Newcastle University

School of Electrical, Electronic and Computer Engineering

Interpreted Graph Models

by Ivan Poliakov

PhD Thesis

2011

Abstract

A model class called an *Interpreted Graph Model (IGM)* is defined. This class includes a large number of graph-based models that are used in asynchronous circuit design and other applications of concurrecy. The defining characteristic of this model class is an underlying static graph-like structure where behavioural semantics are attached using additional entities, such as tokens or node/arc states. The similarities in notation and expressive power allow a number of operations on these formalisms, such as visualisation, interactive simulation, serialisation, schematic entry and model conversion to be generalised.

A software framework called Workcraft was developed to take advantage of these properties of IGMs. Workcraft provides an environment for rapid prototyping of graph-like models and related tools. It provides a large set of standardised functions that considerably facilitate the task of providing tool support for any IGM.

The concept of *Interpreted Graph Models* is the result of research on methods of application of lower level models, such as Petri nets, as a back-end for simulation and verification of higher level models that are more easily manipulated. The goal is to achieve a high degree of automation of this process. In particular, a method for verification of speed-independence of asynchronous circuits is presented. Using this method, the circuit is specified as a gate netlist and its environment is specified as a Signal Transition Graph. The circuit is then automatically translated into a behaviourally equivalent Petri net model. This model is then composed with the specification of the environment. A number of important properties can be established on this compound model, such as the absence of deadlocks and hazards. If a trace is found that violates the required property, it is automatically interpreted in terms of switching of the gates in the original gate-level circuit specification and may be presented visually to the circuit designer.

A similar technique is also used for the verification of a model called *Static Data Flow Structure (SDFS)*. This high level model describes the behaviour of an asynchronous data path. SDFS is particularly interesting because it models complex behaviours such as *preemption*, *early evaluation* and *speculation*. Preemption is a technique which allows to destroy data objects in a computation pipeline if the result of computation is no longer needed, reducing the power consumption. Early evaluation allows a circuit to compute the output using a subset of its inputs and preempting the inputs which are not needed. In speculation, all conflicting branches of computation run concurrently without waiting for the selecting condition; once the selecting condition is computed the unneeded branches are preempted. The automated Petri net based verification technique is especially useful in this case because of the complex nature of these features.

As a result of this work, a number of cases are presented where the concept of IGMs and the Workcraft tool were instrumental. These include the design of two different types of arbiter circuits, the design and debugging of the SDFS model, synthesis of asynchronous circuits from the Conditional Partial Order Graph model and the modification of the workflow of Balsa asynchronous circuit synthesis system.

Acknowledgements

I am very grateful to my supervisor, Alex Yakovlev, for his invaluable guidance and constant support.

I would like to thank my friends and colleagues for their input, especially Victor Khomenko for the numerous fruitful discussions on the topics of verification, synthesis, and Interpreted Graph Models; Danil Sokolov who formalised the SDFS model and Arseniy Alekseyev who contributed a lot to the development of Workcraft.

I would also like to thank Jordi Cortadella, Oriol Roig and Tomohiro Yoneda for their help setting up the verification tools which made it possible to do the benchmarks for Chapter 4.

And a very special thanks to my family, my father Valery, my mother Natalia and my brother Ilya who were always supporting me, even when the times were dire for themselves; and of course to my wife Katerina, always loving and patient.

This work was supported by the EPSRC grants EP/D053064/1 (SEDATE) and EP/G037809/1 (VERDAD).

Contents

List of Figures

List of Publications

Conference papers

2007 **Workcraft: A Static Data Flow Structure Editing, Visualisation and Analysis Tool** (Ivan Poliakov, Danil Sokolov, Andrey Mokhov), *In Petri Nets and Other Models of Concurrency (ICATPN '2007)*

2007 **Asynchronous Data Path Models** (Danil Sokolov, Ivan Poliakov, Alexandre Yakovlev), *In Proceedings of the Seventh International Conference on Application of Concurrency to System Design (ACSD '07)*

2008 **Automated Verification of Asynchronous Circuits Using Circuit Petri Nets** (Ivan Poliakov, Andrey Mokhov, Ashur Rafiev, Danil Sokolov, and Alex Yakovlev), *In Proceedings of the 2008 14th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '08)*

2008 **Static Data Flow Structures with Dynamic Elements** (Ivan Poliakov, Danil Sokolov, Alex Yakovlev, Charles Brej), *In Proceedings of the 20th UK Asynchronous Forum (UKAF '2008)*

2009 **Workcraft — a Framework for Interpreted Graph Models** (Ivan Poliakov, Victor Khomenko, Alex Yakovlev), *In Proceedings of the 30th International Conference on Applications and Theory of Petri Nets (PETRI NETS '09)*

Journal papers

2008 **Analysis of Static Dataflow Structures** (Danil Sokolov, Ivan Poliakov, Alex Yakovlev), *Fundamenta Informaticae, Vol. 88, No.4, pp. 581-610, IOS Press*

Public tool demonstrations

2007 *Design and Testing in Europe (DATE)*

2007 *International conference on Petri Nets and Other Models of Concurrency (ICAPTN)*

2009 *International Conference on Applications and Theory of Petri Nets (PETRI NETS)*

2008 – 2011 *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*

Chapter 1

Introduction

With the continuous increase of the number of transistors that can be put on a single VLSI chip, configurations known as system-on-a-chip (SoC) are becoming more and more popular [96]. Such systems consist of a number of interconnected heterogeneous blocks, such as processors, memory and I/O controllers, built as a single chip (Figure 1.1)[1]. Tighter integration reduces the size of the system and the cost to produce, resulting in portable, high-performance and power efficient consumer products, such as modern smartphones and tablet PCs.

Most of the circuits produced by industry today are synchronous. The operation of their components is controlled by one or more globally distributed periodic signals called clock. Combining pre-designed components into a globally clocked SoC is not a trivial task. Each component (usually called an *IP core*) is designed for a certain clock frequency, and its correct functionality relies on the clock signal being delivered at the same time to all parts of the system. But it is not always possible to use a common clock frequency for the whole SoC. Additionally, the variations in interconnects between IP cores lead to the clock skew problem: reliable distribution of the global clock signal becomes an extremely complex task when the size of the system is in the order of billions of transistors [48, 113].

Therefore, when designing such systems, it is no longer possible to rely solely on the traditional approach of using the globally distributed clock signal.

Solving the problem of communication between the various IP cores clocked with different

[1] Image by Colin M.L. Burnett, used under the terms of GFDL: http://www.gnu.org/copyleft/fdl.html

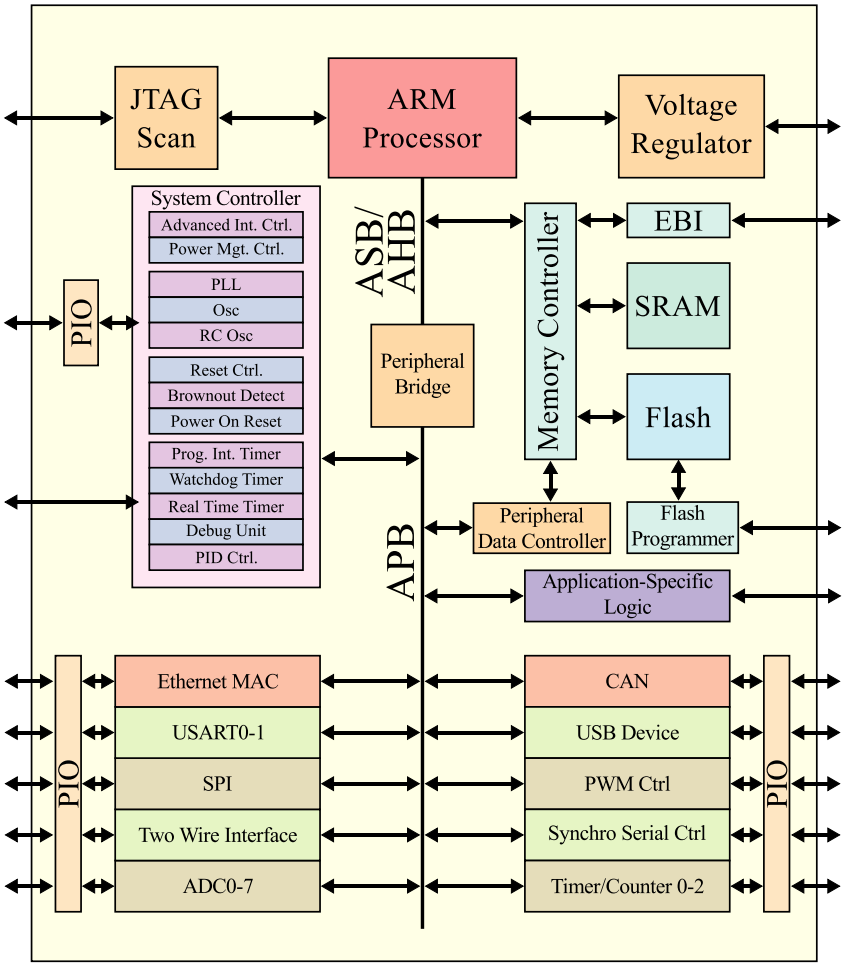


Figure 1.1: Architecture of an ARM-based system-on-a-chip.

frequencies, some potentially self-timed, and all running in parallel becomes one of the central requirements for a successful IC design.

One of the widely accepted solutions to this problem is the use of asynchronous circuits to bridge the IP cores, resulting in what is called a GALS (globally asynchronous, locally synchronous) design [37]. In a GALS system, each synchronous block is interfaced with an asynchronous wrapper that provides communication facilities between blocks in an asynchronous manner.

Asynchronous logic holds several important advantages over synchronous designs. For example, in the context of SoCs, asynchronous circuits can be made to interface with a clocked circuit independent of the clock frequency. Without an asynchronous communication layer, communication between blocks with unrelated clock frequencies (e.g. 333 MHz and 500 MHz) would be very hard to implement efficiently. Having a reliable method of interfacing blocks with arbitrary frequencies not only eliminates the need for the global clock, but greatly increases the potential for component re-use.

A logical evolution of the GALS paradigm is a fully self-timed system, where components do not have local clocks but instead operate asynchronously, using causal relations between events. In addition to better modularity and the lack of clock distribution problem, fully asynchronous circuits possess properties that may be invaluable for certain applications. Among such properties are the inherent concurrency which can exploit parallelism, generally lower power requirements, better tolerance to voltage fluctuations and the ability to automatically adjust operating speed according to changing environmental conditions [85, 43, 110, 118]. These properties allow asynchronous circuits to be used not only in systems where robust concurrent operation is important (e.g. SoCs), but also in specialised devices, such as those operating on harvested power, where a synchronous circuit would be unusable. The absence of the global clock results in lower electromagnetic interference, as well as better device security due to resistance to power analysis attacks (e.g. differential power analysis [71]) – a property that is essential for security-sensitive applications [78].

Unfortunately, asynchronous design technology has its own share of drawbacks that prevent the technology from becoming mainstream. Mitigating some of these problems, which are out-

lined further in this chapter, is the motivation for this work.

1.1 Motivation

The complexity of the asynchronous circuit design process is generally seen as a major drawback preventing the wider adoption of this technology by the industry. There are numerous factors contributing to this perception, and one of the most important ones is the lack of appropriate design tools. The availability of mature, robust tools for the design of synchronous circuits is much higher than that of the tools aimed at an asynchronous approach. In fact, many asynchronous design groups rely on modifying the tools that exist for synchronous design. Because those tools will treat an asynchronous circuit as if it were synchronous, various tricks and workarounds are required to ensure the correct functionality of the resulting design. In many cases, such workarounds are inefficient. Additionally, important design stages, such as asynchronous logic verification, may be very difficult to integrate into the design flow.

Tools that are available specifically for the purpose of asynchronous logic design tend to suffer from limited efficiency due to the phenomenon of state space explosion [117]. This often means that the designer of an asynchronous system is forced to design or verify parts of the circuit manually to overcome these limitations, which is not only time-consuming, but also implies that the designer must have vast knowledge and experience to produce reliable solutions.

The design process of any relatively large asynchronous circuit generally consists of three stages: behavioural specification, implementation and verification. During specification, the expected behaviour of the circuit is described using either a general-purpose hardware description language, such as Verilog [114], VHDL [87] and SystemC [53], or a language designed specifically for asynchronous circuits, such as Tangram [119] or Balsa [47]. The high-level language description is decomposed into a set of communicating processes, or components. The interaction between the processes is captured using a formal model. Numerous formalisms are used for this purpose in different design flows, including Petri nets [84, 90], Signal Transition Graphs [126], process algebras [24, 29], Communicating Sequential Processes [56] and other. Different parts of the system may be described using different formalisms, in contrast to synchronous systems, where the finite state machine (FSM) model is most often the fundamental construct underlying

the whole design process.

For implementation, the specifications of control and data paths are extracted from the formal model. The control and data paths are implemented and optimised separately to improve the design features of each path independently. For example, the control path is often optimised for low latency and size, while the data path is optimised for power consumption and throughput. There are two distinct methodologies used to obtain the implementation of the control and data paths: logic synthesis [68, 41] and direct mapping [47, 119, 106]. In direct mapping, elements of the formal model are mapped into pre-designed components using one-to-one correspondence. In logic synthesis, the implementation is produced through the analysis of the state graph of the system. Logic synthesis produces more efficient solutions, however it is only applicable for small controllers due to its algorithmic complexity. Direct mapping method is very fast and may be used to build very large circuits, but it often produces slow circuits.

To formally verify an asynchronous circuit, various schemes are applied to mathematically prove that the circuit does not exhibit incorrect behaviour following any possible input sequence. Completely automated verification of a complete system is often computationally infeasible because of the state space explosion phenomenon. The designer must be able to produce such abstractions of the system that are small enough for automated verification but are still representative of the actual behaviour of the modelled system. In practice, many levels of abstractions may be required to adequately model and analyse the system in question.

For large projects, it is often a mixture of various formalisms, implementation techniques and verification methods that is used to produce the final design. There is no universally accepted design workflow for asynchronous systems. Despite the fact that a robust theoretical and technological kernel exists for the most of the individual tasks outlined above, a truly complete production pipeline, such that would cover all stages from specification to verification to mapping the net-list to silicon, is indeed very hard to set up. The fragmented state of the tool base is one of the reasons for that. There is not a package, either commercial or open-source, targeted at asynchronous circuit design, that could be compared to the complete synchronous design solutions provided by companies such as Synopsys, Cadence, or Mentor Graphics. Instead, there exists a multitude of standalone tools, coming mostly from academia, that target one particular task without much re-

gard to interaction with other tools that may be doing tasks that precede or follow it in the design process. It is up to the designer to organise such an interaction, which may not be a trivial task; unless such an interaction is automated, it quickly becomes very cumbersome. In addition, the tools coming from the academic environment are mostly focused on solving the mathematical and algorithmic problems and rarely focus on the aspects of user interaction. The interfaces to those tools are mostly command-line- or file-based, which may be daunting even to an experienced user. The output often requires post-processing to become human readable.

The complexities in the asynchronous circuit design therefore exist not only on the purely computational level, but also on the level of organisation of the human-machine interaction during the design process. This concern is the main motivation behind this work.

The specific problems that will be addressed in this thesis are outlined below. They include the automation of specific tasks related to asynchronous circuit design, assisting in the organisation of interaction between existing tools, providing user friendly visual representation of various formal models and the results of their analysis using techniques such as interactive simulation and visualisation of violation traces.

1.1.1 Automated verification of asynchronous circuits using Petri nets

The designs of relatively large circuits are often hierarchical and compositional. Individual blocks of such designs are built after their sub-blocks have been designed and validated. Appropriate forms of interface of the sub-blocks are required, in which the behavioural complexity of the internal implementation of sub-blocks is hidden behind a subset of interface signals. For example, one can abstract away from the timing conditions used inside the sub-blocks, thereby considering the system at the higher level of abstraction from the point of view of its delay-independent behaviour. Conversely, the design may assume a block to be operating in a delay-independent context but the actual internal behaviour of the implementation needs to be validated in terms of its freedom from hazards.

Compositional approach can be achieved in different ways depending on the modelling method used for verification. For example, in the context of Petri nets, a block with an implementation that satisfies its Petri net based specification can be represented in a complex design not by the Petri

net model of its implementation but rather by the Petri net specification, which can be significantly more compact, thereby helping to reduce the complexity of analysis at the higher level.

Aside from good support for modularity, the choice of Petri nets as the verification back-end for asynchronous circuits is justified for several other reasons. A gate-level circuit implementation can be efficiently converted into a Petri net model. Once such a model is obtained, it is very flexible with respect to the methods that can be applied for analysis. For instance, it is possible to compose the Petri net model of a circuit with a Petri net model of its environment, with the model of another circuit, or even with a net generated from a totally different formalism to produce the model of the complete system. Alternatively, the net may be simplified using techniques such as dummy contraction [99] and serve as an environment specification for another model.

The flexibility also comes from the very rich tool base. There are a large number of tools that apply drastically different techniques for analysis (LoLA [101], MPSat [68], Petrify [41], etc.), which provides the possibility to choose the tool that copes best with the structure of the system in question. In addition to performance considerations, modern Petri net tools support the verification of non-standard property specifications which may be very helpful during the debugging of specific systems [66].

Finally, it is possible to interpret a Petri net violation trace in terms of the signal switching activity in the original circuit, which makes it easy to present the verification result to the designer visually and using the original gate-level model.

