Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 69
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 69
Espoo 2001 HUT-TCS-A69

A REACHABILITY ANALYSER FOR ALGEBRAIC SYSTEM NETS

Marko Mäkelä

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 69
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 69
Espoo 2001 HUT-TCS-A69

A REACHABILITY ANALYSER FOR ALGEBRAIC SYSTEM NETS

Marko Mäkelä

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Marko Mäkelä

ISBN 951-22-5541-3
ISSN 1457-7615

Picaset Oy
Helsinki 2001

ABSTRACT: Concurrent and distributed systems are difficult to manage without using formal analysis methods. The user of formal methods has to find a balance between expressive power and tractability. A formalism with small expressive power may not suit well to describing all real-life systems, but on the other hand, performing exhaustive reachability analysis on a system defined using a highly expressive formalism may be an unsolvable problem.

Using extended many-sorted algebras, this work defines the data type system and the set of algebraic operations that the author has implemented in a reachability analyser intended for modelling computer software based communications protocols. The work also introduces the modelling formalism of the analyser, algebraic system nets.

The report discusses some implementation details of the reachability analyser both from a theoretical and from a software technology point of view. Finally, the work shows how some constructs difficult for other tools can be modelled using the new formalism, and describes how different programming language constructs can be transformed to parts of a model by a semi-automated compiler.

KEYWORDS: many-sorted algebras, algebraic system nets, distributed systems, reachability analysis

Contents

PREFACE

The author would like to thank Professor Nisse Husberg, Kimmo Varpaaniemi, D.Sc. (Tech.), Tommi Junttila, Keijo Heljanko and Tommi Syrjänen from the Laboratory for Theoretical Computer Science of Helsinki University of Technology for discussions, feedback and numerous valuable ideas. Also, Tommi Junttila introduced the author to Algebraic System Nets, the formalism the formal notations of this work are based on. Special thanks go to Professor Leo Ojala for his encouragement in the final stages of the work.

The research took place in the MARIA project financed by the National Technology Agency of Finland (TEKES), Nokia Research Center, Nokia Networks, the Helsinki Telephone Corporation and the Finnish Rail Administration. The work was also supported by the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and by a personal grant from Tekniikan Edistämissäätiö.

I would like to thank my friends for all kinds of support I have received during this research project, and Mikko Suonio in particular for reviewing this work and for his constructive comments.

Marko Mäkelä
Espoo, June 2001

1 INTRODUCTION

The rapid development of telecommunications has considerably increased the use of concurrent and distributed systems. Such systems are much more difficult to manage than stand-alone systems controlled by sequential program code. Concurrent and distributed systems call for new analysis tools, because those used for sequential programs are not very useful in an environment where errors are difficult to reproduce due to the asynchronous nature of the communication between processes.

One solution to this problem is the use of formal models, investigating them in reachability analysers. This technique ensures that the complete state space of the system is checked for certain properties and that complete execution traces to possible errors can be recorded.

The big problem with this approach is, however, the vast state space of all real systems. Therefore, reachability analysis should be applied at an early stage of the development, preferably already when the system is specified. In the specification, there usually are few implementation details that would complicate the analysis and explode the state space. It is also easier and cheaper to correct errors at this stage.

A problem that has not been addressed much is the creation of the model. This is a main obstacle that prevents the introduction of this analysis in the industry in a large scale. Existing analysers often require that a formal model is constructed by hand. This means that the creator must be an expert both in the real system design and in the formalism the analyser uses. Such experts are very hard to find.

One solution is to generate the models automatically, to translate the specification into a formal model. This requires that the specification language has solid semantics and does not leave any room for interpretation. There will always be a need for experts, but their workload can be reduced and the coverability of formal analysis techniques greatly improved by introducing appropriate computer aided tools.

There has been some work in this field [21, 28, 40] connected to the TNSDL programming language [49] used in digital telephone exchanges. A major difficulty was that the reachability analyser [57] did not support the data types used in the specification language. Thus the decision was made to design a new analyser that supports a large class of data types [38] and lets the modeller or translator writer concentrate on other things than the representation of data.

This work is about the design of data types for the formalism used in the MARIA analyser and about the implementation of the analyser. The analyser was designed to meet the demands of industrial systems. It is important to notice that while the data type system has been designed in such a way that it satisfies the *practical needs* of specification and programming languages, the modelling language of the analyser is *completely formal*.

The analyser applies a variant of Algebraic System Nets [31, 32, 48] with a fixed set of algebraic operations. Algebraic System Nets, inspired by many-sorted nets [2] and coloured nets [26], are a high-level enhancement of ordinary Petri Nets [46, 47].

Throughout the work we have strictly followed the requirements of reachability analysis, so that all models that can be entered to our tool can be analysed in a formal way. The whole reachability analyser has been designed with the following requirements in mind:

1. satisfaction of practical needs, ease of use
   - structured data types
   - aggregate operations on multi-sets
2. reachability analysis
   - strictly formal semantics
   - very tight representation of states
   - theoretical possibility to unfold all models to low-level nets
3. modular design; replaceable modules for
   - expression evaluation and handling
   - reachability analysis and model checking
   - reachability graph management

When constructing a formal model of a system, one should pay attention not only to correctness but also to possible difficulties in reachability analysis. The expressive power of the formalism is not merely important for easy modelling but also for efficient analysis. If the formalism contains high-level data types like queues and stacks, the set of reachable states can be reduced considerably.

The definition of Algebraic System Nets given in [32] was quite useful for this work, but some additions had to be made in order to achieve better expressiveness and efficiency in the analysis. For instance, our implementation allows multi-set terms with non-constant multiplicity and a special case of multi-set valued variables. Algebraic System Nets define a fairly generic framework for describing computations; this work defines a set of concrete data types and algebraic operations that should be adequate for the succinct description and efficient analysis of practical concurrent systems.

Designing data types for a formal analyser is quite different from designing data types for a programming language. The data types must not only be compatible with the formalism but also with the analysis techniques. Thus it was impossible to allow general pointers in our formalism, as we shall see in Chapter 5. Unbounded data types like lists and trees were also omitted. However, after careful examination it was possible to include most of the data types used in common programming languages. It was also possible to impose a total order on all data types—a useful feature in the analysis and a requirement for some powerful algebraic operations.

1.1 REACHABILITY ANALYSERS IN THE PAST AND PRESENT

Formal analysis tools based on exhaustive state space exploration, or reachability analysers, have greatly developed since the 1980s. There are numerous research groups both at universities and in commercial companies, and many tools have been developed for internal use, or just to see whether a theoretical idea might work in practice, often analysing theoretical models that do not directly have any roots in the real world.

The work on automated protocol validation and tools for Petri Nets began in the 1970's. In the beginning, when the memory capacity of computers was severely limited, reachability analysis could only be applied to rather simple systems. In the early 1980's, the most advanced analysers could handle systems of up to tens of thousands of reachable states, and model checking—checking whether there are execution paths that violate a property expressed in temporal or modal logic—was considered impractical by some researchers. [60]

In Finland, one of the first research projects on computer tools for reachability analysis started in the summer of 1980. Financed by the then Posts and Telecommunications of Finland, the three-year project resulted in a theorem prover for modal logic [33]. In 1984, the research continued in a joint four-year project of the Computer Technology Laboratory of Technical Research Centre of Finland (VTT) and the Digital Systems Laboratory of Helsinki University of Technology. Financed by the then Technology Development Centre of Finland (TEKES) and four industrial partners, the Rimst project (Rinnakkaisjärjestelmien määrittelyn ja suunnittelun tukijärjestelmä, or Support System for Specification and Design of Concurrent Systems) identified a number of practical and theoretical problems, some of which could be addressed during the project. When the project ended in 1988, the Digital Systems Laboratory continued to explore Petri Nets, while the group at VTT eventually developed a tool set for labelled transition systems [29].

At Helsinki University of Technology, the joint project resulted in a set of analysis tools for Predicate/Transition Nets [11] called PRENA [30]. The worst limitation of the tool was that it only could handle some thousands of reachable states. By that time, there were several reachability analysers in existence. A survey from 1988 [34] lists 22 tools for high-level Petri Nets, 13 of which perform reachability analysis. An earlier survey from 1985 [9] that includes tools for low-level nets lists 26 tools, most of which have been written in nonportable languages for proprietary systems. The situation seems to have stabilised a little: a survey from 1998 [52] focuses on ten tools for high-level nets.

The initial version of PROD, the successor of PRENA, was developed in 1989–1991 mostly as a student project. Capable of exhaustively analysing systems with millions of reachable states and performing model checking while generating the set of reachable states, the tool has been a flagship of the Digital Systems Laboratory, later Laboratory for Theoretical Computer Science, and it has been extended with advanced algorithms, such as a model checker for the branching time temporal logic [19] and the stubborn set method [56].

Difficulties in modelling some practical systems presented the need for a reachability analyser that supports a more powerful modelling formalism. One of the main goals of the three-year Maria project, which started in 1998, was to develop a reachability analyser that supports a highly expressive data type system. This goal has been achieved: our tool is capable of performing brute-force exhaustive reachability analysis both interactively and in batch mode. Still, much work remains to be done, and many of the advanced algorithms of Prod are being implemented in Maria. There are some promising reduction methods that have not been implemented in either analyser yet, such as symmetry reductions [27].

There are many institutions with a long tradition in reachability analysis. At Bell Labs, reachability analysers have been developed and used since the early 1980's. Their current analyser Spin [20] is designed for analysing computer protocols, and it is one of the few analysers that have successfully been applied to industrial-size systems. The tool is based on a process-oriented modelling language Promela whose syntax resembles some programming languages. The somewhat informal application-oriented approach of Spin has turned out to suit extremely well to modelling protocols.

Spin supports probabilistic verification, representing the set of reachable states with a very large hash table, a bit vector indexed by hash values. The larger the table and the better the hash function that converts system states to indices of the hash table, the smaller is the probability that an unexplored state is mistaken for an already explored one. When using this probabilistic method, one cannot be completely sure that the whole state space has been explored. Nevertheless, it is hard to match Spin in performance, especially with a tool meant for analysing all kinds of concurrent systems and not only protocols.

SDL, the CCITT Specification and Description Language [25], which is mainly used by the telecommunications industry for specifying protocols, has raised some interest both in the academic world and among commercial tool vendors. The verification tool by Telelogic [8] is probably the most capable reachability analyser for SDL, when it comes to the extent of supported language constructs. However, the tool does not appear to be able to perform model checking of properties expressed in temporal logic, and it generates the reachability graph in the system memory, which makes the exhaustive analysis option useless for nontrivial specifications. In academic projects, small subsets of SDL have been translated to the internal formalisms of some reachability analysers, such as Pep [16] and Prod [57].

1.2 RELATED WORK

Not all tool authors aim for the ability to efficiently perform exhaustive reachability analysis and model checking. For teaching purposes and for running simulations, a graphical user interface and an inscription language with a lot of expressive power seem to be more important than a solid theoretical background.

One tool, Design/CPN [41], is based on a formalism called Coloured Petri Nets [26]. While the tool is capable of performing exhaustive reachability analysis, the high expressive power of its inscription language—including the ability to define data types of an unbounded domain—makes it very difficult to unfold the net to a lower level, which is the domain of many advanced analysis methods. Models constructed in our formalism can always be unfolded in principle; see Section 7.1.3.

The efforts in modelling high-level languages using Petri Nets have shown that the underlying formalism must have enough expressive power to facilitate a straightforward transformation. Otherwise the transformation is awkward [28, 40], or the domain of the source language has to be severely restricted [17], or both, as we illustrate in Chapter 9. However, the introduction of new constructs must be carefully considered in order not to sacrifice tractability for expressiveness.

1.3 OUTLINE

We begin with some general observations on the properties of typical computer programs and suggesting how they should be exploited in formal analysis. In Chapter 3 we list some of the techniques that can be applied in order to detect errors in different stages of software development.

The second part of this work introduces a class of many-sorted algebras and defines Algebraic System Nets in Chapter 4. Chapter 5 gives an inductive definition of the concrete data types implemented in our reachability analyser [38] for Algebraic System Nets, and Chapter 6 defines the algebraic operations.

Chapter 7 discusses some implementation details, such as finding all enabled actions in a state, evaluating algebraic terms efficiently, and representing the states and enabled actions of a system as very short bit vectors. Finally, Chapters 8 and 9 motivate some of the design choices of our analyser by showing how certain common constructs, which are difficult to represent in other formalisms, can be translated to our class of nets in a straightforward way.

2 VERIFICATION OF CONCURRENT PROGRAMS

Many concurrent and distributed systems include components implemented in computer software.[1] Due to the complex nature of such systems, it is difficult to convince oneself of their proper operation under all circumstances just by looking at the program code. Concurrency related errors that manage to go undetected in the manufacturer's tests can be expensive to correct. The manufacturer would rather experience a system crash in the laboratory and not at the customer's premises.

2.1 CLASSIFYING PROGRAMS

In the software industry, there is a famous law referred to as the 20 %–80 % rule or the 10 %–90 % rule. Usually it refers to the size of a program and to its execution time: when a program is executed, most of the time is spent executing statements in a small fraction of the program text. This law could also be used to classify concurrent programs. Typically, only a small part of a concurrent program deals with concurrency, and assuming that the errors in the program are evenly distributed and that formal methods are not being applied, most of the time devoted to debugging will be spent trying to figure out why the concurrent algorithm fails under some circumstances.

2.1.1 Sequential Behaviour

Errors in the sequential parts of a computer program are relatively easy to locate. Modern compilers perform quite a lot of static analysis and can issue warnings for things like unused or uninitialised variables, for enumeration constants not handled in a `switch` statement, and so on.

Static checking can only detect a small class of errors. In practice, the majority of other errors in sequential code can be found by testing. Since sequential programs typically are fully deterministic, i.e. they always produce the same output given the same input, it is easy to reproduce erratic situations and to find their cause by stepping through the code in a debugger. This approach, although it is far from formal, works astonishingly well for sequential program code.

2.1.2 Concurrent Behaviour

Constructing or understanding event or message driven programs requires a different way of thinking than traditional algorithm driven programs. The event driven approach is common in reactive or embedded systems, in graphical user interfaces and in applications working over computer networks. As the input events or messages are typically generated by entities running concurrently with the system, the system as a whole tends to behave in a nondeterministic and irreproducible way.

[1] Verifying the correct operation of hardware circuits is beyond the scope of this work.

Exhaustive analysis of the concurrent behaviour of a system calls for the use of formal methods. Tests or simulations can only prove the presence of a bug (by accidentally finding one), but only exhaustive formal analysis can prove the absence of bugs in an abstract model of the implemented system.[2]

2.2 MAKING ABSTRACTIONS

Although it is possible to formally model any program written for a finite-memory computer system, it is often undesirable to do so, since the set of reachable global states of the system can be excessively large.

Large systems can be formally analysed by omitting details that are considered irrelevant or uninteresting. Abstractions can be made in many different ways, by altering a formal model or its interpretation. In this section, we give two examples of making abstractions in a model.

2.2.1 Atomising Sequences of Actions

Consider a system consisting of n concurrently executing processes. If each process performs k actions before communicating with other processes, the system will have $(k+1)^n$ possible global states and $nk(k+1)^{n-1}$ possible transitions between them. Figure 2.1 illustrates the reachability graph of such a system. The example is from the introductory part of Antti Valmari's dissertation [54, Section 3.5.1], which also contains proofs for the above numbers.

The k actions of each process could represent purely sequential behaviour, which can be abstracted away, efficiently letting $k = 1$. Still, the size of the thus abstracted state space grows exponentially with n. If we make the assumption that the processes execute synchronously, we have $n = 1$ and get a easily manageable number of states.

Since errors in sequential program code can be treated pretty well by using semi-formal techniques such as testing, it is reasonable to assume that there are no errors in the sequential parts when constructing a model for verifying the concurrent behaviour of a system. Thus, the parts internal to processes can be abstracted away from the model without losing any suspicious behaviour.

In our example system, n processes perform k internal actions. Since the processes cannot communicate with each other until each of them have completed the internal actions, there is no way the internal state of one process could affect the behaviour of other processes.

In the lattice-shaped reachability graph presented in Figure 2.1, the vertices in the middle represent intermediate states where each process has performed some but not all of its actions. If the actions are treated atomically—that is, once a process starts performing a sequence of internal actions, it will complete the whole sequence before other processes do anything—we will get rid of those intermediate states.

[2] Also formal analysis can produce wrong results if the model does not adequately represent the system and its environment.

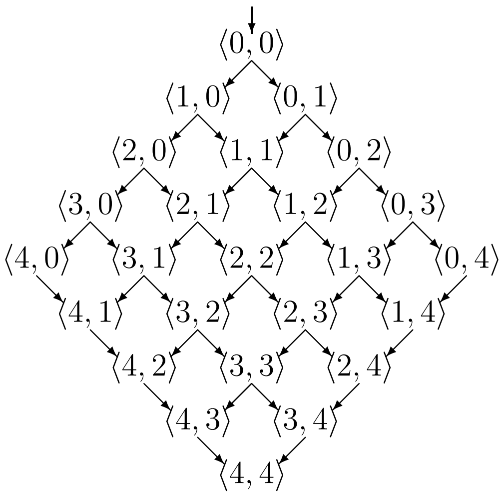


Figure 2.1: Actions in $k = 4$ Steps by $n = 2$ Processes

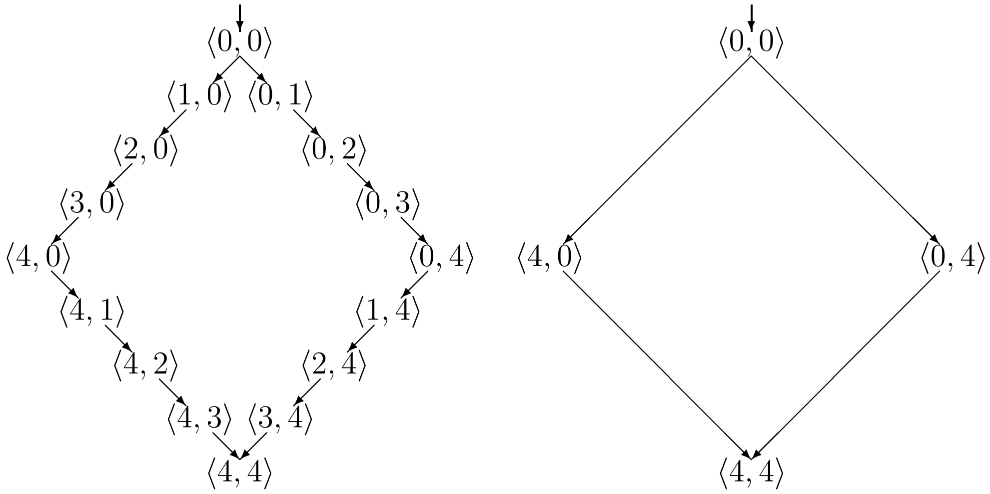


Figure 2.2: Atomic Actions in $k = 4$ Steps by $n = 2$ Processes

Figure 2.2 illustrates the effect of atomising sequences of local events. It is straightforward to see that instead of $(k+1)^n$ states and $nk(k+1)^{n-1}$ transitions, the reachability graph now only has $(n(k-1)+2)2^{n-1}$ states and $nk2^{n-1}$ transitions. Furthermore, if the sequences of events are collapsed to single events as shown in the right part of the figure, we will end up with 2^n states and $n2^{n-1}$ transitions.

This simple example shows that the reachability graph of a concurrent system can be vastly reduced by optimising the model. Further savings can be achieved during the reachability graph generation with advanced analysis techniques such as partial order reductions and symmetry reductions, which are summarised in [55].

2.2.2 Introducing Nondeterminism

It may seem that all concurrent systems can be efficiently modelled by converting complex local computations to single events, thus pruning most intermediate states. Unfortunately this turns out not to be the case. Some analysis techniques become utterly inefficient when the source and target state of a transition differ very much or when extremely many transitions are possible in one state. Introducing a few intermediate states could be of some help, but there is a better solution.

The sequential part of a concurrent system usually constrains and limits the behaviour of the system. For example, in a distributed game, the state of the game field and some decision algorithms will determine the type of messages sent to different counterparts. When we are only interested in generic properties like the absence of deadlocks in the communications protocol of the game, we can abstract away all information regarding the game field (also from the messages sent by different processes) and model the outcome of the decision algorithm by a nondeterministic choice of all possible outcomes, e.g. "game over" or "your turn."[3]

Omitting the typically large data structures needed by the sequential parts of a system from the abstract verification model will not merely reduce the amount of memory needed for storing the global state of the system; also the computational complexity involved with the reproduction of the exact behaviour of the sequential part will be reduced to a trivial linear-time operation of making a nondeterministic choice between all possible outcomes of the computation. Often also the outcome can be presented at a higher abstraction level, which can drastically reduce the domain.

The use of libraries characterises modern programming. It is difficult to imagine a large object-oriented program that does not make any use of built-in libraries for basic data structures, such as lists and search trees. Sometimes libraries are part of the run-time environment and completely transparent for the user. For instance, the virtual machine of the popular interpreted language Perl [58] originally designed for text processing has an instruction that matches character strings by generalised regular expressions [51, p. 324]. Including all these complex algorithms in the abstract model of a system would be like reinventing the wheel and certainly not worth the effort. It is rather unlikely that the library routines of a widely used language implementation contain errors.

Hidden Errors

There is no free lunch, not even in the world of abstractions. Consider a ring network where messages are relayed from node to node. If the communications protocol has been built so that a node may only send a message to another once it has received a message either for itself or for relaying, it can happen that a node has to wait for a long time before it can send a message. If the abstract model of the nodes is totally nondeterministic, so that any node can decide to send a message to another node at any time, the problem will fail to arise in the formal analysis.

Also abstracting chunks of sequential code can hide severe errors. If

[3]The original decision algorithm would probably have several outcomes equivalent to "your turn" in the abstracted version. As the state of the game field has been abstracted away, there is no need to distinguish "I pass" and "I move the piece from place x to place y."

the original piece of code under some circumstances can break the bounds of an array or be trapped in an infinite loop, such errors will be absent in a model that replaces the chunk with a nondeterministic choice.

False Alarms

Abstractions can not only hide errors; they can also cause false alarms. For instance, consider the distributed game described earlier. The communications protocol may have been specified so that it is an error to begin the game with the message "game over." Replacing the decision algorithms with nondeterministic choices can make it possible to start the protocol with that message. If the model has some kind of a guard against this, the analysis will yield a false alarm.

False alarms may seem harmless in comparison to hidden errors, but there is a problem involving the human element. If the person who interprets the results of the analysis receives an extensive list of error traces, he will probably check the realisability of some of them and ignore all further error messages of that type. Realisable errors can then also go unnoticed.

3 COMPUTER TOOLS FOR ANALYSIS

Applying formal methods in software development may seem straightforward. All sequential behaviour will be modelled with nondeterminism, and only the fraction of the system expressing concurrent behaviour must be translated to an abstract model. But the reality can sometimes be quite different. If the project manager wants to have a 50,000-line program verified by tomorrow or even by next week, manually constructing a verification model is not an option, especially if the work has to be done by someone who is not very familiar with formal methods or with the formalism applied by the analysis tool.

It is sad but true that in many software development projects, quality issues and formal methods are typically forgotten until the project encounters problems too severe to be solved in a reasonable time period of hacking and debugging. Designing the system incrementally, verifying and testing each refinement step, will help to detect errors earlier. For the computer tools described here, however, it does not make any difference whether they are used from the very beginning of a project or as a last resort to patch a sinking ship sailing on a stormy ocean.

3.1 INCOMPLETE METHODS

Practical systems tend to be so complex that exploring the complete state space, checking that the system performs correctly on all possible input sequences, is out of the question. Many errors can be found with less ambitious methods, some of which will be described here.

3.1.1 Static Analysis

Some errors can be discovered by relatively simple analysis. In the early days of computing when memory was scarce, compiler front-ends were rather simple, and separate programs were developed for detecting simple mistakes such as using the assignment operator where an equality comparison is likely to be intended.

Modern compilers perform all kinds of optimisations, which require fairly thorough static analysis of the input. Many sanity checks are performed as a side effect. It is common to check for unreachable program statements or for expressions whose value is not used, and often warnings about them indicate typing mistakes or even an error in the program logic.

Many compilers have some very strict static analysis options that are disabled by default, since they would cause numerous warnings for existing programs or even for the built-in libraries. Enabling such options only makes sense if the program does not need too big modifications in order to pass the checks. A good example is the check for uninitialised member variables by a C++ [23] compiler, which requires that all constructors make use of the member initialisation list, a less known feature of the language. Normally it makes sense to filter out certain warning messages reported for library interfaces, so that only relevant, correctable mistakes will be pointed out.

3.1.2 Instrumenting Program Code

Errors in large programs tend to have global effects. For instance, if an algorithm corrupts some data structures, the corruption may cause another, correctly implemented algorithm to fail. It depends on the underlying safety net, e.g. on the granularity of memory protection, how far the effects will propagate before the system crashes.

A tight safety net that would e.g. check the validity of arguments and return values of all procedure invocations will detect errors closer to their cause. Also in this case there is no free lunch: a program *instrumented* with all kinds of sanity checks will run slower, and if the program is free of errors, all the checks are redundant and unnecessary. Usually programs are instrumented during the development and testing phase, and the checks are omitted from the published version if the performance penalty is an issue.

Assertions

The C programming language [24] defines a macro for making *assertions*, for ensuring that a condition holds whenever the macro is evaluated. The macro is supplied with one argument, a truth-valued expression. If the expression evaluates to false, an error message will be displayed and the execution of the program will be aborted. Assertions can be disabled by defining a preprocessor symbol when compiling the program. Therefore, the same source code can be used for producing a fast, optimised executable as well as an instrumented executable for testing and debugging.

Detecting Resource Leakages

A program that allocates a resource but does not ever give it up is said to have a *resource leakage*. Probably the most common type of resource leakages is related to dynamic memory allocation. In programming languages whose run-time environments do not automatically release unused dynamically allocated memory in a process called *garbage collection*, it is the programmer's responsibility to explicitly deallocate memory areas that are no longer needed.

