thesis. The general terminology on EBNF is further adapted from [105].

The *Extended Backus-Nauer Form (EBNF)* is a formal notation, that can be used to describe the syntax of a programming language. The EBNF consists of an unordered set of production rules, where a production rule shall consist of three parts: a left-hand side, a right-hand side, and a symbol separating the two sides, depicted mostly by ::= and read as "is defined as". The left-hand side represents the non-terminal node for which the production rule associates a sequence of terminal or/and nonterminal nodes, that is the right-hand side. Thus, for each nonterminal node, an EBNF rule is defined. Note that, besides the sequence of terminals and nonterminals, the right-hand may contain combinations of the following control forms:

- *choice*: depicted by |(stroke) and meaning that one item is chosen from a list of alternatives separated by |,
- *option*: depicted by [] (brackets), which enclose optional items,
- *repetition*: depicted by {} (braces), which enclose items that can be repeated zero or more times.

Note that the upper defined convention for specifying *repetitions* is not used in our EBNF variant. Instead, we use the following two forms:

- *(...)**: the enclosed items can be repeated zero or more times.
- *(...)*+: the enclosed items can be repeated one or more times.

For instance, in our EBNF variant, the propagation rule defining a nonterminal A can be written as:

$$A ::= B|(C)*|((D)+[E]),$$

where B, C, D, E denote sequences of nonterminals and terminals.

The meaning of the upper defined rule is: the nonterminal A is extended by *any one of the sequences* B, $(C)*$, or $(D)+[E]$, where $(C)*$ means we can have *zero or more occurrences of* C in A, whereas $(D)+[E]$ means that we can have *one or more occurrences of D, followed by an optional occurrence* of E. Note that the expansion choice $((D)+[E])$ appears parenthesized in the rule, in order to group the contained expansions.

.2. SIMOL Syntax

This section contains the complete set of propagation rules used to specify the grammar for SIMOL. In the followings, terminals are written in underlined text style, whereas nonterminals begin with capitals.

```
SIMOLProgram ::=
       [ KBDeclaration ]( ImportDeclaration )* ( ComponentDeclaration )* eof

KBDeclaration ::=
       kbase id ;

ImportDeclaration ::=
       import id [ . * ] ;

ComponentDeclaration ::=
       component id [ extends id (, id)*] { ComponentBodyDeclaration }

ComponentBodyDeclaration ::=
       AttributeDeclaration [ConstraintsDefinition] [TransitionBlock]
       | ConstraintsDefinition [TransitionBlock]

AttributeDeclaration ::=
       attribute (Type id ( , id )* ;)+

Type ::=
       PrimitiveType | ComponentType ( [ int_lit ] )*

PrimitiveType ::=
       bool|int

ComponentType ::=
       id
```

```
ConstraintsDefinition ::=
       constraints
       (
       (( id ) { ConstraintsBodyDefinition }
         ( constraints ( id ) { ConstraintsBodyDefinition } )*)
       | { ConstraintsBodyDefinition }
       )

ConstraintsBodyDefinition ::=
       (Statement)+

TransitionBlock ::=
       transition
       {
         ( Statement )*
       }

Statement ::=
       EmptyStatement
       | Block
       | VariableDeclarationOrExpressionStatement ;
       | IfStatement
       | ForAllBlock
       | ExistStatement

EmptyStatement ::=
       ;

Block ::=
       { ( BlockStatement )* }

BlockStatement ::=
       Statement
```

```
VariableDeclarationOrExpressionStatement ::=
       ComponentTypeVariableDeclarationOrIdExpressionStatement
       | NonIdExpressionStatament

ComponentTypeVariableDeclarationOrIdExpressionStatement ::=
       id
       (
       ComponentTypeVariableDeclaration
       | ( [ [( int_lit | id ) ] ] [ . id] [.next] IdExpressionStatement)
       )

ComponentTypeVariableDeclaration ::=
       VariableDeclarator ( , VariableDeclarator )*

IdExpressionStatement ::=
       (
       ( * | /) ( AccessAttName | (int_lit [Unit][Time] ))
         (
         ( ( * | / | + | -) ExpOperand (RelExpOperator ExpOperand)
           (RelExpOperator ExpOperand)*)
         | (( = | < |> | <= | >= | !=) ArithmeticBoolExp)
         )
       )
       | (((+ |-) ExpOperand)(RelExpOperator ExpOperand)*)
       | (= ((ArithmeticBoolExp) | AttributeDomain))
       | ((< | > | <= |>= |!=) ArithmeticBoolExp)

NonIdExpressionStatament ::=
       ((int_lit [Unit][Time] )| bool_lit )
       (
         (
         (* | /) ( AccessAttName | (int_lit [Unit][Time] ))
```

```
        (
        ( ( * | / | + | -) ExpOperand (RelExpOperator ExpOperand)
          (RelExpOperator ExpOperand)* )
        | (( = | < | > | <= | >= | !=) ArithmeticBoolExp))
        )
      | (((+ |-) ExpOperand)( RelExpOperator ExpOperand)*)
      | (= ArithmeticBoolExp )
      | ((< | > | <= | >= | !=) ArithmeticBoolExp)
      )

IfStatement ::=
      if ( IFConditionExtended )
          { Statement }
          [ else
            { Statement } ]

IFConditionExtended ::=
      Expression ((or | and ) Expression )*

ForAllBlock ::=
      forall ( ComponentType ) Statement

ExistStatement ::=
      exist (
      ( (at_least( int_lit ))
        | (at_most( int_lit ))
        | (int_lit )
      ), int_lit , int_lit = int_lit )

VariableDeclarator ::=
      id ( [ int_lit ] )*
      [{ id =
        (
```

```
          (int_lit)
          | (true)
          | (false)
        )
        ( ,id = ((int_lit)
        | (true)
        | (false)) )*
      }]

AccessAttName ::=
      ( id [[( (int_lit ) | (id ) ) ]] [. id][.next])
      | (sum ([ id (, id)* ], id ))
      | (min ([ id (, id)* ], id ))
      | (max ([ id (, id)* ], id ))
      | (product ([ id (, id)* ], id ))

ExpOperand ::=
      ArithmeticExp
      | bool_lit

RelExpOperator ::=
      =
      | <
      | >
      | <=
      | >=
      | !=

ArithmeticBoolExp ::=
      ExpOperand( RelExpOperator ExpOperand )*

AttributeDomain ::=
      in_range [Unit][Time]
```

```
Unit ::=
       KB | MB | GB | kbit
       | V| Hz | A | %

Time ::=
       /sec | /2min | /day
```