1.1.2 Modelling, simulation and automated verification of the data path of asynchronous circuits

The synthesis of asynchronous control path is well developed and supported by tools, e.g., Petrify [41] and MPSat [68]. The synthesis of asynchronous data path, on the other hand, has not been studied as thoroughly. Usually, a designer makes the whole data path either *bundled-data* (to achieve a smaller size of the circuit) or *dual-rail* (to get an average-case performance and greater robustness to delay variations). In both cases the conventional EDA tools, such as Cadence or Synopsis, are used to obtain the data path combinational logic. The synchronous nature of those tools does not allow exploiting the full potential of the asynchronous data path. For instance,

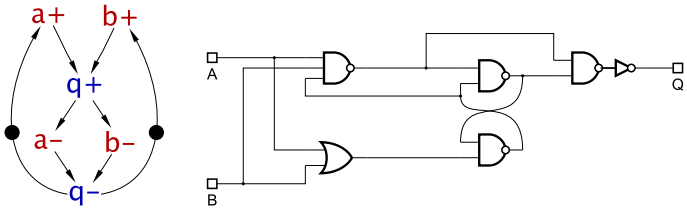
early evaluation cannot be controlled and influenced at an early synthesis stage. Data encoding is forced to single-rail, which is the norm for synchronous designs, but results in redundancy when the obtained circuits are converted to dual-rail.

The combinational logic for the asynchronous data path can alternatively be produced by the tools traditionally used for synthesis of asynchronous controllers, for example Petrify [41]. These tools are based on the state-space exploration, and hence can only handle relatively small specifications. To successfully use them, the data path has to be decomposed into small fragments the state space of which would not exceed the limitations of the available computers. These fragments are then synthesised separately and connected together to form the complete data path circuit.

The choice of the suitable synthesis methods for the various parts of data path is also relevant. It may be the case that implementing some branch of a data path as "expensive" dual-rail does not give any speed advantage because there is a concurrent branch that is very slow and never exhibits early evaluation (the benefits of early evaluation are explained in Section 5.3).

In order to decompose, optimise and efficiently synthesise the asynchronous data path it must be analysed at the level of a formal, technology independent model. At the moment of writing, there is no formal model which adequately represents all the features of the asynchronous data path. Traditional models, such as *Petri nets* (PNs) [90] and *finite state machines* (FSMs), are abstract and low-level, and it is hard to use them to model the high-level behaviour of asynchronous data path. The models that naturally capture the asynchronous data path, such as Static Data Flow Structures (SDFS) [110], are not formally defined and require further research. In particular, modelling *preemption*, *early evaluation* and *speculation* in the asynchronous data path by SDFS is of great interest. Preemption is a technique which allows data items to be destroyed in a computation pipeline if the result of computation is no longer needed, thus reducing the power consumption. Early evaluation and speculation techniques are based on the preemption idea. Early evaluation allows a circuit to compute the output using a subset of its inputs and preempting the inputs which are not needed. In speculation, all conflicting branches of computation run concurrently without waiting for the selecting condition; once the selecting condition is computed the incorrect branches are preempted.

Due to the presence of such complex behaviours, automated verification of SDFS models is



(a) C-element environment STG

(b) C-element implementation

Figure 1.2: A circuit model specified using a gate-level net-list and an environment STG.

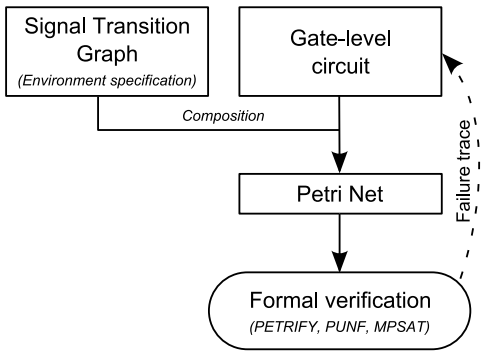


Figure 1.3: The interaction between different formalisms.

very important: it is necessary to ensure that the system is free of problems such as deadlocks. A verification technique based on the low-level Petri net representation, similar to the technique proposed for the verification of circuits in Section 1.1.1 has several important advantages. In particular, it provides a clear graphical presentation of problematic sequences of events found during the verification, which may greatly assist the designer in isolating and fixing the issue.

1.1.3 Multi-formalism models and interaction between formalisms

One of the issues encountered in asynchronous circuit design is the large number of available formal models. To increase productivity, it is often reasonable for the designer to use different formalisms to model various aspects of a complex system. For example, whether a circuit design functions correctly or not almost always depends on the behaviour of its environment, i.e. a circuit can be functional in one environment and not functional in another. In most cases it is impractical or even impossible to provide the specification of the environment as a gate-level circuit. The *abstraction* of the environment is often given in the form of a Signal Transition Graph (STG). This means that the overall specification is inherently heterogeneous, as one part of it is a gate net-list

and another part is an STG (Figure 1.2).

Analysis and verification of such a system is problematic, because very few tools support compound system specification. Development of specialised tools for every new modelling solution is not feasible: it takes years for an algorithmically complex tool to become stable and robust.

In order to re-use existing tools, an attractive solution is to convert the system specification into one of the formalisms with highly developed tool support, such as Petri nets. In this case the analysis is performed on the resulting Petri net, but the results (the violation traces) are then interpreted in terms of the original model (Figure 1.3). Unfortunately, there are no tools that provide adequate support for tasks such as conversion between different formalism types and re-interpreting the low-level analysis results in terms of another, higher level model.

1.1.4 The tool interoperability problem

Even when working with one particular formal model, an asynchronous circuit designer may need to routinely use several tools to perform various tasks. For example, there are different tools that can be used to perform an analysis of a Petri net (e.g. Petrify [41], MPSat [68, 12], LoLA [101]). All of them are based on different techniques and their performance depends greatly on the structure of the given Petri net. If there is a high degree of concurrency present in the net, MPSat may be the best choice, because it is based on the concept of Petri net unfoldings that naturally exploits concurrency [64]. On the other hand, if there are many parts involving choices, MPSat's performance may be unacceptable. The tools also have different sets of supported properties, and their property specification methods differ significantly.

It is therefore impossible to choose one universally applicable tool even for one particular task (verification). And yet besides verification, there are many other practical things that can be done with Petri nets, e.g. composition (supported by PComp [12]), decomposition (supported by DesiJ [99]), synthesis (supported by SIS [103], Petrify [41], MPSat [68, 12] and several other tools), graphical layout (supported by Dot [7]), etc.

This issue becomes even more evident when working with larger, more practical projects, where the number of various model types (and their possible combinations) that need to be managed grows quickly. Consider Figure 1.4, which shows the process of asynchronous system design

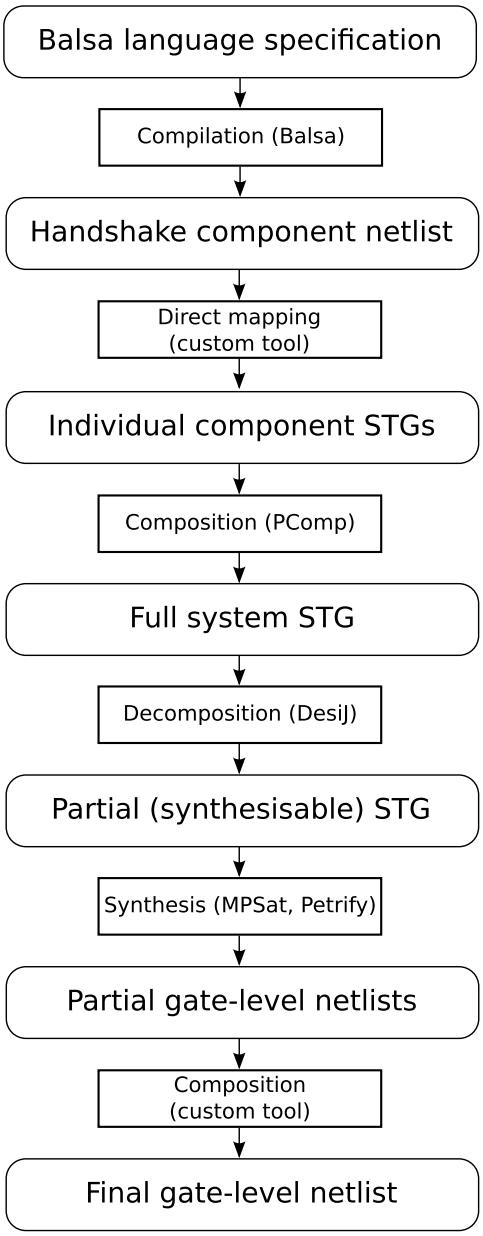


Figure 1.4: Schematic of the asynchronous circuit design technique called “resynthesis”.

known as *resynthesis* [89, 129]. In this design flow, the original system specification is written using Balsa language [47]. The Balsa compiler converts the specification into a network of so-called *handshake components* – pre-defined primitive parts that synchronise their operation using handshake signals. Then these components are replaced with Signal Transition Graphs that describe the behaviour of each individual component. These small STGs are composed according to the connections present in the original handshake component network to produce one large STG that describes the expected behaviour of the whole system. To obtain the implementation in the form of logic gates, this STG must be processed by a logic synthesis tool. However, for any practical system, the STG will be too complex to obtain the synthesis result in a reasonable time. To eliminate this problem, the STG needs to be decomposed into smaller directly synthesisable fragments. Finally, the synthesised fragments of logic must be composed to form the final system net-list.

In this example, there are several standalone tools that are involved in the different stages of the process. Additionally, some of them are interchangeable as mentioned above, e.g., the logic synthesis can be done using one of the many available alternatives. Organising the interaction between all of these tools, with their quirks and peculiarities, is a very tedious and error-prone task.

1.1.5 Interactive graphical environment

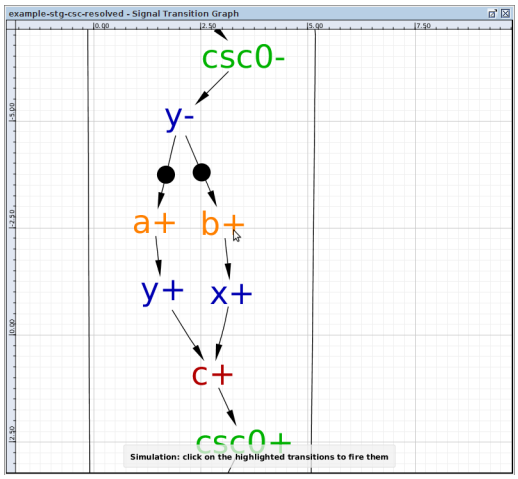


Figure 1.5: Interactive STG simulation. Note that the enabled transitions (a+ and b+) are highlighted.

An interactive graphical environment may be very helpful during the process of system design and debugging. Traditionally, graph layout tools such as Dot [7] have been used to produce the graphical representation of graph-like models such as Petri nets, STGs, or state graphs. Using this approach, it is only possible to produce a static snapshot of the system in question. However, the nature of the majority of the models is dynamic. In Petri nets, for example, transitions get enabled, fired and transfer tokens between places. While it is possible to produce a series of images of different states describing the evolution of the net, a dynamic, interactive visualisation is much more helpful.

Using tools such as PEP [11, 30, 112], it is possible to interactively simulate Petri nets. The tool will highlight the currently enabled transitions, and the user can click on them to cause them to fire (Figure 1.5). This way, the user can experiment with the net and analyse its behaviour by triggering different execution paths.

Petri nets and Petri net-derived formalisms are among the most widely used to model and analyse concurrent systems. Many natural and man-made systems can be classified as concurrent, and due to this reason numerous people from various areas of knowledge have contributed to the theory of Petri nets and to the development of relevant software packages such as PEP.

But there are also a significant number of other useful models that are similarly dynamic in nature, but have been introduced recently and are not as mature, e.g., Static Data Flow Structures (SDFS) (Chapter 5) and Conditional Partial Order Graphs [80]. Because they are relatively new, and may not be as widely applicable as Petri nets, they do not yet have a dedicated tool base. To successfully apply these models in a practical design workflow, however, an adequate tool support is extremely important. Designing and implementing a custom graphical environment to support functionality such as visual entry and interactive simulation for each of those models is not a task that is readily undertaken by researchers.

A common feature of all of the formalisms mentioned above is a static underlying graph structure augmented by a set of properties that describe the state of the system, and a set of rules that govern the evolution of those properties. By exploiting this similarity, it is possible to abstract the visualisation task from the mathematical definition of the model, allowing researchers to provide an accessible and consistent visual interface to their models without being bothered with the

implementation details beyond the model itself.

1.2 Contribution

The underlying issue of this thesis is the problem of interaction between formal models and tools that provide a framework for asynchronous circuit design. The organisation of the thesis and the way its contribution is presented is best seen as the evolution of a modelling approach where a larger system is described using a number of different formal models either to describe its parts, or to provide alternative "views" of the system in order to make it more amenable to analysis and verification.

Automated verification of asynchronous circuits using Petri nets A novel method for automated verification of asynchronous circuits is proposed. The method is based on converting the gate-level net-list of the circuit into a special type of Petri net called a *circuit Petri net*. This net is then composed with the environment specification that is given in form of a Signal Transition Graph. Verification is carried out on the compound system. If the property that is being verified is not satisfied, the Petri net-level violation trace may be re-interpreted in terms of switching activity of the gates in the original net-list. The method is successfully applied to show that a previously published version of a counterflow-style data path controller is incorrect.

Modelling, simulation and automated verification of the data path of asynchronous circuits A new high-level model called a Static Data Flow Structure (SDFS) is presented. SDFS is a token based model of asynchronous circuits involved in the data paths, and can be viewed as an analogous to the *register transfer level* (RTL) in synchronous design. The model allows advanced concurrency techniques such as *preemption*, *early evaluation* and *speculation* to be modelled. This is achieved by applying different sets of token game rules depending on the desired functionality of a particular data path fragment. The key feature of the model is the design of token game rules, specifically done in such a way that the model can be converted into a low-level Petri net for the purpose of verification. Similarly to the technique applied to the gate-level circuit models, the mapping method allows the low-level violation traces to be propagated up to the high-level model.

Interpreted Graph Models A concept of an *Interpreted Graph Model* is introduced. This concept allows to exploit similarities between various graph-like models in order to provide a generalised implementation of several important methods, such as using Petri nets for the verification of the higher-level models and implementing the visualisation and simulation logic, which helps to quickly set up new models by inheriting the basic functionality from the Workcraft framework.

Workcraft framework A software framework called Workcraft is presented. Workcraft is designed to provide a flexible common framework for the rapid development of Interpreted Graph Models, including visual entry, interactive simulation, inter-model conversion and the application of third-party analysis tools. The framework is targeted at two distinct classes of users. For system designers the framework provides the means to model a particular system using the most appropriate formalism (or different formalisms for subsystems) and apply a wide range of external tools for analysis and verification in a consistent and user-friendly fashion. The second class of users are the researchers who wish to introduce new Interpreted Graph Models. For this class of users, Workcraft provides numerous extension points that allow to customise the functionality of the framework in order to accommodate for the new formalism, while retaining the important basic features such as the GUI and the consistent interface to the external tools.

1.3 Organisation of the thesis

The thesis is organised as follows:

Chapter 1. Introduction. This chapter outlines the motivation for this work and its main contribution.

Chapter 2. Background. This chapter provides the definitions of asynchronous circuits, their properties, key stages in their design and contains an overview of the software tools relevant to those stages.

Chapter 3. Petri nets. This chapter introduces the Petri net model which plays a fundamental role in the design and verification of concurrent systems, particularly asynchronous circuits.

Chapter 4. Automated verification of asynchronous circuits using Petri nets. A method for the verification of asynchronous circuits using a special class of Petri nets is introduced in this chapter. The problems that need to be detected and eliminated in order to ensure the correct functionality of an asynchronous circuit are explained. An algorithm to convert a gate-level circuit specification into a Petri net and to compose the obtained Petri net with a specification of the environment is given. The violation of the speed independence property is formulated in terms of the Petri net reachability problem that may be solved using the well-known Petri net tools. Finally, the method is applied to verify the correctness of a number of practical circuits, including a previously published data path controller.

Chapter 5. Modelling, simulation and automated verification of the data path of asynchronous circuits. This chapter presents a new model called a Static Data Flow Structure (SDFS). The definition of the fundamental structure of an SDFS is given. Three different behavioural semantics (atomic token, spread token and counterflow) that are realised using distinct sets of token game rules are defined and their advantages and disadvantages analysed. A method of interfacing fragments of SDFS having different behavioural semantics is proposed. A Petri net verification technique that is an extension of the verification technique proposed in the previous chapter is described. A possible extension of the basic SDFS model with elements modelling the influence of the control path is discussed.

Chapter 6. Interpreted Graph Models. This chapter introduces the concept of an *Interpreted Graph Model* (IGM). A formal definition of an IGM is given. An general-purpose algorithm that uses the IGM concept to generate a graphical representation of any graph-like model is detailed. Several extensions to the algorithm are proposed, such as the use of an additional IGM to produce the graphical representation without enforcing one-to-one correspondence between the mathematical and graphical objects. A generalised STG mapping algorithm and STG-based verification technique is also proposed.

Chapter 7. Workcraft framework. A software framework called Workcraft is presented in this chapter. The general idea behind the tool is explained. A comparison with previously existing

similar tools is given. The software architecture of the tool is described.