The need for explicit memory deallocation causes two major problems. First, programs that constantly allocate memory but do not deallocate it will eventually run out of memory. Second, if a program is too eager about deallocating memory, it can end up with dangling pointers pointing to deallocated memory. Accessing deallocated memory through these pointers or trying to deallocate a memory block more than once will probably corrupt data structures and may cause a crash somewhere else in the program code.

These problems can be addressed with programs called memory access debuggers. They work either by replacing the library routines for memory allocation and deallocation [45], by instrumenting the compiled code [7] or by instrumenting the program at compile time, a technique widely applied both by free [12, 59] and commercial tools.

Memory access debuggers have severe limitations. The instrumented program typically executes an order of magnitude slower than the original, and it will also require much more memory. This is due to the fact that memory access debuggers typically do not deallocate the memory deallocated by the instrumented program. By doing so, they can detect accesses to deallocated memory blocks. Also, the instrumented program code will allocate bigger memory blocks than the program actually asks for, so that it will be possible to detect accesses outside the bounds of the allocated memory area.

3.1.3 Regression Testing

One way to make software development resemble a controlled, deterministic process rather than the efforts of a panicking fire brigade is systematic testing. Whenever new features are added to a piece of software, a set of test cases are defined. The test cases, consisting of input sequences and expected output sequences, are stored in a version control system. Before accepting modifications to the program, it must pass all tests recorded so far. If it does not, the program has to be corrected or the test case must be updated.

The key problem with regression testing is the need for manual intervention. It is difficult to design tests that cover all of the program code. Moreover, when a test fails, one must ensure that it is not the test case that violates the specification, which often is not carved in stone but lives with the development cycle. In any case, it must be kept in mind that testing can only prove the presence of errors, not the absence of them.

3.2 FORMAL METHODS

Hardly anyone believes that the correct operation of a complex system could be formally verified simply by pushing a button. Expertise in formal methods will always be needed to be able to specify the properties the system should fulfill, to interpret the results when a complicated error is detected and to simplify the model of the system by means of abstractions. However, there is no reason why verification could not be made to appear as a push-button technology for system designers and engineers. It is enough to have experts behind the scenes.

Figure 3.1 illustrates our vision of applying formal methods to software verification. There are several reasons why model construction should be automated like this instead of manually constructing models compatible to verification. First, it is not reasonable to assume that everyone can learn formal methods to the necessary extent. Only a few people would have the necessary skills to construct the model. Second, modelling big systems involves much work, and humans make mistakes when performing mechanical tasks. Also, if the system is updated frequently, it would be hard to keep the model up to date.

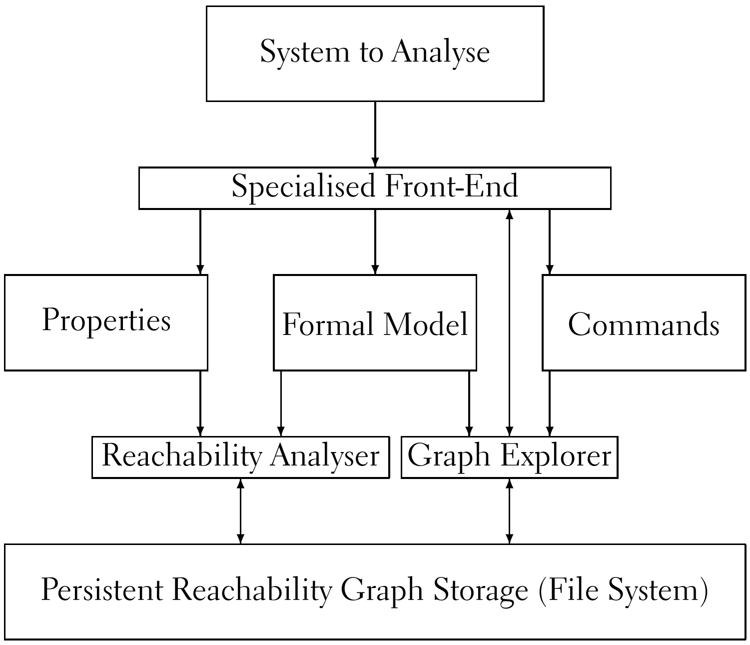


Figure 3.1: Automatic Verification of Computer Software

3.2.1 Constructing the Model

Since most designers are not familiar with other formalisms than the one they are directly working with, the verification model has to be hidden from them by means of a *specialised front-end*. The front-end will translate the implementation expressed in an application-specific high-level language to an analysable formal model, making reasonable abstractions. The model will be checked for the absence of deadlocks and for properties that have been entered in a database by an expert. Counterexamples, that is, execution paths leading to a state violating a desired property, will be translated back to the original formalism. In this way, the average user does not need to know the details of the underlying machinery.

Automatic model construction has many prerequisites. The system has to be implemented in a reasonably high-level language that has a well-defined semantics. If the system description cannot be represented with a single abstract syntax tree, it is difficult to construct a formal model of it. In practice, a system can be implemented using several programming languages and some electronic circuits. Models for low-level subsystems have to be constructed manually. Also the abstraction rules for omitting certain things from the model have to be specified by an expert. The databases of desired properties and abstraction rules may need to be updated during the development process if the specification changes.

Modelling Data

Incorporating big amounts of data in models intended for verification is generally a bad idea, since it raises the memory requirements for storing the global state of the model and often splits logically equivalent states to a number of states, multiplying the reachability graph. Also, complex data structures are typically maintained by sequential code, which should be omitted from the model for reasons given in Chapter 2.

Nevertheless, all models need some data, and if a system is simple enough, it can be formally analysed without omitting anything. Sometimes it is nice to experiment with a model, to see which parts need to be abstracted in order to be able to analyse the behaviour. This kind of prototyping and experimenting requires that the data types and algebraic expressions used in the system can be automatically translated to corresponding structures in the model.

If a system implementation uses a construct that cannot be directly expressed in the modelling language, constructing an automatic transformation can be a major challenge, especially when attention is paid to the performance of the generated model. Data types are very problematic in this aspect. It is cumbersome to represent arrays or first-in-first-out buffers in a formalism that is based on tuples of integers or enumerated constants. Doing so may be of academic interest [17, 21], but if the intention is to accomplish some real work, applying an inefficient transformation that only works in special cases will clearly be out of question.

A modelling language equipped with powerful data types such as structures, unions, arrays and variable-length buffers is an easy target formalism for compilers of programming languages. If the automatically constructed model generates too big a state space, simplifications and abstractions can be specified in the source formalism or in the abstraction rules. This can be done by someone familiar with the system but not necessarily knowing the modelling formalism.

Modelling the Environment

In order to analyse a program, it has to be given some input. In distributed systems, the input typically consists of events generated by the environment. Usually the environment does not act randomly, but it follows some disciplines. For instance, if one wants to place a telephone call, he will first dial the number and will not start talking until the call is answered.

The environment has to be part of the verification model, for various reasons. Connecting the model to a totally nondeterministically acting environment, which can synchronise with all kinds of actions at all times, could result in an unmanageable number of states and in a large number of errors that are impossible in the real system. Also, a nondeterministic environment may drive the system to erroneous states that are impossible in practice, or let the system continue from a situation that would be a deadlock if there were some constraints for the environment.

3.2.2 Analysing the Model

Merely constructing a formal model of a system does not solve anything. The model has to be analysed to see whether it (and the system it represents) fulfills some desired properties. Analysis can be carried out at different levels; here we will present some possibilities.

Reachability Analysis

Exhaustive reachability analysis, considering each possible action in each reachable state of the model, is the most complete analysis method. Its major drawback is that for many practical systems, the set of reachable states tends to be so large that the state space cannot be efficiently stored in a computer memory. There are several techniques that address this *state space explosion* problem; see [55] for an introduction.

Model Checking

Only a limited set of properties, such as the presence or absence of deadlock states, can be found out from the reachability graph generated by exhaustive reachability analysis. Model checking can prove or find violating execution paths for properties typically expressed in temporal logic. If a property does not hold, a counterexample will usually be found after generating only a part of the reachability graph.

Simulation

Like software development, also modelling can be an incremental, experimental process. Simulation is a very useful tool that lets one to experiment with a model, to make sure that it behaves in the intended way. An incorrectly working model will often produce an infinite state space. Performing exhaustive reachability analysis on such a model would be a waste of time; often hours or even days can pass before the analyser will run out of memory.

Simulation is also very useful for illustration and teaching purposes. In simulation, only a small subset of possible actions will be carried out. Simulations typically work on one trace, always performing an action on the state generated by the previously performed action. The next action can be chosen either randomly among the set of possible actions, or it can be chosen by the user.

A somewhat different approach has been implemented in [38]. Instead of letting the user to choose actions, the analyser will perform all possible actions that are possible in the state picked up by the user. This kind of simulation will not produce a single trace, but a (partial) reachability graph, which can be explored at all times. It is also possible to evaluate temporal logic formulae in a state. Only if the model checker does not find a counterexample for the formula, the complete reachability graph will have to be generated, unless the query is interrupted. Interactive on-demand reachability graph generation appears to combine the advantages of simulation with the advantages of reachability analysis at a relatively small price.

4 ALGEBRAIC SYSTEM NETS

Our formalism for modelling and analysing concurrent systems is the class of Algebraic System Nets [31, 32, 48], which we will describe in this chapter. Later parts of this work will refine and restrict the formalism, as used by our implementation [38] of a reachability analyser.

4.1 SIGNATURES AND ALGEBRAS

Algebraic System Nets, as their name suggests, are based on algebras. The algebras we represent here contain two extensions: error checking and short-circuit evaluation. Our notion of errors, inspired by [27], bears similarities with exception conditions [15] and abstract errors [14]. To keep the notation simple, we do not distinguish between different kinds of errors.

The motivation behind the second extension of our algebras, short-circuit evaluation of algebraic terms, is to narrow the gap between algebras and optimised computer implementations of expression evaluators.

Languages have two major properties: syntax (appearance) and semantics (interpretation). For algebraic systems, the syntax is defined in a signature, which contains symbols and terms but does not define any interpretation for them. The semantics is given in the algebra, which consists of the signature and of functions corresponding to the operation symbols given in the signature.

Our definition of algebras makes use of a concept of families, collections of sets. We assume that the reader has a knowledge in some mathematical preliminaries such as sets.

Definition 4.1 (Families) *For a set I,*

$$A = \bigcup_{i \in I} A_i$$

is a family *of sets if A_i is a set for each $i \in I$. Furthermore, a family A is* pairwise disjoint *if for all $i, j \in I$, $i \neq j \Rightarrow A_i \cap A_j = \emptyset$.*

4.1.1 Signatures, Variables and Terms

Definition 4.2 (Signatures) *A* signature

$$\mathbf{S} = \langle \mathcal{S}, \mathcal{F}, \mathcal{G} \rangle$$

consists of

1. *a non-empty set $\mathcal{S}$ of sort names (*sorts*)*
2. *a pairwise disjoint family $\mathcal{F} = \bigcup_{\sigma \in \mathcal{S}^*, s \in \mathcal{S}} \mathcal{F}_{\sigma,s}$ of* functional operation symbols

3. *a pairwise disjoint family* $\mathcal{G} = \bigcup_{s',s \in \mathcal{S}} \mathcal{G}_{s',s^n}$ *of* short-circuit operation symbols *where* s^n *stands for* $\underbrace{s, \ldots, s}_{n \text{ times}}$*;* $\mathcal{F} \cap \mathcal{G} = \emptyset$.

An operation symbol $f \in \mathcal{F}_{s_1 \ldots s_n, s}$ stands for a functional operation from the *domain sorts* $s_1, \ldots, s_n$ to the *range sort* s of f. The set $\mathcal{F}_{\lambda,s}$ is called the set of $\mathbf{S}$-*constant symbols of sort* s, where λ denotes the empty sequence.[1]

An operation symbol $g \in \mathcal{G}_{s',s^n}$ stands for an operation from s' to an operation from s^n to s, where s' is the *selection sort* and s the *range sort* of g.

Definition 4.3 (Variables) *A pairwise disjoint family*

$$\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$$

of symbols is called a family of $\mathbf{S}$-variables.

Using variables and the two kinds of operation symbols, we can build $\mathbf{S}$-terms, sequences of symbols, according to the following definition, which defines a kind of grammar. Algebraic terms can be viewed as strings of symbols; their interpretation is defined separately.

Definition 4.4 (Terms) *The set* $\mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$ *of* $\mathbf{S}$-terms of sort $s \in \mathcal{S}$ over $\mathcal{V}$ *is the minimal set defined inductively by the following rules.*

1. $\mathcal{V}_s \subseteq \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$.
2. *For* $n \geq 0$*, if* $f \in \mathcal{F}_{s_1 \ldots s_n, s}$ *and* $T_i \in \mathbf{T}^{\mathbf{S}}_{s_i}(\mathcal{V})$ *for* $1 \leq i \leq n$*, then* $f(T_1, \ldots, T_n) \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$.
3. *For* $n \geq 0$*, if* $g \in \mathcal{G}_{s',s^n}$ *and* $T' \in \mathbf{T}^{\mathbf{S}}_{s'}(\mathcal{V})$ *and* $T_i \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$ *for* $1 \leq i \leq n$*, then* $g(T', T_1, \ldots, T_n) \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$.

The set $\mathbf{T}^{\mathbf{S}}_s(\emptyset)$ *is the set of* $\mathbf{S}$-ground terms of sort s.

4.1.2 Algebras, Assignments and Evaluations

An algebra corresponding to a signature gives an interpretation for the sorts and operations of the signature. It assigns each sort a carrier and each functional operation symbol a function. Chapter 5 defines one possible family of carriers (called data types), and Chapter 6 defines a family of operations on them. This chapter proceeds in a more abstract level, independent of the actual carriers and functions.

Definition 4.5 (Short Circuit Error Algebras) *Let* $\mathbf{S} = \langle \mathcal{S}, \mathcal{F}, \mathcal{G} \rangle$ *be a signature. An* $\mathbf{S}$-short circuit error algebra*, or* $\mathbf{S}$-algebra

$$\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$$

consists of:

[1] By our convention, sequence indices are written in ascending order. When the index of the last element of a sequence is smaller than the index of the first element, the sequence is empty. For instance, if $n < 1$, the sequence $s_1 \ldots s_n$ equals λ, the empty sequence.

1. *a pairwise disjoint family* $\mathcal{D}^{\mathcal{A}} = \bigcup_{s\in\mathcal{S}} \mathcal{D}^{\mathcal{A}}_s$ *of non-empty* carriers

2. *the* undefined symbol $\epsilon \notin \mathcal{D}^{\mathcal{A}}$

3. *a pairwise disjoint family* $\check{\mathcal{D}}^{\mathcal{A}} = \bigcup_{s\in\mathcal{S}} \check{\mathcal{D}}^{\mathcal{A}}_s$ *of* augmented carriers*; for all* $s \in \mathcal{S}$, $\check{\mathcal{D}}^{\mathcal{A}}_s = \mathcal{D}^{\mathcal{A}}_s \cup \{\epsilon\}$

4. *a pairwise disjoint family* $\mathcal{F}^{\mathcal{A}} = \bigcup_{f\in\mathcal{F}} f^{\mathcal{A}}$ *of* functional operations*; for all* $f \in \mathcal{F}_{s_1\ldots s_n,s}$ *with* $n \geq 0$, $f^{\mathcal{A}}$ *is a mapping*
$$f^{\mathcal{A}} : \check{\mathcal{D}}^{\mathcal{A}}_{s_1} \times \cdots \times \check{\mathcal{D}}^{\mathcal{A}}_{s_n} \to \check{\mathcal{D}}^{\mathcal{A}}_s$$
such that the image of the subset
$$(\check{\mathcal{D}}^{\mathcal{A}}_{s_1} \times \cdots \times \check{\mathcal{D}}^{\mathcal{A}}_{s_n}) \setminus (\mathcal{D}^{\mathcal{A}}_{s_1} \times \cdots \times \mathcal{D}^{\mathcal{A}}_{s_n})$$
equals $\{\epsilon\}$*; that is, whenever an argument equals* ϵ*, so will also the result*

5. *bijective mappings*
$$o_{\mathcal{D}^{\mathcal{A}}_{s'}} : \mathcal{D}^{\mathcal{A}}_{s'} \to \left\{0, \ldots, \left|\mathcal{D}^{\mathcal{A}}_{s'}\right| - 1\right\}$$
for each short-circuit term $g \in \mathcal{G}_{s',s^n}$*, where the selection sort* s' *has a finite carrier* $\mathcal{D}^{\mathcal{A}}_{s'}$ *with* $\left|\mathcal{D}^{\mathcal{A}}_{s'}\right| = n$*.*

Definition 4.6 (Assignments) *Let* $\mathcal{A}$ *be an algebra. The set*
$$V^{\mathcal{A}}(\mathcal{V}) = \left\{ v \,\middle\|\, v : \bigcup_{s\in\mathcal{S}} \left(\mathcal{V}_s \to \check{\mathcal{D}}^{\mathcal{A}}_s\right) \right\}$$
is the set of assignments *to the variables of the family* $\mathcal{V}$*.*

Note that we allow also *undefined variables*, which are assigned the undefined value ϵ. Given an assignment, $\mathbf{S}$-terms can be evaluated as follows:

Definition 4.7 (Evaluations of Terms) *An assignment* $v \in V^{\mathcal{A}}(\mathcal{V})$ *is carried to the corresponding* evaluation of terms
$$e^{\mathcal{A}}_v : \bigcup_{s\in\mathcal{S}} \left(\mathbf{T}^{\mathbf{S}}_s(\mathcal{V}) \to \check{\mathcal{D}}^{\mathcal{A}}_s\right)$$
in the following inductive definition for each $T \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$*:*

1. *If* $T \in \mathcal{V}_s$*, then*
$$e^{\mathcal{A}}_v(T) = v(T).$$

2. *If* $T = f(T_1, \ldots, T_n)$*,* $f \in \mathcal{F}_{s_1\ldots s_n,s}$ *and* $T_i \in \mathbf{T}^{\mathbf{S}}_{s_i}(\mathcal{V})$ *for* $1 \leq i \leq n \geq 0$*, then*
$$e^{\mathcal{A}}_v(T) = f^{\mathcal{A}}(e^{\mathcal{A}}_v(T_1), \ldots, e^{\mathcal{A}}_v(T_n)).$$

3. *If* $T = g(T', T_1, \ldots, T_n)$*,* $g \in \mathcal{G}_{s',s^n}$*,* $T' \in \mathbf{T}^{\mathbf{S}}_{s'}(\mathcal{V})$ *and* $T_i \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$ *for* $1 \leq i \leq n \geq 1$*, then*
$$e^{\mathcal{A}}_v(T) = \begin{cases} \epsilon & \text{if } e^{\mathcal{A}}_v(T') = \epsilon \\ e^{\mathcal{A}}_v(T_{k+1}) & \text{if } \exists k \in \{0, \ldots, n-1\} : k = o_{\mathcal{D}^{\mathcal{A}}_{s'}}(e^{\mathcal{A}}_v(T')) \\ \epsilon & \text{otherwise.} \end{cases}$$

4.1.3 Multi-Set Signatures and Algebras

For convenience, we shall introduce a special kind of signatures and algebras based on multi-sets. First we shall define multi-sets and some basic operations on them.

Definition 4.8 (Multi-Set) *A* multi-set *over a finite non-empty set A is a function*

$$\mu : A \to \mathbb{N}$$

from the set A to the set of natural numbers. For an element $a \in A$, $\mu(a)$ is called the multiplicity *of a in μ.*

Definition 4.9 (Set of Multi-Sets) *The* set of all multi-sets *over A is denoted by*

$$\mathcal{M}(A) = \{\mu \parallel \mu : A \to \mathbb{N}\}\,.$$

Definition 4.10 (Multi-Set Relations and Operations) *Let there be a finite non-empty set A and $\mu_1, \mu_2 \in \mathcal{M}(A)$. We define the following relations and operations:*

1. $\mu_1 = \mu_2$ *if* $\mu_1(a) = \mu_2(a)$ *for all* $a \in A$ *(equality)*
2. $\mu_1 \le \mu_2$ *if* $\mu_1(a) \le \mu_2(a)$ *for all* $a \in A$ *(containment)*
3. $a \in \mu_1$ *if* $\mu_1(a) > 0$ *(membership)*
4. $\mu_1 + \mu_2 = \{\langle a, \mu_1(a) + \mu_2(a)\rangle \parallel a \in A\}$ *(union)*
5. $\mu_1 - \mu_2 = \{\langle a, \max\{0, \mu_1(a) - \mu_2(a)\}\rangle \parallel a \in A\}$ *(difference)*
6. $n \cdot \mu_1 = \{\langle a, n \cdot \mu_1(a)\rangle \parallel a \in A\}$, $n \in \mathbb{N}$ *(scalar multiplication)*
7. $|\mu_1| = \sum_{a \in A} \mu_1(a)$ *(cardinality)*

A multi-set signature distinguishes two different kinds of sorts: basic sorts and multi-set sorts over basic sorts.

Definition 4.11 (Multi-Set Signature) *Let $\mathbf{S} = \langle \mathcal{S}, \mathcal{F}, \mathcal{G}\rangle$ be a signature with a finite set of sorts $\mathcal{S}$. Let $\mathcal{S}_\beta, \mathcal{S}_\mu \subseteq \mathcal{S}$ such that $\mathcal{S}_\beta \cup \mathcal{S}_\mu = \mathcal{S}$ and $\mathcal{S}_\beta \cap \mathcal{S}_\mu = \emptyset$, and let $\mu : \mathcal{S}_\beta \to \mathcal{S}_\mu$ be a bijective mapping from* basic sorts *$\mathcal{S}_\beta$ to* multi-set sorts *$\mathcal{S}_\mu$. Then*

$$\mathbf{S}_\mu = \langle \mathcal{S}, \mathcal{F}, \mathcal{G}, \mu\rangle$$

is a multi-set signature.

A multi-set algebra is a straightforward extension of an algebra. Based on a multi-set signature, it requires that the carrier of its each multi-set sort is the set of multi-sets over the carrier of the corresponding basic sort.

Definition 4.12 (Multi-Set Algebras) *Let $\mathbf{S}$ be a signature and $\mathbf{S}_\mu$ be the corresponding multi-set signature. An $\mathbf{S}$-algebra $\mathcal{A} = \langle \mathcal{D}^\mathcal{A}, \mathcal{F}^\mathcal{A}\rangle$ is an $\mathbf{S}_\mu$-*algebra *if for each $s \in \mathcal{S}_\beta$, $\mathcal{D}^\mathcal{A}_{\mu(s)} = \mathcal{M}(\mathcal{D}^\mathcal{A}_s)$.*

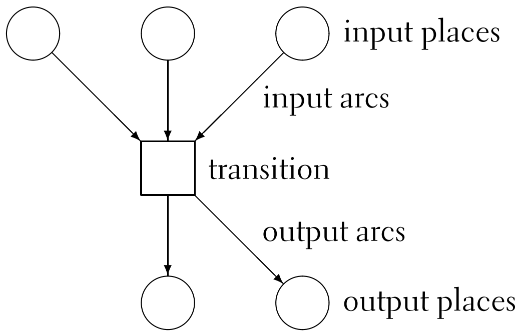


Figure 4.1: Graphical Representation of a Petri Net

4.2 ALGEBRAIC SYSTEM NETS

The algebras defined in the previous section form the core of a modelling formalism called Algebraic System Nets [31, 32, 48]. The formalism is a generalisation of Petri Nets [46, 47], which are a kind of generalised automata. Finite state automata have *states* and *actions* leading from one state to another, which are usually represented with circles and labelled directed arcs connecting circles representing states. Only one state is *active* at a time in a finite state automaton.

Petri Nets have *places*, graphically represented with circles, and *transitions*. Unlike the actions in finite state automata, transitions in Petri Nets may connect an arbitrary number of places. A transition, graphically represented with a rectangle, may have a number of *input and output places* connected to it via directed *arcs*. Input places are connected via input arcs (arcs leading to the transition) and output places via output arcs (arcs leading from the transition). Figure 4.1 illustrates the graphical notation typically used with Petri Nets.

In a finite state automaton, one state may be marked active at a time. In a Petri Net, a number of places may be marked with a *token*.[2] A finite state automaton may take an action if the source state of the action is active. A transition in a Petri Net is *enabled* if all its input places contain enough tokens. Only an enabled transition may *fire*, removing some tokens from its input places and producing some to its output places.[3]

Petri Nets and Algebraic System Nets clearly are more expressive than finite state automata. Once we have defined the formal semantics of Algebraic System Nets, we shall introduce the concept of a *reachability graph*, an automaton or a labelled transition system representing the complete behaviour of an Algebraic System Net.

Definition 4.13 (Algebraic System Nets) *An* algebraic system net

$$\Sigma = \langle \mathcal{N}, \mathcal{A}, \mathcal{V}, i \rangle$$

over $\mathcal{A}$ consists of

[2] In Algebraic System Nets, each place may be marked with any number of tokens.

[3] In Algebraic System Nets, tokens are elements of a multi-set; in low-level Petri Nets, tokens are not associated with values.