.3. SIMOL - MINION Mapping. Further Examples

1. The **c17** combinational circuit from the ISCAS-85 benchmark suite, depicted in Figure 3.4, from Section 3.5.2.

```
   component Nand_Gate_2 {
       attribute bool out, in1, in2;
       constraints (nab) {
           if (in1 = true and in2 = true)
           {
              out = false;
           }
           else {
                      out = true;
                   }
           if (out = false) {
               in1 = true;
               in2 = true;
           }
       }
       constraints (ab) {; }
  }
```

```
component ISCAS {
     attribute bool var_1gat, var_2gat, var_3gat, var_6gat, var_7gat,
          var_10gat, var_11gat, var_16gat, var_19gat, var_22gat, var_23gat;

     constraints {
         Nand_Gate_2  gat10;
         Nand_Gate_2  gat11;
         Nand_Gate_2  gat16;
         Nand_Gate_2  gat19;
         Nand_Gate_2  gat22;
         Nand_Gate_2  gat23;

         gat10.in1 = var_1gat;
         gat10.in2 = var_3gat;
         gat10.out = var_10gat;
         gat11.in1 = var_3gat;
         gat11.in2 = var_6gat;
         gat11.out = var_11gat;
         gat16.in1 = var_2gat;
         gat16.in2 = var_11gat;
         gat16.out = var_16gat;
         gat19.in1 = var_11gat;
         gat19.in2 = var_7gat;
         gat19.out = var_19gat;
         gat22.in1 = var_10gat;
         gat22.in2 = var_16gat;
         gat22.out = var_22gat;
         gat23.in1 = var_16gat;
         gat23.in2 = var_19gat;
         gat23.out = var_23gat;
```

```
      }
  }
```