Chapter 8. Use cases. This chapter gives an overview of several practical use cases where the Workcraft framework and the IGM concept were instrumental. Among those cases are:

- Verification of an asynchronous data path controller;
- Verification of two different types of arbiters;
- Design and debugging of the SDFS model;
- Synthesis of asynchronous controllers using the Conditional Partial Order graph model;
- Modification of the workflow of Balsa asynchronous circuit synthesis system.

Chapter 9. Conclusions. This chapter concludes the thesis, providing an evaluation of the contribution and considering the ways of further development of the theoretical concepts, practical methods and pieces of software that are discussed throughout the thesis.

Appendix A. Workcraft user manual. This appendix gives an overview of the basic concepts of Workcraft's user interface and is aimed at those users who would like to use Workcraft for a particular practical application.

Appendix B. Extending Workcraft. This appendix gives several practical examples of extension classes in Workcraft. A step-by-step instruction is provided explaining how to design a new Interpreted Graph Model class, how to define the way that its nodes are visualised, and how to add custom tools to the new or to the previously existing models. This chapter is aimed at those users who would like to use Workcraft as the base platform for development of their own models and/or tools.

Appendix C. Working with Signal Transition Graphs This appendix contains a tutorial on using Workcraft to create, edit an simulate the Signal Transition Graph models. It also explains how to use the interface to external tools such as Punf, PComp, MPSat, Petrify and DesiJ to carry out advanced operations on the STG models.

Chapter 2

Background

This chapter provides the basic definitions pertaining to the asynchronous circuits (delay models, operation modes, classes and common signalling protocols) as well as an introduction to the methods used in the asynchronous circuit design and a brief overview of the CAD tools implementing those methods.

2.1 Asynchronous circuits

In its basic form, an asynchronous circuit is a set of gates connected to each other through a set of wires, where a *wire* is a conducting medium that connects an output of a single gate to one or more inputs of other gates and a *gate* is an element that generates its output signal based on a logical function that depends on the level of input signals. The simplest gate, an inverter, produces the inversion of its input signal as its output, that is logical one if the input is logical zero, and logical zero if the input is logical one. The process of switching from one value of the output signal to another is never instantaneous due to a number of limiting factors, such as the finite propagation speed of the electric signal in the wires and the non-zero capacitance of the wires and the transistors that the circuit is built from.

The phenomenon of delays is responsible for numerous potential problems that need to be accounted for during the design process.

2.1.1 Delay models

There are two generally accepted delay models. An element exhibiting *pure delay* eventually produces all expected output signal transitions regardless of the shape of the input signal's waveform. The behaviour of an *inertial delay* element may be dependent on the input waveform, in particular it requires the input signal to stay on the same level for a certain period of time before an output signal transition may occur. If the input pulse is too short, it will not result in a change in the output.

The length of the delays are characterised using one of the following timing models. In the *unbounded delay* model, the delay time is assumed to be finite (i.e. the output signal transition will eventually occur), but the upper bound is unknown. The *bounded delay* model assumes that the transition will occur within a known time interval. The *fixed delay* model assumes that the delay time is always the same.

2.1.2 Operation modes

The protocol that a circuit environment uses to interact with the circuit is called *operation mode.* In the *input-output* operation mode, the environment may produce a transition of an input signal in response to any output signal transition. In the *fundamental mode* the environment is only allowed to change an input signal if the circuit is stable, i.e. no further output transitions may be produced given the current state of inputs.

The fundamental mode is further divided into several sub-modes as follows. If only one input signal transition is allowed to occur before the environment has to wait for the circuit to become stable again, such an operation mode is called *single input change fundamental mode* (SIC). If multiple input signals are allowed to change, the mode is called the *multiple input change fundamental mode* (MIC). A circuit operating in the MIC mode is usually faster than a similar circuit operating in the SIC mode, however it is much more difficult to design a circuit operating correctly in the MIC mode. To receive certain benefits of the MIC mode without overly complicating the circuit implementation it may be helpful to group the input signal transitions into sets called *bursts*. Inputs in a burst may arrive in arbitrary sequence, however it is guaranteed that no signal transitions from another burst arrive until the previous burst completes. The circuit is allowed to

stabilise between bursts. A circuit operating in this fashion is called a *burst-mode* circuit.

2.1.3 Classes

Asynchronous circuits are often classified based on their tolerance to delays in various elements of the circuit. The most robust class is called *delay insensitive* (DI). Circuits that belong to this class are guaranteed to function as intended regardless of delays in gates and wires. This implies the highest tolerance to environmental and manufacturing process variations. There are not many practical circuits that can be constructed as DI [116, 39] using the standard set of simple gates.

A less robust, but more practical class is called *quasi delay insensitive* (QDI) [73]. QDI circuits are characterised by the existence of *isochronic forks*, branching wires where delays in different branches are assumed to be equal for all practical purposes. Except for this assumption, QDI circuits must still behave in the same robust fashion as DI with respect to the wire and gate delays.

Speed independent (SI) [83] circuits are designed to function correctly given any (bounded) delay in gates. Wire delays are assumed to be negligible.

Self-timed circuits [102] are constructed from a set of sub-circuits. There is no restriction on the internal implementation of each sub-circuit, but they must be tolerant to any delay in the external communication channels.

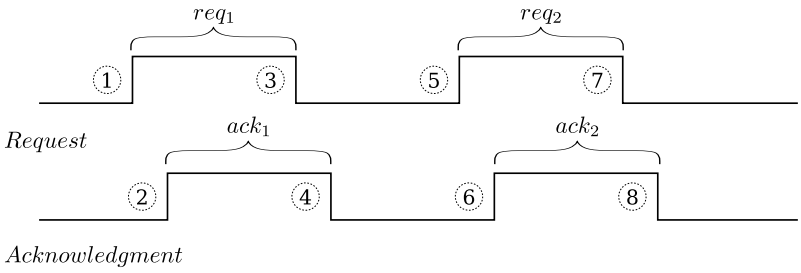
In each of the above cases, the circuit is assumed to function in the input-output mode.

2.1.4 Handshake protocols

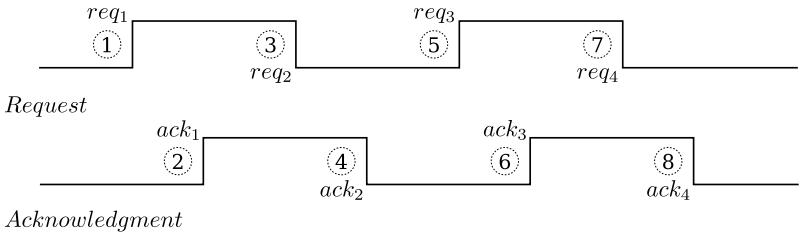
The communication between the components of an asynchronous circuit is typically based on two types of signals, called the *request* and the *acknowledgement*. The request signal is issued by a component to initiate a certain procedure. When the procedure is complete, the component receives the acknowledgement signal and may act on the results.

Generally, the exchange of these signals follows a strict protocol called *a handshake.* A handshake consists of one component issuing a request signal to another component, and waiting until that component responds with an acknowledgement, which concludes the handshake. The requests and the acknowledgements are not allowed to interleave.

There are several ways to interpret the transitions of these signals that are called the *handshake*



(a) four-phase protocol



(b) two-phase protocol

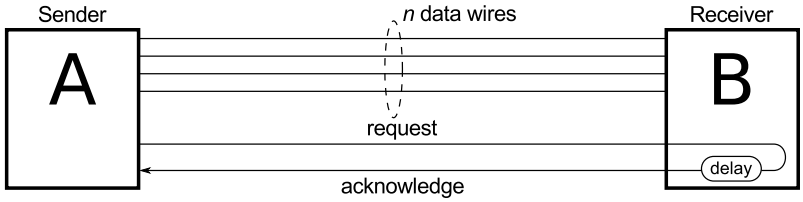
Figure 2.1: Handshake protocols

protocols. Two most commonly used protocols are the four-phase and two-phase. The four-phase protocol, also called return-to-zero (RTZ), is shown in Figure 2.1a. The numbers in circles represent the ordering of signal transitions. The delay between the two consecutive transitions is not specified. In this protocol there are four signal transitions (two on the request and two on the acknowledgement) comprising a single handshake.

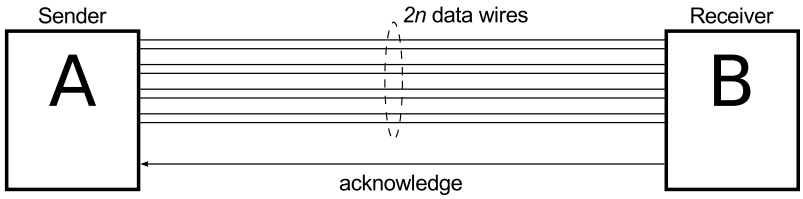
The two-phase protocol, also called non-return-to-zero, is shown in Figure 2.1b. The difference is that every transition on both the request and acknowledgement wires indicates a new event. Although this protocol appears more compact the control circuits are usually smaller for the four-phase version [61].

2.1.5 Data protocols

As with the control signals, there are several ways to organise the transfer of data. The most popular protocol is called *bundled data*. In this protocol, the data is transferred over an n-bit wide data bus and two additional signals, *request* and *acknowledge,* are used to negotiate data transfer and to signal its completion (Figure 2.2a). This protocol is the most efficient one in terms of wires used per bit of data, however it relies on an assumption that the control signals propagate faster



(a) Bundled data



(b) Dual-rail

Figure 2.2: Data protocols

than the data signals.

In practice, the acknowledgement signal is often generated by routing the request through a delay element. The length of the delay is determined at design time in such a way that it guarantees that the computations on the receiving end will complete and the data will no longer be needed by the time the acknowledgement signal is produced. Although this approach is optimal with respect to the number of wires, the delay should be long enough to guarantee completion in the worst-case scenario.

The common alternative to the bundled-data approach is dual-rail encoding. In this case, a bit of data is encoded with using two wires. A standard dual-rail encoding convention is as follows:

00	data not valid (“spacer”)
10	zero
01	one
11	not used

In this case, for an n-bit data value, the link between sender and receiver must contain 2n+1 wires: two wires for each bit of data plus another bit for the acknowledgement signal (Figure 2.2b). In a standard four-phase variant of the dual-rail protocol, sending a bit requires the transition from the spacer state to either the valid one or valid zero state and then, upon receiving the acknowledgement, the transition back to the spacer state. The acknowledgement wire must be reset prior

to a subsequent transmission of a valid data bit.

Compared to the bundled data protocol, dual-rail protocol is more robust as it does not rely on any timing assumptions, however the additional logic that detects transitions between spacers and valid data values (called *completion detection* logic) may incur large overheads.

M-of-n codes [122] are a more general encoding scheme in which data is represented using n wires, m of which are set to an active level. The simplest example is 1-of-2 code which is called dual-rail and is discussed above. An important feature of all m-of-n encodings is their balanced power consumption which improves the security of the circuit with respect to power analysis attacks [71]. In particular, switching from a spacer to any code word consumes the same amount of power due to the symmetry between rails.

2.2 Asynchronous circuit design paradigms

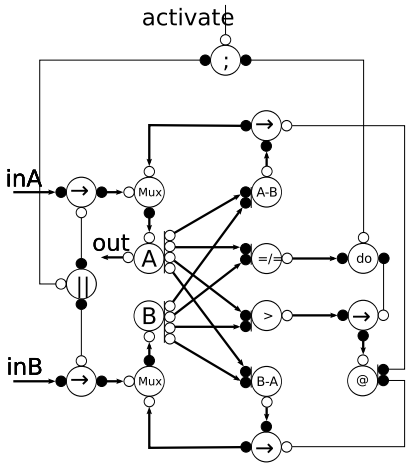
The majority of the integrated circuits designs are called synchronous because they rely on a global clock signal to synchronise the changes of states throughout the circuit. The global clock allows hiding certain features of the underlying technology from the designer and providing a highly abstract, discrete view of the system (called the register transfer level, or RTL) where all state transitions can be viewed as occurring simultaneously.

This design approach relies on the assumption that the clock signal is distributed evenly, without delays or phase shifts, to all parts of the circuit. When the circuit grows more and more complex, however, it becomes difficult to satisfy this assumption. The clock distribution networks grow to constitute considerable parts of such circuits and draw significant amount of power. In addition, dependency on the clock signal of particular frequency makes interfacing circuits that were designed for different clock frequencies very problematic.

Asynchronous circuits are naturally composable and avoid the clock distribution overheads. Due to the absence of global clock, however, the problems of local synchronisation, relative timing and hazards become prevalent and have to be dealt with. Although in many historical cases asynchronous controllers have been designed manually by experienced designers, it is clear that for any large scale application automated methods are required to ensure adequate designer efficiency [42].

```
input (a,b);
while a = b do
   if a > b then a ← a − b;
            else b ← b − a;
output (a);
```

(a) GCD algorithm in Balsa language



(b) GCD implementation in handshake components

Figure 2.3: An implementation of the greatest common divisor (GCD) algorithm in Balsa

A number of distinct techniques have been developed to meet that requirement. These techniques are discussed briefly in the rest of this section along with an overview of the tools that implement them. These are the tools that come mostly from academia and are available in the public domain. Some of the tools from larger commercial packages, such as Synopsys, Cadence or Mentor Graphics (packages targeted primarily at the design process of synchronous circuits) are widely used as a back-end for tasks such as circuit layout and routing in the asynchronous circuit design groups, but they lack the capacity to provide a complete workflow for the design of asynchronous circuits.

In this section of the thesis only those tools that were created to solve problems directly related to asynchronous circuit design will be reviewed, as one of the major contributions of this thesis is a software framework specifically designed to integrate those useful, but scattered tools. At the time of writing, several of the tools, namely Petrify, Punf, MPSat and DesiJ have been tightly integrated with Workcraft.

2.2.1 Direct mapping and syntax-driven translation

In direct mapping, a representation of the system that is given as a graph is translated into a gate-level circuit in such a way that the graph nodes correspond to the circuit elements and graph arcs correspond to the connecting wires [106]. Direct mapping can be applied at various abstraction

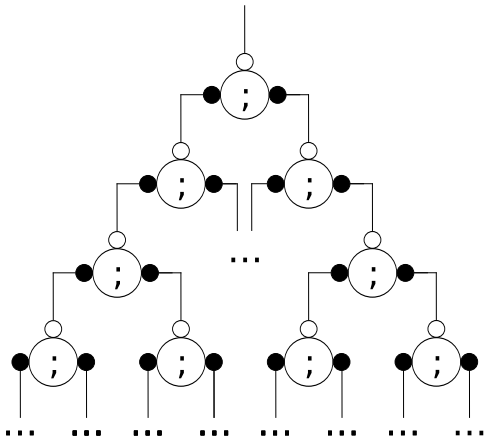


Figure 2.4: A tree of sequence elements

levels. An example of high-level direct mapping is syntax-driven translation. It is a widely used design paradigm in which a system specification is written in a high-level programming-like language. This specification is then translated into a network of handshake components, pre-designed control modules that directly implement the basic language constructs [47, 119, 88, 27].

The major advantages of the syntax-driven approach are the low algorithmic complexity of the translation process and the transparency that is provided by a strict one-to-one mapping from language constructs to the components that implement them [91]. The latter gives the designer a greater ability to control the features of the resulting circuit and decide the trade-offs between performance, area and power consumption, compared to methods such as logic synthesis, while the former allows very large systems to be successfully designed. Another notable advantage is that small changes in the source description generally result in small, predictable changes in the resulting circuit.

The main drawback of this method is the considerable overheads required to implement the control path. Because the translation is done directly from the syntax parsing tree, the control path often contains redundant constructs that may be slower than the computational blocks that process data. This causes stalls in the data path while the control path catches up [91]. An example of such redundant control path is shown in Figure 2.4. When such a tree of sequence elements is activated, the components that are connected to the leafs of the tree will simply be activated in sequence. The control signals, however, will have to travel up and down the branches between the sequence components, performing handshakes that are in this case redundant, which will hurt the

overall performance of the circuit.

Prominent examples of this approach is the Tangram language with its “silicon compiler” [119], and Balsa, an asynchronous circuit synthesis system [47]. In Figure 2.3a, a Balsa language specification of a simple algorithm (finding a greatest common divisor) is shown next to the network of handshake components that is generated by the Balsa compiler from this specification (Figure 2.3b).

As an alternative to the direct mapping from a special language, several techniques use Petri nets as an intermediate specification format. The Petri nets are extracted from a traditional hardware description language, such as Verilog, VHDL or SystemC. The control path and the data path are optimised and synthesised separately [34]. This allows different mapping and optimisation techniques to be applied to the different parts of the circuit. A method proposed in [106] further splits the control path Petri net into a device and an environment, which synchronise via a communication net that models wires. The device is represented as a tracker and a bouncer. The tracker follows the state of the environment and provides reference points to the device outputs. The bouncer interfaces to the environment and generates output events in response to the input events according to the state of the tracker. Such architecture is easily mapped into a gate net-list.