1. a net $\mathcal{N} = \langle \mathcal{P}, \mathcal{T}, \mathcal{F} \rangle$ where
   (a) $\mathcal{P}$, *a finite pairwise disjoint family of sort-indexed* places $\mathcal{P} = \bigcup_{s \in \mathcal{S}_\mu} \mathcal{P}_s$, *is a set of* $\mathbf{S}_\mu$*-variables whose assignments are multi-set-valued*
   (b) $\mathcal{T}$, *a finite set of* transitions, *is disjoint from the family of places:* $\mathcal{T} \cap \mathcal{P} = \emptyset$
   (c) $\mathcal{F} \subseteq (\mathcal{T} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{T})$ *is a* flow relation*; the items of* $\mathcal{F}$ *are called* arcs*; for* $t \in \mathcal{T}$ *and* $p \in \mathcal{P}$, $\langle p, t \rangle \in \mathcal{F}$ *is an* input arc *and* $\langle t, p \rangle \in \mathcal{F}$ *is an* output arc
2. *an* $\mathbf{S}_\mu$*-algebra* $\mathcal{A}$ *for a multi-set signature* $\mathbf{S}_\mu$=$\langle \mathcal{S}_\beta \cup \mathcal{S}_\mu, \mathcal{F}, \mathcal{G}, \mu \rangle$*; one basic sort* $b \in \mathcal{S}_\beta$ *is the Boolean sort of truth values with* $\mathcal{D}_b^\mathcal{A} = \mathbb{B} = \{\bot, \top\}$
3. *a sorted* $\mathbf{S}_\mu$*-variable set* $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$ *such that* $\mathcal{P}_s \cap \mathcal{V}_s = \emptyset$ *for all* $s \in \mathcal{S}_\mu$
4. *a* net inscription $i : (\mathcal{P} \cup \mathcal{T} \cup \mathcal{F}) \to \bigcup_{s \in \mathcal{S}} \mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ *such that*
   (a) $i(p) \in \mathbf{T}_s^{\mathbf{S}_\mu}(\emptyset)$ *for each* $s \in \mathcal{S}_\mu$ *and* $p \in \mathcal{P}_s$*; i.e.* $\bigcup_{p \in \mathcal{P}} i(p)$ *are the* initialisation expressions
   (b) $i(t) \in \mathbf{T}_b^{\mathbf{S}_\mu}(\mathcal{V})$ *for each* $t \in \mathcal{T}$*, where the* b *is the Boolean sort; i.e.* $\bigcup_{t \in \mathcal{T}} i(t)$ *are the* transition guards
   (c) $i(f) \in \mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ *for each* $f \in \mathcal{F}$ *such that* $f = \langle p, t \rangle$ *or* $f = \langle t, p \rangle$ *where* $s \in \mathcal{S}_\mu$*,* $p \in \mathcal{P}_s$ *and* $t \in \mathcal{T}$*; i.e.* $\bigcup_{f \in \mathcal{F}} i(f)$ *are the* arc expressions.

So far, we have formally defined a structure for modelling concurrent systems. We shall now give the structure a dynamic semantics, which can be applied to determine all possible executions (state–transition sequences) of the model. First we will define the states of the model.

Definition 4.14 (Marking, Local Marking and Tokens) *Let* Σ *be an algebraic system net. The mapping*

$$M : \bigcup_{s \in \mathcal{S}_\mu} \left(\mathcal{P}_s \to \mathcal{D}_s^\mathcal{A} \right),$$

is a marking *of* Σ*, and the set of all such markings* M *is denoted by* $\mathcal{M}_\Sigma$*.*

For each $s \in \mathcal{S}_\beta$ *and* $p \in \mathcal{P}_{\mu(s)}$*, we call* $M(p)$ *the* local marking *of place* p *and the items* $d \in \mathcal{D}_s^\mathcal{A}$ tokens *of sort* s*. We say that the place* p *contains* n *tokens carrying the value* d *if* $M(p)(d) = n$*.*

Definition 4.15 (Initial Marking) *Let* $v_\emptyset \in V^\mathcal{A}(\emptyset)$ *be an empty assignment. The marking* $M_0 : \mathcal{P} \to \mathcal{D}^\mathcal{A}$ *with*

$$M_0 : \bigcup_{s \in \mathcal{S}_\mu} \left\{ \left\langle p, e_{v_\emptyset}^\mathcal{A}(i(p)) \right\rangle \,\middle\|\, p \in \mathcal{P}_s \right\}$$

is called the initial marking *of* Σ*. Note that the initial marking is undefined if* $e_{v_\emptyset}^\mathcal{A}(i(p)) = \epsilon$ *for some* $p \in \mathcal{P}_s$*.*

The rôle of the places of an algebraic system net has now been covered. The places are associated with the global state of the model, the marking. In each state M of a net Σ, a place $p \in \mathcal{P}$ of Σ contains a multi-set $M(p)$ of tokens. The initial marking yields the initial distribution of tokens in the model, its initial state.

In an algebraic system net, places are connected via arcs and transitions. They define the behaviour of the system, the possible actions leading from one global state to another. In the following definitions, the possible actions in a state are referred to as the enabled transition modes in a marking. Furthermore, a transition can be fired in a marking in an enabled mode to transform the marking to another marking representing another state of the system.

Definition 4.16 (Input and Output Effect) *Let Σ be an algebraic system net, $t \in \mathcal{T}$ a transition and $\hat{v}, v \in V^{\mathcal{A}}(\mathcal{V})$ assignments such that v augments $\hat{v}$, i.e. for each $s \in \mathcal{S}$ and $x \in \mathcal{V}_s$, $\hat{v}(x) = \epsilon$ or $\hat{v}(x) = v(x)$. The two substitutions $t^-_{\hat{v}}, t^+_v : \bigcup_{s \in \mathcal{S}_\beta}(\mathcal{P}_{\mu(s)} \rightarrow \check{\mathcal{D}}^{\mathcal{A}}_{\mu(s)})$ are called the* input effect *and the* output effect*, respectively, and they are defined by:*

$$t^-_{\hat{v}}(p) = \begin{cases} e^{\mathcal{A}}_{\hat{v}}(i(\langle p, t\rangle)) & \text{if } \langle p, t\rangle \in \mathcal{F} \\ \mathcal{D}^{\mathcal{A}}_s \rightarrow \{0\} & \text{otherwise} \end{cases} \quad t^+_v(p) = \begin{cases} e^{\mathcal{A}}_{v}(i(\langle t, p\rangle)) & \text{if } \langle t, p\rangle \in \mathcal{F} \\ \mathcal{D}^{\mathcal{A}}_s \rightarrow \{0\} & \text{otherwise} \end{cases}$$

Definition 4.17 (Pre-Enabling Rule) *Let Σ be an algebraic system net, M its marking, $t \in \mathcal{T}$ a transition of Σ and $\hat{v} \in V^{\mathcal{A}}(\mathcal{V})$ an assignment. Transition t is* pre-enabled in mode $\hat{v}$ at marking M of Σ *if the following conditions hold for each $s \in \mathcal{S}_\mu$ and $p \in \mathcal{P}_s$:*

1. *$e^{\mathcal{A}}_{\hat{v}}(i(t)) = \top$; i.e. the transition guard holds*
2. *$t^-_{\hat{v}}(p) \neq \epsilon$; i.e. the input effect is defined*
3. *$t^-_{\hat{v}}(p) \leq M(p)$; i.e. each place contains enough tokens*

Definition 4.18 (Enabling Rule) *Let Σ, M, $t \in \mathcal{T}$ and $\hat{v} \in V^{\mathcal{A}}(\mathcal{V})$ be such that t is pre-enabled in mode $\hat{v}$ at marking M of Σ. Let $v \in V^{\mathcal{A}}(\mathcal{V})$ such that for each $s \in \mathcal{S}$ and $x \in \mathcal{V}_s$, $\hat{v}(x) = \epsilon$ or $\hat{v}(x) = v(x)$. Transition t is* enabled in mode v' *if for each $s \in \mathcal{S}_\mu$ and $p \in \mathcal{P}_s$ it holds that $t^+_v(p) \neq \epsilon$.*

Our definition of the transition enabling rule is divided into two parts, one based on the input effect and another one based on the output effect. Our definition distinguishes a set of variables

$$\mathcal{V}_o = \bigcup_{s \in \mathcal{S}} \{x \in \mathcal{V}_s \parallel \hat{v}(x) \neq v(x)\}$$

that are defined (not ϵ) in v but undefined in $\hat{v}$. These variables are called *output variables*, since they can only be evaluated on the output arcs of an enabled transition. An assignment[4] $\hat{v}$ can be extended to a number of assignments v e.g. by enumerating through the domain of each variable in $\mathcal{V}_o$, picking values such that a user-defined condition $c \in \mathbf{T}^{\mathbf{S}_\mu}_b(\mathcal{V})$ holds:

[4] Also the words "instance" and "valuation" are usual.

$e_v^{\mathcal{A}}(c) = \top$. The implementation in [38] does this, and it also provides variables for $t_{\hat{v}}^-(p)$ for each place p.[5] Formally, for each sort $s \in \mathcal{S}_\mu$ and place $p \in \mathcal{P}_s$, we have a variable $x_p \in \mathcal{V}_s$, and for each enabled instance $\hat{v}, v \in V^{\mathcal{A}}(\mathcal{V})$ of a transition $t \in \mathcal{T}$, it holds that

$$\begin{aligned} \hat{v}(x_p) &= \epsilon \\ v(x_p) &= t_{\hat{v}}^-(p). \end{aligned}$$

Note that Definition 4.13 does not allow variables to refer to the global state of the model, which would make it possible to simulate Turing machines [44, Chapter 2] with Algebraic System Nets.[6]

Definition 4.19 (Firing Rule) *Let Σ, M, $t \in \mathcal{T}$ and $\hat{v}, v \in V^{\mathcal{A}}(\mathcal{V})$ be such that t is enabled in mode v at marking M of Σ. The* firing of transition t *in mode v at marking M produces a marking*

$$M' = \bigcup_{s \in \mathcal{S}_\mu} \left\{ \langle p, M(p) - t_{\hat{v}}^-(p) + t_v^+(p) \rangle \parallel p \in \mathcal{P}_s \right\}.$$

The fact that M'' is the result of firing t in mode v at M can be written $M[t_v\rangle M''$.

The firing rule allows us to determine the set of markings that are reachable in the model, the reachable state space of the model.

Definition 4.20 (Reachable States) *Let Σ be an algebraic system net and M_0 the initial marking of Σ. The set of* reachable states *of Σ is the smallest set $R \subseteq \mathcal{M}_\Sigma$ fulfilling the following conditions:*

1. *$M_0 \in R$*
2. *$\{M' \parallel M[t_v\rangle M'\} \subseteq R$ for all $M \in R$, $t \in \mathcal{T}$ and $v \in V^{\mathcal{A}}(\mathcal{V})$ such that t is enabled in mode v at marking M.*

Definition 4.21 (Reachable Actions) *Let Σ be an algebraic system net and M_0 the initial marking of Σ and R the set of reachable states of Σ. The set of* reachable actions *of Σ is defined to be the smallest set $E \subseteq R \times (\mathcal{T} \times V^{\mathcal{A}}(\mathcal{V})) \times R$ for which*

$$\{\langle M, \langle t, v \rangle, M' \rangle \parallel M[t_v\rangle M'\} \subseteq E$$

for all $M \in R$, $t \in \mathcal{T}$ and $v \in V^{\mathcal{A}}(\mathcal{V})$ such that t is enabled in mode v at M.

Definition 4.22 (Reachability Graph) *Let Σ be an algebraic system net and M_0 the initial marking of Σ. Let R be the set of reachable states and E the set of reachable actions of Σ. The* reachability graph *of Σ is the directed graph*

$$G = \langle R, E \rangle.$$

[5] This extension to the formalism is redundant in the sense that a model making use of output variables can be transformed to a model without output variables. However, using output variables will speed up the reachability analysis and can make models more intuitive.

[6] As one of the consequences, the reachability problem for Algebraic System Nets would become undecidable, since an Algebraic System Net could decide whether a Turing machine halts.

5 DATA TYPES

Digital devices such as computers represent all data with binary digits, also known as bits, of zeros and ones. They are often grouped to fixed-width *machine words* of $n = 2^m$ bits, often interpreted as integer numbers between 0 and $2^n - 1$ or between -2^{n-1} and $2^{n-1} - 1$. This is fine for numerical applications, but many other applications would benefit from more sophisticated structures for managing data. The data model of practically all high-level programming languages is based on *data types*.

Mathematically, a data type can be represented as a set comprising all the acceptable values, the *domain* of the type. A computer implementation of data types has to be more specific. We will present an inductive definition of a data type system that has been implemented in [38].

5.1 DESIGN CRITERIA

Our data type system is based on the following design criteria.

Limited domain
: All data types $\mathcal{D}$ have a limited domain, $|\mathcal{D}| > 0$, facilitating a conversion between data items and sequences of machine words. The domains can be limited further by specifying ranges of allowed values.

Total order
: For all data types $\mathcal{D}$ and for all data items $i, j \in \mathcal{D}$, there is an asymmetric irreflexive transitive relation $<_\mathcal{D} \subset \mathcal{D} \times \mathcal{D}$,[1] i.e. at most one of the following holds: $i <_\mathcal{D} j$ or $j <_\mathcal{D} i$. If neither property holds, i and j refer to the same data item: $i = j$.

Tight representation
: For a data type $\mathcal{D}$ with a domain consisting of $n = |\mathcal{D}|$ data items, we construct a bijective mapping $o_\mathcal{D} : \mathcal{D} \to \{0, \ldots, n-1\}$ such that for all $d, d' \in \mathcal{D}$, $o_\mathcal{D}(d) < o_\mathcal{D}(d')$ if and only if $d <_\mathcal{D} d'$. The mapping allows each $d \in \mathcal{D}$ to be represented with $\lceil \log_2 n \rceil$ binary digits.

Expressive power
: There should be a straightforward transformation to our data types from most data types used in programming languages. We have to omit pointers to data items, since they would violate all the preceding properties, and unbounded data types such as lists and trees. We do have variable-length buffers of limited capacity.

[1] We deviate from the commonly used definition that includes reflexivity, i.e. $i \leq_\mathcal{D} i$.

5.1.1 Tight Representation

For a data type $\mathcal{D}$ with $|\mathcal{D}| = n$ and with a total order $<_{\mathcal{D}} \subset \mathcal{D} \times \mathcal{D}$, we define the mapping

$$o_{\mathcal{D}} : \mathcal{D} \to \{0, \ldots, n-1\} : d \mapsto |\{k \in \mathcal{D} \parallel k <_{\mathcal{D}} d\}| \,.$$

It is easy to see that the mapping is bijective and that it preserves the order of the mapped items. Because $<_{\mathcal{D}}$ is a total order, $\mathcal{D}$ can be written as

$$\mathcal{D} = \{d_0, \ldots, d_{n-1}\}$$

such that $d_{i-1} <_{\mathcal{D}} d_i$ for all $0 < i < n$. Now $o_{\mathcal{D}}$ maps each d_i, $0 \leq i < n$, to a unique value:

$$\begin{aligned} o_{\mathcal{D}}(d_i) &= |\{k \in \mathcal{D} \parallel k <_{\mathcal{D}} d_i\}| \\ &= |\{d_0, \ldots, d_{i-1}\}| \\ &= i. \end{aligned}$$

Since $o_{\mathcal{D}}(d_i) = i$, it holds that $o_{\mathcal{D}}(d_i) < o_{\mathcal{D}}(d_j)$ if and only if $i < j$, or $d_i <_{\mathcal{D}} d_j$. Thus, $o_{\mathcal{D}}$ is an order-preserving mapping.

5.1.2 Expressive Power

Pointers to data items are a problematic issue. We cannot allow them for several reasons. First of all, pointers having a large number of possible addressees (data objects they may point to) would destroy the modularity and locality properties of the verification model, thus restricting the applicability of compositional verification techniques. Second, pointers do not mix with the fundamental concept of Petri Nets: transitions affect the state only locally. Furthermore, pointers can be used to define data types of unlimited domain, which cannot be handled by analysis techniques that require unfolding; see Section 7.1.3. Last but not least, no total order can be defined for pointers in an obvious way independent of the underlying computer system.

Some restricted cases of pointer usage can be modelled by using the identifier type that will be introduced in Section 5.2.5; this will be discussed in Section 9.3.2.

5.2 SIMPLE TYPES

There are a number of data types whose data items can be directly represented with machine words. Usually machine words are interpreted as integers, but they can easily be interpreted as other enumerable unstructured data items as well.

5.2.1 Boolean

The Boolean type $\mathbb{B} = \{\bot, \top\}$ represents truth values. It has the total order $<_{\mathbb{B}} = \{\langle \bot, \top \rangle\}$.

5.2.2 Character

The Character type $\mathbb{K}$ represents the character set of the underlying computer system. The size of the alphabet is typically $|\mathbb{K}| = 2^8 = 256$. A bijective mapping

$$c : \mathbb{K} \to \{0, \ldots, |\mathbb{K}| - 1\}$$

assigns the character set an ordering. The total order $<_{\mathbb{K}} \subset \mathbb{K} \times \mathbb{K}$ is defined as

$$<_{\mathbb{K}} = \{\langle i, j\rangle \in \mathbb{K} \times \mathbb{K} \parallel c(i) < c(j)\}$$

based on the bijective mapping c and the total order in the set of integer numbers.

5.2.3 Integer

The Integer type $\mathbb{I}$ is a straightforward representation of machine words. Machine words can be interpreted either as signed or as unsigned integers.[2] The domain of the signed variant usually is $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, where n is the length of the machine word in bits, typically 2^5 or 2^6. The total order $<_{\mathbb{I}}$ is defined in the obvious way:

$$<_{\mathbb{I}} = \{\langle i, j\rangle \in \mathbb{I} \times \mathbb{I} \parallel i < j\}\,.$$

5.2.4 Enumerated Types

Enumerated data types are similar to integers. An enumerated type consists of named integer constants

$$\mathbb{E}_N \;= \{\langle k, n_k\rangle \parallel k \in N \wedge n_k \in \mathbb{I}\}$$

having distinct values:

$$\forall k \in N : \forall l \in N : n_k = n_l \Rightarrow k = l.$$

The total order $<_{\mathbb{E}_N}$ is defined in terms of the integer values of the named constants:

$$<_{\mathbb{E}_N} = \{\langle\langle i, n_i\rangle, \langle j, n_j\rangle\rangle \in \mathbb{E}_N \times \mathbb{E}_N \parallel n_i <_{\mathbb{I}} n_j\}\,.$$

5.2.5 Identifier Type

The identifier type is comparable to a pointer or a resource handle in a programming language environment. From the user's point of view, the type does not have any literals (constants), and identifier values can only be compared for equality; for the user, there is no total order or conversion to integers. Symmetry reductions are very effective on models making use of this type.

[2] Like many programming languages, our implementation [38] provides both unsigned and signed integers, but for the sake of simplicity, we make the assumption that all integers are of the same type.

The identifier type $\mathbb{J}_n$ is represented with integers,

$$\mathbb{J}_n = \{0, \ldots, n-1\},$$

and the total order $<_{\mathbb{J}_n}$ is defined in the obvious way:

$$<_{\mathbb{J}_n} = \{\langle i, j\rangle \in \mathbb{J}_n \times \mathbb{J}_n \parallel i < j\}.$$

5.3 STRUCTURED TYPES

Structured data types add expressive power to the data type system, especially when structured types can be constructed of other structured types without limitation. As all types in our data type system must have finite domains, we cannot allow any kind of recursion in data type definitions. For instance, it is impossible to define a tuple A with a component belonging to a union type containing the tuple A.

5.3.1 Tuple

A tuple data type is a Cartesian product of data types $\mathcal{D}_k$,

$$\mathbb{T}_{\mathcal{D}_1 \ldots \mathcal{D}_n} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n.$$

Syntactically, the data types $\mathcal{D}_k$ in a tuple are called the *components* of the tuple. Typically components are identified by names; our formal description shall use index numbers:

$$f_{\mathbb{T}_{\mathcal{D}_1 \ldots \mathcal{D}_n}} : \mathbb{T}_{\mathcal{D}_1 \ldots \mathcal{D}_n} \times \{1, \ldots, n\} \to \bigcup_{k=1}^{n} \mathcal{D}_k : \langle\langle d_1, \ldots, d_n\rangle, k\rangle \mapsto d_k$$

The total order $<_{\mathcal{D}}$ of $\mathcal{D} = \mathbb{T}_{\mathcal{D}_1 \ldots \mathcal{D}_n}$ is defined lexicographically in the little endian [5] way: The first component is the least significant one, and it will thus determine the ordering only when all other components are equal:

$$<_{\mathcal{D}} = \left\{\Big\langle\langle i_1, \ldots, i_n\rangle, \langle j_1, \ldots, j_n\rangle\Big\rangle \in \mathcal{D} \times \mathcal{D} \,\Bigg\|\, \bigvee_{k=1}^{n} (i_k <_{\mathcal{D}_k} j_k \wedge \bigwedge_{l=k+1}^{n} i_l = j_l)\right\}.$$

When $n = 0$, $\mathcal{D} = \mathbb{T} = \{\langle\rangle\}$ and $<_{\mathcal{D}} = \emptyset$, as we define the empty disjunction to be false.

Why is $<_{\mathcal{D}}$ a total order? For $l > k$, $i_l = j_l$ implies that $i_l <_{\mathcal{D}_l} j_l$ cannot hold (since $<_{\mathcal{D}_l}$ is a total order). On the other hand, if $i_k <_{\mathcal{D}_k} j_k$, we cannot have $i_k = j_k$. Therefore, at most one of the disjuncts in the condition can be true at a time. We shall now show that for any $i, j \in \mathcal{D}$, either $i <_{\mathcal{D}} j$, $j <_{\mathcal{D}} i$ or $i = j$. There are two cases. Either one disjunct k' is true or all disjuncts are false. When a disjunct k' is true, we have that for $l > k'$, $i_l = j_l$, and that $i_{k'} <_{\mathcal{D}_{k'}} j_{k'}$. The symmetric comparison yields $j_l = i_l$ for $l > k'$, but neither $j_{k'} <_{\mathcal{D}_{k'}} i_{k'}$ nor $j_{k'} = i_{k'}$ holds, because $<_{\mathcal{D}_{k'}}$ is a total order. Therefore $j <_{\mathcal{D}} i$ does not hold in this case. When all disjuncts in $i <_{\mathcal{D}} j$ are false, either $i_k = j_k$ for all $1 \leq k \leq n$, in which case $i = j$, or there exists a k'' such that $j_{k''} <_{D_{k''}} i_{k''}$ and thus $j <_{\mathcal{D}} i$.

5.3.2 Associative Array

An associative array is a collection of elements of a data type $\mathcal{D}_e$ indexed by data items of $\mathcal{D}_x$. Formally,

$$\mathbb{A}_{\mathcal{D}_x,\mathcal{D}_e} = \underbrace{\mathcal{D}_e \times \cdots \times \mathcal{D}_e}_{|\mathcal{D}_x| \text{ terms}}.$$

Arrays are similar to tuples, except that all components (elements) are of the same type $\mathcal{D}_e$ and that the elements are indexed by integer representations $o_{\mathcal{D}_x}(d)$ of the index values $d \in \mathcal{D}_x$ rather than by component names or numbers:

$$x_{\mathbb{A}_{\mathcal{D}_x,\mathcal{D}_e}} : \mathbb{A}_{\mathcal{D}_x,\mathcal{D}_e} \times \mathcal{D}_x \to \mathcal{D}_e : \big\langle \langle d_1, \ldots, d_{|\mathcal{D}_x|} \rangle, d \big\rangle \mapsto d_{1+o_{\mathcal{D}_x}(d)}.$$

The total order $<_{\mathcal{D}}$ of $\mathcal{D} = \mathbb{A}_{\mathcal{D}_x,\mathcal{D}_e}$ is defined lexicographically, in a way similar to the total order among tuples:

$$<_{\mathcal{D}} = \left\{ \Big\langle \langle i_1, \ldots, i_{|\mathcal{D}_x|} \rangle, \langle j_1, \ldots, j_{|\mathcal{D}_x|} \rangle \Big\rangle \in \mathcal{D} \times \mathcal{D} \,\middle\|\, \bigvee_{k=1}^{|\mathcal{D}_x|} (i_k <_{\mathcal{D}_e} j_k \wedge \bigwedge_{l=k+1}^{|\mathcal{D}_x|} i_l = j_l) \right\}.$$

Since the construction is essentially the same as for tuples (substituting $|\mathcal{D}_x|$ for n), it is easy to see that $<_{\mathcal{D}}$ is a total order.

5.3.3 Variable-Length Buffer

Linked lists, first-in-first-out queues and last-in-first-out stacks can be seen as variable-length buffers of data elements of type $\mathcal{D}_e$. Since all our data types have a bounded domain, our variable-length buffer has a maximum number n of data elements. Formally,

$$\mathbb{V}_{\mathcal{D}_e,n} = \bigcup_{k=0}^{n} \mathcal{D}_e^k$$

where the superscript indicates a Cartesian product

$$\mathcal{D}_e^k = \underbrace{\mathcal{D}_e \times \cdots \times \mathcal{D}_e}_{k \text{ times}}.$$

The total order $<_{\mathcal{D}}$ of $\mathcal{D} = \mathbb{V}_{\mathcal{D}_e,n}$ is defined by considering a missing item to be the smallest. A buffer value can only contain missing items as a contiguous sequence in its tail. Because of this, the lexicographical comparison can be implemented by comparing the actual buffer length first:

$$<_{\mathcal{D}} = \left\{ \Big\langle \langle (i)_1^p \rangle, \langle (j)_1^q \rangle \Big\rangle \in \mathcal{D} \times \mathcal{D} \,\middle\|\, p < q \vee \Big(p = q \wedge \bigvee_{k=1}^{m} (i_k <_{\mathcal{D}_e} j_k \wedge \bigwedge_{l=k+1}^{m} i_l = j_l) \Big) \right\}$$

where m denotes the minimum of p and q, and the abbreviation $(i)_1^p$ stands for the sequence $i_1, \ldots, i_p$.