The MINION mapping will be:

```
MINION 3

**VARIABLES**
BOOL ISCAS_var_1gat
BOOL ISCAS_var_2gat
BOOL ISCAS_var_3gat
BOOL ISCAS_var_6gat
BOOL ISCAS_var_7gat
BOOL ISCAS_var_10gat
BOOL ISCAS_var_11gat
BOOL ISCAS_var_16gat
BOOL ISCAS_var_19gat
BOOL ISCAS_var_22gat
BOOL ISCAS_var_23gat
BOOL ISCAS_gat10_Nand_Gate_2_out
BOOL ISCAS_gat10_Nand_Gate_2_in1
BOOL ISCAS_gat10_Nand_Gate_2_in2
BOOL ifCond0
BOOL ifCond1
BOOL ifCond2
BOOL ifCond3
BOOL ISCAS_gat11_Nand_Gate_2_out
BOOL ISCAS_gat11_Nand_Gate_2_in1
BOOL ISCAS_gat11_Nand_Gate_2_in2
BOOL ifCond4
BOOL ifCond5
BOOL ifCond6
```

```
BOOL ifCond7
BOOL ISCAS_gat16_Nand_Gate_2_out
BOOL ISCAS_gat16_Nand_Gate_2_in1
BOOL ISCAS_gat16_Nand_Gate_2_in2
BOOL ifCond8
BOOL ifCond9
BOOL ifCond10
BOOL ifCond11
BOOL ISCAS_gat19_Nand_Gate_2_out
BOOL ISCAS_gat19_Nand_Gate_2_in1
BOOL ISCAS_gat19_Nand_Gate_2_in2
BOOL ifCond12
BOOL ifCond13
BOOL ifCond14
BOOL ifCond15
BOOL ISCAS_gat22_Nand_Gate_2_out
BOOL ISCAS_gat22_Nand_Gate_2_in1
BOOL ISCAS_gat22_Nand_Gate_2_in2
BOOL ifCond16
BOOL ifCond17
BOOL ifCond18
BOOL ifCond19
BOOL ISCAS_gat23_Nand_Gate_2_out
BOOL ISCAS_gat23_Nand_Gate_2_in1
BOOL ISCAS_gat23_Nand_Gate_2_in2
BOOL ifCond20
BOOL ifCond21
BOOL ifCond22
BOOL ifCond23
BOOL nab_ISCAS_gat10_Nand_Gate_2
BOOL nab_ISCAS_gat11_Nand_Gate_2
BOOL nab_ISCAS_gat16_Nand_Gate_2
BOOL nab_ISCAS_gat19_Nand_Gate_2
```

```
BOOL nab_ISCAS_gat22_Nand_Gate_2
BOOL nab_ISCAS_gat23_Nand_Gate_2
BOOL ab_ISCAS_gat10_Nand_Gate_2
BOOL ab_ISCAS_gat11_Nand_Gate_2
BOOL ab_ISCAS_gat16_Nand_Gate_2
BOOL ab_ISCAS_gat19_Nand_Gate_2
BOOL ab_ISCAS_gat22_Nand_Gate_2
BOOL ab_ISCAS_gat23_Nand_Gate_2

**SEARCH**

VARORDER [nab_ISCAS_gat10_Nand_Gate_2,nab_ISCAS_gat11_Nand_Gate_2,
nab_ISCAS_gat16_Nand_Gate_2,nab_ISCAS_gat19_Nand_Gate_2,
nab_ISCAS_gat22_Nand_Gate_2,nab_ISCAS_gat23_Nand_Gate_2,
ab_ISCAS_gat10_Nand_Gate_2,ab_ISCAS_gat11_Nand_Gate_2,
ab_ISCAS_gat16_Nand_Gate_2,ab_ISCAS_gat19_Nand_Gate_2,
ab_ISCAS_gat22_Nand_Gate_2,ab_ISCAS_gat23_Nand_Gate_2,]

VALORDER [a, a, a, a, a, a, a, a, a, a, a, a]

PRINT ALL
**CONSTRAINTS**
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_in1,1), ifCond0),
                                    nab_ISCAS_gat10_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_in2,1), ifCond1),
                                     nab_ISCAS_gat10_Nand_Gate_2)
reify(watched-and({eq(ifCond0,1),eq(ifCond1,1)}), ifCond2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_out,0), ifCond2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_out,1), !ifCond2)
reifyimply(reify(eq(ISCAS_gat10_Nand_Gate_2_out,0), ifCond3),
                                    nab_ISCAS_gat10_Nand_Gate_2)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_in1,1), ifCond3)
reifyimply(eq(ISCAS_gat10_Nand_Gate_2_in2,1), ifCond3)
```

```
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_in1,1), ifCond4),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_in2,1), ifCond5),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reify(watched-and({eq(ifCond4,1),eq(ifCond5,1)}), ifCond6)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_out,0), ifCond6)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_out,1), !ifCond6)
reifyimply(reify(eq(ISCAS_gat11_Nand_Gate_2_out,0), ifCond7),
                                    nab_ISCAS_gat11_Nand_Gate_2)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_in1,1), ifCond7)
reifyimply(eq(ISCAS_gat11_Nand_Gate_2_in2,1), ifCond7)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_in1,1), ifCond8),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_in2,1), ifCond9),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reify(watched-and({eq(ifCond8,1),eq(ifCond9,1)}), ifCond10)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_out,0), ifCond10)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_out,1), !ifCond10)
reifyimply(reify(eq(ISCAS_gat16_Nand_Gate_2_out,0), ifCond11),
                                    nab_ISCAS_gat16_Nand_Gate_2)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_in1,1), ifCond11)
reifyimply(eq(ISCAS_gat16_Nand_Gate_2_in2,1), ifCond11)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_in1,1), ifCond12),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_in2,1), ifCond13),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reify(watched-and({eq(ifCond12,1),eq(ifCond13,1)}), ifCond14)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_out,0), ifCond14)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_out,1), !ifCond14)
reifyimply(reify(eq(ISCAS_gat19_Nand_Gate_2_out,0), ifCond15),
                                    nab_ISCAS_gat19_Nand_Gate_2)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_in1,1), ifCond15)
reifyimply(eq(ISCAS_gat19_Nand_Gate_2_in2,1), ifCond15)
```

```
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_in1,1), ifCond16),
                                    nab_ISCAS_gat22_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_in2,1), ifCond17),
                                    nab_ISCAS_gat22_Nand_Gate_2)
reify(watched-and({eq(ifCond16,1),eq(ifCond17,1)}), ifCond18)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_out,0), ifCond18)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_out,1), !ifCond18)
reifyimply(reify(eq(ISCAS_gat22_Nand_Gate_2_out,0), ifCond19),
                                    nab_ISCAS_gat22_Nand_Gate_2)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_in1,1), ifCond19)
reifyimply(eq(ISCAS_gat22_Nand_Gate_2_in2,1), ifCond19)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_in1,1), ifCond20),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_in2,1), ifCond21),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reify(watched-and({eq(ifCond20,1),eq(ifCond21,1)}), ifCond22)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_out,0), ifCond22)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_out,1), !ifCond22)
reifyimply(reify(eq(ISCAS_gat23_Nand_Gate_2_out,0), ifCond23),
                                    nab_ISCAS_gat23_Nand_Gate_2)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_in1,1), ifCond23)
reifyimply(eq(ISCAS_gat23_Nand_Gate_2_in2,1), ifCond23)
eq(ISCAS_gat10_Nand_Gate_2_in1,ISCAS_var_1gat)
eq(ISCAS_gat10_Nand_Gate_2_in2,ISCAS_var_3gat)
eq(ISCAS_gat10_Nand_Gate_2_out,ISCAS_var_10gat)
eq(ISCAS_gat11_Nand_Gate_2_in1,ISCAS_var_3gat)
eq(ISCAS_gat11_Nand_Gate_2_in2,ISCAS_var_6gat)
eq(ISCAS_gat11_Nand_Gate_2_out,ISCAS_var_11gat)
eq(ISCAS_gat16_Nand_Gate_2_in1,ISCAS_var_2gat)
eq(ISCAS_gat16_Nand_Gate_2_in2,ISCAS_var_11gat)
eq(ISCAS_gat16_Nand_Gate_2_out,ISCAS_var_16gat)
eq(ISCAS_gat19_Nand_Gate_2_in1,ISCAS_var_11gat)
eq(ISCAS_gat19_Nand_Gate_2_in2,ISCAS_var_7gat)
```

```
eq(ISCAS_gat19_Nand_Gate_2_out,ISCAS_var_19gat)
eq(ISCAS_gat22_Nand_Gate_2_in1,ISCAS_var_10gat)
eq(ISCAS_gat22_Nand_Gate_2_in2,ISCAS_var_16gat)
eq(ISCAS_gat22_Nand_Gate_2_out,ISCAS_var_22gat)
eq(ISCAS_gat23_Nand_Gate_2_in1,ISCAS_var_16gat)
eq(ISCAS_gat23_Nand_Gate_2_in2,ISCAS_var_19gat)
eq(ISCAS_gat23_Nand_Gate_2_out,ISCAS_var_23gat)
watchsumleq([nab_ISCAS_gat10_Nand_Gate_2,ab_ISCAS_gat10_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat11_Nand_Gate_2,ab_ISCAS_gat11_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat16_Nand_Gate_2,ab_ISCAS_gat16_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat19_Nand_Gate_2,ab_ISCAS_gat19_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat22_Nand_Gate_2,ab_ISCAS_gat22_Nand_Gate_2,], 1)
watchsumleq([nab_ISCAS_gat23_Nand_Gate_2,ab_ISCAS_gat23_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat10_Nand_Gate_2,ab_ISCAS_gat10_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat11_Nand_Gate_2,ab_ISCAS_gat11_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat16_Nand_Gate_2,ab_ISCAS_gat16_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat19_Nand_Gate_2,ab_ISCAS_gat19_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat22_Nand_Gate_2,ab_ISCAS_gat22_Nand_Gate_2,], 1)
watchsumgeq([nab_ISCAS_gat23_Nand_Gate_2,ab_ISCAS_gat23_Nand_Gate_2,], 1)

 **EOF**
```