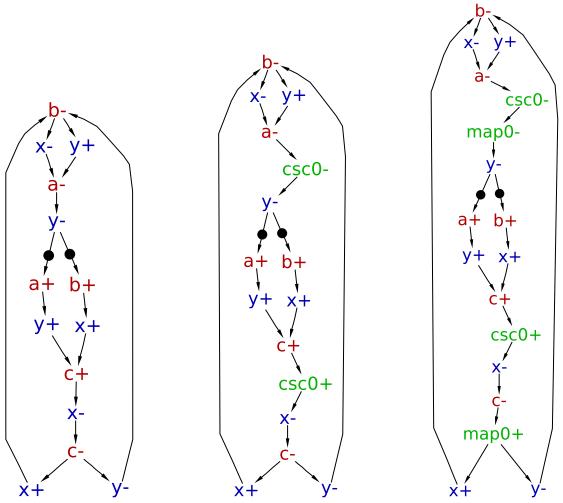
The Gate Transfer Level (GTL) method [105] is a direct mapping method that is applied at the very low level of individual gates. In this method the gates of a standard RTL net-list are replaced by pipeline stages. Each stage contains the gate itself, a register to buffer the output, and a controller that implements communication of the stage with its neighbours.

2.2.2 Logic synthesis

Logic synthesis is a method for automatic construction of asynchronous circuits from the specification of the expected behaviour given by the designer. It is based on the construction of a state graph, in which each reachable state is assigned a binary code that holds the value of each signal. This allows the generation of a circuit using state-of-the-art Boolean minimisation techniques.

Generally, the specification is given in the form of a Signal Transition Graph. To produce a circuit from an STG, the following steps are required [68]:

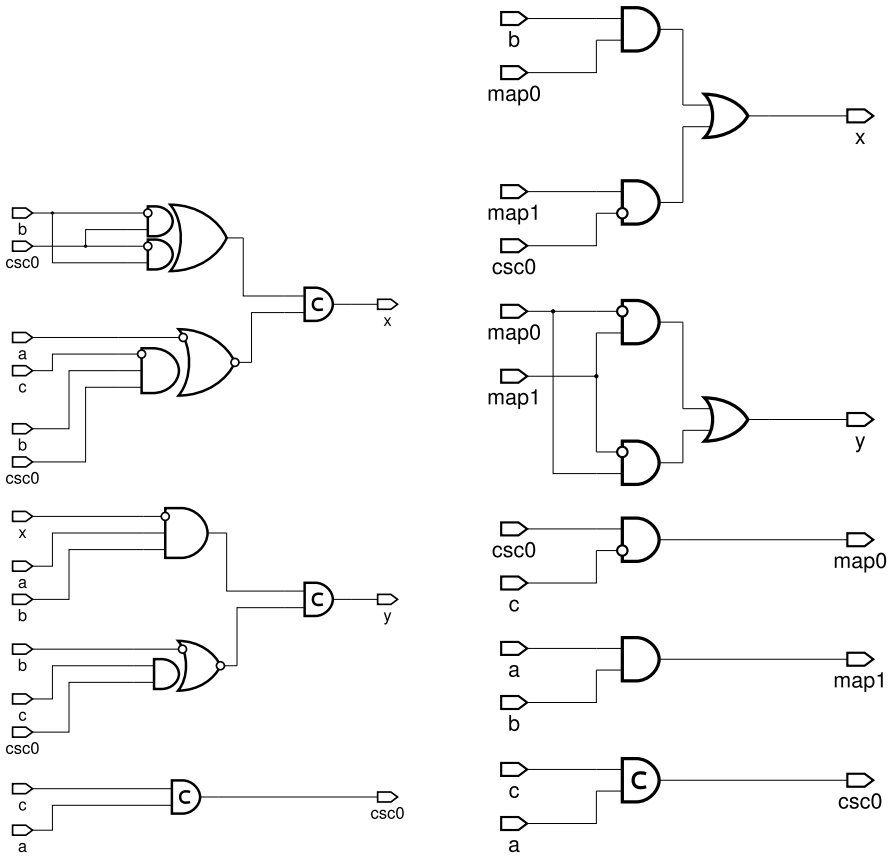
1. Checking the necessary conditions for the implementability of the given STG as a logic



(a) Original STG specification

(b) Modified STG specification with complete state coding (CSC)

(c) Modified STG specification with CSC and mappable into simple gates



(d) Synthesis result (complex gates)

(e) Synthesis result (technology mapped)

Figure 2.5: Logic synthesis

circuit;

2. Modifying, if necessary, the initial STG to make it implementable;

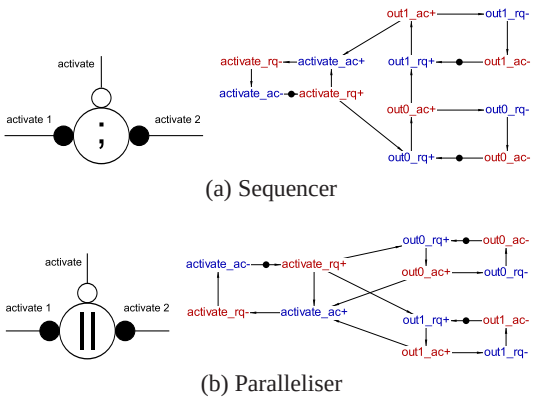
3. Finding the appropriate Boolean covers for the next-state functions of output and internal signals and obtaining them in the form of Boolean equations for the (complex) logic gates of the circuit;

4. Mapping the Boolean equations of the complex logic gates onto the set of standard gates available in the gate library, preserving the speed-independence (logic decomposition).

The above steps are illustrated in Figure 2.5 (as performed by the tool Petrify [41]). In this example, Figure 2.5a is the original STG specification, which is not immediately implementable because it does not satisfy the Complete State Coding (CSC) condition (which means that there are two semantically different states that share the same binary encoding of signal states). Figure 2.5b is the STG modified by the tool to have CSC, which allows a circuit to be derived from it (Figure 2.5d). Note that the added signals, e.g., *csc0*, are treated as internal signals (as opposed to inputs or outputs), and therefore the externally observed behaviour of the STG is not changed by this modification. The circuit shown in Figure 2.5d is composed out of so-called complex gates that implement non-trivial Boolean functions in a speed-independent manner. Such gates, however, are unlikely to be available in the industrial gate libraries. In order to make the circuit implementable in hardware, an additional step is required to map the Boolean equation obtained during the previous step onto the set of gates available from the library. Figure 2.5c is the final STG transformation performed by Petrify to produce the final circuit built out of simple 2-input gates, shown in Figure 2.5e.

Petrify performs all of these steps with the help of a reachability graph that is extracted from the initial STG specification (in the form of a Binary Decision Diagram (BDD) [21]). For highly concurrent STGs the reachability graph can be prohibitively large due to the state space explosion problem. This limits the practical size of circuits that can be synthesised using Petrify.

An alternative technique, Petri net analysis based on causal partial order semantics (in the form of Petri net unfoldings), can also be applied to the circuit synthesis problem. Experimental results produced by the MPSat logic synthesis tool [12], which works on Petri net unfoldings, show



(a) Sequencer

(b) Paralleliser

Figure 2.6: Handshake components and their corresponding STGs

significant performance improvements and more efficient memory usage when compared to the methods based on reachability graphs, while producing similar solutions [68]. Although MPSat is able to handle larger specifications than Petrify, it still suffers from the state space explosion problem and is therefore also limited to relatively small specifications.

To tackle complex specifications, the initial STG has to be broken down into smaller fragments directly synthesisable by the aforementioned tools. An efficient structural STG decomposition method, specifically designed for synthesis of large asynchronous controllers, is described in [99].

Several other synthesis techniques work with different specification formats, e.g., [54, 115].

Compared to the direct mapping method, logic synthesis usually produces highly efficient solutions, but is not applicable to large system specifications due to extremely high computational cost. Additionally, small changes to the specification can result in unexpectedly large and unpredictable changes in the resulting circuit, making it hard for the designer to fine-tune the result.

2.2.3 Mixed approach

The syntax-directed translation method greatly enhances the designer's productivity, but has several important drawbacks, of which the control-path overhead is the most decisive. The controllers obtained by syntax-directed translation are usually far from optimal, because the pre-designed components are required to implement their declared protocols fully and strictly in order to be reusable in all possible circuit configurations. However, it is often the case that a significant part of their functionality becomes redundant due to the peculiarities of a specific configuration, e.g. in many cases full handshaking between the components can be avoided.

This redundancy can be eliminated by replacing the manually designed gate-level implementation of the high level components with an implementation synthesised automatically. The goal is to produce an efficient implementation of a set of interconnected handshake components, as opposed to the composition of the pre-designed general implementations of individual components [89, 129]. This approach is often called *resynthesis*.

One method to accomplish this is to design a Signal Transition Graph describing the expected behaviour of each of the handshake components control circuits (Figure 2.6). Then, given a handshake component network produced by the compiler, each component is replaced with its STG specification. The separate component STGs are composed together via an operation called *parallel composition* to form a complete system STG [89]. An optimal gate-level implementation can then be automatically produced from the STG using tools such as petrify [41], SIS [103] and MPSat [68]. Automatic synthesis can become problematic when the size of the STG becomes too large: at the time of writing, the largest STG size that the synthesis tools can process in an acceptable time is about 100 signals. The impact of this can be lessened by including STG decomposition tools, such as DesiJ [99] into the workflow. The decomposition tool is able to break the large optimised STG down into several smaller STGs that are synthesisable in reasonable time. A schematic of this workflow, as implemented in Workcraft, is shown in Figure 1.4 (compare to the standard workflow used in Balsa, shown in Figure 2.7).

It should be noted that although this method can significantly improve the efficiency of the control circuits associated with the handshake components, it is not applicable to the majority of the data path elements, such as registers and combinational logic, because their behaviour is too complex to be automatically synthesised. These elements of the circuit are unaffected by the resynthesis method, and therefore it should only be applied if the control path is actually the bottleneck.

2.3 CAD tools for the design of asynchronous circuits

This section contains an overview of the notable tools that are often applied to the design of asynchronous circuits.

2.3.1 Direct mapping/syntax-driven translation tools

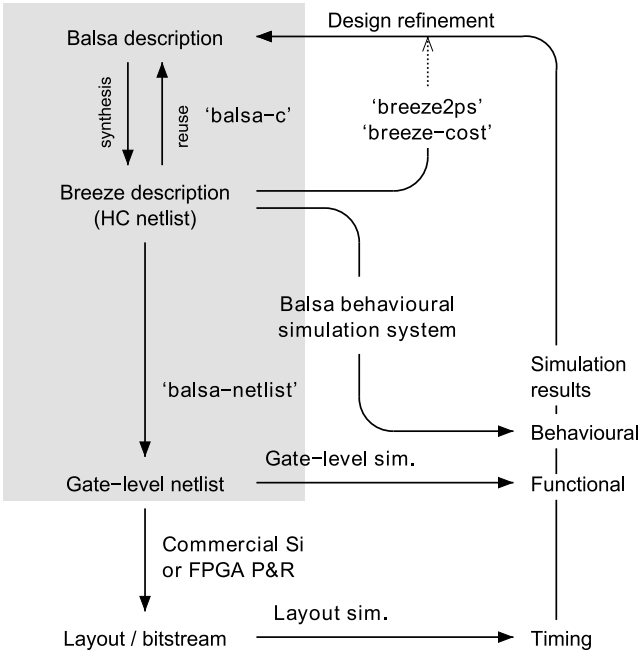


Figure 2.7: Balsa design workflow

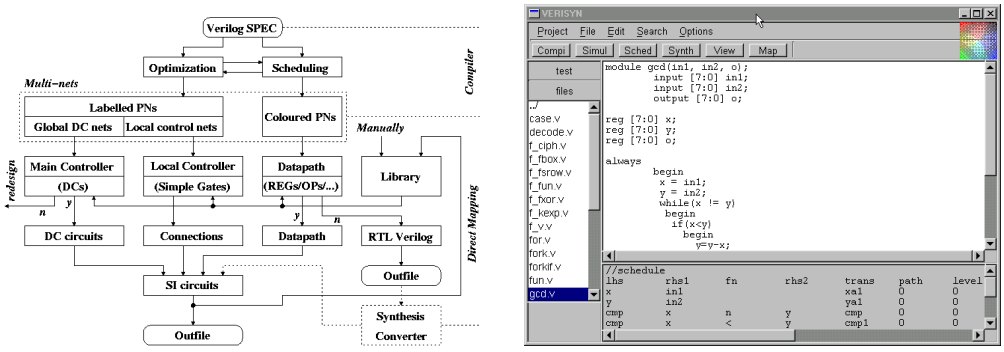
Balsa Balsa is an asynchronous circuit synthesis system [47] developed over a number of years at the APT group of the School Of Computer Science, The University of Manchester.

Balsa is built around the handshake circuits methodology and can generate gate level net-lists from high-level descriptions written in the Balsa language. Both dual-rail (QDI) and single-rail (bundled data) circuits can be generated.

The approach adopted by Balsa is that of syntax-directed compilation into communicating handshaking components (Figure 2.7) and is similar to the Tangram system of Philips [119] (later distributed as Haste by Handshake Solutions). The advantage of this approach is that the compilation is transparent: there is a one-to-one mapping between the language constructs in the specification and the intermediate handshake circuits that are produced. It is relatively easy for an experienced user to envisage the architecture of the circuit that results from the original description. Incremental changes made at the language level result in predictable changes at the circuit implementation level.

Although this property greatly enhances designer productivity, the controllers obtained by syntax-directed mapping are usually not optimal, because the pre-designed components are re-

quired to implement their declared protocols fully and correctly in order to be reusable in all possible circuit configurations. It is often the case that a significant part of their functionality becomes redundant due to the peculiarities of the specific configuration, e.g. in many cases full handshaking between the components can be avoided. There have been attempts [89, 129] to enhance Balsa's synthesis results by introducing a logic synthesis step for the control path instead of direct mapping.



(a) VeriSyn design flow (b) VeriSyn interface

Figure 2.8: VeriSyn

VeriSyn In the VeriSyn tool [104, 18] the high-level language descriptions (Verilog, VHDL) are initially compiled and converted into an intermediate Petri net format. The intermediate format is subsequently used as a medium for direct mapping to asynchronous circuits. The control nets are split into two types for mapping: global control nets which are used for direct mapping to David Cells and local control nets for mapping to simple control gates.

The intermediate format is subsequently passed to optimisation tools and mapping tools where it is directly mapped into asynchronous data path and control circuits using David Cells. Hardware components are selected from a basic library for mapping. An RTL Verilog description can also be output to a synthesis converter: i.e. a synchronous synthesis tool (e.g. Synopsys Design Compiler) generates circuits which are converted back to asynchronous circuits using a tool called VeriMap. Finally logic optimisation tools are applied to generate speed independent (SI) circuits.

The design flow schematic and a screen shot of the tool's interface are shown in Figure 2.8.

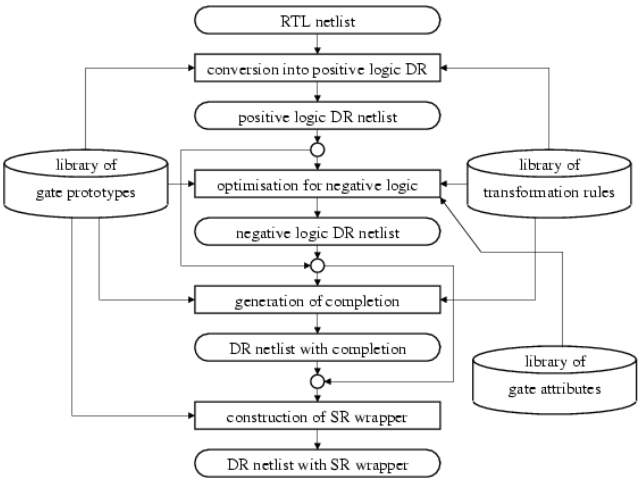


Figure 2.9: VeriMap design flow

VeriMap The VeriMap design kit [17] converts single-rail RTL net-lists into dual-rail circuits which are resistant to Differential Power Analysis (DPA) attacks. VeriMap design kit successfully interfaces to the Cadence CAD tools. It takes as input a structural Verilog net-list file, created by Cadence Ambit (or another logic synthesis tool), and converts it into dual-rail net-list. The resulting net-list can then be processed by Cadence or other EDA tools. All Design For Testability (DFT) features incorporated at the logic synthesis stage are preserved.

The VeriMap design flow is shown in Figure 2.9.

2.3.2 Logic synthesis tools

Pipefitter Pipefitter [31] is a tool chain that implements a fully automated synthesis flow for asynchronous circuits. It can be used to design simple asynchronous microcontrollers using RTL-like Verilog HDL as the input format.

Pipefitter directly synthesises the control path as a hazard-free standard cell net-list, and uses a genetic algorithm to perform binding and multiplexer optimisation for the data path. It produces a synthesisable Verilog specification for the data path, as well as a set of scripts driving both its synthesis and timing analysis by state-of-the-art commercial synchronous RTL and logic synthesis tools. The automated insertion of matched delays completes the logic design, and hands off the net-list to the standard cell-based layout tools. The schematics of the tool's design flow is shown in figure 2.10.

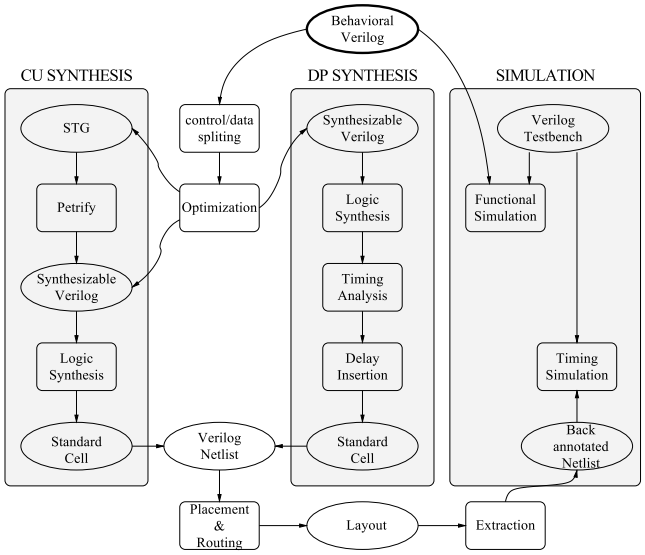


Figure 2.10: Pipefitter design flow.