5.3.4 Tagged Union

The union type $\mathbb{U}_{\mathcal{D}_1 \dots \mathcal{D}_n}$ can hold values of different types $\mathcal{D}_1, \dots, \mathcal{D}_n$. In order to ease type conversions, a value of type $\mathcal{D}_k$ will be tagged with the index number k.

$$\mathbb{U}_{\mathcal{D}_1 \dots \mathcal{D}_n} = \bigcup_{k=1}^{n} \{\langle k, d_k \rangle \parallel d_k \in \mathcal{D}_k\}$$

The total order $<_{\mathcal{D}}$ for $\mathcal{D} = \mathbb{U}_{\mathcal{D}_1 \dots \mathcal{D}_n}$ first compares index numbers and then values:

$$<_{\mathcal{D}} = \left\{ \langle \langle k_i, i \rangle, \langle k_j, j \rangle \rangle \in \mathcal{D} \times \mathcal{D} \,\middle\|\, k_i < k_j \lor (k_i = k_j \land i <_{\mathcal{D}_{k_i}} j) \right\}.$$

5.4 CONSTRAINTS

Type constraints limit the domain of a data type. A constraint function f maps each data item of a data type $\mathcal{D}$ to a Boolean value that specifies whether the data item is allowed in the constrained type

$$\mathcal{C}(\mathcal{D}, f : \mathcal{D} \to \mathbb{B}) = \{d \in \mathcal{D} \parallel f(d) = \top\}.$$

The total order $<_{\mathcal{C}(\mathcal{D},f)}$ is defined in terms of the total order $<_{\mathcal{D}}$ of the underlying data type $\mathcal{D}$:

$$<_{\mathcal{C}(\mathcal{D},f)} = \{\langle i, j \rangle \in \mathcal{C}(\mathcal{D}, f) \times \mathcal{C}(\mathcal{D}, f) \parallel i <_{\mathcal{D}} j\}.$$

In [38], constraints are implemented as lists of non-overlapping semi-open ranges interpreted as unions, and constraints can be defined for all data types. The grammar of the input language allows both unions and intersections of ranges to be entered. Figure 5.1 depicts how unions and intersections of ranges can be evaluated to convert constraints to the canonic form. Symmetric cases (obtained by swapping the rôles of r_1 and r_2) are omitted from the figure.

5.4.1 Computing the Union

When computing the union of two constraints c_1 and c_2, the algorithm implemented in [38] compares each range r_i^1 in c_1 with each range r_j^2 in c_2. If one of the cases depicted in Figure 5.1 applies, then the union of the two ranges will be added to the result. Otherwise the algorithm will compare whether the upper limit of r_i^1 is one less than the lower limit of r_j^2 or vice versa, and add the combination of the two ranges to the result if this is the case. This is how $\{1, 2\} \cup \{3\}$ will become $\{1, 2, 3\}$. If even this does not apply, the two ranges are disjoint, and both will be stored in the resulting constraint.

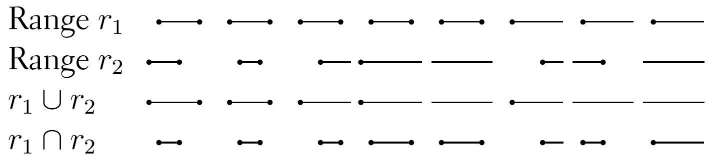


Figure 5.1: The Union and the Intersection of Ranges

5.4.2 Computing the Intersection

The intersection of two constraints c_1 and c_2 is computed by comparing each range r_i^1 in c_1 with each range r_j^2 in c_2. If one of the cases described in Figure 5.1 applies, the intersection of the two ranges will be added to the resulting constraint. Otherwise the two ranges are disjoint, and the resulting constraint will not be augmented. To simplify our implementation, we made all ranges closed. This eliminates six of the eight of cases illustrated in Figure 5.1.

6 ALGEBRAIC OPERATIONS

Computers work by performing operations on data. Big operations are implemented in terms of smaller operations, and at the lowest level, there are *basic operations* performed by the underlying computing machinery. Because operations are performed on data items, they are tightly coupled with the data type system. Here we shall present the basic operations implemented in [38]; the data type system is defined in Chapter 5.

It is quite common that there is some redundancy among basic operations, even in low-level languages. For instance, if the basic operations of an algebra include logical conjunction, logical disjunction and logical negation, the conjunction or the disjunction can be constructed in terms of the two remaining operations by applying De Morgan's law: $\neg a \wedge \neg b = \neg(a \vee b)$. High-level languages tend to have more redundant operations than low-level languages, so called *syntactic sugar* that adds expressive power to the language and allows for compact notation.

High-level programming languages define expressions and statements. In the Algebraic System Nets defined in Chapter 4, expressions do not have side effects (they cannot affect the assignment they are evaluated in), and there is only one form of a statement, the transition. A transition can be viewed as a statement that changes the value of some variables in an assignment.[1]

The algebraic operations defined in this chapter use the notational conventions

$$\begin{aligned} \mathsf{op} &\in \mathcal{F}_{s_1,\ldots,s_n,s} \\ \mathsf{op}^{\mathcal{A}} &: \check{\mathcal{D}}^{\mathcal{A}}_{s_1} \times \cdots \times \check{\mathcal{D}}^{\mathcal{A}}_{s_n} \to \check{\mathcal{D}}^{\mathcal{A}}_{s} \end{aligned}$$

and the implicit definition

$$\mathsf{op}^{\mathcal{A}}(d) = \epsilon \qquad \text{for all } d \in \left(\check{\mathcal{D}}^{\mathcal{A}}_{s_1} \times \cdots \times \check{\mathcal{D}}^{\mathcal{A}}_{s_n}\right) \setminus \left(\mathcal{D}^{\mathcal{A}}_{s_1} \times \cdots \times \mathcal{D}^{\mathcal{A}}_{s_n}\right).$$

Further algebraic operation definitions in this chapter augment the implicit definition. In further definitions, the domains consist of the carriers of the domain sorts (excluding the symbol ϵ), and the ranges are the carriers of the range sorts augmented with the symbol ϵ.

6.1 DESIGN CRITERIA

The basic operations were chosen according to the following criteria:

No side effects
: No operation modifies the environment it is evaluated in.

Total order
: With the exception of the identifier type defined in Section 5.2.5, there are operations for accessing the total order of data items e.g. via comparisons and via predecessor and successor operations.

[1] In Algebraic System Nets, these variables (in the algebraic sense) are the local markings of the places attached to the transition via input and output arcs.

Expressive power

All basic operations of popular programming languages like C [24] are covered, except for pointer arithmetics and operations having side effects. In addition, there are some operations that deal with multi-sets, which are an extension over the data type system defined in Chapter 5.

6.2 VARIABLES

In computer languages, there typically are two things that can be done to variables: they can be *declared* and *referenced*. In various high-level Petri Net formalisms, variables are often declared implicitly, at the points when the variables are first referenced. Declaring variables corresponds to extending the family of variables $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$ with new items.

Variable references $x \in \mathcal{V}_s$ for $s \in \mathcal{S}$ are not operations, but direct members of the set of terms: $x \in \mathbf{T}_s^{\mathbf{S}}(\mathcal{V})$.

6.3 OPERATIONS ON BASIC SORTS

The lowest level operations of our algebra are defined for the family of basic sorts $\mathcal{S}_\beta$, which is mapped to data types constructed using the structures defined in Chapter 5.

6.3.1 Constants

Constants in our algebra include literals of simple types, minimum and maximum value of ordered types, and integer constants representing the number of elements in a type. For each basic sort $s \in \mathcal{S}_\beta$ and $d \in \mathcal{D}_s^{\mathcal{A}}$, we define

$$\begin{aligned} \mathsf{constant}_d &\in \mathcal{F}_{\lambda,s} \\ \mathsf{constant}_d^{\mathcal{A}}() &= d. \end{aligned}$$

Note that we do not fix the syntax here. Especially if an implementation performs *constant folding*, replacing terms $g \in \mathbf{T}_s^{\mathbf{S}}(\emptyset)$ with terms $\mathsf{constant}_d()$ where $d = e_v^{\mathcal{A}}(g)$ for an empty assignment $v \in V^{\mathcal{A}}(\emptyset)$, there are numerous ways of writing constants.

There is also an undefined constant, whose purpose is to detect errors in an Algebraic System Net model. Remember, a transition instance can only be fired when all its arc expressions evaluate to defined values. In [38], there are two variations of the undefined constant: one that only causes an error message to be displayed for the current state, and another that causes the reachability analysis to be aborted.

$$\begin{aligned} \mathsf{undefined} &\in \mathcal{F}_{\lambda,s} \\ \mathsf{undefined}^{\mathcal{A}}() &= \epsilon. \end{aligned}$$

6.3.2 Total Order

Successor and Predecessor

For each sort $s \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_s^\mathcal{A}$ and the bijective mapping $o_{\mathcal{D}_s^\mathcal{A}} : \mathcal{D}_s^\mathcal{A} \to \left\{0, \ldots, \left|\mathcal{D}_s^\mathcal{A}\right| - 1\right\}$ defined in Section 5.1.1, we define the *successor* and *predecessor* operations as follows:

$$
\begin{aligned}
\mathsf{succ} &\in \mathcal{F}_{s,s} \\
\mathsf{succ}^\mathcal{A}(d) &= d_+ \\
\mathsf{pred} &\in \mathcal{F}_{s,s} \\
\mathsf{pred}^\mathcal{A}(d) &= d_-
\end{aligned}
$$

such that

$$
\begin{aligned}
o_{\mathcal{D}_s^\mathcal{A}}(d) + 1 &\equiv o_{\mathcal{D}_s^\mathcal{A}}(d_+) \pmod{|\mathcal{D}_s^\mathcal{A}|} \\
o_{\mathcal{D}_s^\mathcal{A}}(d) - 1 &\equiv o_{\mathcal{D}_s^\mathcal{A}}(d_-) \pmod{|\mathcal{D}_s^\mathcal{A}|}.
\end{aligned}
$$

Comparison

For the Boolean sort $b \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_b^\mathcal{A} = \mathbb{B} = \{\bot, \top\}$ and for each sort $s \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_s^\mathcal{A}$ and the total order $<_{\mathcal{D}_s^\mathcal{A}} \subset \mathcal{D}_s^\mathcal{A} \times \mathcal{D}_s^\mathcal{A}$, we define following comparison operations:

$$
\begin{aligned}
\mathsf{equal} &\in \mathcal{F}_{s,s,b} \\
\mathsf{unequal} &\in \mathcal{F}_{s,s,b} \\
\mathsf{less} &\in \mathcal{F}_{s,s,b} \\
\mathsf{greater} &\in \mathcal{F}_{s,s,b} \\
\mathsf{lessequal} &\in \mathcal{F}_{s,s,b} \\
\mathsf{greaterequal} &\in \mathcal{F}_{s,s,b} \\
\mathsf{equal}^\mathcal{A}(d_1, d_2) &= \begin{cases} \top & \text{if } d_1 = d_2 \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{unequal}^\mathcal{A}(d_1, d_2) &= \begin{cases} \bot & \text{if } d_1 = d_2 \\ \top & \text{otherwise} \end{cases} \\
\mathsf{less}^\mathcal{A}(d_1, d_2) &= \begin{cases} \top & \text{if } d_1 <_{\mathcal{D}_s^\mathcal{A}} d_2 \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{greater}^\mathcal{A}(d_1, d_2) &= \begin{cases} \top & \text{if } d_2 <_{\mathcal{D}_s^\mathcal{A}} d_1 \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{lessequal}^\mathcal{A}(d_1, d_2) &= \begin{cases} \bot & \text{if } d_2 <_{\mathcal{D}_s^\mathcal{A}} d_1 \\ \top & \text{otherwise} \end{cases} \\
\mathsf{greaterequal}^\mathcal{A}(d_1, d_2) &= \begin{cases} \bot & \text{if } d_1 <_{\mathcal{D}_s^\mathcal{A}} d_2 \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that equality and inequality comparisons are defined for all basic sorts. The implementation in [38] restricts the use of the other comparison operations on sorts that do not contain the identifier sort.

6.3.3 Logical Operations

For the Boolean sort $b \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_b^{\mathcal{A}} = \mathbb{B} = \{\bot, \top\}$, we define the following logical operations:

$$
\begin{aligned}
\textsf{not} &\in \mathcal{F}_{b,b} \\
\textsf{and} &\in \mathcal{F}_{b,b,b} \\
\textsf{or} &\in \mathcal{F}_{b,b,b} \\
\textsf{not}^{\mathcal{A}}(d_1) &= \begin{cases} \bot & \text{if } d_1 = \top \\ \top & \text{otherwise} \end{cases} \\
\textsf{and}^{\mathcal{A}}(d_1, d_2) &= \begin{cases} \top & \text{if } d_1 = \top \text{ and } d_2 = \top \\ \bot & \text{otherwise} \end{cases} \\
\textsf{or}^{\mathcal{A}}(d_1, d_2) &= \begin{cases} \top & \text{if } d_1 = \top \text{ or } d_2 = \top \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Our parser implementation [38] defines logical implication, equivalence and logical exclusive disjunction in terms of these operations.

6.3.4 Integer Arithmetics

For the integer sort $s \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_s^{\mathcal{A}} = \mathbb{I}$, we define the following basic arithmetic operations based on addition, subtraction, multiplication and truncated division of integer numbers. To simplify the following definitions, we do not explicitly state that when the result of an integer arithmetic operation does not belong to $\mathbb{I}$, it will be ϵ. Also, our definition of the division operation rounds the values towards negative infinity. Our implementation [38] uses the underlying C++ [23] implementation, which can as well round the values towards zero. To reduce implementation-defined behaviour, our modulus operation is only defined for positive integers.

$$
\begin{aligned}
\textsf{negate} &\in \mathcal{F}_{s,s} \\
\textsf{plus} &\in \mathcal{F}_{s,s,s} \\
\textsf{minus} &\in \mathcal{F}_{s,s,s} \\
\textsf{times} &\in \mathcal{F}_{s,s,s} \\
\textsf{divide} &\in \mathcal{F}_{s,s,s} \\
\textsf{modulus} &\in \mathcal{F}_{s,s,s} \\
\textsf{negate}^{\mathcal{A}}(d_1) &= 0 - d_1 \\
\textsf{plus}^{\mathcal{A}}(d_1, d_2) &= d_1 + d_2 \\
\textsf{minus}^{\mathcal{A}}(d_1, d_2) &= d_1 - d_2 \\
\textsf{times}^{\mathcal{A}}(d_1, d_2) &= d_1 \cdot d_2 \\
\textsf{divide}^{\mathcal{A}}(d_1, d_2) &= \begin{cases} \epsilon & \text{if } d_2 = 0 \\ \left\lfloor \frac{d_1}{d_2} \right\rfloor & \text{otherwise} \end{cases}
\end{aligned}
$$

$$\mathsf{modulus}^{\mathcal{A}}(d_1, d_2) = \begin{cases} \epsilon & \text{if } d_1 < 0 \\ \epsilon & \text{if } d_2 \leq 0 \\ d_1 - d_2 \cdot \left\lfloor \frac{d_1}{d_2} \right\rfloor & \text{otherwise} \end{cases}$$

Furthermore, in [38] we define the following binary arithmetic operations for integers. Their definition depends on the underlying implementation of the built-in C++ [23] operators ~, &, |, ^, << and >>, respectively. The bitwise logical operators will be evaluated without errors; the shift operators will evaluate to ϵ if the second argument, denoting the amount of bits to be shifted, is negative or at least the width of the machine word.

$$\begin{aligned} \mathsf{bitnot} &\in \mathcal{F}_{s,s} \\ \mathsf{bitand} &\in \mathcal{F}_{s,s,s} \\ \mathsf{bitor} &\in \mathcal{F}_{s,s,s} \\ \mathsf{bitxor} &\in \mathcal{F}_{s,s,s} \\ \mathsf{shiftl} &\in \mathcal{F}_{s,s,s} \\ \mathsf{shiftr} &\in \mathcal{F}_{s,s,s} \end{aligned}$$

6.3.5 Structure Operations

Structured data types, defined in Section 5.3, can be viewed as containers for data items. The operations defined for structured data types mostly deal with composing and decomposing data items of structured types.

Tuple

The tuple type, defined in Section 5.3.1, has three operations: composition, decomposition and substitution. Formally, for all $s, s_1, \ldots, s_n \in \mathcal{S}_\beta$ such that $\mathcal{D}_s^{\mathcal{A}} = \mathbb{T}_{\mathcal{D}_{s_1}^{\mathcal{A}} \ldots \mathcal{D}_{s_n}^{\mathcal{A}}}$ and for $1 \leq k \leq n$, we define the following operations:

$$\begin{aligned} \mathsf{cons}_s &\in \mathcal{F}_{s_1,\ldots,s_n,s} \\ \mathsf{component}_{s,k} &\in \mathcal{F}_{s,s_k} \\ \mathsf{assign}_{s,k} &\in \mathcal{F}_{s,s_k,s} \\ \mathsf{cons}_s^{\mathcal{A}}(d_1, \ldots, d_n) &= \langle d_1, \ldots, d_n \rangle \\ \mathsf{component}_{s,k}^{\mathcal{A}}(\langle d_1, \ldots, d_n \rangle) &= d_k \\ \mathsf{assign}_{s,k}^{\mathcal{A}}(\langle d_1, \ldots, d_n \rangle, e) &= \langle d_1, \ldots, d_{k-1}, e, d_{k+1}, \ldots, d_n \rangle \end{aligned}$$

Associative Array

For the two variants of the associative array defined in Section 5.3.2, there are four basic operations, three of which are analogous to the basic operations defined for tuples. The fourth operation shifts the elements of the array by a specified number of positions.

For all $s, s_1, s_2, s' \in \mathcal{S}_\beta$ such that $\mathcal{D}_s^{\mathcal{A}} = \mathbb{A}_{\mathcal{D}_{s_1}^{\mathcal{A}}, \mathcal{D}_{s_2}^{\mathcal{A}}}$ and $\mathcal{D}_{s'}^{\mathcal{A}} = \mathbb{I}$ and for $n = \left|\mathcal{D}_{s_1}^{\mathcal{A}}\right|$, we define the following basic operations:

$$\mathsf{cons}_s \in \mathcal{F}_{s_2^n,s}$$

$$
\begin{array}{rcl}
\mathsf{index}_s & \in & \mathcal{F}_{s,s_1,s_2} \\
\mathsf{assign}_s & \in & \mathcal{F}_{s,s_1,s_2,s} \\
\mathsf{shift}_s & \in & \mathcal{F}_{s,s',s} \\
\mathsf{cons}_s^{\mathcal{A}}(d_1,\ldots,d_n) & = & \langle d_1,\ldots,d_n\rangle \\
\mathsf{index}_s^{\mathcal{A}}(\langle d_1,\ldots,d_n\rangle, d) & = & d_k \\
\mathsf{assign}_s^{\mathcal{A}}(\langle d_1,\ldots,d_n\rangle, d, e) & = & \langle d_1,\ldots,d_{k-1},e,d_{k+1},\ldots,d_n\rangle \\
\mathsf{shift}_s^{\mathcal{A}}(\langle d_1,\ldots,d_n\rangle, l) & = & \langle d_{l'},\ldots,d_n,d_1,\ldots,d_{l'-1}\rangle
\end{array}
$$

where $k = 1 + o_{\mathcal{D}_{s_1}^{\mathcal{A}}}(d)$, $l' = l - \left|\mathcal{D}_{s_1}^{\mathcal{A}}\right| \left\lfloor \frac{l}{\left|\mathcal{D}_{s_1}^{\mathcal{A}}\right|} \right\rfloor$ and s_2^n stands for $\underbrace{s_2,\ldots,s_2}_{n \text{ times}}$.

Variable-Length Buffer

For the variable-length buffer defined in Section 5.3.3, our basic operations extend the set of operations traditionally defined for queues and stacks. For all $s, s_1, s' \in \mathcal{S}_\beta$ and $n \geq 0$ such that $\mathcal{D}_s^{\mathcal{A}} = \mathbb{V}_{\mathcal{D}_{s_1}^{\mathcal{A}},n}$ and $\mathcal{D}_{s'}^{\mathcal{A}} = \mathbb{I}$, and for $0 \leq k \leq n$, using the short-hand notations $(d)_l^m$ for the sequence $d_l,\ldots,d_m$ and s_1^k for a sequence of s_1 of length k, we define the following operations:

$$
\begin{array}{rcl}
\mathsf{cons}_s & \in & \mathcal{F}_{s_1^k,s} \\
\mathsf{enqueue}_s & \in & \mathcal{F}_{s,s_1,s} \\
\mathsf{enqueue\text{-}at}_s & \in & \mathcal{F}_{s,s_1,s',s} \\
\mathsf{push}_s & \in & \mathcal{F}_{s,s_1,s} \\
\mathsf{push\text{-}at}_s & \in & \mathcal{F}_{s,s_1,s',s} \\
\mathsf{remove}_s & \in & \mathcal{F}_{s,s} \\
\mathsf{remove\text{-}at}_s & \in & \mathcal{F}_{s,s',s} \\
\mathsf{peek}_s & \in & \mathcal{F}_{s,s_1} \\
\mathsf{peek\text{-}at}_s & \in & \mathcal{F}_{s,s',s_1} \\
\mathsf{free}_s & \in & \mathcal{F}_{s,s'} \\
\mathsf{used}_s & \in & \mathcal{F}_{s,s'} \\
\mathsf{cons}_s^{\mathcal{A}}((d)_1^k) & = & \langle (d)_1^k \rangle \\
\mathsf{enqueue}_s^{\mathcal{A}}(\langle (d)_1^k\rangle, d') & = & \begin{cases} \langle (d)_1^k, d'\rangle & \text{if } k < n \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{enqueue\text{-}at}_s^{\mathcal{A}}(\langle (d)_1^k\rangle, l, d') & = & \begin{cases} \langle (d)_1^{k-l}, d', (d)_{k-l+1}^k\rangle & \text{if } 0 \leq l \leq k < n \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{push}_s^{\mathcal{A}}(\langle (d)_1^k\rangle, d') & = & \begin{cases} \langle d', (d)_1^k\rangle & \text{if } k < n \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{push\text{-}at}_s^{\mathcal{A}}(\langle (d)_1^k\rangle, l, d') & = & \begin{cases} \langle (d)_1^l, d', (d)_{l+1}^k\rangle & \text{if } 0 \leq l \leq k < n \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{remove}_s^{\mathcal{A}}(\langle (d)_1^k\rangle) & = & \begin{cases} \langle (d)_2^k\rangle & \text{if } k \geq 1 \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{remove\text{-}at}_s^{\mathcal{A}}(\langle (d)_1^k\rangle, l) & = & \begin{cases} \langle (d)_1^l, (d)_{l+2}^k\rangle & \text{if } 0 \leq l < k \geq 1 \\ \epsilon & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{aligned}
\mathsf{peek}^{\mathcal{A}}_s(\langle (d)^k_1 \rangle) &= \begin{cases} d_1 & \text{if } k \geq 1 \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{peek\text{-}at}^{\mathcal{A}}_s(\langle (d)^k_1 \rangle, l) &= \begin{cases} d_{l+1} & \text{if } 0 \leq l < k \geq 1 \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{free}^{\mathcal{A}}_s(\langle (d)^k_1 \rangle) &= n - k \\
\mathsf{used}^{\mathcal{A}}_s(\langle (d)^k_1 \rangle) &= k.
\end{aligned}
$$

The non-orthodox operations that allow the buffer to be accessed in the middle will be needed for an efficient transformation of some language constructs discussed in Section 9.2.2.

Tagged Union

For the tagged union defined in Section 5.3.4, we define three operations. Let $s_1, \ldots, s_n \in \mathcal{S}_\beta$ for some $n > 0$, and let the corresponding carriers be $\mathcal{D}^{\mathcal{A}}_{s_1}, \ldots, \mathcal{D}^{\mathcal{A}}_{s_n}$. Furthermore, let the sorts $s, b \in \mathcal{S}_\beta$ have the carriers $\mathcal{D}^{\mathcal{A}}_s = \mathbb{U}_{\mathcal{D}^{\mathcal{A}}_{s_1} \ldots \mathcal{D}^{\mathcal{A}}_{s_n}}$ and $\mathcal{D}^{\mathcal{A}}_b = \mathbb{B} = \{\bot, \top\}$. We define the following operations for all $1 \leq k \leq n$:

$$
\begin{aligned}
\mathsf{cons}_{s,k} &\in \mathcal{F}_{s_k, s} \\
\mathsf{component}_{s,k} &\in \mathcal{F}_{s, s_k} \\
\mathsf{defined}_{s,k} &\in \mathcal{F}_{s,b} \\
\mathsf{cons}^{\mathcal{A}}_{s,k}(d) &= \langle k, d_k \rangle \\
\mathsf{component}^{\mathcal{A}}_{s,k}(\langle l, d_l \rangle) &= \begin{cases} d_l & \text{if } l = k \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{defined}^{\mathcal{A}}_{s,k}(\langle l, d_l \rangle) &= \begin{cases} \top & \text{if } l = k \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

6.3.6 Type Conversions

We define three kinds of type conversion operations. The first kind will convert values between simple elementary types. The other two kinds are specific to data type systems having constraints. It is possible to convert values between types that differ only in the constraint, or between structured types whose component types can be pairwise converted to each other. The implementation in [38] performs some static analysis and does not allow conversions between types that have no common values.

The formal definition for the generic conversions is simple: For all $s, s' \in \mathcal{S}_\beta$ with the carriers $\mathcal{D}^{\mathcal{A}}_s$ and $\mathcal{D}^{\mathcal{A}}_{s'}$, we define

$$
\begin{aligned}
\mathsf{convert}_{s,s'} &\in \mathcal{F}_{s,s'} \\
\mathsf{convert}^{\mathcal{A}}_{s,s'}(d) &= \begin{cases} d & \text{if } d \in \mathcal{D}^{\mathcal{A}}_{s'} \\ \langle k, d \rangle & \text{if } \mathcal{D}^{\mathcal{A}}_{s'} = \mathbb{U}_{\mathcal{D}^{\mathcal{A}}_{s_1} \ldots \mathcal{D}^{\mathcal{A}}_{s_n}} \text{ and } \exists_1 k : d \in \mathcal{D}^{\mathcal{A}}_{s_k} \\ d' & \text{if } \mathcal{D}^{\mathcal{A}}_{s} = \mathbb{U}_{\mathcal{D}^{\mathcal{A}}_{s_1} \ldots \mathcal{D}^{\mathcal{A}}_{s_n}},\ d = \langle k, d' \rangle \text{ and } d' \in \mathcal{D}^{\mathcal{A}}_{s'} \\ \epsilon & \text{otherwise} \end{cases}
\end{aligned}
$$

We use the short-hand notation $(d)^n_1$ for the sequence $d_1, \ldots, d_n$. For tuple sorts $s, s' \in \mathcal{S}_\beta$ of the respective component sorts $s_1, \ldots, s_n \in \mathcal{S}_\beta$

and $s'_1, \dots, s'_n \in \mathcal{S}_\beta$, with

$$\mathcal{D}^{\mathcal{A}}_s = \mathbb{T}_{\mathcal{D}^{\mathcal{A}}_{s_1} \dots \mathcal{D}^{\mathcal{A}}_{s_n}} \qquad \text{and} \qquad \mathcal{D}^{\mathcal{A}}_{s'} = \mathbb{T}_{\mathcal{D}^{\mathcal{A}}_{s'_1} \dots \mathcal{D}^{\mathcal{A}}_{s'_n}},$$

we extend the type conversion operation with the rule

$$\mathsf{convert}^{\mathcal{A}}_{s,s'}(\langle (d)^n_1 \rangle) \quad = \quad \begin{cases} \mathsf{cons}^{\mathcal{A}}_{s'}((d')^n_1) & \text{if } d'_k = \mathsf{convert}^{\mathcal{A}}_{s_k,s'_k}(d_k) \neq \epsilon \\ \epsilon & \text{otherwise,} \end{cases}$$

quantifying the condition for all $1 \leq k \leq n$.