2. The well-known **D74** circuit, discussed in Section 6.3.

```
kbase circuits;

component Gate {
 attribute int in1, in2, out;

}

component Multiplier extends Gate{
 constraints (nab){
```

```
        out=in1*in2;
       }

      }

      component Adder extends Gate{
       constraints (nab){
        out=in1+in2;
       }
constraints (s10){
       out =10;
       }

      }

      component  D74{
       attribute int a, b, c, d, e,  f, g; // given in OBS
       constraints{
        Multiplier  m1, m2, m3;
        Adder a1, a2;

        m1.in1=a;
        m1.in2=c;
        m1.out=a1.in1;

        m2.in1=b;
        m2.in2=d;
        m2.out=a1.in2;

        m3.in1=c;
        m3.in2=e;
        m3.out= a2.in2;
```

```
                a1.out=f;

                a2.in1=m2.out;
                a2.out=g;

 // OBS
 a=3;
 b=2;
 c=2;
 d=3;
 e=3;
 f=10;
 g=12;
                }
               }
```

The MINION mapping will be:

```
MINION 3

**VARIABLES**
DISCRETE D74_a {0..1000}
DISCRETE D74_b {0..1000}
DISCRETE D74_c {0..1000}
DISCRETE D74_d {0..1000}
DISCRETE D74_e {0..1000}
DISCRETE D74_f {0..1000}
DISCRETE D74_g {0..1000}
DISCRETE D74_m1_Multiplier_in1 {0..1000}
DISCRETE D74_m1_Multiplier_in2 {0..1000}
DISCRETE D74_m1_Multiplier_out {0..1000}
```

```
DISCRETE aux1 {0..20000}
DISCRETE D74_m2_Multiplier_in1 {0..1000}
DISCRETE D74_m2_Multiplier_in2 {0..1000}
DISCRETE D74_m2_Multiplier_out {0..1000}
DISCRETE aux3 {0..20000}
DISCRETE D74_m3_Multiplier_in1 {0..1000}
DISCRETE D74_m3_Multiplier_in2 {0..1000}
DISCRETE D74_m3_Multiplier_out {0..1000}
DISCRETE aux4 {0..20000}
DISCRETE D74_a1_Adder_in1 {0..1000}
DISCRETE D74_a1_Adder_in2 {0..1000}
DISCRETE D74_a1_Adder_out {0..1000}
DISCRETE aux2 {0..20000}
DISCRETE D74_a2_Adder_in1 {0..1000}
DISCRETE D74_a2_Adder_in2 {0..1000}
DISCRETE D74_a2_Adder_out {0..1000}
DISCRETE aux5 {0..20000}
BOOL nab_D74_m1_Multiplier
BOOL nab_D74_m2_Multiplier
BOOL nab_D74_m3_Multiplier
BOOL nab_D74_a1_Adder
BOOL nab_D74_a2_Adder
BOOL s10_D74_a1_Adder
BOOL s10_D74_a2_Adder

**SEARCH**
VARORDER [nab_D74_m1_Multiplier,nab_D74_m2_Multiplier,nab_D74_m3_Multiplier,
       nab_D74_a1_Adder,nab_D74_a2_Adder,s10_D74_a1_Adder,s10_D74_a2_Adder,]
VALORDER [a, a, a, a, a, a, a]
PRINT ALL
**CONSTRAINTS**
product(D74_m1_Multiplier_in1, D74_m1_Multiplier_in2, aux1)
reifyimply(eq(D74_m1_Multiplier_out,aux1), nab_D74_m1_Multiplier)
```

```
product(D74_m2_Multiplier_in1, D74_m2_Multiplier_in2, aux3)
reifyimply(eq(D74_m2_Multiplier_out,aux3), nab_D74_m2_Multiplier)
product(D74_m3_Multiplier_in1, D74_m3_Multiplier_in2, aux4)
reifyimply(eq(D74_m3_Multiplier_out,aux4), nab_D74_m3_Multiplier)
weightedsumgeq([1, 1], [D74_a1_Adder_in1, D74_a1_Adder_in2], aux2)
weightedsumleq([1, 1], [D74_a1_Adder_in1, D74_a1_Adder_in2], aux2)
reifyimply(eq(D74_a1_Adder_out,aux2), nab_D74_a1_Adder)
reifyimply(eq(D74_a1_Adder_out,10), s10_D74_a1_Adder)
weightedsumgeq([1, 1], [D74_a2_Adder_in1, D74_a2_Adder_in2], aux5)
weightedsumleq([1, 1], [D74_a2_Adder_in1, D74_a2_Adder_in2], aux5)
reifyimply(eq(D74_a2_Adder_out,aux5), nab_D74_a2_Adder)
reifyimply(eq(D74_a2_Adder_out,10), s10_D74_a2_Adder)
eq(D74_m1_Multiplier_in1,D74_a)
eq(D74_m1_Multiplier_in2,D74_c)
eq(D74_m1_Multiplier_out,D74_a1_Adder_in1)
eq(D74_m2_Multiplier_in1,D74_b)
eq(D74_m2_Multiplier_in2,D74_d)
eq(D74_m2_Multiplier_out,D74_a1_Adder_in2)
eq(D74_m3_Multiplier_in1,D74_c)
eq(D74_m3_Multiplier_in2,D74_e)
eq(D74_m3_Multiplier_out,D74_a2_Adder_in2)
eq(D74_a1_Adder_out,D74_f)
eq(D74_a2_Adder_in1,D74_m2_Multiplier_out)
eq(D74_a2_Adder_out,D74_g)
eq(D74_a,3)
eq(D74_b,2)
eq(D74_c,2)
eq(D74_d,3)
eq(D74_e,3)
eq(D74_f,10)
eq(D74_g,12)
watchsumleq([nab_D74_m1_Multiplier,], 1)
watchsumleq([nab_D74_m2_Multiplier,], 1)
```

```
watchsumleq([nab_D74_m3_Multiplier,], 1)
watchsumleq([nab_D74_a1_Adder,s10_D74_a1_Adder,], 1)
watchsumleq([nab_D74_a2_Adder,s10_D74_a2_Adder,], 1)
watchsumgeq([nab_D74_m1_Multiplier,], 1)
watchsumgeq([nab_D74_m2_Multiplier,], 1)
watchsumgeq([nab_D74_m3_Multiplier,], 1)
watchsumgeq([nab_D74_a1_Adder,s10_D74_a1_Adder,], 1)
watchsumgeq([nab_D74_a2_Adder,s10_D74_a2_Adder,], 1)

 **EOF**
```