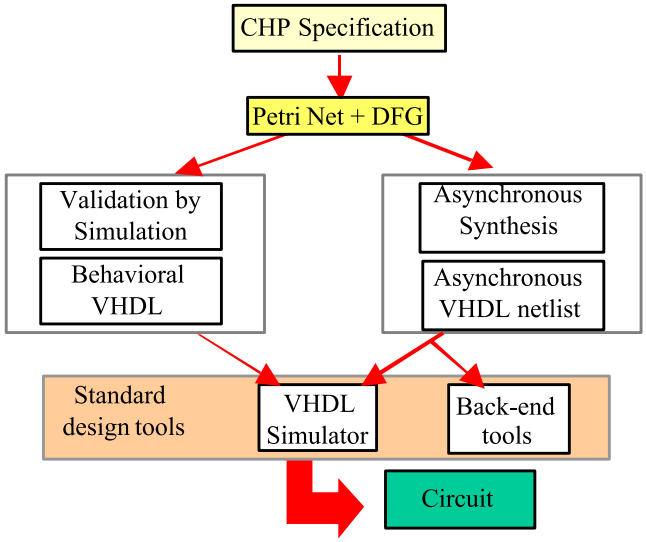


Figure 2.11: TAST design flow

TAST (Tima Asynchronous Synthesis Tools) TAST (Tima Asynchronous Synthesis Tools) [46] is an open design framework devoted to asynchronous circuits. It consists of three parts: a compiler, a synthesiser and a simulation-model generator. TAST offers the capability of targeting several outputs from a high level, CSP-like description language called CHP (Communicating Hardware Processes). The compiler translates CHP programs into Petri nets (PN) associated to Data Flow Graphs (DFG). The synthesiser generates asynchronous circuits from the PN representation of the CHP programs (Figure 2.11). It provides a set of rules to guarantee that

PN-DFG graphs are synthesisable into QDI circuits.

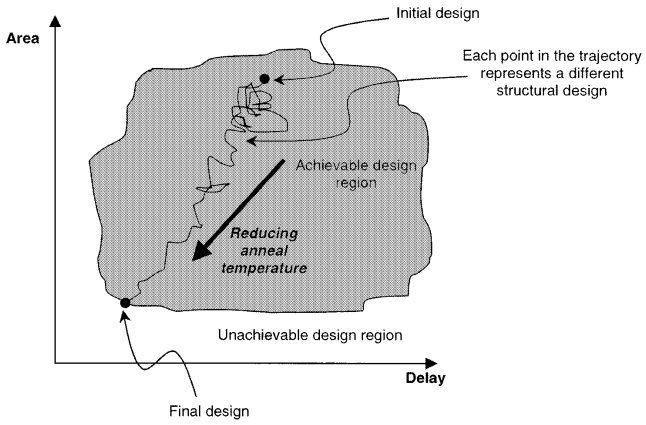


Figure 2.12: MOODS design space traversal algorithm

MOODS (Multiple Objective Optimisation for Data and control Synthesis) The multiple objective optimisation for data and control synthesis (MOODS) [25] system implements global optimisation of a design data flow and control graph by the repeated application of small reversible (behaviour preserving) transforms under the control of a simulated annealing algorithm. The system is designed to support overall optimisation with respect to various criteria, such as area, delay, and power dissipation. The operation of the system is usually characterised by a design trajectory – the entire structural design is represented by its values of area, delay, and power dissipation and these numbers form the coordinates of a point in design space. The algorithm moves the design through this space as shown in Figure 2.12 from an initial point created from a line-by-line translation of the user-defined goal (typically, minimum area, delay and dissipation). The speed of this process allows the designer to interactively study the trade-offs possible between the three criteria.

Petrify Petrify [41] is a tool for manipulating concurrent specifications and synthesis and optimisation of asynchronous control circuits. Given a Petri Net (PN), a Signal Transition Graph (STG), or a Transition System (TS) it generates another PN or STG which is simpler than the original description and produces its implementation in the form of a net-list of an asynchronous controller in the target gate library with the specified input-output behaviour.

For transforming a specification Petrify performs token flow analysis of the initial PN and pro-

duces a transition system (TS). In the initial TS, all transitions with the same label are considered as one event. The TS is then transformed and transitions relabelled to fulfil the conditions required to obtain a safe irredundant PN. For synthesis of an asynchronous implementation Petrify performs state assignment by solving the Complete State Coding problem. State assignment is coupled with logic minimisation and speed-independent technology mapping to a target library. The final net-list is guaranteed to be speed-independent, i.e., hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification.

Punf/MPSat Punf [55, 64, 12] is a parallel Petri net unfolder: it takes a Petri net (which may be an STG or a high-level Petri net) and produces a finite and complete prefix of its unfolding. Such a prefix is a concise representation of the net's state space and can be used for efficient model checking and, in case of STGs, synthesis of circuits. For STGs such a representation is often superior to that based on explicit state graphs and BDDs due to the fact that STGs usually contain a lot of concurrency but rather few choices. As a result, the memory requirements of synthesis algorithms based on unfoldings are very moderate.

MPSat [67, 68, 12, 65] is a tool for model-checking and for synthesis of asynchronous circuits. It works on an unfolding prefix (e.g. one produced by Punf) and has several modes of operation. Among those are model-checking (such as deadlock checking and reachability analysis), encoding conflicts detection and resolution, and logic synthesis modes. MPSat supports an expressive language called Reach [66] for the specification of reachability-like properties. It allows to formulate non-trivial reachability-like conditions in a concise and human-readable form. Internally, MPSat translates the problem into Boolean satisfiability (SAT) and employs one of the high performance SAT-solvers to obtain a solution.

Punf and MPSat are used as a back-end for a large number of Petri net- and STG-related tasks in Workcraft.

2.3.3 Analysis and verification tools

LoLA (Low Level Petri Net Analyser) LoLA (a Low Level Petri Net Analyser) [101] is a space state reduction based tool for Petri net verification. It includes a large number of available reduction techniques many of which may be applied jointly. Dedicated variations of state space

reduction techniques for several frequently used properties are available. The tool's interface is text-based and designed for integration into other tools. Standard properties (liveness, reversibility, boundedness, reachability, dead transitions, deadlocks, home states) as well as satisfiability of state predicates and CTL model checking are supported. Reduction techniques include stubborn sets, sweep line method, cycle coverage, invariant based compression and other. Most techniques may be applied in combination. In many cases, variations of a technique are used which are particularly optimised for the analysed property.

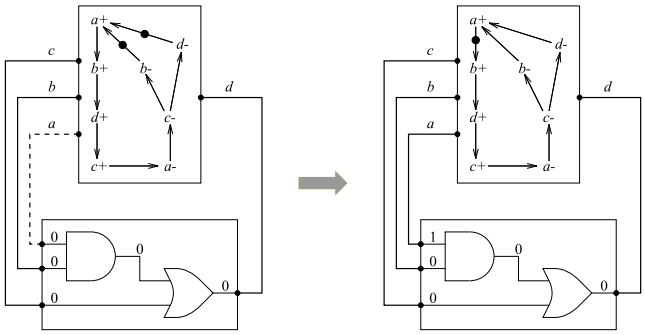


Figure 2.13: Composition of a circuit and its environment in Versify

Versify Versify [19, 97] is a tool that verifies the speed-independence of a given circuit and its specification. The specification is described as a Petri net and the circuit as a flat net-list of gates. The Petri net describes both the behaviour of the environment and the expected behaviour of the circuit. Circuit and environment are composed (Figure 2.13) forming a closed system, and the reachability analysis of such a system is performed. Both specification and circuit are modelled by Boolean functions and, therefore, the whole system can be represented and manipulated by using binary decision diagrams (BDDs). Two approaches are used: the first one uses all the variables of the circuit, whereas the second one automatically eliminates internal combinational signals. With this reduction in the number of signals, complexity is made dependent on the number of memory elements rather than on the number of signals.

A circuit is deemed to be correct if it does not generate any unexpected behaviour following any possible input sequences that correspond to the environment specification.

zeta zeta [75] is an asynchronous circuit verification tool that checks the conformance of sequences of input and output signal changes (traces) in the circuit against a Petri net specification. The tool is based on an algorithm that uses zero-suppressed binary decision diagrams (ZBDDs) [79], which are a variant of BDD that is specifically optimised for the representation of binary vectors that contain only a small number of ones. Because Petri nets often have sparse state spaces, they can be handled very efficiently using a ZBDD representation [128]. Benchmark results of the tool compare favourably to those of Versify.

GENET (GEneralised NET Synthesis) GENET (for GEneralised NET Synthesis) [36] is a tool for mining and synthesis of Petri nets from transition systems. The tool is based on the theory of regions. The input of the tool is a transition system from which it can generate a Petri net with a reachability graph that is either bisimilar to the input transition system (synthesis) or is a a superset of the input transition system's language (mining).

GENET allows the user to transform a system with a state-based representation into a system with event-based representation. If the system in question exhibits a high level of concurrency, event-based representation is often more efficient for visualisation and model-checking.

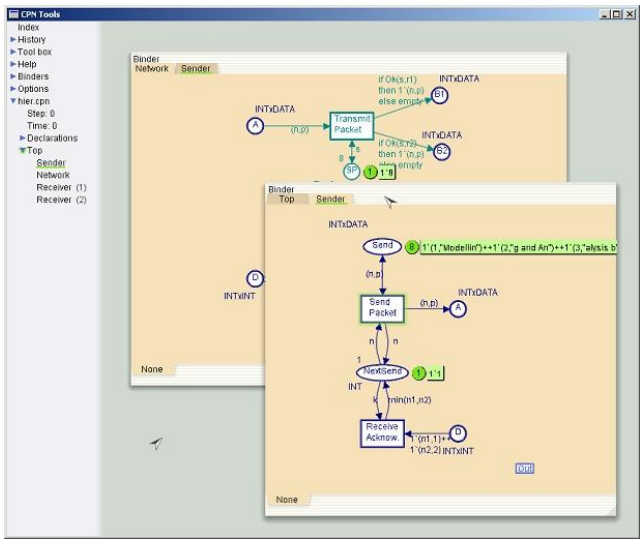


Figure 2.14: CPN Tools GUI

CPN tools CPN Tools [4] is a set of tools for editing, simulating and analysing Coloured Petri Nets. The GUI (Figure 2.14) is based on advanced interaction techniques, such as tool glasses,

marking menus, and bi-manual interaction. Feedback facilities provide contextual error messages and indicate dependency relationships between net elements. The tool features incremental syntax checking and code generation which take place while a net is being constructed.

The simulator handles both untimed and timed nets. Full and partial state spaces can be generated and analysed, and a standard state space report contains information such as boundedness properties and liveness properties. The functionality of the simulation engine and state space facilities are similar to the corresponding components in Design/CPN, which is a widespread tool for Coloured Petri Nets [5].

Spin Spin [58, 57, 14] is a tool that primarily targets software verification as opposed to hardware verification. The tool supports a high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signalling protocols, etc. The tool detects deadlocks, unspecified receptions, race conditions, and unwarranted assumptions about the relative speeds of processes. Spin works on-the-fly, which means that it avoids the need to preconstruct a global state graph, as a prerequisite for the verification of system properties.

Spin can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.

Spin has also been applied for the analysis of Petri nets, both in a standalone tool [49] and as a part of PEP tool [52].

PEP tool (Programming Environment based on Petri nets) The PEP tool (Programming Environment based on Petri nets) [30, 112, 11] is a comprehensive set of modelling, compilation, simulation and verification components, linked together within a Tcl/Tk-based graphical user interface.

The programming component allows the user to design concurrent algorithms in an easy to use imperative language, and the PEP system then generates Petri nets from such programs. The simulation of a Petri net can even trigger the simulation of the corresponding program.

PEP's verification component contains various Petri net indigenous algorithms to check reachability properties and deadlock-freeness, as well as verification algorithms.

2.3.4 Modelling tools

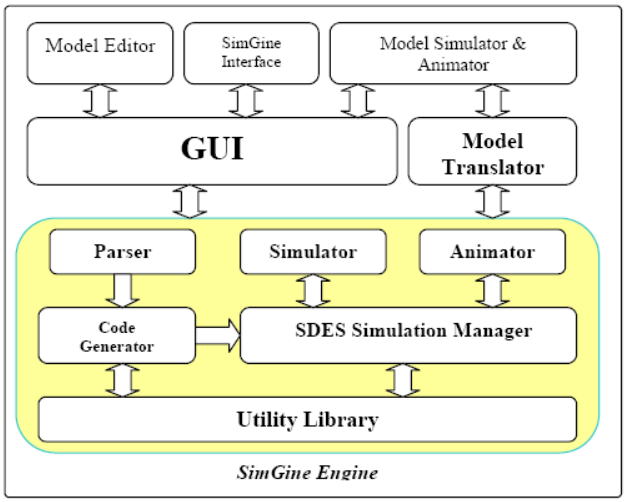


Figure 2.15: PDETool architecture

PDETool PDETool [62] is a multi-formalism modelling and simulation tool for stochastic discrete-event systems which uses a simulation engine based on a unified abstract description called SDES [130]. This modelling tool provides features for construction and translation of models into the XML-based input language of PDETool's simulation engine. Currently, some useful extensions of Petri nets have been implemented in the tool, including generalised stochastic Petri nets, stochastic reward nets, stochastic activity networks and coloured stochastic activity networks.

PDETool is designed to be extensible (Figure 2.15) to support a wide range of graphical and non-graphical formalisms.

Yasper (Yet Another Smart Process EditoR) Yasper (Yet Another Smart Process EditoR) [120] is a tool for modelling and simulating discrete processes. Yasper uses extended Petri nets as its modelling back-end. It supports manual simulation, in which the user selects execution

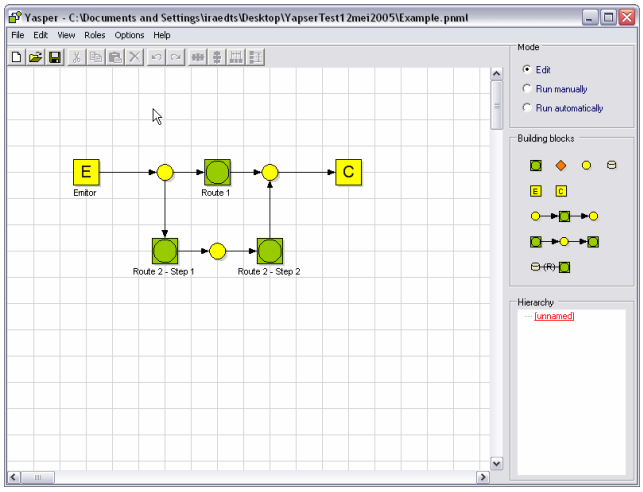


Figure 2.16: Yasper GUI

steps by clicking on the model, and automatic simulation, which randomises the choice of steps and produces an aggregated report with relevant statistics. Yasper models directly support some popular Petri net extensions, and can emulate several other techniques, such as state machines, flowcharts, UML 1 activity diagrams, and EPCs.

The Moebius framework The Moebius framework [44] is an environment for implementation of multiple modelling formalisms and solution techniques. Models expressed in formalisms that are compatible with the framework are translated into equivalent models using Moebius framework components. This translation preserves the structure of the models, allowing efficient solutions. The framework is implemented in the tool by a well-defined abstract functional interface. Models and solution techniques interact with one another through the use of the standard interface, allowing them to interact with Moebius framework components, not formalism components. This permits novel combinations of modelling techniques to be used for research.

Draw-Net The Draw-Net Modelling System (DMS), a framework supporting the design and the solution of models expressed in a graph-based formalism. The system is characterised by an open architecture and includes an XML based language family that can be used to define existing as well as new formalisms, and multi-formalism models expressed through such formalisms. The idea behind Draw-Ne, that differentiates it from the other approaches, is the possibility of easily adding new formalisms via a GUI, favouring the reuse and integration of existing tools for solving

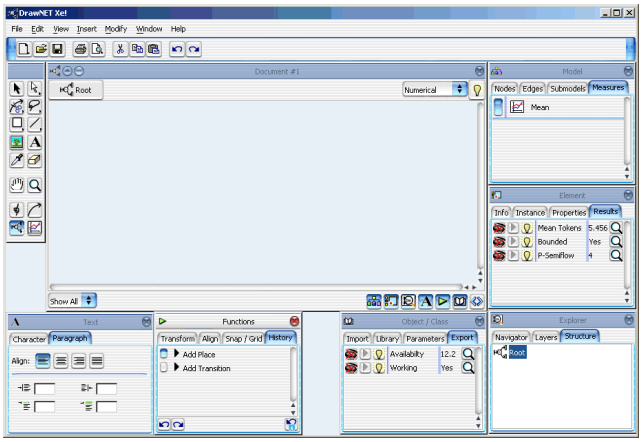


Figure 2.17: Draw-Net GUI

multi-formalism models, rather than the creation of new tools.