Similarly, for array sorts $s, s' \in \mathcal{S}_\beta$ with the index sort $s_x \in \mathcal{S}_\beta$ and the item sorts $s_e, s'_e \in \mathcal{S}_\beta$ and with the carriers such that

$$\mathcal{D}^{\mathcal{A}}_s = \mathbb{A}_{\mathcal{D}^{\mathcal{A}}_{s_x}, \mathcal{D}^{\mathcal{A}}_{s_e}} \qquad \text{and} \qquad \mathcal{D}^{\mathcal{A}}_{s'} = \mathbb{A}_{\mathcal{D}^{\mathcal{A}}_{s_x}, \mathcal{D}^{\mathcal{A}}_{s'_e}} \qquad \text{and} \qquad \left|\mathcal{D}^{\mathcal{A}}_{s_x}\right| = n,$$

we define

$$\mathsf{convert}^{\mathcal{A}}_{s,s'}(\langle (d)^n_1 \rangle) \quad = \quad \begin{cases} \mathsf{cons}^{\mathcal{A}}_{s'}((d')^n_1) & \text{if } d'_k = \mathsf{convert}^{\mathcal{A}}_{s_e,s'_e}(d_k) \neq \epsilon \\ \epsilon & \text{otherwise.} \end{cases}$$

When $\mathcal{D}^{\mathcal{A}}_s$ and $\mathcal{D}^{\mathcal{A}}_{s'}$ are one of the simple types $\mathbb{B}$, $\mathbb{K}$, $\mathbb{I}$ or $\mathbb{E}_N$ defined in Section 5.2, conversions are based on numeric interpretations of the values. Let $s'' \in \mathcal{S}_\beta$ with $\mathcal{D}^{\mathcal{A}}_{s''} = \mathbb{I}$.

$$\begin{aligned}
\mathsf{int}_s &\in \mathcal{F}_{s,s''} \\
\mathsf{int}^{\mathcal{A}}_s(d) &= \begin{cases} 0 & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{B} \text{ and } d = \bot \\ 1 & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{B} \text{ and } d = \top \\ c(d) & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{K} \\ d & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{I} \\ n_i & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{E}_N \text{ and } d = \langle i, n_i \rangle \\ \epsilon & \text{otherwise} \end{cases} \\
\mathsf{ext}^{\mathcal{A}}_s &\in \mathcal{F}_{s'',s} \\
\mathsf{ext}^{\mathcal{A}}_s(d) &= \begin{cases} \bot & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{B} \text{ and } d = 0 \\ \top & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{B} \text{ and } d = 1 \\ d' & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{K} \text{ and } d = c(d') \\ d & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{I} \\ \langle i, n_i \rangle & \text{if } \mathcal{D}^{\mathcal{A}}_s = \mathbb{E}_N \text{ and } \exists i \in N : d = n_i \\ \epsilon & \text{otherwise} \end{cases}
\end{aligned}$$

The implementation in [38] combines the two operations as

$$\mathsf{ext}^{\mathcal{A}}_s(\mathsf{int}^{\mathcal{A}}_{s'}) : \mathcal{D}^{\mathcal{A}}_{s'} \to \check{\mathcal{D}}^{\mathcal{A}}_s$$

and provides uniform syntax for all type conversion operations.

6.4 OPERATIONS ON MULTI-SET SORTS

All operations introduced so far have basic sorts as their carriers. As the arc expressions in Algebraic System Nets are sorted over sets of multi-sets, we will need to define some operations with multi-set sorts as their carriers.

6.4.1 Multi-Set Constructor

For $s, s' \in \mathcal{S}_\beta$ with the carriers $\mathcal{D}_s^\mathcal{A}$ and $\mathcal{D}_{s'}^\mathcal{A} = \mathbb{I}$, respectively, the multi-set constructor

$$\begin{aligned} \mathsf{mset} &\in \mathcal{F}_{s,s',\mu(s)} \\ \mathsf{mset}^\mathcal{A}(d,n) &= \begin{cases} \mathcal{D}_s^\mathcal{A} \times \{0\} \setminus \{\langle d, 0\rangle\} \cup \{\langle d, n\rangle\} & \text{if } d \neq \epsilon \text{ and } n \geq 0 \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

constructs a multi-set containing some number of a single element.

6.4.2 Empty Multi-Set

For $s \in \mathcal{S}_\beta$ with a carrier $\mathcal{D}_s^\mathcal{A}$, the empty multi-set over s is defined as

$$\begin{aligned} \mathsf{empty}_s &\in \mathcal{F}_{\lambda,\mu(s)} \\ \mathsf{empty}_s^\mathcal{A}() &= \mathcal{D}_s^\mathcal{A} \times \{0\}. \end{aligned}$$

6.4.3 Multi-Set Sum and Filter

For $s, s', b \in \mathcal{S}_\beta$ with the respective carriers $\mathcal{D}_s^\mathcal{A}$, $\mathcal{D}_{s'}^\mathcal{A}$ and $\mathcal{D}_b^\mathcal{A} = \mathbb{B} = \{\bot, \top\}$, and for the terms $T \in \mathbf{T}_b^{\mathbf{S}_\mu}(\mathcal{V})$ and $U \in \mathbf{T}_{\mu(s')}^{\mathbf{S}_\mu}(\mathcal{V})$ where $\mathcal{V} = \mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_n} \cup \mathcal{V}_s$ for some $n \geq 0$ such that $x \in \mathcal{V}_s$, $x_i \in \mathcal{V}_{s_i}$ for $s_i \in \mathcal{S}$ and $1 \leq i \leq n$, we define the following operations:

$$\begin{aligned} \mathsf{sum}_{x,T,U} &\in \mathcal{F}_{s_1,\ldots,s_n,\mu(s')} \\ \mathsf{filter}_{x,T} &\in \mathcal{F}_{\mu(s),s_1,\ldots,s_n,\mu(s)} \\ \mathsf{sum}_{x,T,U}^\mathcal{A}(d_1,\ldots,d_n) &= \begin{cases} \epsilon & \text{if } g(V_{\mathcal{D}_s^\mathcal{A}}^\mathcal{A}(\mathcal{V}), T) \\ \epsilon & \text{if } g(V_{\mathcal{D}_s^\mathcal{A}}^\mathcal{A}(\mathcal{V}), U) \\ \mathsf{empty}_{s'}^\mathcal{A}() & \text{if } V_{\mathcal{D}_s^\mathcal{A},T}^\mathcal{A}(\mathcal{V}) = \emptyset \\ \displaystyle\sum_{v \in V_{\mathcal{D}_s^\mathcal{A},T}^\mathcal{A}(\mathcal{V})} e_v^\mathcal{A}(U) & \text{otherwise} \end{cases} \\ \mathsf{filter}_{x,T}^\mathcal{A}(\mu, d_1,\ldots,d_n) &= \begin{cases} \epsilon & \text{if } g(V_\mu^\mathcal{A}(\mathcal{V}), T) \\ \mathsf{empty}_s^\mathcal{A}() & \text{if } V_\mu^\mathcal{A}(\mathcal{V}) = \emptyset \\ \displaystyle\sum_{v \in V_\mu^\mathcal{A}(\mathcal{V})} \mathsf{mset}^\mathcal{A}\Big(v(x), \mu_v(v(x), T)\Big) & \text{otherwise} \end{cases} \end{aligned}$$

where the summations are multi-set unions as in Definition 4.10 and

$$\begin{aligned} V_{\mathcal{D}_s^\mathcal{A}}^\mathcal{A}(\mathcal{V}) &= \bigcup_{d \in \mathcal{D}_s^\mathcal{A}} \Big\{ \{\langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \langle x, d\rangle\} \Big\} \\ V_{\mathcal{D}_s^\mathcal{A},T}^\mathcal{A}(\mathcal{V}) &= \Big\{ v \in V_{\mathcal{D}_s^\mathcal{A}}^\mathcal{A}(\mathcal{V}) \;\|\; e_v^\mathcal{A}(T) = \top \Big\} \\ V_\mu^\mathcal{A}(\mathcal{V}) &= \bigcup_{d \in \mu} \Big\{ \{\langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \langle x, d\rangle\} \Big\} \\ g(V, T) &\equiv \exists v \in V : e_v^\mathcal{A}(T) = \epsilon \\ \mu_v(d, T) &= \begin{cases} \mu(d) & \text{if } e_v^\mathcal{A}(T) = \top \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The sum and the filter operations overlap in functionality. In typical models, the multi-set sum is used mostly for initial markings, whereas the filter is applied in temporal logic formulae and on output arcs.

6.4.4 Multi-Set Transformations

Item Mapping

For $s \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}_s^{\mathcal{A}}$, for $s' \in \mathcal{S}_\mu$ and for the term $T \in \mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ with $\mathcal{V} = \mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_n} \cup \mathcal{V}_s$ for some $n \geq 0$ such that $x \in \mathcal{V}_s$, $x_i \in \mathcal{V}_{s_i}$ for $s_i \in \mathcal{S}$ and $1 \leq i \leq n$, we define the multi-set item mapping operation

$$\begin{aligned}
\mathsf{map}_{x,T} &\in \mathcal{F}_{s',s_1,\ldots,s_n,\mu(s)} \\
\mathsf{map}^{\mathcal{A}}_{x,T}(\mu, d_1, \ldots, d_n) &= \begin{cases} \epsilon & \text{if } g(V^{\mathcal{A}}_\mu(\mathcal{V}), T) \\ \epsilon & \text{if } \exists d \in \mu : \mu(d) \notin \mathbb{I} \\ \mathsf{empty}^{\mathcal{A}}_s() & \text{if } V^{\mathcal{A}}_\mu(\mathcal{V}) = \emptyset \\ \displaystyle\sum_{v \in V^{\mathcal{A}}_\mu(\mathcal{V})} \mathsf{mset}^{\mathcal{A}}\big(e^{\mathcal{A}}_v(T), \mu(v(x))\big) & \text{otherwise} \end{cases}
\end{aligned}$$

where the summation is a multi-set union as in Definition 4.10 and

$$\begin{aligned}
V^{\mathcal{A}}_\mu(\mathcal{V}) &= \bigcup_{d \in \mu} \Big\{ \{\langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \langle x, d\rangle\} \Big\} \\
g(V, T) &\equiv \exists v \in V : e^{\mathcal{A}}_v(T) = \epsilon.
\end{aligned}$$

Multiplicity Mapping

For $s, s' \in \mathcal{S}_\beta$ with the carriers $\mathcal{D}_s^{\mathcal{A}}$ and $\mathcal{D}_{s'}^{\mathcal{A}} = \mathbb{I}$ and for the term $T \in \mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ with $\mathcal{V} = \mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_n} \cup \mathcal{V}_s \cup \mathcal{V}_{s'}$ for some $n \geq 0$ such that $x \in \mathcal{V}_s$, $y \in \mathcal{V}_{s'}$ and $x_i \in \mathcal{V}_{s_i}$ for $s_i \in \mathcal{S}$ and $1 \leq i \leq n$, we define the multi-set multiplicity mapping operation

$$\begin{aligned}
\mathsf{mmap}_{x,y,T} &\in \mathcal{F}_{\mu(s),s_1,\ldots,s_n,\mu(s)} \\
\mathsf{mmap}^{\mathcal{A}}_{x,y,T}(\mu, d_1, \ldots, d_n) &= \begin{cases} \epsilon & \text{if } h(V^{\mathcal{A}}_\mu(\mathcal{V}), T) \\ \epsilon & \text{if } \exists d \in \mu\colon \mu(d) \notin \mathbb{I} \\ \mathsf{empty}^{\mathcal{A}}_s() & \text{if } V^{\mathcal{A}}_\mu(\mathcal{V}) = \emptyset \\ \displaystyle\sum_{v \in V^{\mathcal{A}}_\mu(\mathcal{V})} \mathsf{mset}^{\mathcal{A}}\big(v(x), e^{\mathcal{A}}_v(T)\big) & \text{otherwise} \end{cases}
\end{aligned}$$

where the summation is a multi-set union as in Definition 4.10 and

$$\begin{aligned}
V^{\mathcal{A}}_\mu(\mathcal{V}) &= \bigcup_{d \in \mu} \Big\{ \{\langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \langle x, d\rangle, \langle y, \mu(d)\rangle\} \Big\} \\
h(V, T) &\equiv \exists v \in V : e^{\mathcal{A}}_v(T) = \epsilon \vee e^{\mathcal{A}}_v(T) < 0.
\end{aligned}$$

6.4.5 Union and Intersection

For $s \in \mathcal{S}_\beta$, we define the following multi-set operations:

$$\begin{aligned}
\mathsf{union}_s &\in \mathcal{F}_{\mu(s),\mu(s),\mu(s)} \\
\mathsf{minus}_s &\in \mathcal{F}_{\mu(s),\mu(s),\mu(s)} \\
\mathsf{union}^{\mathcal{A}}_s(\mu_1, \mu_2) &= \mu_1 + \mu_2 \\
\mathsf{minus}^{\mathcal{A}}_s(\mu_1, \mu_2) &= \mu_1 - \mu_2
\end{aligned}$$

6.4.6 Scalar Multiplication

With the operations defined so far, it is possible to construct all the multi-sets needed in practice. The following operation will not add any expressive power to the set of defined operations. Multi-set-valued terms can be shortened by making use of this operation, just like scalar expressions can be shortened by making use of grouping, e.g.: $a \cdot b + a \cdot c = a \cdot (b + c)$.

For $s \in \mathcal{S}_\mu$ and $s' \in \mathcal{S}_\beta$ with the carrier $\mathcal{D}^{\mathcal{A}}_{s'} = \mathbb{I}$, we define the scalar multiplication of a multi-set by an integer as follows:

$$\begin{aligned} \mathsf{mul}_s &\in \mathcal{F}_{s,s',s} \\ \mathsf{mul}^{\mathcal{A}}_s(\mu, d) &= \begin{cases} d \cdot \mu & \text{if } d \geq 0 \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

6.4.7 Comparison

For $s \in \mathcal{S}_\mu$ and $b \in \mathcal{S}_\beta$ with the carriers $\mathcal{D}^{\mathcal{A}}_s$ and $\mathcal{D}^{\mathcal{A}}_b = \mathbb{B} = \{\bot, \top\}$, we define the following binary relations $\mathcal{D}^{\mathcal{A}}_s \times \mathcal{D}^{\mathcal{A}}_s \to \mathbb{B}$ for testing multi-set equality and containment:

$$\begin{aligned} \mathsf{equalset}_s &\in \mathcal{F}_{s,s,b} \\ \mathsf{subset}_s &\in \mathcal{F}_{s,s,b} \\ \mathsf{equalset}^{\mathcal{A}}_s(\mu_1, \mu_2) &= \begin{cases} \top & \text{if } \mu_1 = \mu_2 \\ \bot & \text{otherwise} \end{cases} \\ \mathsf{subset}^{\mathcal{A}}_s(\mu_1, \mu_2) &= \begin{cases} \top & \text{if } \mu_1 \leq \mu_2 \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

6.4.8 Minimum and Maximum Multiplicity and Cardinality

For $s, s' \in \mathcal{S}_\beta$ with the carriers $\mathcal{D}^{\mathcal{A}}_s$ and $\mathcal{D}^{\mathcal{A}}_{s'} = \mathbb{I}$, we define the following operations:

$$\begin{aligned} \mathsf{min}_s &\in \mathcal{F}_{\mu(s),s'} \\ \mathsf{max}_s &\in \mathcal{F}_{\mu(s),s'} \\ \mathsf{card}_s &\in \mathcal{F}_{\mu(s),s'} \\ \mathsf{min}^{\mathcal{A}}_s(\mu) &= \min(\{k \parallel \langle d, k\rangle \in \mu \wedge k > 0\} \cup \max \mathbb{I}) \\ \mathsf{max}^{\mathcal{A}}_s(\mu) &= \max \{k \parallel \langle d, k\rangle \in \mu\} \\ \mathsf{card}^{\mathcal{A}}_s(\mu) &= \begin{cases} |\mu| & \text{if } |\mu| \in \mathbb{I} \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

6.5 SHORT-CIRCUIT OPERATIONS

Some programming languages make use of a technique called *short-circuit evaluation*. Under some circumstances, it suffices to evaluate only part of an expression. The built-in binary operators for logical conjunction && and disjunction || and the ternary if-then-else operator ?: of the programming languages C [24] and C++ [23] are a good example. The binary operators can be represented with the ternary operator:

$$\begin{aligned} a \,\&\&\, b &= a \,?\, b : 0 \\ a \,\|\, b &= a \,?\, 1 : b \end{aligned}$$

When the ternary operation is evaluated, its leftmost operand will be evaluated first. If it evaluates to zero or false, the result of the ternary operation will be the evaluation of its third operand; otherwise the outcome will be the evaluation of the second operand.

In [38], there are short-circuit versions of the logical conjunction and disjunction operators and a generalised $(n+1)$-ary variant of the selection operator ?:, which takes a left-hand-side term with a carrier of n elements, and n right-hand-side terms separated by colons, one of which will be selected for evaluation based on the outcome of the evaluation of the left-hand-side argument. The selected right-hand-side term will determine the outcome of the operation, as defined in Definition 4.7. We shall define only the selection operation here, since short-circuit disjunctions and conjunctions can be directly represented with it.

For $s' \in \mathcal{S}_\beta$ and $s \in \mathcal{S}$ with the carriers $\mathcal{D}^{\mathcal{A}}_{s'}$ and $\mathcal{D}^{\mathcal{A}}_{s}$ and $n = \left|\mathcal{D}^{\mathcal{A}}_{s'}\right|$, we define

$$\mathsf{select}_{s',s^n} \in \mathcal{G}_{s',s^n}$$

where s^n denotes the sequence $\underbrace{s, \ldots, s}_{n \text{ times}}$.

7 IMPLEMENTING THE ANALYSER

There are some implementation details we would like to point out, since they affect the exact semantics of the class of algebraic system nets we have implemented in [38]. Moreover, the way how an algorithm is implemented can be important or even crucial in practice, even if the difference is just an n-fold increase in execution speed or decrease in memory usage for some constant n. We will discuss some optimisations that have turned out to be extremely useful in practice.

7.1 TRANSITION INSTANCE ANALYSIS

Definition 4.17, the transition pre-enabling rule, does not mention how the assignments $v \in V^{\mathcal{A}}(\mathcal{V})$ that pre-enable a transition $t \in \mathcal{T}$ in a marking M can be found. A naïve approach would be to enumerate through all assignments, without paying any attention to the marking M. Doing so is possible when all input variables of a transition have a finite carrier, but such an algorithm would have exponential time complexity. Nevertheless, this is the usual way when a net is unfolded; see e.g. [32, Definition 13]. This approach does not work very well if the transitions have a large (or infinite) number of possible assignments (firing modes), and the transitions are enabled in only a few firing modes in the reachable states.

Fortunately, there is a more efficient approach for the case when the input places of a transition are marked sparsely. The process of finding assignments or substitutions under which two algebraic terms are equivalent is often referred to as *unification*, e.g. [3, pp. 74–76]. In algebraic system nets, we can unify input arc inscriptions with a marking of the net. In this case, a unifier is an assignment for the transition variables under which the evaluations of the input arc inscriptions are contained in the corresponding input place markings.

If the algebraic operations are not restricted, there might be prohibitively many unifiers. For instance, consider the constant $2 \in \mathbb{I}$ and the expression $x+y$. If the variables x and y are known to be non-negative integers, then three assignments are possible unifiers: $\{\langle x, 0\rangle, \langle y, 2\rangle\}$, $\{\langle x, 1\rangle, \langle y, 1\rangle\}$, and $\{\langle x, 2\rangle, \langle y, 0\rangle\}$. If the constant was n, there would be $n + 1$ different unifiers. If either variable was allowed to be negative, there would be infinitely many unifiers.

In order to avoid a combinatorial explosion, we have to restrict the set of algebraic terms that the unification algorithm examines to find values for variables. A natural way of making this restriction is to limit the set of operations the unification algorithm recognises in such a way that the choice of unifiers is always unique. This rules out the operation $+$ in our previous example.

In [39], we distinguish two classes of operations that are recognised by our algorithm. Reversible unary operations, such as taking the successor of an element in a sequence, can be “neutralised” by applying a reverse operation, such as the predecessor operator. Other operations that the algorithm must know are constructors that tie terms together. For instance, we want to be able to unify the variables in the term $\mathtt{cons}_s(x, y)$ with the constant $\langle 1, 2\rangle$.

We will give some definitions before we shed some light on the algorithm implemented in [38]. In some pathological cases, the algorithm reports an error as it fails to find values for the variables needed for evaluating the transition expressions.

Our algorithm makes use of *unifier candidates*. Given an algebraic term and a value the term should evaluate to, the definition yields a set of variable bindings under which the algebraic term could be *compatible* with the value, i.e. evaluate either to the value or fail to be evaluated because of an undefined variable.

Our definition of unifier candidates reflects our implementation [38]. Although the definition could cover all reversible algebraic operations—operations computable with injective mappings—it only covers variables and constructors for structured data types, defined in Section 6.3.5. For instance, the successor and predecessor operations defined in Section 6.3.2 are outside its scope.[1]

Definition 7.1 (Unifier candidate) *Let $\mathbf{S}_\mu = \langle \mathcal{S}_\beta \cup \mathcal{S}_\mu, \mathcal{F}, \mathcal{G}, \mu\rangle$ be a multiset signature and $\mathcal{A}$ be an $\mathbf{S}_\mu$-algebra. Let $T \in \mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ be a term. A variable $x \in \mathcal{V}$ is said to be* unifiable from T, *denoted $x \triangleleft T$, if*

1. *$T = x$, or*
2. *for some $n \geq 0$ and $s \in \mathcal{S}_\beta$, $T = \mathtt{cons}_s(T_1, \ldots, T_n)$ and for some $k \in \{1, \ldots, n\}$, $x \triangleleft T'_k$.*

Furthermore, let $v \in V^{\mathcal{A}}(\mathcal{V})$ be a valuation and $T' \in \mathbf{T}_s^{\mathbf{S}_\mu}$ a ground term, and let $x \triangleleft T$. A unifier candidate *$x \triangleleft_{T'} T$ is inductively defined as follows:*

1. *T', if $T = x$*
2. *$x \triangleleft_{T'_k} T_k$, if for some $n \geq 0$ and $s \in \mathcal{S}_\beta$, $T = \mathtt{cons}_s(T_1, \ldots, T_n)$, $T' = \mathtt{cons}_s(T'_1, \ldots, T'_n)$, and for some $k \in \{1, \ldots, n\}$, $x \triangleleft T_k$, and there is no $1 \leq j < k$ such that $x \triangleleft T_j$.*[2]

Finally, let $d = e^{\mathcal{A}}_{v_\emptyset}(T')$ be the value a ground term evaluates to. We use the short-hand notation $x \triangleleft_d T$ for the unifier candidate $x \triangleleft_{T'} T$.

Definition 7.2 (Term Compatibility) *Let $\mathcal{A}$ be an algebra with the multiset signature $\mathbf{S}_\mu = \langle \mathcal{S}_\beta \cup \mathcal{S}_\mu, \mathcal{F}, \mathcal{G}, \mu\rangle$. Let $T \in \mathbf{T}^{\mathbf{S}_\mu}(\mathcal{V})$ be a term, $T' \in \mathbf{T}^{\mathbf{S}_\mu}$ a ground term and $v \in V^{\mathcal{A}}(\mathcal{V})$ a valuation. The terms are* compatible under *v, denoted*

$$T \sim_v T',$$

if either

[1] This choice not only benefits the programmer but also persons studying a model, who do not need to know which operations are considered injective or reversible.

[2] Requiring the smallest k to be chosen ensures that unifier candidates are unique.

1. $e_v^{\mathcal{A}}(T) = e_{v_\emptyset}^{\mathcal{A}}(T')$ or
2. for some $n \geq 0$ and $s \in \mathcal{S}_\beta$, the terms T and T' are of the form $\mathsf{cons}_s(T_1, \ldots, T_n)$ and $\mathsf{cons}_s(T'_1, \ldots, T'_n)$, respectively, and for each k between 1 and n, either $T_k \sim_v T'_k$ or $e_v^{\mathcal{A}}(T_k) = \epsilon$.

Let $d = e_{v_\emptyset}^{\mathcal{A}}(T')$ be the value a ground term evaluates to. We use the shorthand notation $T \sim_v d$ for the compatibility check $T \sim_v T'$.

We have not presented any algorithms yet, but we are about to face a somewhat philosophical question. Should a unification algorithm be able to find all possible assignments that enable the transition, or does it suffice for the algorithm to deal with real models, and report errors for cases it cannot handle? An implementation that restricts the sets of supported operations is likely to be more efficient and less prone to errors than one that tries to handle everything. For instance, when making basic arithmetic operations reversible, one must take care of arithmetic precision and exceptional situations.