3. The SIMOL program from Figure 3.1. For further information on the components defined within the program, we refer the reader to Chapter 5.

```
kbase GPRSCell;
component P2PMeter{
attribute int mdist, codeset, mRate, dataAmount;
 constraints{
  mdist={1..3};
  codeset= {1..4};
  }
}

component FPC {
 attribute int value;
 constraints(default){
  value= 1;
  }
 constraints(x1){
  value= {2..4};
  }
```

```
 constraints(unknown) {
  ;
  }
}

component ATS{
 attribute int value;
 constraints(default){
  value= 1;
  }
 constraints(a1){
  value= 2;
  }
 constraints(a2){
  value= 4;
  }
 }

component BTS{
 attribute int fpc, availableTimeSlots;
 constraints{
  FPC fpc1;
  ATS ats1;
  availableTimeSlots= ats1.value;
  fpc= fpc1.value;
  }
 }

component Cell{
 attribute int neededR, P2PNo, realR, usedTimeSlots;
 constraints{
  P2PNo=3;
  P2PMeter  s[3];
```

```
  BTS b;
  forall(P2PMeter ){
   mdist=1;
   if(codeset=1){
    mRate=8;
    }
   if(codeset=2){
    mRate=12;
    }
   if(codeset=3){
    mRate=14;
    }
   if(codeset=4){
    mRate=20 ;
    }
   }
  neededR=16;
  realR=sum([s], mRate)/P2PNo;
  realR>=neededR;
  usedTimeSlots=neededR/realR;
  }
 transition{
  forall(P2PMeter){
  if(mdist=1 and codeset=2){
   codeset.next={2,3};
   }
  if(mdist=3 and codeset=2){
   codeset.next={2,1};
   }
  }
 }
}
```