2.4 Conclusions

A number of fundamental definitions related to asynchronous circuits (circuit classes, operation modes, control and data protocols) were given in this chapter. The most popular approaches to the design of asynchronous circuits were discussed. An overview of the tools implementing stages in the design workflow was included.

Chapter 3

Petri nets

The process of *communication* is a key concept that is the basis of any information-driven system. The more complex a system becomes, the more intricate is the communication between its parts. Failure in communication between the smallest components can quickly lead to the malfunctioning or even collapse of the whole system. Therefore, a system designer must have a very good understanding of all the possible ways that the components could interact. It is easily seen that for any practical system the number of such possibilities is unmanageable without automated tools. For example, a tool may be able to prove that the system will not behave in any unwanted manner given every possible combination of control actions, or, if a behaviour is found violating the required system property, assist the designer by identifying the sequence of events leading to the undesired state. For this purpose, the system in question must be described using a formally defined model. The choice of the model is generally based on two characteristics: expressive power and ease of analysis, i.e. the possibility and computational feasibility of verification of essential properties. Unfortunately, there is always a trade-off between these features.

A Petri net is a mathematical model that in many ways hits the "sweet spot" in this trade-off. Petri nets are expressive just enough to model the most important features of a concurrent system, and yet are amenable to automated analysis techniques [68, 67, 49, 30, 84, 101]. Petri nets allow modelling major aspects of behaviour of such systems, including concurrency, causality and conflict [126]. Modern analysis tools are able to exploit the true concurrency representation in Petri nets (as opposed to, for example, interleaving representation in automata) to drastically reduce the

size of data required for state space representation, and hence memory and computational cost of verification process. Moreover, Petri nets have a very simple and intuitive graphical representation which is very helpful for human understanding.

Petri nets are, of course, not without drawbacks. Due to their simplicity, the size of the net required to model a system with complex behaviour can be very large, quickly becoming unobservable for the designer. To overcome this limitation, a designer may be presented with a higher-level view of the system, where the blocks of the underlying Petri net are represented as compact high-level objects, while the Petri net itself is used "behind-the-scenes" for verification tasks. This approach is one of the fundamental ideas behind this thesis.

3.1 Definitions

Definition 3.1. A *Petri net (PN)* is a quadruple $N = \langle P, T, F, m_0 \rangle$, where P is a finite non-empty set of places, T is a finite non-empty set of transitions, $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation between places and transitions and m_0 is the initial marking. A pair $a \in F$ is called an arc. A Petri net marking is a function $m : P \to \mathbb{Z}_+$, where $m(p)$ is called the number of tokens in place $p \in P$ in the marking m. The set of places $\bullet t = \{p \in P \mid (p,t) \in F\}$ is called the preset of a transition $t \in T$, and $t\bullet = \{p \in P \mid (t,p) \in F\}$ is called the postset of t. A transition $t \in T$ is *enabled* at marking m if $\forall p \in \bullet t, m(p) > 0$. A transition $t \in T$ enabled at marking m can fire, producing a new marking m' (denoted $m[t\rangle m'$), such that

$$\begin{cases} m'(p) = m(p) - 1, p \in \bullet t \setminus t\bullet \\ m'(p) = m(p) + 1, p \in t\bullet \setminus \bullet t \\ m'(p) = m(p), p \in t\bullet \cap \bullet t \end{cases}$$

thus achieving the flow of information within the net.

Graphically, places of a PN are represented as circles (○), transitions as boxes (□), consuming and producing arcs are shown using arrows (→), and tokens of the PN marking are depicted by dots in the corresponding places (⦿) (see e.g. Figure 3.2).

A very useful extension of a plain Petri net is a labelled Petri net:

Definition 3.2. A *labelled Petri net (LPN)* is a 6-tuple $S = \langle P, T, F, m_0, \Sigma, \lambda \rangle$, where $\langle P, T, F, m_0 \rangle$ is a Petri net, Σ is a finite alphabet and λ is a function $\lambda : T \to \Sigma$ associating each transition of a Petri net with a label.

This allows for some meaningful semantics to be attached to the transitions. For example, each transition may be labelled with the name of an event. Then, having observed a sequence of transitions firing, one can judge from that a sequence of events that happened in the system modelled by the Petri net.

3.1.1 An example system: the Sleeping Barber's Shop

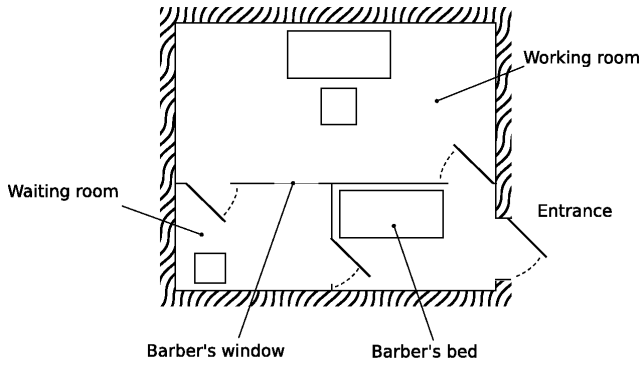


Figure 3.1: The Sleeping Barber's Shop

Let us take a variation of the Sleeping Barber Problem [45] to build an example Petri net model.

A barber (who notably likes to sleep a lot) has set up the following routine for his work. He has put a bed by the entrance to his shop which he uses to sleep at the first opportunity. If he is sleeping in his bed, the customer who enters the shop sees it and wakes up the barber, who then takes the customer to his working room and cuts his or her hair. If he is not in his bed, the customer assumes that the barber is busy with another client and goes to the waiting room, where he or she waits until the barber comes for them. The barber, having finished serving a client, takes a quick look through a small window (Figure 3.1). If there is another client waiting, he invites them to the

working room and cuts their hair. Otherwise, he proceeds to his bed using a personal door that he has set up for quickest access to the bed, and falls asleep. What is the flaw in this routine?

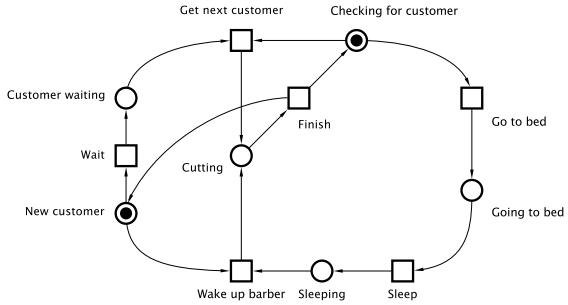


Figure 3.2: Petri net model

A Petri net that models the barber's work routine is shown in Figure 3.2. In the initial state, the barber has just finished serving a client and is checking if another client is waiting for him (note the tokens in the correspondingly labelled places). At this point, the transition labelled "Go to bed" is enabled as per definition 3.1. If it fires, it transfers the token from the place labelled "Checking for customer" to the place labelled "Going to bed", reflecting the change in the state of the barber. Transition labelled "Get next customer" is not enabled, however, because there is no token in the place labelled "Customer waiting". This transition requiring two tokens to fire reflects the precondition that for the next customer to be served, both he/she must be waiting and the barber must have finished serving the previous client. For simplicity, we assume that only one client can be in the barber's shop at one time.

By randomly firing enabled transitions, one can see that the system indeed behaves in the way described in the problem statement. But where is the problem, and how the Petri net helps to identify it?

By running the net through an automated analysis tool, such as MPSat [63, 12] or Petrify [41], the problem is quickly found: the net has a deadlock state. A *deadlock* is a state where no transition is enabled, hence no progress can be made in the system. Unless such a state is recognised as one of the accepted final states, this means that the system does something wrong. Using the failure trace given by the tool, one can reproduce the sequence of events that need to have happened for the system to fail. In this case, this sequence is as follows: "Go to bed", "Wait", "Sleep". This means that the barber has checked his window and found no client waiting, so he started walking

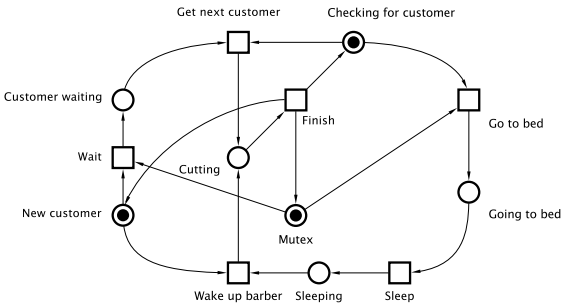


Figure 3.3: Improved Petri net model

to his bed. However at the same time, a client has entered the shop, and having seen no barber sleeping in the bed, goes to the waiting room. The barber has successfully reached his bed and went to sleep. Now, the barber is waiting for a client to wake him up, but the client is waiting for the barber to invite him to the working room. Neither ever happens, and the system stops functioning.

This problem is caused by the fact that the customer and the barber can both be changing state at the same time. In the Petri net that is reflected by two simultaneously enabled transitions, “Wait” and “Go to bed”, neither of which disables the other. In a real-life situation this could happen if they were simultaneously taking different routes through the barber’s shop, each unaware of the others actions, and thus had missed each other. A good way to prevent that would be introducing a mutual exclusion, i.e. some action that can only be performed by one party at the same time, and by performing which one party prevents the other from taking further actions. This is easily implemented in the Petri net: only one additional place is required, as shown in (Figure 3.3). Both state changing transitions now need to take a token from that place, and, because there is only one token, only one transition can fire and by doing so disable the opposing transition. The token must be put back into the mutex place at some point in time for the system to keep functioning. A real-life analogy for this process could be a declaration of intentions by both parties: when a barber is about to go to bed, he would announce that loud enough for the customer to hear, and after hearing that the customer would not go to the waiting room. Similarly, if the customer, upon seeing no sleeping barber, decides to go to the waiting room, he or she would announce that, and the barber would not go to sleep. In this example, declaration of intentions is analogous to removing a token from the mutex place (alternatively, one could think of the mutex as a mechanism that prevents

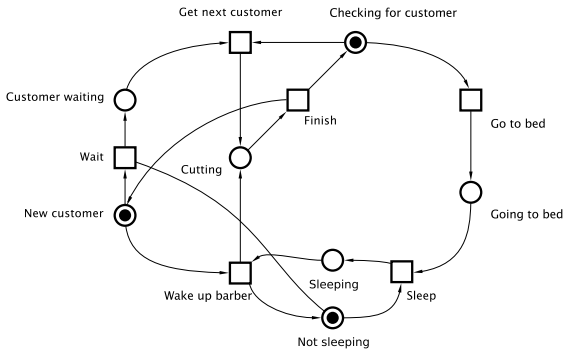


Figure 3.4: A Petri net model with a complementary place and a read arc

opening both doors at the same time).

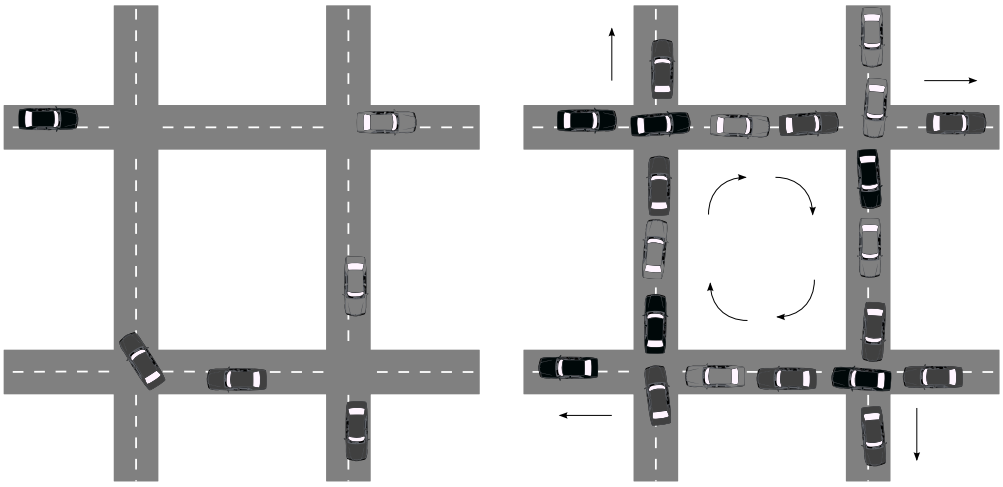
3.1.2 Contextual nets

An extension of a Petri net model is a *contextual net* [82]. It uses additional elements such as *non-consuming arcs*, which only control the enabling of transitions and do not influence their firing.

Definition 3.3. *A contextual net [82]* is a Petri net extended with a special type of non-consuming arcs, namely *read-arcs*, is defined as $PN = \langle P, T, F, R, m_0 \rangle$, where $\langle P, T, F, m_0 \rangle$ is a Petri net and R is the set of read arcs. A set of read-arcs R is defined as $R \subseteq (P \times T)$, there is a read-arc between p and t iff $(p,t) \in R$. The *read-preset* of a transition $t \in T$ is defined as $\star t = \{p \mid (p,t) \in R\}$, and the *read-postset* of a place $p \in P$ as $p\star = \{t \mid (p,t) \in R\}$. A transition t is enabled iff $\forall p,\ p \in \bullet t \cup \star t \Rightarrow m(p) > 0$. The rules for firing of the transitions are preserved. Graphically, a read-arc is shown as a line without arrows.

Read arcs prove to be a very practical mechanism for modelling certain features of asynchronous systems that otherwise would require much more complex and non-intuitive Petri net constructs. They are particularly useful to model systems controlled by switching binary signals, such as asynchronous circuits, as shown in Chapter 4.

To give an example, let us return to the original Petri net model of the barber's shop (Figure 3.2). Assume that the barber has gone to bed after serving a customer and the new customer has just arrived. In this case, there are two transitions enabled for the new customer: "Wait" and "Wake up barber". This contradicts the problem specification because the customer must wake the



(a) Normal traffic – no problems (b) Heavy traffic – potential deadlock

Figure 3.5: The gridlock problem

barber up if he or she sees him sleeping. To fix this flaw, we have to ensure that the "Wait" transition is only enabled if the barber is not sleeping. A read arc allows us to achieve this behaviour with minimal effort. We cannot use a read arc directly in the original net, however, because the read arc has to check the "Barber is *not* sleeping" state, and the net only has a place that represents the "Sleeping" state. In such situation a *complementary place* is helpful.

Definition 3.4. Given a place p, a place p' is called *complementary* to p if $\forall m \in \mathscr{RM} : m(p') = N - m(p)$ where N is the maximum number of tokens that may appear in place p.

In a simple case where p can hold at most one token, a complimentary place p' is always marked with a token when p is not, and, vice versa, p' is never marked when p is.

In Figure 3.4, the place labelled "Not sleeping" is complementary to the place labelled "Sleeping". There is also a read arc between transition "Wait" and place "Not sleeping" that prevents the transition from being enabled when there is no token in that place, i.e. prevents the customer from going to the waiting room if the barber is sleeping. Note that the deadlock problem is still present in this net and it can be resolved in the same way as shown in Figure 3.3. Incidentally, the introduction of mutex place resolves the conflict between the two transitions as well, and it can be seen that using it for this reason is not as obvious as using a read arc.

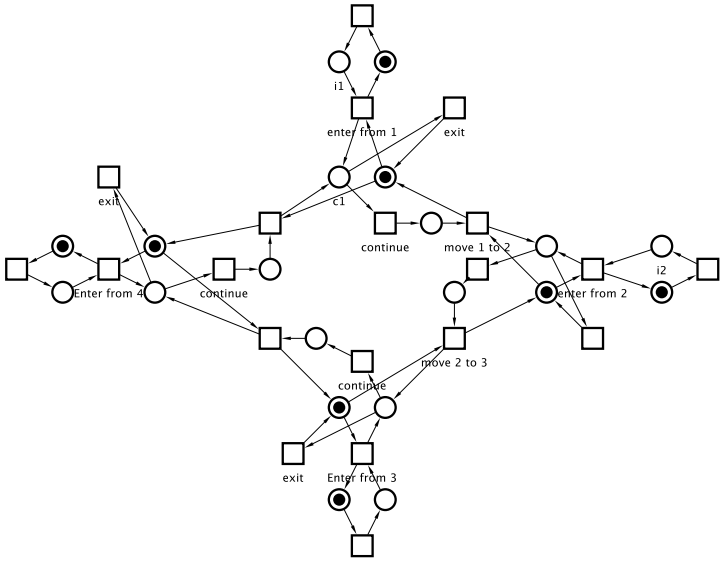


Figure 3.6: A Petri net model of four intersecting roads

3.1.3 Another example: a traffic network

Another typical example of a system where a Petri net model is very useful for finding problems is shown in Figure 3.5. This model illustrates something called a gridlock, a situation that arises both in vehicular traffic in the road network as well as in the network traffic between a set of routers in System-on-a-Chip. For clarity, let us consider the road network example. Figure 3.5a shows the system of four intersecting one-way roads under normal conditions. The cars can freely go around the inner ring formed by the four roads, which in this case is akin to a roundabout, although such an arrangement is usually controlled by traffic lights. The cars may enter from any of the four sides and exit to any other side after spending some time in the ring. The problem appears when enough cars enter the system, as shown in Figure 3.5b. A car that has entered a junction on the green light can become blocked in the middle, unable to move forward because of the heavy traffic moving across, and unable to go back because there are too many cars behind it. If this happens in all four junctions at the same time, then the whole system becomes deadlocked because no car can either move along the ring or exit the junction.