Variables that are not unifiable by Definitions 7.1 and 7.2 could be handled by nondeterministically picking values for them from their domains and by checking the terms for compatibility, but doing so is computationally expensive if the domains are large or there are many such variables. It is easier to report "variables cannot be unified" even though it might be possible to unify them. According to our experience with practical models, this works pretty well. One can always gain expressive power by replacing problematic terms with new variables and guards.

7.1.1 Splitting the Arcs

In typical models, arc expressions consist of elementary multisets (created with the **mset** operation) combined with multiset summation (union_s). In order to improve the granularity of our algorithm, we write each input arc inscription as a such combination of terms. Sanders refers to this as *arc unfolding* [50, Section 3].

The claim "any arc with a non-elementary multi-set may be 'unfolded' into multiple arcs" by Sanders [50, Section 3] is difficult to fulfil if the arcs contain multiset-valued variables or other multiset operations than the two we defined above. Our approach does not restrict the set of multiset operations, since we do not require that all split arc inscriptions be elementary multisets.

We distinguish three kinds of split arc inscriptions: ones that contain unifiable variables, ones that can be evaluated under a partial assignment incrementally constructed by our algorithm, and others. What matters is that whenever the unification algorithm finds a complete assignment, all arc inscriptions are compatible under it with the constants corresponding to the given marking of the model.

7.1.2 The Unification Algorithm

Our unification algorithm performs a depth-first search on the input arc inscriptions of the transition, split as described earlier. The algorithm is remarkably simple, since it processes the arcs in a fixed order produced in static analysis. Static analysis also determines which variables are unified from which arcs, and verifies that all variables can be unified.

However, when variable-multiplicity arcs are present, the algorithm cannot guarantee that all variables can always be unified. A variable unified from a variable-multiplicity arc may remain undefined if the multiplicity of the arc evaluates to zero. Even this does not prevent a transition from being enabled, if the variable is never evaluated due to short-circuit evaluation or arc multiplicities that evaluate to zero.

The input arc inscriptions $i(\langle p, t\rangle)$ of a transition t are split into items $S_k = \langle T_k, \mathcal{V}_k, p_k, m_k\rangle \in \mathbf{T}^{\mathbf{S}_\mu}(\mathcal{V}) \times 2^{\mathcal{V}} \times \mathcal{P} \times \mathcal{D}^{\mathcal{A}}$, $k \in \{1, \ldots, n\}$ for some n, such that

- the variable sets $\mathcal{V}_k$ are pairwise disjoint: $\mathcal{V}_j \cap \mathcal{V}_k = \emptyset$ if $j \neq k$
- no T_k refers to a variable outside $\bigcup_{j=1}^{n} \mathcal{V}_j$: $T_k \in \mathbf{T}^{\mathbf{S}_\mu}(\bigcup_{j=1}^{n} \mathcal{V}_j)$
- if $\mathcal{V}_k = \emptyset$: $T_k \in \mathbf{T}^{\mathbf{S}_\mu}(\bigcup_{j=1}^{k-1} \mathcal{V}_j)$
- if $\mathcal{V}_k \neq \emptyset$: $T_k = \mathsf{mset}(T_k'', T_k')$ and $T_k' \in \mathbf{T}^{\mathbf{S}_\mu}(\bigcup_{j=1}^{k-1} \mathcal{V}_j)$ and $\forall x \in \mathcal{V}_k : x \triangleleft T_k''$
- the original input arc inscriptions $i(\langle p, t\rangle)$ can be obtained from the places p_k and split inscriptions T_k by combining terms T_k and T_l having equal places ($p_k = p_l$) via multiset summation (union_s)

The last component, m_k, is a place-holder for the multiset the term is supposed to evaluate to. Our algorithm does not refer to it before initialising it; for convenience, here we can assign it to the empty multiset.

Our unification algorithm is presented in Figure 7.1. The computation is initiated by invoking ANALYSE with the split input arc inscriptions S and their amount n and a marking $M \in \mathcal{M}_\Sigma$ of the net. The computation step of the depth-first search is divided into two alternatives: processing a "constant" arc (arc with no new bindable variables), and obtaining new variable bindings from an arc.

Despite its limitations, the algorithm works except in a few pathological cases, e.g. when a transition has only one input arc of the form

$$\mathsf{union}_s\,(\mathsf{mset}(x, y), \mathsf{mset}(y, x))$$

where $x, y \in \mathcal{V}_s$ and $\mathcal{D}_s^{\mathcal{A}} = \mathbb{I}$ (the integer type defined in Section 5.2.3). Since there are no assignment candidates for x and y, no symbolic tokens can be extracted from the input arc.

An Example

We shall illustrate the algorithm with a simple example, presented in Figure 7.2. For the sake of simplicity, place inscriptions are omitted from the example. Let us simulate the algorithm with a marking M having

$$\begin{aligned} M(\mathsf{A}) &= \{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,1\rangle, \langle 3,1\rangle\} \\ M(\mathsf{B}) &= \{\langle 0,3\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3,0\rangle\} \end{aligned}$$

Analyse arcs $S_1..S_n$ *w.r.t. marking* M
ANALYSE(S, n, M):
$v \leftarrow (\bigcup_{k=1}^{n} \mathcal{V}_k) \times \{\epsilon\}$
ARCS($S, 1, n, M, v$)

Analyse arcs $S_k..S_n$
ARCS(S, k, n, M, v):
$\triangleright\ S_k = \langle T_k, \mathcal{V}_k, p_k, m_k \rangle$
if $k = n + 1$ **then**
  print v
  return
if $\mathcal{V}_k = \emptyset$ **then**
  CONSTANT(S, k, n, M, v)
else
  VARIABLE(S, k, n, M, v)

Evaluate arc S_k
CONSTANT(S, k, n, M, v):
$\triangleright\ S_k = \langle T_k, \mathcal{V}_k, p_k, m_k \rangle$
$m_k \leftarrow e_v^{\mathcal{A}}(T_k)$
if $m_k = \epsilon$ **then**
  print "undefined arc", v, T_k
  return
if $M(p_k) \geq m_k$ **then**
  $M' \leftarrow M$
  $M'(p_k) \leftarrow M(p_k) - m_k$
  ARCS($S, k + 1, n, M, v$)

Analyse arc S_k*, augment* v
VARIABLE(S, k, n, M, v):
$\triangleright\ S_k = \langle T_k, \mathcal{V}_k, p_k, m_k \rangle$
$\triangleright\ T_k = \mathsf{mset}(T_k'', T_k')$
$c \leftarrow e_v^{\mathcal{A}}(T_k')$
if $c = \epsilon$ **then**
  print "undefined multiplicity", v, T_k
  return
if $c = 0$ **then**
  $m_k \leftarrow \emptyset$
  ARCS($S, k + 1, n, M, v$)
  return
for each $m : M(p_k) \geq \langle m, c \rangle$ **do**
  $m_k \leftarrow \langle m, c \rangle$
  $v' \leftarrow v$
  for each $x \in \mathcal{V}_k$ **do**
    $v'(x) \leftarrow (x \triangleleft_m T_k'')$
  if $\bigwedge_{j=1}^{k} T_j \sim_{v'} m_j$ **then**
    $M' \leftarrow M$
    $M'(p_k) \leftarrow M(p_k) - m_k$
    ARCS($S, k + 1, n, M', v'$)

Figure 7.1: The Unification Algorithm.

$$
\begin{aligned}
\Sigma &= \langle \mathcal{N}, \mathcal{A}, \mathcal{V}, i \rangle \\
\mathcal{N} &= \langle \mathcal{P}, \mathcal{T}, \mathcal{F} \rangle \\
s &\in \mathcal{S}_\beta \\
\mathcal{D}_s^{\mathcal{A}} &= \{0, 1, 2, 3\} \\
\mathcal{P}_{\mu(s)} &= \{\mathsf{A}, \mathsf{B}\} \\
\mathcal{P} &= \mathcal{P}_{\mu(s)} \\
\mathcal{T} &= \{\mathsf{T}\} \\
\mathcal{F} &= \{\langle \mathsf{A}, \mathsf{T} \rangle, \langle \mathsf{B}, \mathsf{T} \rangle\} \\
\mathcal{V} &= \mathcal{V}_s \\
\mathcal{V}_s &= \{x, y\} \\
i(\langle \mathsf{A}, \mathsf{T} \rangle) &= \mathsf{union}_s(\mathsf{mset}(x, \mathsf{constant}_1()), \mathsf{mset}(\mathsf{succ}(x), \mathsf{constant}_1())) \\
i(\langle \mathsf{B}, \mathsf{T} \rangle) &= \mathsf{mset}(y, \mathsf{pred}(x)) \\
i(\mathsf{T}) &= \mathsf{constant}_{\mathsf{T}}()
\end{aligned}
$$

Figure 7.2: An Example Model for Transition Instance Analysis

and the input arcs of T split as follows:

$$\begin{aligned}
S_1 &= \langle T_1, \mathcal{V}_1, p_1, m_1 \rangle = \langle \mathsf{mset}(x, \mathsf{constant}_1()), \{x\}, \mathsf{A}, \emptyset \rangle \\
S_2 &= \langle T_2, \mathcal{V}_2, p_2, m_2 \rangle = \langle \mathsf{mset}(\mathsf{succ}(x), \mathsf{constant}_1()), \emptyset, \mathsf{A}, \emptyset \rangle \\
S_3 &= \langle T_3, \mathcal{V}_3, p_3, m_3 \rangle = \langle \mathsf{mset}(y, \mathsf{pred}(x)), \emptyset, \mathsf{B}, \emptyset \rangle .
\end{aligned}$$

The call to ANALYSE makes all variables undefined in the assignment and invokes ARCS to analyse the first inscription. Since S_1 has a nonempty variable set $\{x\}$, the method VARIABLE is invoked in order to unify the variable x.

The multiplicity of the first inscription evaluates to $c = 1$. Thus the loop over m considers all items in $M(p_1 = \mathsf{A})$ having a nonzero multiplicity. Let us assume that it starts with $m = 1$. The assignment is augmented to $v'(x) = m$. The compatibility check passes, and the token $m_k = \langle m, 1 \rangle$ is subtracted from the marking. The method ARCS is invoked to analyse the next arc.

As the variable set of the second arc S_2 is empty, it is analysed with the method CONSTANT. The term evaluates to $m_k = \langle m + 1, 1 \rangle$. Since $M(p_2 = \mathsf{A})$ is a super-multiset of the evaluation, the analysis can proceed to the third arc inscription.

Since the third arc S_3 has a variable, it is analysed by VARIABLE. On this first round with $v'(x) = 1$, its multiplicity evaluates to zero. The method ARCS is invoked with $k = 4 > n$, and the search starts to backtrack. At this point, the valuation is $\{\langle x, 1 \rangle, \langle y, \epsilon \rangle\}$.

The search backtracks to the invocation of VARIABLE that is analysing S_1. Let us assume that the next candidate for m is 2. The compatibility check is passed, and also S_2 can be unified, since $M(\mathsf{A})$ contains both $\langle 2, 1 \rangle$ and $\langle 3, 1 \rangle$. Now the multiplicity of S_3 evaluates to $c = 1$, which leaves two choices for picking a token from the place B: one valued 1 or another with the value 2. Both turn out to be compatible with $M(\mathsf{B})$, and the valuations $\{\langle x, 2 \rangle, \langle y, 1 \rangle\}$ and $\{\langle x, 2 \rangle, \langle y, 2 \rangle\}$ are printed.

The algorithm backs up again to S_1, and the only choice left is $v'(x) = 3$. Again S_2 passes the compatibility check, and the multiplicity of S_3 evaluates to $c = 2$. $M(\mathsf{B})$ contains only one item with a big enough multiplicity, and we get one more assignment, $\{\langle x, 3 \rangle, \langle y, 2 \rangle\}$. To sum up, the set of assignments reported by the algorithm is

$$V = \{\{\langle x, 1 \rangle, \langle y, \epsilon \rangle\}, \{\langle x, 2 \rangle, \langle y, 0 \rangle\}, \{\langle x, 2 \rangle, \langle y, 1 \rangle\}, \{\langle x, 2 \rangle, \langle y, 2 \rangle\}\} .$$

This example shows that our unification algorithm does not necessarily provide values for all variables. If an output arc of the transition T needs to evaluate the variable y, the assignment $\{\langle x, 1 \rangle, \langle y, \epsilon \rangle\}$ that pre-enables T will not enable T.

Our unification algorithm could probably be optimised considerably. The pattern matching algorithm presented in [43] would be advantageous in situations where the marking M does not change much between successive invocations of the algorithm. This could be useful when generating the set of reachable states in a depth-first order.

7.1.3 Unfolding to Place/Transition Nets

Some classes of high-level Petri Nets can be mapped to Place/Transition Nets [47], which consist of low-level places and transitions connected via weighted arcs. In a Place/Transition Net, places may contain a number of black tokens, which are moved around by the firing of low-level transitions. The transformation to a lower-level formalism is motivated, since many reduction methods for reachability analysis are only defined for the low-level Place/Transition Nets.

High-level Petri Nets are usually *unfolded* to Place/Transition Nets by mapping each place $p \in \mathcal{P}_{\mu(s)}$ with $s \in \mathcal{S}_\beta$ to $\left|\mathcal{D}_s^{\mathcal{A}}\right|$ low-level places and each possible enabled transition instance $\langle t, \hat{v}, v\rangle$ with $t \in \mathcal{T}$ and $\hat{v}, v \in V^{\mathcal{A}}(\mathcal{V})$ and $e_{\hat{v}}^{\mathcal{A}}(i(t)) = \top$, to a low-level transition connected with the low-level places corresponding to the input and output effects $t_{\hat{v}}^-$ and t_v^+. See e.g. [32, Section 5.1] for an illustration of this approach.

It is worth noting that this kind of a transformation is independent of the initial marking and the set of reachable markings; the unfolded net may contain many places whose markings will never change or transitions that will never fire. A more efficient unfolding would construct the set of low-level places incrementally, starting from the low-level places that are non-empty in the initial marking, and adding necessary low-level places for each transition instance. Also, some data types, such as the buffer type, can be modelled with more efficient constructs in some special cases.

The algorithm described in the previous section can be fairly easily modified for unfolding. Instead of taking the speculative tokens from a given marking, the unfolding algorithm assumes an "infinite" marking.

7.2 THE EXPRESSION EVALUATOR

Implementing an expression evaluator, the computation of $e_v^{\mathcal{A}}(T)$, is a straightforward task, once the interfaces have been defined. In an interpreting approach, it is customary to implement the expression evaluator in a set of functions that recursively call each other. An object-oriented implementation language, such as our choice C++, allows the expression evaluator to be implemented in virtual methods, which makes it fairly easy to add new types of subexpressions, each of which corresponds to some operations defined in Chapter 6.

The principle of an expression evaluator is very simple: it can be implemented as a procedure that takes an expression (an algebraic term) and an assignment (values for variables), and evaluates the expression by calling itself recursively, and by using the evaluated values of the subexpressions for computing the value of the expression. The assignment is only needed for evaluating variable references.

7.2.1 Error Handling

What should happen when a subexpression is undefined, if for instance it contains a division by zero? One approach is to sweep the problem under the carpet by using special values for undefined subexpressions, such as the ϵ defined in our algebra, or the NaN (Not a Number) values used by many floating-point evaluators. Another way is to use an exception handling mechanism to abort evaluations when an error occurs.

Our implementation uses a special value (a null pointer) together with a more specific error reporting mechanism. The expression evaluator is implemented as a virtual method that takes one parameter, an object containing the assignment for the variables. When an error occurs, the expression evaluator will return a special value after passing the assignment object an error code and a reference to the failed expression. A diagnostic method in the assignment object displays the error code together with the complete assignment and the subexpression that caused the error. This is a very useful way of reporting errors, especially those that occur during transition instance analysis.

7.2.2 Optimisations

One cannot expect much performance from an interpreter-based expression evaluator. The performance can be improved by introducing optimisations. Some of them transform expressions to simpler equivalent ones, while others affect the expression evaluator itself.

Constant Folding

Evaluating a variable-free expression always yields the same result. If the expression is evaluated more than once, computing power will be wasted. Such expressions, called ground terms or constant expressions, can be evaluated by the expression parser and replaced with constants representing their values. This technique is called *constant folding*. [1]

Constant folding can also improve error checking. If there is an error in a constant expression, the error can be detected already in the parsing stage, and it is easy to associate a diagnostic message with the exact location of the error in the input. Would the error occur at a later stage, when evaluating the expression in a dynamic context, reporting it precisely would be more difficult. Also, if not all expressions will ever be evaluated, errors found during constant folding could go undetected.

Common Subexpression Elimination

Expressions can be viewed as trees, which are a special case of directed acyclic graphs. A well-known optimisation technique called common subexpression elimination [1] views the expression as a directed graph. Identical subexpressions are not represented by identical subtrees, but by multiple arcs leading to the same subgraph representing the subexpression. Instead of parsing an expression to a tree, it can be parsed to a directed acyclic graph. A simple example is illustrated in Figure 7.3.

Common subexpression elimination has many advantages, and a typical expression tends to have common subexpressions, at least variable references. One advantage of the graph representation of expressions is space efficiency. But only when this technique is combined with other optimisation techniques, performance issues will become evident. When

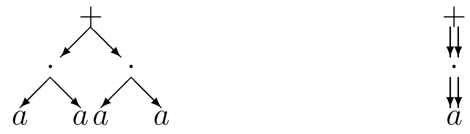


Figure 7.3: An Expression $a \cdot a + a \cdot a$ as a Tree and as a Graph

an expression is translated to machine code, each subexpression is typically assigned one (pseudo) register of the underlying machine. The fewer subexpressions there are, the less registers will be used and the less computations will be performed, and the easier it will be to schedule the use of the registers.

Cached Evaluation

During the transition instance analysis process described in Section 7.1, the same expressions will be evaluated several times, and typically only one variable will have changed its value between two successive evaluations of an expression.

One way to speed up the expression evaluator is to store the result of the last evaluation in a cache and return the cached value when the evaluator is called again. There is only one problem. When the value of a variable is changed in the assignment, the values of all expressions and subexpressions that depend on the value of the variable must be wiped out of the cache.

The overhead involved with managing the cache might over-weigh the advantage gained from using the cache. In our implementation, we experimented with a cache that could be disabled as a compile-time option. Using the cache only slightly improved the performance at the price of complicating the algorithms and maintaining two copies of expression evaluation code. We finally decided to replace the cache with an option that generates executable machine code.

Other Optimisations

It would be tempting to devise transformations for optimising the expression evaluator further. In a student project of a compiler construction course, the author implemented, among others, a transformation that was able to optimise an expression like $a + c + b + (b + a) \cdot 3$ to $(a + b) \cdot 4 + c$. Implementing and especially testing this kind of optimisations consumes much time, and the gain is often marginal, since the structures affected by such optimisations may seldom occur in practice.[3]

7.2.3 Interpreting vs. Compiling

A computer program that implements an abstract machine that executes computations expressed in some input language is called an *interpreter*. The interpreter itself may be executed by another interpreter, but at the

[3] This also means that thorough testing of such optimisation algorithms is essential: errors in them are not likely to be discovered with simple test cases.

lowest level there are some physical circuits that actually perform the computations.

Simulating an abstract machines involves an overhead. Each expression or instruction expressed in the input language must be translated to a sequence of computations on the underlying machine. If the same computations are repeated several times, maybe varying some parameters, the computations could be speeded up by translating the input language to a lower-level language that can be executed on a simpler and faster abstract machine, or directly on a physical machine.

Expanding the Multi-Set Sum

The multi-set sum operation, described in Section 6.4.3, provides a compact way of representing initial markings and modelling complex actions, such as broadcasting a message to a number of neighbours.

One of the reasons why this powerful operation is rare in reachability analysers are implementation difficulties. When a multi-set sum occurs in an initialisation expression, it can be evaluated in a straightforward way, since there are no unknown variables. Multi-set sums on cause much more headache when they occur on arc expressions, since both the summation condition and the summand may depend on unbound variables.

Originally, we implemented the multi-set sum as a genuine operation, and the sums were expanded during the transition instance analysis. This complicated the bookkeeping, since the set of symbolic tokens associated with a transition varied during the instance analysis. Furthermore, a multi-set sum whose summation condition depends on other variables than summation indices cannot be fully expanded at all times. In addition to maintaining a "processed" flag for each symbolic token handled by the instance analyser, we had to keep track on the multi-set sum expressions that were skipped due to an undefined summation condition.

To simplify and to speed up the instance analysis, we decided to expand the multi-set sums already in the parsing stage. Such a static transformation trades memory space for speed, since the algebraic term representing the symbolic expansion of the sum has to be kept in the memory all the time and not only when the sum actually needs to be evaluated.

When the static expansion takes place, the only variables whose values are available are the summation indices. The summation condition, which selects the index values for which the summand is evaluated, may depend on other variables. The condition can be translated to a scalar multiplication by zero or one.

Let there be the basic sorts $s, s', s'', b \in \mathcal{S}_\beta$ with the respective carriers $\mathcal{D}_s^\mathcal{A}$, $\mathcal{D}_{s'}^\mathcal{A}$, $\mathcal{D}_{s''}^\mathcal{A} = \mathbb{I}$ and $\mathcal{D}_b^\mathcal{A} = \mathbb{B} = \{\bot, \top\}$. Let the terms $T \in \mathbf{T}_b^{\mathbf{S}_\mu}(\mathcal{V})$ and $U \in \mathbf{T}_{\mu(s')}^{\mathbf{S}_\mu}(\mathcal{V})$ have $\mathcal{V}$ such that $x \in \mathcal{V}_s$. The multi-set sum term

$$\mathsf{sum}_{x,T,U}$$

is equivalent to a term containing a chain of applications of the multi-set union operator defined in Section 6.4.5, enumerating through all the values $d \in \mathcal{D}_s^\mathcal{A}$ and combining the terms

$$\mathsf{mul}_s(U_x^d, \mathsf{select}_{b,s'',s''}(T_x^d, \mathsf{constant}_0(), \mathsf{constant}_1()))$$

where T_x^d and U_x^d denote transformations of the terms T and U such that each occurrence of x has been replaced with the term $\mathsf{constant}_d()$.

Our implementation [38] folds constants while transforming the terms, and it omits the scalar multiplication or the whole term if the multiplicity is one or zero.

Compiling Expressions

The most obvious performance bottleneck in [38] is the expression evaluator, which makes heavy use of dynamically allocated objects. According to our work [36], the performance improves vastly when the expressions are not interpreted but translated to something that can be compiled to machine code.

The programming languages C [24] and C++ [23] are the most obvious choices for an intermediate language. Our implementation generates C, since it is simpler and often more efficient than C++. Each transition in the model is translated into a function that analyses and fires all its enabled instances in the specified marking. The code for evaluating expressions is embedded in the code. We also generate encoding and decoding functions for managing the reachability graph (see Section 7.3). These do not need to evaluate expressions.

7.3 MANAGING THE REACHABILITY GRAPH

The limited amount of system memory is a major bottleneck in exhaustive reachability analysis. Algorithms for reachability analysis and model checking need to keep track on the states that have been explored. In that way, they can detect cyclic behaviour and limit the investigation of successors to truly new states.

There are some techniques that only manage the set of reachable states and utilise similarities between the states. One of them, Binary Decision Diagrams [4, Chapter 5], has been successfully applied in the verification of digital circuits. Techniques applied on the analysis of software systems include a state compaction method for product automata [13] and a method known as Graph Encoded Tuple Sets [18].

One problem with these techniques is that inserting a state may involve global changes, slowing down disk-based implementations. Another problem is that states have no identities: there is no way to retrieve a state from the structure by specifying an index number. It is difficult to use such a structure for anything else than determining whether a particular state has been explored.

Since we want to be able to make all sorts of queries on the generated reachability graph, our approach represents each marking (state) separately and records also the transition instances that lead from one marking to another. Our approach encodes the markings in a compact way, and it saves system memory by maintaining the encoded markings and transition instances on disk.

The reachability graph storage implemented in [38, 37] builds heavily on two encoding routines. One inputs a number in a set $\{0, \ldots, n-1\}$

and appends a string of $\lceil \log_2 n \rceil$ binary digits to an encoding buffer. Another routine encodes a value $d \in \mathcal{D}^{\mathcal{A}}$ using around $\lceil \log_2 |\mathcal{D}^{\mathcal{A}}| \rceil$ binary digits.[4] This is similar to the approach represented in [10], but it was developed independently.

Our encoding is optimal if $n = 2^k$ for some integer $k \geq 0$ and if each item of $\mathcal{D}$ occurs with equal probability. More efficient encodings would be possible if the data to be encoded were known in advance. For a parameterised model, one could try to estimate probabilities for different values by analysing smaller state spaces generated with suitable parameters. We did not consider this option further.

7.3.1 Encoding Markings

The marking M of a net $\mathcal{N} = \langle \mathcal{P}, \mathcal{T}, \mathcal{F} \rangle$ is encoded by processing the places in a systematic order. For each place $p \in \mathcal{P}$, the local marking

$$\mu = M(p)$$

must be encoded. With the assumption that $\mu : \mathcal{D} \to \mathbb{N}$ maps most data items in $\mathcal{D}$ to zero, it makes sense to explicitly represent the data items with nonzero multiplicity. Obviously, each data item can be represented using $\lceil \log_2 |\mathcal{D}| \rceil$ binary digits. Therefore, we shall concentrate on the encoding of the multiplicities and the representation of empty places.

Representing Multiplicities

A multi-set μ can be characterised by two quantities: the total number of items

$$t = |\mu|$$

and the number of distinct items

$$d = |\{a \parallel \mu(a) > 0\}| \, .$$

The total number of items can theoretically be any natural number, but a finite-memory implementation imposes a limit on it, typically $0 \leq t < 2^n$ for some n.

An user-defined *capacity constraint* [38] can reduce the number of bits required for representing t. If there are m different possibilities for the total number of tokens in a place, the actual number t can be represented using $\lceil \log_2 m \rceil$ bits.