The MINION mapping will be:

```
MINION 3

**VARIABLES**
DISCRETE Cell_neededR_S0 {0..20000}
DISCRETE Cell_P2PNo_S0 {0..1000}
DISCRETE Cell_realR_S0 {0..20000}
DISCRETE Cell_usedTimeSlots_S0 {0..1000}
DISCRETE Cell_s_P2PMeter_mdist_S0[3] {0..1000}
DISCRETE Cell_s_P2PMeter_codeset_S0[3] {0..1000}
DISCRETE Cell_s_P2PMeter_mRate_S0[3] {0..1000}
DISCRETE Cell_s_P2PMeter_dataAmount_S0[3] {0..1000}
DISCRETE Cell_b_BTS_fpc_S0 {0..1000}
DISCRETE Cell_b_BTS_availableTimeSlots_S0 {0..1000}
DISCRETE Cell_b_BTS_fpc1_FPC_value_S0 {0..1000}
DISCRETE Cell_b_BTS_ats1_ATS_value_S0 {0..1000}
BOOL ifCond0_S0
BOOL ifCond1_S0
BOOL ifCond2_S0
BOOL ifCond3_S0
BOOL ifCond4_S0
BOOL ifCond5_S0
BOOL ifCond6_S0
BOOL ifCond7_S0
BOOL ifCond8_S0
BOOL ifCond9_S0
BOOL ifCond10_S0
BOOL ifCond11_S0
DISCRETE aux1_S0 {0..20000}
DISCRETE aux2_S0 {0..20000}
DISCRETE aux3_S0 {0..20000}
```

```
BOOL ifCond12_S0
BOOL ifCond13_S0
BOOL ifCond14_S0
BOOL ifCond15_S0
BOOL ifCond16_S0
BOOL ifCond17_S0
BOOL ifCond18_S0
BOOL ifCond19_S0
BOOL ifCond20_S0
BOOL ifCond21_S0
BOOL ifCond22_S0
BOOL ifCond23_S0
BOOL ifCond24_S0
BOOL ifCond25_S0
BOOL ifCond26_S0
BOOL ifCond27_S0
BOOL ifCond28_S0
BOOL ifCond29_S0
BOOL ifCond30_S0
BOOL ifCond31_S0
BOOL ifCond32_S0
BOOL ifCond33_S0
BOOL ifCond34_S0
BOOL ifCond35_S0
DISCRETE Cell_neededR_S1 {0..20000}
DISCRETE Cell_P2PNo_S1 {0..1000}
DISCRETE Cell_realR_S1 {0..20000}
DISCRETE Cell_usedTimeSlots_S1 {0..1000}
DISCRETE Cell_s_P2PMeter_mdist_S1[3] {0..1000}
DISCRETE Cell_s_P2PMeter_codeset_S1[3] {0..1000}
DISCRETE Cell_s_P2PMeter_mRate_S1[3] {0..1000}
DISCRETE Cell_s_P2PMeter_dataAmount_S1[3] {0..1000}
DISCRETE Cell_b_BTS_fpc_S1 {0..1000}
```

```
DISCRETE Cell_b_BTS_availableTimeSlots_S1 {0..1000}
DISCRETE Cell_b_BTS_fpc1_FPC_value_S1 {0..1000}
DISCRETE Cell_b_BTS_ats1_ATS_value_S1 {0..1000}
BOOL ifCond0_S1
BOOL ifCond1_S1
BOOL ifCond2_S1
BOOL ifCond3_S1
BOOL ifCond4_S1
BOOL ifCond5_S1
BOOL ifCond6_S1
BOOL ifCond7_S1
BOOL ifCond8_S1
BOOL ifCond9_S1
BOOL ifCond10_S1
BOOL ifCond11_S1
DISCRETE aux1_S1 {0..20000}
DISCRETE aux2_S1 {0..20000}
DISCRETE aux3_S1 {0..20000}
BOOL ifCond12_S1
BOOL ifCond13_S1
BOOL ifCond14_S1
BOOL ifCond15_S1
BOOL ifCond16_S1
BOOL ifCond17_S1
BOOL ifCond18_S1
BOOL ifCond19_S1
BOOL ifCond20_S1
BOOL ifCond21_S1
BOOL ifCond22_S1
BOOL ifCond23_S1
BOOL ifCond24_S1
BOOL ifCond25_S1
BOOL ifCond26_S1
```

```
BOOL ifCond27_S1
BOOL ifCond28_S1
BOOL ifCond29_S1
BOOL ifCond30_S1
BOOL ifCond31_S1
BOOL ifCond32_S1
BOOL ifCond33_S1
BOOL ifCond34_S1
BOOL ifCond35_S1
DISCRETE Cell_neededR_S2 {0..20000}
DISCRETE Cell_P2PNo_S2 {0..1000}
DISCRETE Cell_realR_S2 {0..20000}
DISCRETE Cell_usedTimeSlots_S2 {0..1000}
DISCRETE Cell_s_P2PMeter_mdist_S2[3] {0..1000}
DISCRETE Cell_s_P2PMeter_codeset_S2[3] {0..1000}
DISCRETE Cell_s_P2PMeter_mRate_S2[3] {0..1000}
DISCRETE Cell_s_P2PMeter_dataAmount_S2[3] {0..1000}
DISCRETE Cell_b_BTS_fpc_S2 {0..1000}
DISCRETE Cell_b_BTS_availableTimeSlots_S2 {0..1000}
DISCRETE Cell_b_BTS_fpc1_FPC_value_S2 {0..1000}
DISCRETE Cell_b_BTS_ats1_ATS_value_S2 {0..1000}
BOOL ifCond0_S2
BOOL ifCond1_S2
BOOL ifCond2_S2
BOOL ifCond3_S2
BOOL ifCond4_S2
BOOL ifCond5_S2
BOOL ifCond6_S2
BOOL ifCond7_S2
BOOL ifCond8_S2
BOOL ifCond9_S2
BOOL ifCond10_S2
BOOL ifCond11_S2
```

```
DISCRETE aux1_S2 {0..20000}
DISCRETE aux2_S2 {0..20000}
DISCRETE aux3_S2 {0..20000}
BOOL ifCond12_S2
BOOL ifCond13_S2
BOOL ifCond14_S2
BOOL ifCond15_S2
BOOL ifCond16_S2
BOOL ifCond17_S2
BOOL ifCond18_S2
BOOL ifCond19_S2
BOOL ifCond20_S2
BOOL ifCond21_S2
BOOL ifCond22_S2
BOOL ifCond23_S2
BOOL ifCond24_S2
BOOL ifCond25_S2
BOOL ifCond26_S2
BOOL ifCond27_S2
BOOL ifCond28_S2
BOOL ifCond29_S2
BOOL ifCond30_S2
BOOL ifCond31_S2
BOOL ifCond32_S2
BOOL ifCond33_S2
BOOL ifCond34_S2
BOOL ifCond35_S2
BOOL default_Cell_b_BTS_fpc1_FPC
BOOL default_Cell_b_BTS_ats1_ATS
BOOL x1_Cell_b_BTS_fpc1_FPC
BOOL a1_Cell_b_BTS_ats1_ATS
BOOL a2_Cell_b_BTS_ats1_ATS
```

```
**SEARCH**
VARORDER [default_Cell_b_BTS_fpc1_FPC,default_Cell_b_BTS_ats1_ATS,
   x1_Cell_b_BTS_fpc1_FPC,a1_Cell_b_BTS_ats1_ATS,a2_Cell_b_BTS_ats1_ATS,]
VALORDER [d, d, a, a, a]
PRINT ALL
**CONSTRAINTS**
eq(Cell_P2PNo_S0,3)
w-inrange(Cell_s_P2PMeter_mdist_S0[0], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S0[1], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S0[2], [1,3])
w-inrange(Cell_s_P2PMeter_codeset_S0[0], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S0[1], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S0[2], [1,4])
reifyimply(eq(Cell_b_BTS_fpc1_FPC_value_S0,1), default_Cell_b_BTS_fpc1_FPC)
reifyimply(w-inrange(Cell_b_BTS_fpc1_FPC_value_S0, [2,4]),
                                                    x1_Cell_b_BTS_fpc1_FPC)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S0,1), default_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S0,2), a1_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S0,4), a2_Cell_b_BTS_ats1_ATS)
eq(Cell_b_BTS_availableTimeSlots_S0,Cell_b_BTS_ats1_ATS_value_S0)
eq(Cell_b_BTS_fpc_S0,Cell_b_BTS_fpc1_FPC_value_S0)
eq(Cell_s_P2PMeter_mdist_S0[0],1)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],1), ifCond0_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[0],8), ifCond0_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],2), ifCond1_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[0],12), ifCond1_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],3), ifCond2_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[0],14), ifCond2_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],4), ifCond3_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[0],20), ifCond3_S0)