This is indeed a problem for heavily congested road networks, especially those arranged in a regular grid pattern (e.g. New York). To prevent this from happening, drivers should never enter the crossing unless there is enough space in the road across for them to clear it, even if there is a green light. Unfortunately, some over-eager traffic wardens may insist that such behaviour

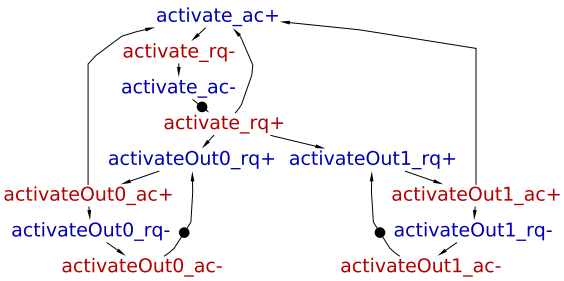


Figure 3.7: Graphical representation of an STG

constitutes deliberate blocking of the way and insist that you move on the green light in any case.

Similar deadlocks may occur in networks-on-chip [23].

3.2 Signal Transition Graphs

The Signal Transition Graph (STG) model, an extension of the Petri net model designed to to formally model the behaviour of circuits, was introduced independently in [38] and [98]. A Signal Transition Graph describes the causality relations between transitions on the input and output signals of the specified circuit. It also allows the explicit description of data-dependent choices between various possible behaviours. Because STGs are a special case of Petri nets, there exists a rich theoretical and tool base for their analysis and specification.

Definition 3.5. A *signal transition graph (STG)* [69] is a tuple $G = \langle P, T, F, m_0, \lambda, I, O, v_0 \rangle$, where $\langle P, T, F, m_0 \rangle$ is an PN, I is a set of input signals, O is a set of output signals, $I \cap O = \emptyset$, $Z = I \cup O = \{z_1, z_2, ..., z_{|Z|}\}$ is a joint set of all signals, $v_0 = \{0,1\}^{|Z|}$is a vector of initial signal values, λ is an injective labelling function $\lambda : T \to Z \times \{+,-\}$, i.e. an STG is an LPN where each transition is labelled with a signal level change event. If different transitions correspond to the same event, an index is used to distinguish them. Note that graphically a signal event and its index are separated using a slash symbol, and if there is only one instance of the event, the index is omitted.

For graphical representation of STGs a short-hand notation is often used (shown in Figure 3.7), where a place is not shown if it has exactly one incoming and one outgoing arc (the tokens are drawn on the arc instead).

3.3 Properties and analysis of Petri nets

Checking whether a Petri net satisfies a certain property is very important for the analysis of system models. In particular, the notions of *marking reachability* and *deadlock* are used throughout this thesis:

Definition 3.6. (Reachability) The set of reachable markings of a Petri net is the smallest (w.r.t. $\subseteq$) set $\mathscr{RM}$ containing m_0 and such that if $m \in \mathscr{RM}$ and $m[t\rangle m'$, for some $t \in T$ then $m' \in \mathscr{RM}$. A marking m is *reachable* if $m \in \mathscr{RM}$.

Definition 3.7. (Deadlock) A marking m is *deadlocked* if at this marking no transitions are enabled. A Petri net is *deadlock-free* if none of its reachable markings is deadlocked.

Definition 3.8. (Boundedness) A *Petri net* is k-bounded if $\forall m \in \mathscr{RM}, m(p) \leq N, p \in P$, i.e. for every reachable marking the number of tokens in any place does not exceed k. A Petri net is safe if it is 1-bounded. A Petri net is simply bounded if $\exists k$ such that the net is k-bounded.

To determine if there exists a reachable marking satisfying certain properties, the set of reachable markings $\mathscr{RM}$ must be computed. This, however, quickly leads to a combinatorial explosion problem, and requires state-space reduction techniques to be employed. One such technique is based on Petri net unfoldings [77].

Given a bounded Petri net N, the unfolding technique aims at building a labelled acyclic net Unf_N (prefix) satisfying two key properties [64]:

- Completeness. Each reachable marking of N is represented by at least one "witness", i.e., one marking of Unf_N reachable from its initial marking. Similarly, for each possible firing of a transition at any reachable state of N there is a suitable "witness" event in Unf_N.
- Finiteness. The prefix is finite and thus can be used as an input to model checking algorithms, e.g., those searching for deadlocks.

A prefix satisfying these two properties can be used for model checking as a condensed representation of the state space of a system. Since its introduction [77], the unfolding-based approach has been extensively improved and is able to deal with more complex and varied applications. In

particular, recent research has shown that many verification problems for unfoldings can be formulated in terms of Boolean satisfiability (SAT) and very efficiently dealt with by existing SAT solvers [63].

The unfolding technique is not the only state-space reduction technique that is applicable to Petri nets. However, it proved to be very efficient for the class of systems discussed throughout this thesis and thus was chosen as the main model-checking method.

3.4 Conclusions

In this chapter, a formal definition of Petri nets was given. Using two illustrative examples, the mechanics of the token game in a Petri net were explained. A number of problems characteristic to concurrent systems are highlighted, and it is shown that Petri nets are highly helpful in discovering such problems. Several properties of Petri nets relevant to the context of the remaining chapters of this thesis were defined.

Chapter 4

Automated verification of asynchronous circuits using Petri nets

During the design of asynchronous circuits that are relatively small in size, but have peculiar behaviour (e.g., arbiters, data path controllers, handshake component implementations), it is often the case that some parts of the circuit are designed manually or generated by the software written specifically for this task. Such circuits cannot be guaranteed to be correct by construction, as opposed to the solutions produced by logic synthesis tools. The circuit implementation obtained this way needs to be validated against its specification to ensure its correctness before it is committed to hardware.

The designer can usually choose between two methods of circuit validation: *simulation* or *formal verification*. Simulation can be used to demonstrate the correct functionality of a circuit under certain stimuli from the environment. However, this cannot reveal all of the possible circuit behaviours since it would require exhaustive enumeration of all allowable sequences of actions of the environment, which quickly leads to the combinatorial explosion problem. The aim of the formal verification methods is to avoid the explicit enumeration of the input sequences to provide a more efficient solution to the validation problem.

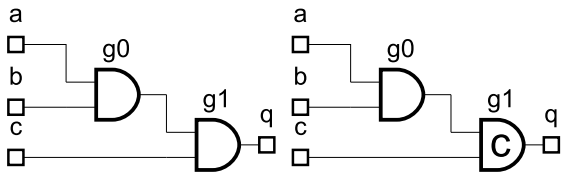
Simulation and verification are particularly different in their results for asynchronous circuits, because the latter often exhibit high degree of concurrency. Moreover, the environment's choice of input signal transitions can be concurrent with the internal signal transitions, making techniques

such as cycle accurate analysis ineffective. In those circumstances, the complexity of validation by simulation increases, and demands for the use of analytic exploration of the behavioural models of the circuit implementations. It is therefore imperative to consider formal verification using models similar to those used for specification.

When compared to synchronous circuits, asynchronous circuits are often described as having significantly better modularity. To produce a robust modular solution, the designer must adhere to a strict definition of the environment that the circuit is expected to work with. The circuit must produce only those changes of output signal levels that are expected by the environment, only in response to corresponding changes of input signals, at the correct time and in the correct sequence.

The "environment" in this case is the circuit that the circuit being designed is to be interfaced with. Note that the circuit can only "see" those signal transition in its environment that are directly connected to its inputs. This implies that even though the implementation of the environment may be complex, the circuit designer does not need to be concerned with it. Only the abstract environment specification that defines the proper ordering of output signal transitions in response to the particular input transitions is required. Such specification is in most cases much smaller than a concrete circuit implementing it. It is therefore practical, for the purpose of verification, to represent a closed system as two parts: the circuit implementation and the specification of the environment. It is also practical to describe these parts using different formal models: the environment specification is most naturally expressed using a Signal Transition Graph, and the circuit implementation is generally given as a network of logic gates.

Analysing such a system, however, is not a straightforward task. Because the different formalisms are used to describe the circuit implementation and the environment specification, they cannot be directly "glued" together to produce a closed system suitable for automated verification. One solution to this problem is to convert both models into another representation. This is the verification method underlying the tool Versify [19, 97]. The tool checks the correctness of a gate level implementation of a circuit against its STG specification, by considering the closed system whose state space is subject to analysis for undesirable conditions. The closed system is formed implicitly, at a symbolic state-space traversal stage, where both the gate-level net-list (i.e. a set of Boolean equations) and the specification STG contribute to the respective state vector components.



(a) 2 AND gates (b) AND and synchronising C-element

Figure 4.1: An intuitive implementation of 3-input AND gate

In this chapter we propose an alternative solution. The main idea is to translate the circuit implementation into the same modelling language as the specification. To accomplish this, the gate net-list is converted into a special type of a Petri net (called a circuit Petri net) using a direct mapping algorithm. Because the environment STG is also a Petri net, the composition of the two parts of the system is simple.

Once the complete system is produced, it is passed to one of the available Petri net analysis tools for automated verification.

4.1 The verification problem

Let us consider an example shown in Figure 4.1 (a), which is a possible implementation of a 3-input AND gate. Intuitively, one would think that since $((a \wedge b) \wedge c) = (a \wedge b \wedge c)$, this circuit is correct. Given enough time for the circuit to stabilise between consecutive computation cycles (which constitutes the synchronous design approach), this is indeed true, but it is obviously advantageous to present the circuit with new data as soon as the computation of previous data is complete. However, since no assumptions about gate delays are made in this approach, this can quickly lead to problems. For example, the following firing sequence:

$$\langle c+, a+, b+, g0+, g1+, c-, a-, b-, g1-, c+, g1+ \rangle$$

leads the gate $g1$ into firing prematurely, which happens because the new wave of inputs arrives before gate $g0$ could return back into a stable state. This produces an incorrect behaviour of the circuit. If one tries to avoid this situation by substituting the second AND gate with a a C-element

(Figure 4.1 (b)) in order to synchronise the two gates, another problematic sequence surfaces:

$$\langle c+, a+, b+, g0+, g1+, c- \rangle$$

after which the output q will remain stable, even though one of the inputs is low, which is sufficient to state that this circuit is not a 3-input AND gate.

This very small example already illustrates the complex nature of interactions of the elements in an asynchronous circuit. Detection of all possible coincidences that may result in the incorrect behaviour of a circuit is a very complex task. Considering that the modern technology requires to take into account not only possible delays of the logic gates, but also delays on the wires, it is also extremely computationally expensive. Several approaches are known that alleviate the state space explosion problem [97, 75], most of them based on explicit, although compressed, representation of the reachability graph.

In this chapter we present an alternative, Petri net based approach to the problem of asynchronous circuit validation. To compress the state space, Petri net unfolding techniques (as outlined in section 3.3) are employed, which represent the state space implicitly.

4.2 Circuits and Petri nets

An idea to represent switching circuits as a special class of Petri nets was first proposed in [51] and further refined in [121]. For a long time, this approach was deemed inefficient due to the fact that several places and transitions, as well as a set of connecting arcs, are required to represent each signal (as opposed to a pair of Boolean equations used in BDD-based approaches [97]). However, in the light of recent developments in Petri net verification techniques, particularly of the tools based on unfolding theory [64, 55, 66, 12] this approach cannot be ignored: the finite prefix of a Petri net unfolding is usually able to represent all of the possible behaviours of the net in a very compact way.

Definition 4.1. A circuit [97] is a triple $C = \langle V, \mathscr{F}, s_0 \rangle$, where $V = \{v, v_1, ..., v_n\}$is a set of signals, $\mathscr{F}$ is a mapping $\mathscr{F} : v_i \rightarrow f_{v_i}, v_i \in V$ where f_{v_i}corresponds to the Boolean function of the logic gate that drives v_i, and $s_0$is the initial state of the circuit.

Definition 4.2. A circuit Petri net R associated with a circuit C is an STG that satisfies the following properties:

1. For each signal $v_i \in V$ there exist exactly two complementary places $\{p_{v_i}, \overline{p_{v_i}}\} \in P$, such that at any reachable marking one and only one of $\{p_{v_i}, \overline{p_{v_i}}\}$ is marked with a single token. If in initial state $s_0 \in C$ signal v_i is high, then at the initial marking $m_0 \in R$ place p_{v_i} is marked. Otherwise, $\overline{p_{v_i}}$ is marked.

2. For each pair of complementary places, there exists a finite number of rising signal transitions $t^+_{v_i} \in T$ that transfer the token from the place $\overline{p_{v_i}}$ to the place p_{v_i}, corresponding to the event of signal v_i going from low to high. Similarly, there exists a finite number of falling signal transitions $t^-_{v_i} \in T$ that transfer the token from p_{v_i} to $\overline{p_{v_i}}$, corresponding to the event of signal v_i going from high to low.

3. Transitions between complementary places are controlled by a set of read arcs [82] that non-destructively test the presence of tokens in other places in P. The read arcs are placed in such a manner that they correspond to the dependence of a signal $v_i \in V$ on other signals in V exactly as defined by $\mathscr{F}(v_i)$.

As can be seen from definition 4.2, a circuit Petri net consists of a number of so-called *elementary cycles* interconnected with read arcs. An elementary cycle is a set of two complementary places and the transitions connecting them, associated with a signal in the circuit via labelling. Figure 4.2 shows the structure of such elementary cycles, and the way of producing different causality relations. In the figure, the regular arcs are shown as thin lines with arrowheads, and the read arcs as thick lines with no arrows. Subfigure (a) is an elementary cycle with only one rising and one falling transition; subfigure (b) is an elementary cycle with two rising transitions and one falling transition, which means that the signal it represents exhibits OR-causality for positive excitation and AND-causality for negative excitation (hence an OR-gate is driving it); subfigure (c) is an elementary cycle with OR-causality for negative excitation and AND-causality for positive excitation, which suggests that an AND-gate is driving the associated signal.

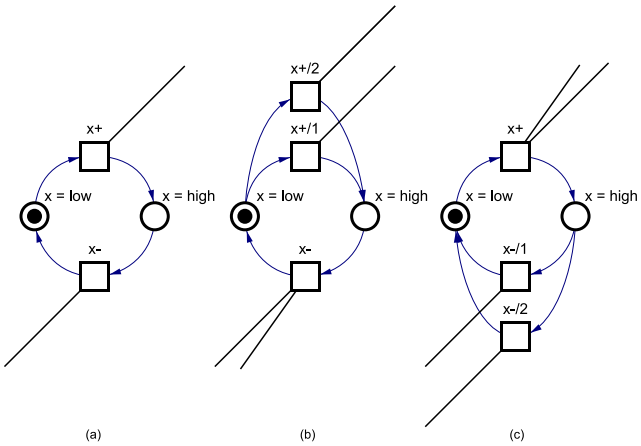


Figure 4.2: Examples of elementary cycles in circuit Petri net

4.3 Construction of a circuit Petri net

Given a source gate-level model, a corresponding circuit Petri can be produced using Algorithm 1. However, several further steps are necessary before the Petri net can be fed to the external tools for verification. These steps are detailed below.

4.3.1 Applying environment interface

After the circuit Petri net has been constructed, it is necessary to compose it with the provided environment interface STG. This is done by superposition of the corresponding transitions of the two Petri nets. Figure 4.3 shows an example of such superposition of transitions corresponding to the output signal Q. In the circuit Petri net, there is a rising transition $Q+$ and a falling transition $Q-$. Environment STG contains two occurrences of rising transition$\{Q+/1, Q+/2\}$ and one falling transition $Q-$ (see Subfigure(a)). The superposition of $Q-$ transition is trivial: it is removed from environment STG and the token flow is redirected through $Q-$ transition in the circuit Petri net. This is not possible with the rising transition $Q+$: it needs to be duplicated in the circuit Petri net to create two transitions $\{Q+/1, Q+/2\}$ with the same preset and postset. After that these two transitions can be superpositioned with the corresponding transitions in the environment STG (see Subfigure(b)).

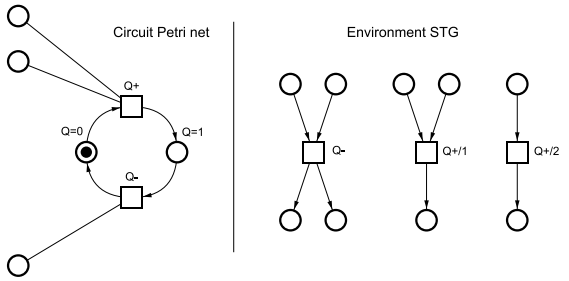
This technique is called parallel composition [125].