Encoding the total number of items t before the number of distinct items d has one advantage: it is straightforward to see that $1 \leq d \leq t$ when t is nonzero. Therefore, d can be represented using $\lceil \log_2 t \rceil$ bits. After this, the distinct items

$$\{a \parallel \mu(a) > 0\}$$

and their multiplicities are encoded in descending order of $\mu(a)$. Clearly

$$\left\lceil \frac{t}{d} \right\rceil \leq \mu(a) \leq 1 + t - d$$

[4] Due to an implementation choice, if $|\mathcal{D}^{\mathcal{A}}|$ cannot be represented in a machine word, the value will be encoded componentwise, which wastes a fraction of a binary digit for each component whose domain is not of size 2^k for some k.

holds for the greatest multiplicity $\mu(a)$. If $\mu(a) = 1 + t - d$, it holds that the other $d-1$ distinct items must have a multiplicity of 1 in order for the total number of items to be t. Similarly, if $\mu(a) = \lceil \frac{t}{d} \rceil$, the multiplicities of the remaining items must equal $\mu(a)$ or $\mu(a) - 1$.

So, the greatest multiplicity $\mu(a)$ can always be represented with

$$\left\lceil \log_2 \left(2 + t - d - \left\lceil \frac{t}{d} \right\rceil \right) \right\rceil$$

binary digits. After decoding $\mu(a)$, the decoder knows the remaining total cardinality $t' = t - \mu(a)$ and the number of remaining distinct items $d' = d - 1$. When the multiplicities are encoded in descending order, the encoder always selects the greatest of the remaining multiplicities and uses less and less bits.

This encoding of multiplicities appears to be quite compact even when capacity constraints are not used. A simpler encoding might represent the multiplicities using a fixed number of binary digits for the multiplicity of each distinct item. For instance, when the multiplicities $\mu(a)$ are limited in the range $0 \leq \mu(a) < 2^n$ for some n, such a simple coding would use up at least dn bits for encoding d multiplicities, not considering the bits needed for signalling the end of the encoded stream.

For representing $d = 5$ multiplicities, the simple encoding would use at least $5n$ bits. The optimised encoding needs n bits for representing the total cardinality. Assuming that it is 8, the number of distinct tokens is encoded in 3 bits. The greatest multiplicity lies between $\lceil \frac{8}{5} \rceil = 2$ and $8 - 5 + 1 = 4$; therefore it can be represented with 2 bits. Clearly, the improved encoding requires less than $n + 3 + 5 \cdot 2 = n + 13$ bits. The difference between $5n$ and $n+13$ is tangible in practical implementations, which typically use $n = 16$ or $n = 32$.

Representing Empty Places

In many practical models, there is a substantial number of empty places in most reachable markings. With our optimised multiplicity encoding, an empty place requires $\lceil \log_2 m \rceil$ bits of storage, if there are m different possibilities for the total number of tokens in the place.

As it is rather uncommon to define tight capacity constraints, representing the total cardinalities typically requires one machine word per place, which easily dominates the encoding of markings where most places are empty. With the assumption that many places are empty in each reachable marking, we use a simple bit vector for indicating empty places. The encoded cardinalities for those places that are marked empty can be omitted. As an optimisation, the bit vector is not used for places having a capacity constraint that can be represented in at most two bits or forbids a cardinality of zero.

As a further optimisation, we use a variable-length code for representing the total cardinalities of places with no capacity constraint. This code favors small cardinalities. Empty places are indicated with one bit, and places containing one to eight tokens with five bits. Codes for larger cardinalities are 11, 20 and $n + 4$ bits long where n is the length of the machine word.

Implicit Places

Some models contain *implicit places*, places whose local marking is a function of the local markings of other places. One might ask why such places occur in models, but they can make models more readable and ease the transition instance analysis. Nevertheless, the local markings of these places need not be encoded. Allowing the user to specify marking-dependent initialisation expressions would be an elegant way to point out implicit places.

This optimisation is likely to be implemented soon in [38]. It, together with the capacity constraint, slightly affects the formal semantics: when the firing of an enabled transition instance would violate a capacity constraint or be in contradiction with the initialisation expression of an "implicit place," the transition instance would not be fired, but it would be reported to be faulty.

Hashing

A hash value will be computed for the encoded marking. If the hash value does not exist in a *hash table*, a search structure that maps hash values to numbers of corresponding markings, the encoded marking will represent a new node in the reachability graph. In this case it will be assigned its own number and stored in the hash table as well as written to a disk file containing the distinct encoded markings generated so far.

If the hash value exists in the hash table, the encoded markings corresponding to it will be fetched from the disk file and compared against the newly encoded marking. If a match is found, the newly encoded marking will be assigned the number of the existing marking. Otherwise the marking will be written to the disk file and added to the hash table.

In order to save memory, we implemented the hash table as a disk-based B-tree. With a block size of 4096 bytes and a branching degree of 512, this structure requires 8 to 16 bytes per stored marking.

7.3.2 Encoding Transition Instances

The encoded reachability graph includes also transition instances, leading from one marking to another. Each transition instance consists of a high-level transition name and an assignment, mapping variables to values.

Each high-level transition $t \in \mathcal{T}$ is assigned a unique number between 0 and $|\mathcal{T}|-1$. The encoded instance consists of a pair of marking numbers (the source and the target marking), followed by the transition number encoded in $\lceil \log_2 |\mathcal{T}| \rceil$ binary digits and by the encoded assignment.

The assignment is encoded by processing the variables in a systematic order. One bit is used for signalling that a variable is undefined, if it is allowed to be undefined.[5] When a variable is defined, its encoded value will be appended to the bit vector representing the assignment.

No search structure for transition instances is needed while generating the reachability graph. The transition instances can be appended to a disk file in a linear manner.

[5] By default, all transition variables must have a value in all enabled transition instances. Undefined variables occur when the multiplicity of an input arc expression evaluates to zero.

Some resources could be saved by allowing the user to specify the variables whose values should be included in the encoded transition instance. The omitted values could be reconstructed on demand by running the instance analysis algorithm in the source marking of the event.

8 CONSTRUCTING AND ANALYSING MODELS

A formalism without any practical applications is a dead formalism. We will present two examples that illustrate some of the advantages of the algebraic operations presented in Chapter 6 over the set of operations conventionally defined in computer tools for high-level Petri Nets.

8.1 POINT-TO-MULTIPOINT COMMUNICATIONS

Many distributed algorithms designed for computer networks involve operations where a node sends messages to or waits for a message from each or some of its neighbours. When constructing parameterised models, e.g. with n nodes, the multi-set sum operator defined in Section 6.4.3 provides a compact way for writing arc expressions describing such situations.

Consider a system of n nodes connected via a broadcasting network. The nodes periodically need to synchronise with each other. This is accomplished by broadcasting messages. When a node is ready for synchronisation, it will send a message to all other nodes and wait for a message from all other nodes. In order to model this system, we need to define the neighbourhood relation

$$N = (U \times U) \setminus \{\langle d, d\rangle \parallel d \in U\}$$

with $|U| = n$.

Without using the multi-set sum operator, this relation, or its projection

$$N(u) = \{u' \parallel \langle u, u'\rangle \in N\}$$

can only be represented for some fixed value of n. Let us consider an example where $n = 4 = |U|$. Let $\mathcal{A}$ be an algebra with $\mathcal{D}_s^{\mathcal{A}} = U$ and $\mathcal{D}_{s'}^{\mathcal{A}} = \mathbb{T}_{U,U}$. Now

$$\begin{aligned} N(u) &= \left\{u' \,\middle\|\, \langle u', 1\rangle \in e^{\mathcal{A}}_{\{\langle x,u\rangle\}}(T_1)\right\} \\ T_1 &= \mathsf{union}_s(\mathsf{mset}(\mathsf{succ}(x), \mathsf{constant}_1()), \\ &\qquad \mathsf{mset}(\mathsf{succ}(\mathsf{succ}(x)), \mathsf{constant}_1()), \\ &\qquad \mathsf{mset}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(x))), \mathsf{constant}_1())). \end{aligned}$$

It is easy to see that in this kind of a construct, the length of the term T_1 presented above increases at least linearly with n. The multi-set sum operator allows for more compact notation, which does not depend on n, the number of items in the set U:

$$\begin{aligned} T_1 &= \mathsf{sum}_{y,T_2,T_3} \\ T_2 &= \mathsf{unequal}(x, y) \\ T_3 &= \mathsf{mset}(y, \mathsf{constant}_1()). \end{aligned}$$

Now we are ready to represent an Algebraic System Net model of the system, illustrated in Figures 8.1 and 8.2. The latter uses graphical notation, representing places (elements of P) with circles encircling sort identifiers, transitions (elements of T) with boxes, and the flow relation F with directed arcs connecting the images of places and transitions. Furthermore, the inscription function i is represented with textual inscriptions written next to the places, transitions and arcs. Trivial "null" inscriptions—empty place initialisers and constantly enabled transition guards—are omitted. We have also labelled the place and transition symbols with the names used in the purely textual representation of Figure 8.1.

The alert reader may notice that the place PENDING in Figure 8.1 is redundant. In all reachable markings of the net, the place BUS contains a set of tuples

$$\bigcup_{x \in A} \{\langle x, y \rangle \parallel y \in B \setminus \{x\}\}$$

where A and B are some subsets of $\mathcal{D}_s^{\mathcal{A}}$. Due to the structure of the net, the place PENDING will contain the set A corresponding to the contents of the place BUS in all reachable markings.

The redundant place PENDING is needed in the model to circumvent the limitations of the transition instance analysis algorithm described in Section 7.1.2. In our algebraic terms, if we allow initialisation expressions of implicit places to depend on the current marking of the net as suggested in Section 7.3.1, the local marking of the redundant place can be represented as follows:

$$\begin{aligned} i(\textsf{PENDING}) &= \textsf{mmap}_{x,y,T}(\textsf{map}_{z,U}(\textsf{BUS})) \\ T &= \textsf{constant}_1() \\ U &= \textsf{component}_{s',1}(z). \end{aligned}$$

Clearly also the place IDLE is a complement place of PENDING. In all reachable states, the union of the two local markings equals $\mathcal{D}_s^{\mathcal{A}}$. In other words,

$$\begin{aligned} i(\textsf{IDLE}) &= \textsf{minus}_s(\textsf{sum}_{x',T',U'}, \textsf{PENDING}) \\ T' &= \textsf{constant}_\top() \\ U' &= \textsf{mset}(x', \textsf{constant}_1()). \end{aligned}$$

It is not always clear whether a place is implicit. The theory for finding implicit or redundant places is called *invariant analysis*, but it is not with the scope of this work. However, it may be noted that our tool [38] is capable of determining whether certain place invariants, as defined in [32], hold in the reachable states of a model.

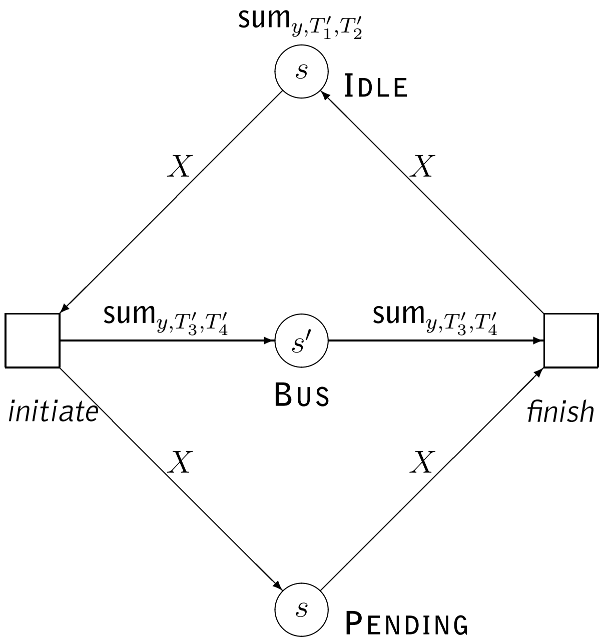
8.2 EXISTENTIAL QUANTIFICATION

In some models one would like to apply existential and universal quantification on algebraic terms. For example, minimisation problems, that is, problems of the form

$$\exists x : \forall y : f(x) \leq g(x, y),$$

$$
\begin{array}{rcl}
\Sigma & = & \langle \mathcal{N}, \mathcal{A}, \mathcal{V}, i \rangle \\
\mathcal{N} & = & \langle \mathcal{P}, \mathcal{T}, \mathcal{F} \rangle \\
s, s' & \in & \mathcal{S}_\beta \\
\mathcal{D}^{\mathcal{A}}_s & = & \{0, \ldots, n-1\} \\
\mathcal{D}^{\mathcal{A}}_{s'} & = & \mathbb{T}_{\mathcal{D}^{\mathcal{A}}_s, \mathcal{D}^{\mathcal{A}}_s} \\
\mathcal{P}_{\mu(s)} & = & \{\textsc{Idle}, \textsc{Pending}\} \\
\mathcal{P}_{\mu(s')} & = & \{\textsc{Bus}\} \\
\mathcal{P} & = & \mathcal{P}_{\mu(s)} \cup \mathcal{P}_{\mu(s')} \\
\mathcal{T} & = & \{\mathit{initiate}, \mathit{finish}\} \\
\mathcal{F} & = & \{f_1, f_2, f_3, f_4, f_5, f_6\} \\
f_1 & = & \langle \textsc{Idle}, \mathit{initiate} \rangle \\
f_2 & = & \langle \mathit{initiate}, \textsc{Pending} \rangle \\
f_3 & = & \langle \mathit{initiate}, \textsc{Bus} \rangle \\
f_4 & = & \langle \textsc{Pending}, \mathit{finish} \rangle \\
f_5 & = & \langle \textsc{Bus}, \mathit{finish} \rangle \\
f_6 & = & \langle \mathit{finish}, \textsc{Idle} \rangle \\
\mathcal{V} & = & \mathcal{V}_s \\
\mathcal{V}_s & = & \{x\} \\
i(\textsc{Idle}) & = & \mathsf{sum}_{y, T'_1, T'_2} \\
i(\textsc{Pending}) & = & \mathsf{empty}_s() \\
i(\textsc{Bus}) & = & \mathsf{empty}_{s'}() \\
i(\mathit{initiate}) & = & \mathsf{constant}_\top() \\
i(\mathit{finish}) & = & \mathsf{constant}_\top() \\
i(f_1) & = & \mathsf{mset}(x, \mathsf{constant}_1()) \\
i(f_2) & = & \mathsf{mset}(x, \mathsf{constant}_1()) \\
i(f_3) & = & \mathsf{sum}_{y, T'_3, T'_4} \\
i(f_4) & = & \mathsf{mset}(x, \mathsf{constant}_1()) \\
i(f_5) & = & \mathsf{sum}_{y, T'_3, T'_4} \\
i(f_6) & = & \mathsf{mset}(x, \mathsf{constant}_1()) \\
T'_1 & = & \mathsf{constant}_\top() \\
T'_2 & = & \mathsf{mset}(y, \mathsf{constant}_1()) \\
T'_3 & = & \mathsf{unequal}(x, y) \\
T'_4 & = & \mathsf{mset}(\mathsf{cons}_{s'}(x, y), \mathsf{constant}_1()).
\end{array}
$$

Figure 8.1: Algebraic System Net Model of a Synchronisation Protocol



$$
\begin{aligned}
s, s' &\in \mathcal{S}_\beta \\
\mathcal{D}^{\mathcal{A}}_s &= \{0, \ldots, n-1\} \\
\mathcal{D}^{\mathcal{A}}_{s'} &= \mathbb{T}_{\mathcal{D}^{\mathcal{A}}_s, \mathcal{D}^{\mathcal{A}}_s} \\
\mathcal{V} &= \mathcal{V}_s \\
\mathcal{V}_s &= \{x\} \\
X &= \mathsf{mset}(x, \mathsf{constant}_1()) \\
T_1' &= \mathsf{constant}_\top() \\
T_2' &= \mathsf{mset}(y, \mathsf{constant}_1()) \\
T_3' &= \mathsf{unequal}(x, y) \\
T_4' &= \mathsf{mset}(\mathsf{cons}_{s'}(x, y), \mathsf{constant}_1())
\end{aligned}
$$

Figure 8.2: Graphical Representation of the Model in Figure 8.1

need to be solved every now and then. One solution to this kind of problems is the introduction of an algebraic operation; another way to attack the problem is to introduce implicit places, e.g. a place that holds the current minimum.

Purely existential problems can be solved by counting. If we want to know whether there exists a token i in a place p such that some logical property $f(i)$ holds, we can introduce an implicit place that holds the number of such tokens i in the place p, in all reachable markings. A similar approach can be taken for problems involving purely universal quantification, since $\exists i : f(i)$ is equivalent to $\neg\forall i : \neg f(i)$.

An existential-universal quantification operation has been considered for implementation in [38], but it has not been implemented yet. One problem with such quantification is that if both quantification variables x and y belong to a data type with n items, the algorithm will need to test n^2 combinations in the worst case. On the other hand, performing the quantification in one atomic step is an order of magnitude more efficient than modelling it "manually" with a sequence of transitions, causing numerous uninteresting intermediate states to be generated.

8.3 THE PERFORMANCE OF EXHAUSTIVE ANALYSIS

We shall demonstrate the performance of our reachability analyser implementation with a small example. The distributed data base system model presented in Figure 8.3 has been translated into the input language of our tool from an example file distributed with PROD [57].

The model has one parameter, the number of service nodes. The original model for PROD, dbm.net, describes a system with ten nodes. Adding or removing nodes involves changes in several places of the model. The MARIA model refers to the number of service nodes only in the definition of the data type db_t. The multi-set summations used in the initialisation expression of the place INACTIVE and in some arc expressions expand according to the domain size of this data type.

8.3.1 The Size of the Encoded State Space

Table 8.1 illustrates the performance of our state encoding on a 32-bit computer system. We analysed the distributed data base model with PROD using the default option and an unfolding option, and with MARIA using three variants of the model: without and with capacity constraints for the places, and with capacity constraints and implicit places indicated. Unlike in the paper [37], this model does not contain the fully redundant place called UNUSED.

The figures exclude the space required for bookkeeping. In MARIA, the bookkeeping record is a table of file offsets, plus a b-tree of hash values and state numbers. On a 32-bit system with 32-bit file offsets, the file offsets occupy 8 bytes per encoded state while the b-tree takes 8 to 16 bytes. The low-level binary string routines in MARIA operate on machine words, but the encoded states are stored as sequences of bytes.

```
typedef unsigned (1..10) db_t;
typedef struct {
  db_t sender;
  db_t recipient;
} db_pair_t;

place INACTIVE (0..#db_t) db_t: db_t d: d;
place WAITING (0..1) db_t;
place PERFORMING (0..#db_t) db_t;
place EXCLUSION (0..1) struct {}: {};
place SENT (0..#db_t) db_pair_t;
place RECEIVED (0..#db_t) db_pair_t;
place ACKNOWLEDGED (0..#db_t) db_pair_t;

trans "update and send messages"
in { INACTIVE: s; EXCLUSION: {}; }
out { WAITING: s; SENT: db_t r (r != s): { s, r }; };
trans "receive acknowledgements"
in { WAITING: s; ACKNOWLEDGED: db_t r (r != s): { s, r }; }
out { INACTIVE: s; EXCLUSION: {}; };
trans "receive message"
in { INACTIVE: r; SENT: { s, r }; }
out { PERFORMING: r; RECEIVED: { s, r }; }
gate s != r;
trans "send acknowledgement"
in { PERFORMING: r; RECEIVED: { s, r }; }
out { INACTIVE: r; ACKNOWLEDGED: { s, r }; }
gate s != r;
```

Figure 8.3: MARIA Model of a Distributed Data Base System

Table 8.1: Encoded State Space Sizes for the Distributed Data Base Model

Model Size		Encoded State Space in Bytes			
$\|\mathcal{D}\|$	States	PROD	MARIA	(cap.)	(red.)
1	2	17	4	2	2
2	7	76	21	14	9
3	28	389	150	111	86
4	109	1,848	724	543	414
5	406	8,113	3,565	3,159	2,396
6	1,459	33,548	15,725	14,266	10,807
7	5,104	132,693	60,786	55,683	43,569
8	17,497	507,400	233,702	217,213	168,089
9	59,050	1,889,585	998,945	952,558	738,397
10	196,831	6,889,068	3,559,389	3,526,147	2,683,381

9 MODELLING COMPUTER PROGRAMS

One of the major goals of this research is to provide a mechanism for semi-automatic verification of concurrent computer software, such as implementations of communications protocols. Presenting a complete set of rules for translating concurrent programs into algebraic system nets is out of the scope of this work, but the data type system has been created with this transformation in mind. We begin by summarising the features of our data type system and comparing them with other approaches. Later on, we sketch the transformation rules for some typical constructs and make some comparisons to related work.

9.1 DATA TYPES

The rôle of powerful data types may not be ignored. While it is theoretically possible to represent all data by using integer numbers, it is very difficult to translate expressions (algebraic terms) manipulating structured data to equivalent expressions that work with the integer representations of the data objects. This has been tried in [28, 40], where problems were encountered e.g. with array indices. Another way to handle data types is to restrict the source language to allow only easily representable data types [17], severely restricting the usability of the verification tool.

Table 9.1 summarises the data types supported by some reachability analysers. A solid circle ($\bullet$) denotes that the analyser supports the data type, whereas a hollow circle ($\circ$) means that a feature is partially supported or there is a mapping to a native data type of the analyser, but not all constraints are enforced or operations are supported.

The four tools we compare to our reachability analyser for Algebraic System Nets, Maria [38], are Prod [57] for Predicate/Transition Nets [11], Design/CPN [41] for Coloured Petri Nets [26], and a protocol analyser Spin [20], which is based on its own formalism, concurrent processes communicating via message queues.

Of the four reachability analysers, Prod has the scarcest data type system. Its data model is based on tuples of nonnegative integer numbers, which may be assigned lower and upper limits. The constraints in Maria and Design/CPN are based on Boolean conditions that allow the definition of types with "holes", e.g. $\{i \in \mathbb{N} \parallel i \leq 42 \vee 64 \leq i \leq 128\}$.

There is not much variation among simple types. Design/CPN, whose algebraic model is based on the Standard Meta Language [42], has two simple data types that fundamentally differ from the integer-like types: unbounded strings of characters and floating-point numbers. Character strings can be mapped to fixed-length character arrays in Maria and Spin. Floating point arithmetics is not supported in the other reachability analysers, probably because it highly depends on the underlying computer system.

The analysers differ more in structured data types. All analysers support the tuple type. Prod does not support nested tuples, but this does

Table 9.1: Data Types Representable in Different Reachability Analysers

	Prod	Spin	Maria	Design/CPN
Constraints	○		●	●
Simple Data Types				
Boolean	○	●	●	●
Character	○	●	●	●
Integer	●	●	●	●
Enumerated	●	○	●	●
Identifier	○	○	●	○
Floating-Point Number				●
Character String		○	○	●
Structured Data Types				
Tuple	●	●	●	●
Array		●	●	●
Tagged Union	○	○	●	○
Variable-Length Buffer		○	●	●
Linked List				●

not limit the expressive power of its type system, since there are no other structured data types in Prod. Spin has fixed-length arrays indexed by integers, and the linked lists of Design/CPN can be constrained to be of a constant length. In Maria, arrays can be declared with any data type for indices, but one should keep in mind that an array with n different elements and m distinct indices has n^m possible values.

Only Maria supports the tagged union data type, which is necessary for modelling object-oriented constructs (Section 9.5) and for supporting the data type systems of some programming languages. Of the other analysers, Prod, whose data type system is limited to tuples of integers, can rather efficiently represent tagged unions of its natively supported data types. The mapping is straightforward: prepend each tuple with a tag element. In the other tools, one can form a tuple of an integer and all the data types belonging to the tagged union. The integer will indicate the active union component. This is utterly inefficient: while a tagged union of data items with domain sizes n_i has a domain of $\sum_{i=1}^{k} n_i$ elements, the domain size of the corresponding tuple mapping is $k \prod_{i=1}^{k} n_i$.

The linked list data type of Design/CPN, like the character string, has an unbounded domain. A list constrained to a maximum length is equivalent to the variable-length buffer of Maria. Spin has a separate construct for defining message queues outside the type system.

9.1.1 Expressive Power

Of the data type systems compared in Table 9.1, the data type system in Prod has the least expressive power. The Spin data type system appears to be a proper subset of the system implemented in Maria, and all data types of Maria can be mapped to Design/CPN data types.

9.1.2 Representation

The fundamental difference between the data type systems supported by MARIA and DESIGN/CPN is the boundedness of data types. Bounded systems have only one disadvantage—possible lack of expressive power.

When a data type has a domain of known size n, each value can be represented with $\lceil \log_2 n \rceil$ bits. This affects especially the representation of the state space. The encoding described in Section 7.3 clearly outperforms the one used in DESIGN/CPN, which maintains the data in the pointer structures of the underlying Standard Meta Language [42] implementation.

The data types in PROD have bounded domains, but the analyser does not encourage the use of constraints. Therefore, the state space encoder often has to deal with tuples of unconstrained machine words. The coding represents small numbers with fewer bits.

9.2 MESSAGE QUEUES

The Open Systems Interconnection Reference Model [22] and related models provide a common basis for communications protocols. Protocol entities in open systems communicate with each other by sending messages via lower-level entities, which are connected via a physical medium to the underlying network. The messages are usually buffered at the receiving end obeying the first-in-first-out discipline. For instance, the dynamic semantics of SDL, the CCITT Specification and Description Language [25], builds heavily on such buffering of messages, which are called "signals" in SDL.

In the idealistic world of SDL, buffers have infinite capacity, while all practical computing systems have a finite amount of memory. In many applications, "astronomically large" is a good substitute for infinite. When applying formal methods, especially in exhaustive reachability analysis, even models with buffer capacities limited to four messages may be unanalysable, and one has to hope that if there are errors in the protocol, they will be discovered also with severely limited capacities.

9.2.1 Previous Approaches

It is possible to represent queues without introducing special data types. If a queue holds only one kind of data items or if the semantics of the "queue" allows items to be dequeued in arbitrary order, the queue can be represented with one simple net place. Usually this is not the case.