eq(Cell_s_P2PMeter_mdist_S0[1],1)
reify(eq(Cell_s_P2PMeter_codeset_S0[1],1), ifCond4_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[1],8), ifCond4_S0)
```

```
reify(eq(Cell_s_P2PMeter_codeset_S0[1],2), ifCond5_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[1],12), ifCond5_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[1],3), ifCond6_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[1],14), ifCond6_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[1],4), ifCond7_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[1],20), ifCond7_S0)
eq(Cell_s_P2PMeter_mdist_S0[2],1)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],1), ifCond8_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[2],8), ifCond8_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],2), ifCond9_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[2],12), ifCond9_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],3), ifCond10_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[2],14), ifCond10_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],4), ifCond11_S0)
reifyimply(eq(Cell_s_P2PMeter_mRate_S0[2],20), ifCond11_S0)
eq(Cell_neededR_S0,16)
sumleq([Cell_s_P2PMeter_mRate_S0], aux1_S0)
sumgeq([Cell_s_P2PMeter_mRate_S0], aux1_S0)
div(aux1_S0, Cell_P2PNo_S0, aux2_S0)
eq(Cell_realR_S0,aux2_S0)
ineq(Cell_neededR_S0,Cell_realR_S0, 0)
div(Cell_neededR_S0, Cell_realR_S0, aux3_S0)
eq(Cell_usedTimeSlots_S0,aux3_S0)
eq(Cell_P2PNo_S1,3)
w-inrange(Cell_s_P2PMeter_mdist_S1[0], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S1[1], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S1[2], [1,3])
w-inrange(Cell_s_P2PMeter_codeset_S1[0], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S1[1], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S1[2], [1,4])
reifyimply(eq(Cell_b_BTS_fpc1_FPC_value_S1,1), default_Cell_b_BTS_fpc1_FPC)
reifyimply(w-inrange(Cell_b_BTS_fpc1_FPC_value_S1, [2,4]),
                                                    x1_Cell_b_BTS_fpc1_FPC)
```

```
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S1,1), default_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S1,2), a1_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S1,4), a2_Cell_b_BTS_ats1_ATS)
eq(Cell_b_BTS_availableTimeSlots_S1,Cell_b_BTS_ats1_ATS_value_S1)
eq(Cell_b_BTS_fpc_S1,Cell_b_BTS_fpc1_FPC_value_S1)
eq(Cell_s_P2PMeter_mdist_S1[0],1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],1), ifCond0_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[0],8), ifCond0_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],2), ifCond1_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[0],12), ifCond1_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],3), ifCond2_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[0],14), ifCond2_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],4), ifCond3_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[0],20), ifCond3_S1)
eq(Cell_s_P2PMeter_mdist_S1[1],1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],1), ifCond4_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[1],8), ifCond4_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],2), ifCond5_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[1],12), ifCond5_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],3), ifCond6_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[1],14), ifCond6_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],4), ifCond7_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[1],20), ifCond7_S1)
eq(Cell_s_P2PMeter_mdist_S1[2],1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],1), ifCond8_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[2],8), ifCond8_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],2), ifCond9_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[2],12), ifCond9_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],3), ifCond10_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[2],14), ifCond10_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],4), ifCond11_S1)
reifyimply(eq(Cell_s_P2PMeter_mRate_S1[2],20), ifCond11_S1)
eq(Cell_neededR_S1,16)
```

```
sumleq([Cell_s_P2PMeter_mRate_S1], aux1_S1)
sumgeq([Cell_s_P2PMeter_mRate_S1], aux1_S1)
div(aux1_S1, Cell_P2PNo_S1, aux2_S1)
eq(Cell_realR_S1,aux2_S1)
ineq(Cell_neededR_S1,Cell_realR_S1, 0)
div(Cell_neededR_S1, Cell_realR_S1, aux3_S1)
eq(Cell_usedTimeSlots_S1,aux3_S1)
eq(Cell_P2PNo_S2,3)
w-inrange(Cell_s_P2PMeter_mdist_S2[0], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S2[1], [1,3])
w-inrange(Cell_s_P2PMeter_mdist_S2[2], [1,3])
w-inrange(Cell_s_P2PMeter_codeset_S2[0], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S2[1], [1,4])
w-inrange(Cell_s_P2PMeter_codeset_S2[2], [1,4])
reifyimply(eq(Cell_b_BTS_fpc1_FPC_value_S2,1), default_Cell_b_BTS_fpc1_FPC)
reifyimply(w-inrange(Cell_b_BTS_fpc1_FPC_value_S2, [2,4]),
                                                    x1_Cell_b_BTS_fpc1_FPC)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S2,1), default_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S2,2), a1_Cell_b_BTS_ats1_ATS)
reifyimply(eq(Cell_b_BTS_ats1_ATS_value_S2,4), a2_Cell_b_BTS_ats1_ATS)
eq(Cell_b_BTS_availableTimeSlots_S2,Cell_b_BTS_ats1_ATS_value_S2)
eq(Cell_b_BTS_fpc_S2,Cell_b_BTS_fpc1_FPC_value_S2)
eq(Cell_s_P2PMeter_mdist_S2[0],1)