We mention two previous approaches for modelling queues. Neither of them maps message queues directly to native data types, which would atomise queue operations.

Managing Queues as Circular Arrays

The approach taken in [28, 40] essentially models the queues of the SDL variant TNSDL [49] as circular arrays. It uses one place containing tokens (items in the buffer) tagged with the buffer position, and other places holding pointers to the front and back of the queue.

The advantage of this approach is that the queue contents never will be shifted. When an item is removed from the buffer, its slot will be marked empty, and the pointer to the front of the queue will be updated.

Unfortunately, the advantage is also a disadvantage. For instance, a first-in-first-out buffer of capacity 4 containing the items a, b and c could be represented as $\langle\langle a, b, c, \epsilon\rangle, \langle 0, 3\rangle\rangle$ but also as $\langle\langle c, \epsilon, a, b\rangle, \langle 2, 1\rangle\rangle$. The two structures are identical only when interpreted appropriately. A simple analysis algorithm may generate many uninteresting "copies" of the essential state space of a system modelled in such a way.

Managing Queues by Shifting Items

A similar approach, represented in [17], avoids the problem of seemingly different states by not using pointers. Instead, the buffer contents is shifted, one item at a time, when an item is removed or inserted. Especially if a model encompasses several buffers of this kind that can be used concurrently, its state space is likely to contain several lattices resembling those depicted in Section 2.2.

9.2.2 Algebraic Support for Queues

The variable-length buffer data type described in Section 5.3.3 suits to modelling message queues. The contents of the message queue can be packed in a variable whose type encapsulates a buffer over the message type. More precisely: assuming that some $s \in \mathcal{S}_\beta$ is the message sort and $\mathcal{D}_s^\mathcal{A}$ its carrier, the carrier of the associated message buffer sort s' could be $\mathcal{D}_{s'}^\mathcal{A} = \mathbb{V}_{\mathcal{D}_s^\mathcal{A},n}$ for some buffer length n.

The carrier of s' could also be some other data type encapsulating $\mathbb{V}_{\mathcal{D}_s^\mathcal{A},n}$. For instance, if the system consists of a number of similar buffers, it could make sense to represent the buffers as an array of buffers, or as sets of tuples with some components identifying the "owner" of the buffer and one component holding the buffer contents.

When the underlying formalism includes basic queue operations, there is no need to pay attention to the shifting operations discussed earlier. In our class of nets, queue operations can be performed atomically.

Some languages include operations that break the first-in-first-out principle of queues. For instance, the `save` operation of SDL [25] makes it possible to skip over certain messages in the front of a queue and to process and dequeue a following message. In order to facilitate simple transformations for this kind of non-orthodox operations, it is advisable to foresee the modelling formalism with enough expressive power. The queue operations defined in Section 6.3.5 have indexed variants.

9.3 DYNAMIC RESOURCE ALLOCATION

It is straightforward to model systems that use a fixed amount of resources over their lifetime. For systems that allocate resources dynamically, it is often difficult to foresee the maximum number of resources it will use in any of its reachable states. In practice, systems always have some limited number of resources, and an allocation operation may fail due to lack of available resources. If the pool of available resources is big enough so that a resource shortage never occurs, the illusion of a truly dynamic allocation of resources from an infinite pool works.

So, resource allocation can be modelled with a finite pool. In such models, there is a fixed amount of resources, some of which are available for allocation. Similar to message queues (Section 9.2), the resource pool usually has to be kept small in order to be able to analyse the model.

Although dynamic resource management is not particularly difficult to model, it has been neglected in many approaches. For instance, the transformation presented in [17] does not allow dynamic process creation.

9.3.1 Process Creation

The computations of a concurrent system are often executed in processes, each one executing its own algorithm, synchronising with other processes from time to time. Some systems are implemented with a dynamic number of processes. For instance, there could be a master process that listens for requests and creates slave processes, each of which serves one request before terminating.

The dynamic resource involved are data structures. Processes have a local state, which has to be maintained somehow, and in order to allow inter-process communication, each process has to be given a process identifier. A straightforward approach of modelling processes with Algebraic System Nets is to introduce a data type for the process identifier and to store the local variables of each process into data structures that can be indexed by the process identifier, e.g. into places whose carrier consists of multi-sets over tuples, one of whose components is the process identifier.

The identifier type defined in Section 5.2.5 suits very well to describing process identifiers. A pool of available process identifiers can be modelled with a place that initially contains all process identifiers. Each process creation operation will consume an identifier from the pool, and each process termination will restore the identifier to the pool.

When a process terminates, it may be useful to verify that no other processes hold the process identifier. Let us assume that there are processes "thinking" that a particular process still exists, even though it has terminated. When the process terminated, it returned its identifier back to the pool, and a new process creation operation may assign the same identifier to some other process. The processes still holding the old identifier may then mistake the new process for the old one.

This kind of errors, *dangling resource identifiers*, can presently be detected with model checking, but the associated temporal logic formulae or property automata may become very large. It would be tempting to introduce an algebraic operation that maps a marking of a net and an identifier value to a truth value:

$$J : \mathcal{M}_\Sigma \times \mathbb{J}_n \to \mathbb{B} : \langle M, j \rangle \mapsto \begin{cases} \top & \text{if "} j \notin M(p) \text{" for all } p \in P \\ \bot & \text{otherwise.} \end{cases}$$

The informal notation "$j \notin M(p)$" denotes that the identifier value j does not occur anywhere in the place marking, not even in a component of a structured value.

9.3.2 Memory Allocation

As discussed in Section 3.2.1, data structures should be abstracted away from formal models. Industrial-size programs exhibiting memory management problems typically are too big to be analysed using formal methods. One must try to cope with the less complete methods described in Section 3.1, such as static analysis and regression testing of instrumented program code.

But what if one absolutely must create a model that involves dynamic memory allocation? Like with dynamic process creation, the answer lies in the identifier data type, which can represent pointers to data items. When a program needs to allocate data items of sort s whose carrier is $\mathcal{D}_s^{\mathcal{A}}$, the generated model will contain an identifier pool (sort s', carrier $\mathcal{D}_{s'}^{\mathcal{A}}$) for the "pointers" of $\mathcal{D}_s^{\mathcal{A}}$, and the dynamically allocated data items of sort s will be represented using a sort s'' having the carrier $\mathcal{D}_{s''}^{\mathcal{A}} = \mathbb{T}_{\mathcal{D}_{s'}^{\mathcal{A}}, \mathcal{D}_s^{\mathcal{A}}}$.

This approach uses disjoint sets of pointers for different data types. There is a fixed pool of available pointers for each dynamically allocated data type. Dangling identifiers are a problem source also here, since a program may deallocate an object without removing all references to it.

With this approach, it is possible to model dynamic memory management that relies on explicit deallocation of unused data items. Systems that perform *garbage collection*, automatically deallocating unused resources, are difficult to model in an efficient way. One option would be to extend the formalism with a garbage collector, but an extension of such a global nature would affect many analysis methods and algorithms.

9.4 PROCEDURE CALLS

Structured programming languages require that program statements are encapsulated into units referred to as procedures. Procedures are very useful for structuring program code, and they can hide obscuring details of low-level operations from high-level procedure code. An Algebraic System Net, however, is a flat formalism: there are just places and transitions, all on the same level. In order to translate a structural program to an Algebraic System Net, the structure must be *flattened*, losing the structural information.[1]

Typical programming languages define procedure calls as part of the expression syntax. In other words, the underlying many-sorted algebra of a language may contain procedure calls as terms. This provides a way of defining operations without extending the language core.

[1] When generating a model of a program, one can provide some structural information by assigning meaningful identifiers to places and transitions. For instance, transition identifiers could consist of the name of the program file and the line and column numbers where the corresponding statement or subexpression of the program begins.

Allowing procedure calls in algebraic terms poses a problem when the terms are to be translated to a less expressive algebra that does not encapsulate procedure calls. Only calls to simple procedures that do not modify their environment e.g. by altering some data structures or by sending messages over the network, can be translated directly.

One solution is to divide the terms to subterms not containing procedure calls, and to evaluate the expression in several steps, making the procedure calls in appropriate places. In this way, one statement in a program can be translated into several transitions in an Algebraic System Net. This approach is analogous with the one outlined in Section 7.2.3. The transformation given in [17] transforms all procedure calls to several transitions, without giving special treatment to purely functional procedures that do not modify their environment. In [28, 40], procedure calls are treated as statements, so they cannot occur in subexpressions.

9.4.1 Scoping

Procedures typically have their own *name spaces*: variable x in procedure a is different from variable x in procedure b. It is thus impossible to represent both variables with just one place `X` that would hold the values of the variables. An obvious solution is to incorporate the *scope* identifier to the label, creating e.g. places `A:X` and `B:X`.

9.4.2 Recursive Procedures

Non-recursive procedure calls can be translated with a simple macro expansion, replacing each procedure call with the body of the called procedure, substituting the procedure parameters. The transformation represented in [17] is equivalent to a macro expansion in its expressive power: it cannot handle cyclic or recursive procedure invocations.

Recursive procedures can easily be implemented with a stack architecture. Each procedure invocation is executed in its own context. The contexts can be identified e.g. with the current recursion depth, the current number of active procedure invocations. The *control flow* of the procedures, the order in which statements will be executed, can be modelled with *control places*. Each statement will be translated to a sequence of transitions moving a *control token* from one control place to another. When several invocations of a procedure can be active at the same time, the control token must identify the context, e.g. the recursion depth.

Since the data types in our class of Algebraic System Nets are finite, we cannot allow infinite recursion depth. Analogous to dynamic resource management, a limit must be fixed for the recursion depth. If it is big enough and if the system being analysed does not exhibit infinite behaviour, the limit will not restrict the behaviour of the system.

9.4.3 Exception Handling

Exception handling [15] affects the semantics of procedure calls. When an unexpected condition occurs during the execution of a procedure, the procedure or the run-time system may raise an exception, transferring control to the nearest applicable exception handler. If no exception handler can be applied in the procedure, the exception will be raised further in the calling procedures, until an applicable exception handler is found.

The framework proposed in [15] allows resumption of the action that raised the exception. Most modern languages do not allow that: it is impossible for an exception handler to return to the statement that raised the exception condition. This means that exceptions can be modelled as a special kind of jump operations, which can return from several levels of procedure calls in one step, similar to the `setjmp` and `longjmp` operations defined in the C programming language [24].

9.5 OBJECT-ORIENTED CONSTRUCTS

The concept of object-oriented programming, the buzzword of the past decade, was introduced in the early 1960s. One of the first implementations, and the original inspiration behind C++ [23] was Simula [6], a language designed for writing simulations.

Object-oriented programming ties data type definitions and operations syntactically together. Traditional programming languages based on many-sorted algebras have typed (sorted) variables and operations whose signatures indicate the argument types. Object-oriented programming languages have a special kind of data types, called *classes*, which have *member variables* and *methods*. A class definition can be seen as a tuple data type definition, with the member variables as the components of the tuple. Methods are algebraic operations whose definition is syntactically bundled with the class definition. The instances of a class, or the variables of a class sort, are called *objects*.

The first C++ implementation [53] translated the object-oriented constructs to C code, mapping classes (object types) to structured data types, and rewriting identifiers to be compatible with the flat name space of C. We suggest a similar approach, flattening the constructs, for translating object-oriented programs to Algebraic System Nets.

9.5.1 Inheritance and Polymorphism

Simple objects, class instances, can be represented as tuples of their member variables. This is not the whole truth. Objects can *inherit* properties from each other. A class *derived* from a *base class* inherits all the member variables of the base class, and it may add new member variables. There can be inheritance on multiple levels, and several classes can derive from the same base class. Nevertheless, the inheritance structure can be represented as a tree or as a directed acyclic graph, if *multiple inheritance*, deriving properties from more than one base class, is allowed.

Objects of derived classes, which can sometimes also be used as objects of a base class, can be represented with a union of structured data types. Consider a base class C whose members can be represented with the carrier $\mathcal{D}_s^{\mathcal{A}}$ of a sort s. A derived class C' whose own members can be represented with the carrier $\mathcal{D}^{\mathcal{A}}_{s'}$ of another sort s', can be represented as the tuple $\mathbb{T}_{\mathcal{D}^{\mathcal{A}}_{s},\mathcal{D}^{\mathcal{A}}_{s'}}$.

It is possible for a derived class to *override* some of the methods defined for the base class. When a particular method is applied to an object, the corresponding method defined for the base class will only be used if it has not been overridden in the class the object belongs to. A method that performs different operations on different kinds of objects is called *polymorphic*.

Polymorphism can be realised by allowing objects to belong either to a base class or to one of its derived classes. Such objects can be represented as a tuple of the carrier of the base class and a tagged union over the carriers of the derived class members, and over a singleton type. The singleton type applies for objects of the base class, which have no derived members. The algebraic operations for the union type defined in Sections 6.3.5 and 6.3.6 can be used to find out the classes whose instance an object is, and to convert objects of derived classes to objects of corresponding base classes.

10 CONCLUSION

There exist numerous reachability analysers for models of concurrent and distributed systems, but most of them lack a powerful data type system and algebraic operations that would facilitate a straightforward transformation from a computer program to a formal analysis model. Such tools are typically used for analysing manually constructed models.

The tedious and time-consuming task of constructing a formal verification model of a system can be eased by developing more powerful tools. We have implemented a freely distributable tool for analysing models employing a data type system that tolerates the comparison with the data type systems used in some state-of-the-art reachability analysers.

While the formalism we deploy has not specifically been designed for constructing models of computer software, the transformation principles outlined in Chapter 9 should cover the intrinsic constructs of commonly used procedural and object-oriented programming languages. Some of the transformations have been presented in earlier work; some of them appear to be new.

Our description formalism, Algebraic System Nets [32], has a solid mathematical foundation, and there is active research on efficient analysis methods for it. A substantial part of the research concentrates on a lower-level formalism, Place/Transition Nets [47]. Analysis methods developed for that level can be lifted to our class of nets, since all models constructed in our formalism can be unfolded to the lower level.

The framework of our formalism is based on many-sorted algebras, defined in Chapter 4. The interpretation rules of our many-sorted algebras include short-circuit evaluation, a common feature in programming languages that is easily overlooked. Short-circuit evaluation is not merely an optimisation; it also affects the semantics when a subexpression skipped due to short-circuit evaluation could not be evaluated due to an error.

Implementing an expression evaluator is a rather irksome task, but there is some place for important design choices, such as different optimisations, or the way errors are reported. Especially the latter greatly affects the usability of the expression evaluator, both from the user and from the programmer perspective.

The biggest user of an expression evaluator in a Petri Net reachability analyser is the transition instance analysis algorithm described in Section 7.1, which finds all actions that can be performed in a given system state. If the instance analyser silently ignored erroneous actions, as at least one implementation [57] does, serious errors could be overlooked, or they could be hard to locate. Our instance analyser reports all erroneous transition instances, even incomplete ones, and the user will always be notified when something is wrong in the model.

In addition to the common data types and basic algebraic operations, our analyser tool implementation [38] defines a total order and a compact encoding for all data types, including structured and constrained types.

Compact encoding is essential for space efficient management of the state spaces of complex systems. In our formalism, the nodes of a reachability graph are sequences of multi-sets over sorted values. We believe the encoding method described in Section 7.3.1 to be an original invention. According to Table 8.1, it performs an order of magnitude better than the method used in a comparable tool.

Currently the reachability analyser we have designed and implemented lacks some essential features, such as on-the-fly model checking of properties specified in temporal logic with the presence of fairness constraints, and various reduction methods that cause uninteresting intermediate states to be omitted from the reachability graph. These areas are being investigated, and the implementation is being worked on.

This work discusses specialised front-ends, which translate a system description given in an application-specific high-level language to a formal model that can be analysed, but does not refer to an actual implementation. Our compiler [35] for SDL [25] is being extended with model generation routines, so that specifications of telecommunications protocols can be input to the reachability analyser. It will be an interesting exercise to implement transformations for true object-oriented constructs and exception handling, which should be included in the upcoming version of SDL.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison–Wesley, Reading, MA, USA, March 1986.

[2] Jonathan Billington. Many-sorted high level nets. In *3rd Workshop on Petri Nets and Performance Models*, pages 166–179, Washington, DC, USA, 11–13 December 1989. IEEE CS Press, Los Alamitos, CA, USA.

[3] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, NY, USA, 1973.

[4] Edmund M. Clarke Jr., Orna Grümberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[5] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, October 1981.

[6] Ole-Johan Dahl and Kristen Nygaad. SIMULA—an Algol based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[7] Jeremy Dion and Louis Monier. *Third Degree: heap usage and leak profiler, and memory-access error checker for C and C++ programs*. Digital Equipment Corporation, Maynard, MA, USA, 1998.

[8] Anders Ek. Automatic debugging of communicating systems using the SDT Validator. Technical paper, Telelogic AB, Malmö, Sweden, September 1998.

[9] Frits Feldbrugge. List of Petri Net tools. *Petri Net Newsletter*, 22:20–32, October 1985.

[10] Jaco Geldenhuys and Pieter de Villiers. Runtime efficient state compaction in SPIN. In Dennis Dams and Mieke Massink, editors, *The 5th International SPIN Workshop on Theoretical Aspects of Model Checking*, pages 3–12, Trento, Italy, July 5 1999.

[11] Hartmann J. Genrich. Predicate/Transition Nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and their Properties—Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247, Bad Honnef, Germany, September 1986. Springer-Verlag, Berlin, Germany, 1987.

[12] Tristan Gingold. *Checker, a memory access detector*, 1996. Documentation for Version 0.8.

[13] Patrice Godefroid and Gerard J. Holzmann. On the verification of temporal properties. In *13th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 109–124, Liège, Belgium, May 1993.

[14] Joseph A. Goguen. Abstract errors for abstract data types. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 491–525. North-Holland Publishing Company, Amsterdam, The Netherlands, 1978.

[15] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12), December 1975.

[16] Bernd Grahlmann. The PEP tool. In Orna Grümberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443, Haifa, Israel, June 1997. Springer-Verlag, Berlin, Germany.

[17] Bernd Grahlmann. *Parallel Programs as Petri Nets*. Dissertation, Universität Hildesheim, Fachbereich Mathematik, Informatik, Naturwissenschaften, Hildesheim, Germany, September 1998.

[18] Jean-Charles Grégoire. State space compression in SPIN with GETSs. In *2nd International SPIN Verification Workshop*, New Brunswick, NJ, USA, August 1996.

[19] Keijo Heljanko. Model checking the branching time temporal logic CTL. Research Report A45, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, May 1997.

[20] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[21] Nisse Husberg. Verifying SDL programs using Petri nets. In *1998 IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 208–213, San Diego, CA, USA, October 1998. Institute of Electrical and Electronics Engineers, Inc.

[22] *Information Processing Systems—OSI Reference Model—The Basic Model*. ISO/IEC 7498-1. International Organization for Standardization, Geneva, Switzerland, 1994.

[23] *Information Technology—Programming Languages—C++*. ISO/IEC 14882. International Organization for Standardization, Geneva, Switzerland, 1998.

[24] *Information Technology—Programming Languages—C*. ISO/IEC 9899. International Organization for Standardization, Geneva, Switzerland, 1999.

[25] *CCITT Specification and Description Language (SDL)*. Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, October 1996.

[26] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1995.

[27] Tommi Junttila. *Detecting and Exploiting Data Type Symmetries of Algebraic System Nets during Reachability Analysis*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, December 1999.

[28] Tero Jyrinki. Dynamic analysis of SDL programs with Predicate/Transition Nets. Technical Report B17, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, April 1997.

[29] Jukka Kemppainen. *ARA Tools, Reference Manual*. VTT Electronics, Oulu, Finland, March 1994.

[30] Esa Kettunen, Esa Montonen, and Timo Tuuliniemi. An interactive PrT-net tool for verification of SDL-specifications. Technical Report B3, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, February 1988.

[31] Ekkart Kindler and Wolfgang Reisig. Algebraic System Nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.

[32] Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–364, Lisbon, Portugal, June 1998. Springer-Verlag, Berlin, Germany.

[33] Raimo Kujansuu, Leo Ojala, and Heikki Tuominen. Verification and validation of communication protocols. In Carl A. Sunshine, editor, *Protocol Testing, Specification and Validation*, pages 311–313, Idyllwild, CA, USA, May 1982. North-Holland Publishing Company, Amsterdam, The Netherlands.

[34] Marek Leszak and Horst Eggert. *Petri-Netz-Methoden und -Werkzeuge*, volume 197 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, Germany, October 1988.

[35] Marko Mäkelä. Implementing the front-end of an SDL compiler. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, December 1998.

[36] Marko Mäkelä. Applying compiler techniques to reachability analysis of high-level models. In Hans-Dieter Burkhard, Ludwik Czaja, Andrzej Skowron, and Peter Starke, editors, *Workshop on Concurrency, Specification & Programming 2000*, number 140 in

Informatik-Bericht, pages 129–142. Humboldt-Universität zu Berlin, Germany, October 2000.

[37] Marko Mäkelä. Condensed storage of multi-set sequences. In Kurt Jensen, editor, *Practical Use of High-Level Petri Nets*, number 547 in DAIMI report PB, pages 111–125. University of Århus, Denmark, June 2000.

[38] Marko Mäkelä. *Maria—Modular Reachability Analyzer for Algebraic System Nets*. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, January 2000. On-line documentation, ⟨URL:http://www.tcs.hut.fi/maria/⟩.

[39] Marko Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. In *Application and Theory of Petri Nets 2001: 21st International Conference, ICATPN'01*, Lecture Notes in Computer Science, Newcastle upon Tyne, England, June 2001. Springer-Verlag, Berlin, Germany. To appear.

[40] Markus Malmqvist. Methodology of dynamical analysis of SDL programs using Predicate/Transition Nets. Technical Report B16, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, April 1997.

[41] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Reference Manual for X-Windows, Version 2.0*, 1993.

[42] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[43] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In Howard Forbus and Kenneth Shrobe, editors, *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 42–47, Seattle, WA, USA, July 1987. Morgan Kaufmann, San Mateo, CA, USA.

[44] Christos H. Papadimitriou. *Computational Complexity*. Addison–Wesley, Reading, MA, USA, 1994.

[45] Bruce Perens. *Electric Fence Malloc Debugger*. Pixar Animation Studios, 1993. Manual page for Version 2.0.5.

[46] Carl Adam Petri. *Kommunikation mit Automaten*. Dissertation, Technische Universität Darmstadt, Fachbereich Mathematik, Physik, Darmstadt, Germany, 1962.

[47] Wolfgang Reisig. *Petrinetze—Eine Einführung*. Springer-Verlag, Berlin, Germany, 1986.

[48] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, March 1991.

[49] Erkki Ruohtula, Esa Kettunen, and Heikki Tuominen. *TNSDL Book*. Nokia Telecommunications, Inc., 3rd edition, November 1995.

[50] Michael J. Sanders. Efficient computation of enabled transition bindings in high-level Petri nets. In *2000 IEEE International Conference on Systems, Man and Cybernetics*, pages 3153–3158, Nashville, TN, USA, October 2000.

[51] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly & Associates, Sebastopol, CA, USA, 1st edition, August 1997.

[52] Harald Störrle. An evaluation of high-end tools for Petri-nets. Bericht 9802, Ludwig-Maximilians-Universität München, Institut für Informatik, Munich, Germany, June 1998.

[53] Bjarne Stroustrup. Adding classes to the C language: An exercise in language evolution. *Software—Practice and Experience*, pages 139–161, February 1983.

[54] Antti Valmari. State space generation: Efficiency and practicality. Publications 55, Tampere University of Technology, Tampere, Finland, 1988.

[55] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, Berlin, Germany, 1998.

[56] Kimmo Varpaaniemi. On the stubborn set method in reduced state space generation. Research Report A51, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, May 1998.

[57] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995.

[58] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Sebastopol, CA, USA, 2nd edition, May 1997.

[59] Gary Watson. *Debug Malloc Library*, January 1998. Documentation for Version 3.3.1.

[60] Colin H. West. Automated protocol validation. In Carl A. Sunshine, editor, *Protocol Testing, Specification and Validation*, pages 361–371, Idyllwild, CA, USA, May 1982. North-Holland Publishing Company, Amsterdam, The Netherlands.

Index

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A56	Keijo Heljanko	Deadlock and Reachability Checking with Finite Complete Prefixes. December 1999.
HUT-TCS-A57	Tommi Junttila	Detecting and Exploiting Data Type Symmetries of Algebraic System Nets during Reachability Analysis. December 1999.
HUT-TCS-A58	Patrik Simons	Extending and Implementing the Stable Model Semantics. April 2000.
HUT-TCS-A59	Tommi Junttila	Computational Complexity of the Place/Transition-Net Symmetry Reduction Method. April 2000.
HUT-TCS-A60	Javier Esparza, Keijo Heljanko	A New Unfolding Approach to LTL Model Checking. April 2000.
HUT-TCS-A61	Tuomas Aura, Carl Ellison	Privacy and accountability in certificate systems. April 2000.
HUT-TCS-A62	Kari J. Nurmela, Patric R. J. Östergård	Covering a Square with up to 30 Equal Circles. June 2000.
HUT-TCS-A63	Nisse Husberg, Tomi Janhunen, Ilkka Niemelä (Eds.)	Leksa Notes in Computer Science. October 2000.
HUT-TCS-A64	Tuomas Aura	Authorization and availability - aspects of open network security. November 2000.
HUT-TCS-A65	Harri Haanpää	Computational Methods for Ramsey Numbers. November 2000.
HUT-TCS-A66	Heikki Tauriainen	Automated Testing of Büchi Automata Translators for Linear Temporal Logic. December 2000.
HUT-TCS-A67	Timo Latvala	Model Checking Linear Temporal Logic Properties of Petri Nets with Fairness Constraints. January 2001.
HUT-TCS-A68	Javier Esparza, Keijo Heljanko	Implementing LTL Model Checking with Net Unfoldings. March 2001.
HUT-TCS-A69	Marko Mäkelä	A Reachability Analyser for Algebraic System Nets. June 2001.

ISBN 951-22-5541-3
ISSN 1457-7615