reify(eq(Cell_s_P2PMeter_codeset_S2[0],1), ifCond0_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[0],8), ifCond0_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[0],2), ifCond1_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[0],12), ifCond1_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[0],3), ifCond2_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[0],14), ifCond2_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[0],4), ifCond3_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[0],20), ifCond3_S2)
eq(Cell_s_P2PMeter_mdist_S2[1],1)
reify(eq(Cell_s_P2PMeter_codeset_S2[1],1), ifCond4_S2)
```

```
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[1],8), ifCond4_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[1],2), ifCond5_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[1],12), ifCond5_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[1],3), ifCond6_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[1],14), ifCond6_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[1],4), ifCond7_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[1],20), ifCond7_S2)
eq(Cell_s_P2PMeter_mdist_S2[2],1)
reify(eq(Cell_s_P2PMeter_codeset_S2[2],1), ifCond8_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[2],8), ifCond8_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[2],2), ifCond9_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[2],12), ifCond9_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[2],3), ifCond10_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[2],14), ifCond10_S2)
reify(eq(Cell_s_P2PMeter_codeset_S2[2],4), ifCond11_S2)
reifyimply(eq(Cell_s_P2PMeter_mRate_S2[2],20), ifCond11_S2)
eq(Cell_neededR_S2,16)
sumleq([Cell_s_P2PMeter_mRate_S2], aux1_S2)
sumgeq([Cell_s_P2PMeter_mRate_S2], aux1_S2)
div(aux1_S2, Cell_P2PNo_S2, aux2_S2)
eq(Cell_realR_S2,aux2_S2)
ineq(Cell_neededR_S2,Cell_realR_S2, 0)
div(Cell_neededR_S2, Cell_realR_S2, aux3_S2)
eq(Cell_usedTimeSlots_S2,aux3_S2)
watchsumleq([default_Cell_b_BTS_fpc1_FPC,x1_Cell_b_BTS_fpc1_FPC,], 1)
watchsumleq([default_Cell_b_BTS_ats1_ATS,a1_Cell_b_BTS_ats1_ATS,
                                         a2_Cell_b_BTS_ats1_ATS,], 1)
watchsumgeq([default_Cell_b_BTS_fpc1_FPC,x1_Cell_b_BTS_fpc1_FPC,], 1)
watchsumgeq([default_Cell_b_BTS_ats1_ATS,a1_Cell_b_BTS_ats1_ATS,
                                         a2_Cell_b_BTS_ats1_ATS,], 1)
reify(eq(Cell_s_P2PMeter_mdist_S0[0],1), ifCond18_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],2), ifCond19_S0)
reify(watched-and({eq(ifCond18_S0,1),eq(ifCond19_S0,1)}), ifCond20_S0)
```

```
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[0], [2,3]), ifCond20_S0)
reify(eq(Cell_s_P2PMeter_mdist_S0[0],3), ifCond21_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[0],2), ifCond22_S0)
reify(watched-and({eq(ifCond21_S0,1),eq(ifCond22_S0,1)}), ifCond23_S0)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[0], [2,1]), ifCond23_S0)
reify(eq(Cell_s_P2PMeter_mdist_S0[1],1), ifCond24_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[1],2), ifCond25_S0)
reify(watched-and({eq(ifCond24_S0,1),eq(ifCond25_S0,1)}), ifCond26_S0)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[1], [2,3]), ifCond26_S0)
reify(eq(Cell_s_P2PMeter_mdist_S0[1],3), ifCond27_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[1],2), ifCond28_S0)
reify(watched-and({eq(ifCond27_S0,1),eq(ifCond28_S0,1)}), ifCond29_S0)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[1], [2,1]), ifCond29_S0)
reify(eq(Cell_s_P2PMeter_mdist_S0[2],1), ifCond30_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],2), ifCond31_S0)
reify(watched-and({eq(ifCond30_S0,1),eq(ifCond31_S0,1)}), ifCond32_S0)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[2], [2,3]), ifCond32_S0)
reify(eq(Cell_s_P2PMeter_mdist_S0[2],3), ifCond33_S0)
reify(eq(Cell_s_P2PMeter_codeset_S0[2],2), ifCond34_S0)
reify(watched-and({eq(ifCond33_S0,1),eq(ifCond34_S0,1)}), ifCond35_S0)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S1[2], [2,1]), ifCond35_S0)
reify(eq(Cell_s_P2PMeter_mdist_S1[0],1), ifCond18_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],2), ifCond19_S1)
reify(watched-and({eq(ifCond18_S1,1),eq(ifCond19_S1,1)}), ifCond20_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[0], [2,3]), ifCond20_S1)
reify(eq(Cell_s_P2PMeter_mdist_S1[0],3), ifCond21_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[0],2), ifCond22_S1)
reify(watched-and({eq(ifCond21_S1,1),eq(ifCond22_S1,1)}), ifCond23_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[0], [2,1]), ifCond23_S1)
reify(eq(Cell_s_P2PMeter_mdist_S1[1],1), ifCond24_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],2), ifCond25_S1)
reify(watched-and({eq(ifCond24_S1,1),eq(ifCond25_S1,1)}), ifCond26_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[1], [2,3]), ifCond26_S1)
```

```
reify(eq(Cell_s_P2PMeter_mdist_S1[1],3), ifCond27_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[1],2), ifCond28_S1)
reify(watched-and({eq(ifCond27_S1,1),eq(ifCond28_S1,1)}), ifCond29_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[1], [2,1]), ifCond29_S1)
reify(eq(Cell_s_P2PMeter_mdist_S1[2],1), ifCond30_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],2), ifCond31_S1)
reify(watched-and({eq(ifCond30_S1,1),eq(ifCond31_S1,1)}), ifCond32_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[2], [2,3]), ifCond32_S1)
reify(eq(Cell_s_P2PMeter_mdist_S1[2],3), ifCond33_S1)
reify(eq(Cell_s_P2PMeter_codeset_S1[2],2), ifCond34_S1)
reify(watched-and({eq(ifCond33_S1,1),eq(ifCond34_S1,1)}), ifCond35_S1)
reifyimply(w-inset(Cell_s_P2PMeter_codeset_S2[2], [2,1]), ifCond35_S1)


 **EOF**
```