Algorithm 1 Conversion to circuit Petri net

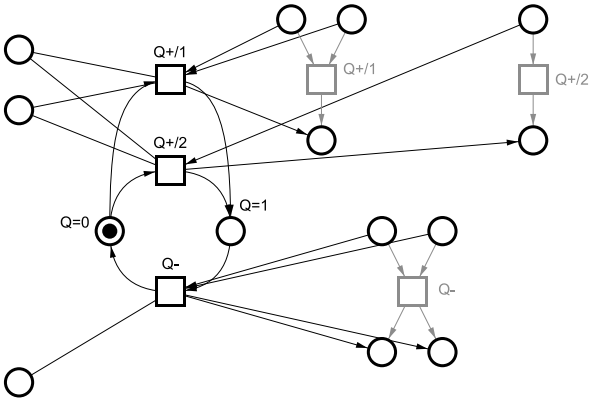
```
for each signal v_i ∈ V:
  insert places {p_{v_i}, p̄_{v_i}} into P
  if v_i is high in s_0 then
    mark p_{v_i}
  else
   mark p̄_{v_i}
  end if
end for

for each signal v_i ∈ V:
  build a DNF DNF_set for function F(v_i)
  perform Boolean minimisation* of DNF_set
  k = 0
  for each clause C∈DNF_set:
    insert a transition t_{v_i}^{+k} into T
    insert arcs {(p_{v_i}, t_{v_i}^{+k}), (t_{v_i}^{+k}, p̄_{v_i})} into F
    for each signal v_j ∈ C:
      if v_j is negated then
         insert arcs {(p̄_{v_j}, t_{v_i}^{+k}), (t_{v_i}^{+k}, p̄_{v_j})} into F
      else
         insert arcs {(p_{v_j}, t_{v_i}^{+k}), (t_{v_i}^{+}, p_{v_j})} into F
      end if
    end for
    increment k
 end for
   build a DNF DNF_reset for function ¬F(v_i)
   perform Boolean minimisation of DNF_reset
   k = 0
   for each clause C∈DNF_set:
     insert a transition t_{v_i}^{-k} into T
     insert arcs {(p̄_{v_i}, t_{v_i}^{-k}), (t_{v_i}^{-k}, p_{v_i})} into F
     for each signal v_j ∈ C:
      if v_j is negated then
          insert arcs {(p̄_{v_j}, t_{v_i}^{-k}), (t_{v_i}^{-k}, p̄_{v_j})} into F
      else
          insert arcs {(p_{v_j}, t_{v_i}^{-k}), (t_{v_i}^{-k}, p_{v_j})} into F
      end if
     end for
     increment k
   end for
 end for

*using Quine-McCluskey algorithm [76]
```



(a) Circuit and environment STGs



(b) Compositional STG

Figure 4.3: Composition of circuit and environment STGs

4.3.2 Read arcs complexity reduction

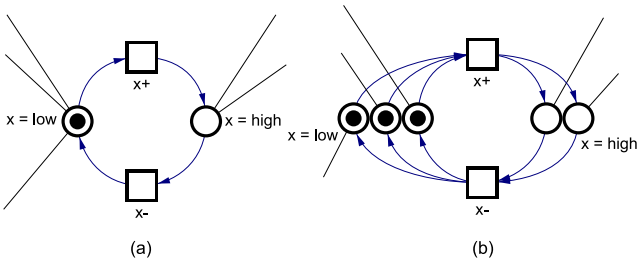


Figure 4.4: Read arcs complexity reduction
(a) multiple read arcs associated with one place
(b) only one read arc per place

At the time of writing, the available Petri net unfolding tools do not recognise read arcs as a special type of arc. Instead, read arcs need to be modelled as double-headed arcs, i.e. if $p \in \bullet t \cap t\bullet, p \in P, t \in T$ then p and t are connected with a read arc. Though behaviourally correct, this representation is semantically different from an actual read arc in that it introduces a choice, which may lead to a drastic growth of the unfolding size. This problem can be resolved to an

extent by ensuring that any place is associated with at most one read arc [124], which can be accomplished by making a necessary number of copies of each place with multiple outgoing read arcs and rearranging the read arcs accordingly, as shown in (Figure 4.4).

4.4 Verification

A circuit is considered speed-independent under a given environment, if

1. It conforms to the environment, i.e. produces only those changes of output signals that do not conflict with the environment's STG.

2. It is hazard-free.

In the scope of this work, a hazard is defined to be an unexpected change of the input signal of a gate, such that it causes an enabled (positively or negatively excited) gate to become disabled (i.e. to return into a stable state without firing). A circuit that never exhibits such behaviour is called *hazard-free*, or *safe*.

4.4.1 Detection of potential hazards

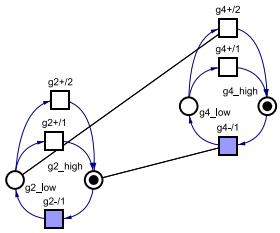
A pair of signals is called *conflicting* if there exists a reachable state of the circuit such that a change in the level of one of them disables the gate driving the other. In terms of a circuit Petri net, a potentially hazardous state is a state which violates the *semi-modularity* property:

Definition 4.3. A Petri net is called semi-modular if any transition in this net, once enabled, cannot be disabled until it has fired.

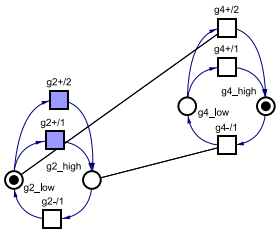
In other words, once each place in the preset of a transition has become marked with a token, thus enabling the transition, no other transition can "steal" any of these tokens. In Figure 4.5 (a), an example of non-semi-modularity is shown: if transition $g2-/1$ fires, it disables transition $g4-/1$ (enabled transitions are depicted as greyed boxes).

Definition 4.4. A pair of transitions $\{t_1, t_2\} \in T$ is called *conflicting* if $\bullet t_1 \cap \bullet t_2 \neq \emptyset$.

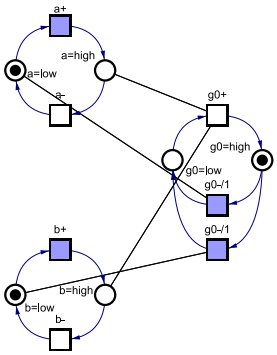
For the purpose of verification, we consider that if a circuit Petri net is semi-modular, then the circuit it was constructed from by using Algorithm 1 is hazard-free.



(a) Conflicting transitions



(b) Transitions in allowable conflict



(c) Transitions in allowable conflict

Figure 4.5: Non-semi-modular states

This statement stems from the following: for each signal in the circuit, there is an elementary cycle (see Section 4.2) in the Petri net, and for each possible combination of the levels of input signals which lead the gate that drives this signal into a positively or negatively excited state, there is a corresponding rising or falling transition in this cycle. Once any of these combinations becomes active (the gate becomes excited), the corresponding transition becomes enabled. If the state of any of the input signals changes in such a way that the excitation condition is no longer fulfilled and the gate has not yet fired, this produces hazard, but it will also cause the corresponding circuit Petri net transition to become disabled, thus violating semi-modularity (see also [83] for Muller's original view of semi-modularity).

However, while the presence of a potential hazard in the source gate-level model will always indicate a violation of the semi-modularity in the circuit Petri net, the reverse is not true. There are two cases in which a violation of semi-modularity in the circuit Petri net does not indicate the presence of a potential hazard in the original circuit.

The first situation arises due to the possibility of several transitions representing the same

signal event, but being caused by different preceding events, as shown in Figure 4.5. In Subfigure (b), the conflicting transitions $g2+/1$ and $g2+/2$ represent the same event, signal $g2$ going high. Hence, the conflict of these transitions does not constitute a signal conflict: they both have the same semantic and thus their firing does not disable any other signal. In Subfigure (c), the conflict between $g0-/1$ and $g0-/2$ is allowed for the same reason, but there is also conflict between $a/+$ and $g0-/1$ which are associated with different signals. However, even if $a/+$fires, disabling $g0-/1$, the enabled transition $g0-/2$ still keeps the signal event $g0-$ enabled, and thus disabling of the transition $g0-/1$ does not lead to the disabling of the negatively excited gate driving signal $g0$, so there is again no signal conflict. On the other hand, if the transition $g0-/2$ was not enabled, then the conflict between $a/+$ and $g0-/1$ would be a signal conflict.

The second situation occurs when the conflicting transitions are both associated with the input signals. Since it is the environment that controls these signals, this situation should be considered a choice of mode of circuit operation made by the environment and not a signal conflict.

To summarise, if for some conflicting pair of transitions $\{t_1, t_2\} \in T$:

1. $\lambda(t_1)$ and $\lambda(t_2)$ are not both input signal events
2. $\lambda(t_1) \neq \lambda(t_2)$
3. there exists a reachable marking such that $\forall p \in \bullet t_1 \cup \bullet t_2, m(p) > 0$ and at this marking there is no enabled transition $t \in T$ such that $(\lambda(t) = \lambda(t_1)) \vee (\lambda(t) = \lambda(t_2))$

then there is a potential hazard in the original circuit.

4.4.2 Detection of interface non-conformance

A circuit Petri net, when composed with its environment, forms a closed system: the outputs of the circuit are the inputs for the environment STG, and vice versa. Thus, the conformance verification is twofold: if the environment part of the composed Petri net is able to produce a sequence of inputs that causes "bad behaviour" of the circuit (i.e. a hazard or a deadlock), the circuit is said not to conform to its environment and this situation is referred to as *α-non-conformance*; on the other hand, if the circuit is ever able to produce an output signal change that is not expected by the

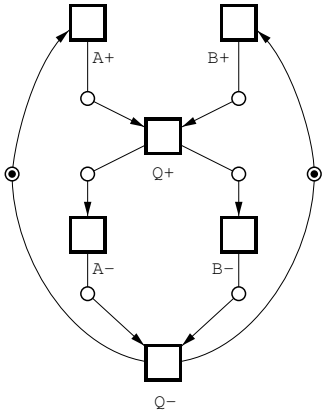


Figure 4.6: A C-element interface STG

environment, it is also said not to conform to the environment, and this situation is referred to as *β-non-conformance.*

An example of α-non-conformance can be demonstrated if a XOR gate is verified against the C-element interface STG (Figure 4.6). If the environment produces events $A+$ and $B+$ almost simultaneously, quickly enough so that the XOR gate becomes excited but does not fire and returns into stable (output signal low) state, this leads to, first, a hazard on one of the inputs, and, second, into a deadlock. The deadlock is present because the C-element environment, having switched both input signals to high, expects an output signal Q to go high. But this never happens: a XOR gate cannot switch output signal to high until one of its input signals goes low, and this will never happen as well, because the STG does not allow to reset the inputs A and B until the output Q is produced. Thus, α-non-conformance is decided by checking the Petri net for hazards and deadlocks. A method for hazard detection is explained in Subsection 4.4.1, and the deadlock problem is solved by external model-checking tools, thus checking for α-non-conformance does not require much additional effort.

If the XOR gate is replaced by an AND gate, however, there is no α-non-conformance: the input goes high only when both outputs go high, thus no hazard is observed. But when either one of the inputs goes low, the AND gate becomes negatively excited, and tries to reset the output, which is not expected by the environment STG. However, in the corresponding compositional Petri net the environment *restricts* the circuit because the two transitions $Q-$(one provided by the circuit, and the other by the environment) become superimposed (Subsection 4.3.1), which introduces

a synchronisation, and thus the transition $Q-$ will only become enabled when the environment resets the second input signal. Hence, the system has no hazards and no deadlock, but the AND gate obviously does not conform to the C-element interface. If it would have not been restricted by environment, it could produce event $Q-$ when it was unexpected, exhibiting β-non-conformance.

Let C be a circuit, R be a circuit Petri net constructed from C and E be the environment STG. Let $R.P$ denote the set of places $P \in R$, $E.P$ the set $P \in E$ and M the set of all transitions which were superimposed during circuit-environment composition. Then, if there exists a reachable marking m such that at this marking for at least one transition from M, all of the places in its preset that belong to R are marked, and there exists at least one place in its preset that belongs to E which is not marked, or, formally, $\exists t \in M : (\forall p \in \bullet t \cap R.P, m(p) > 0) \wedge (\exists p \in \bullet t \cap E.P, m(p) = 0)$ then the circuit C is β-non-conformant under environment E.

4.5 A practical example

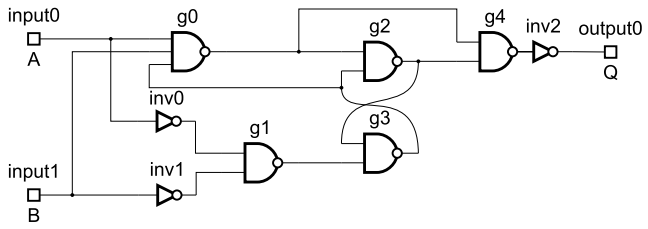


Figure 4.7: NAND C-element implementation

This section presents an example of application of the method proposed in this chapter and implemented in the Workcraft framework (Chapter 7), and demonstrates the achieved integrity of the design workflow. Figure 4.7 shows a NAND-based implementation of the C-element proposed by Maevsky. The gate-level model was created using Workcraft's visual editor and verified. The verification fails and reports a following trace as the shortest firing sequence that leads to a potential hazard:

$$\langle input1, input0, inv1, g0, g1, g2, g3, g0, g4, inv2, output0 \rangle$$

The faulty trace can be simulated and the problematic firing sequence examined, which reveals that indeed, provided the *inv*0 inverter's delay is long enough, it can be excited but still not have

fired after the environment has received output signal Q and resets input signals A and B, disabling $inv0$. However this is very unlikely, because in order for this hazard to actually happen, $inv0$ delay should be longer that the total delay of all other gates. Hence, this potential failure can be safely ignored for any practical application.

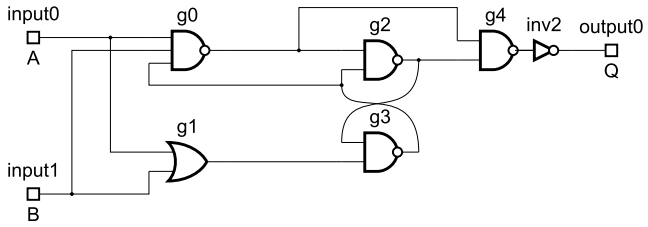


Figure 4.8: NAND-OR C-element implementation (no wire delays)

By replacing the inverters $\{inv0, inv1\}$ and the NAND gate $g1$ with an OR gate (Figure 4.8), this problem is eliminated and the verification reports success, confirming that the implementation of the C-element shown in Figure 4.8 is speed-independent, and the implementation shown of Figure 4.7 is speed-independent under a very reasonable timing assumption.

But while this circuit is speed-independent, it could still produce unexpected behaviour if it is not delay-insensitive. To verify whether it is delay-insensitive, possible wire delays should be taken into account. Since it is enough to demonstrate that delay on any of the wires may lead to a hazard in order to assert that the circuit is not delay-insensitive, it may be reasonable not to model delays on all of the wires in order to minimise verification time. In Figure 4.9, a wire delay is introduced in the form of a buffer into the fork following gate $g3$ output. Verification fails with the following trace:

$$\langle input1, input0, g1, g0, g2, g3, g0 \rangle$$

Examination of this trace shows that the hazard can happen if gate $g0$, after receiving the signal from $g3$, will switch before the same signal from $g3$, but travelling across the other branch of the fork, reaches gate $g2$. In this case, the firing of $g0$ will disable the already excited gate $g2$. This is enough to state that this C-element implementation is not strictly delay-insensitive, but requires a timing assumption that the delay of signal reaching $g0$ plus $g0$ switching delay is more than wire delay on the other branch of the fork.

It may still be helpful to check this circuit considering all the possible wire delays. This

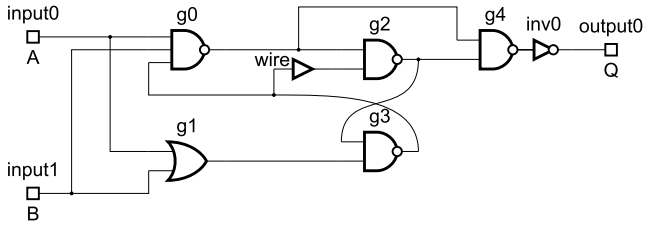


Figure 4.9: NAND-OR C-element implementation
(wire delay present on one fork only)

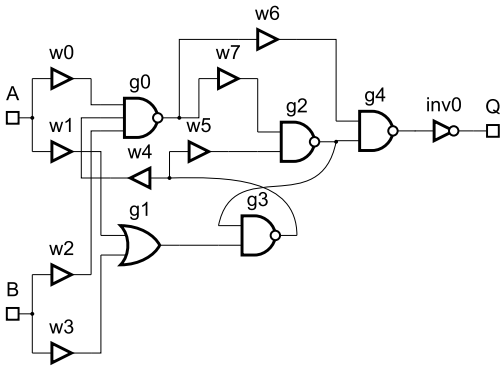
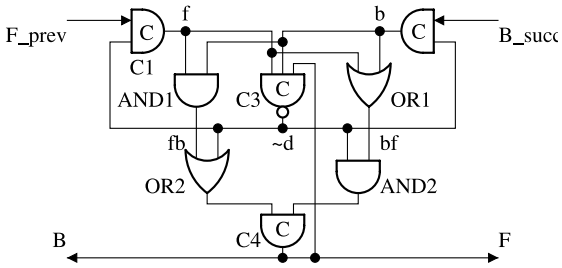


Figure 4.10: NAND-OR C-element implementation
(full set of wire delays)

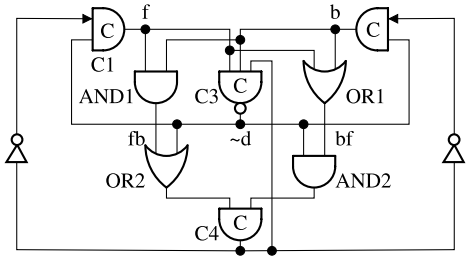
can be done by providing branches of all forks with buffers (the buffers are not needed on non-branching wires, and on the sections of wire preceding forks, because in this case it may simply be considered that the delay of the gate producing signal on this wire includes the wire delay), as shown in Figure 4.10. Verification in this case produces the following failure trace:

$$\langle input1, input0, w2, w0, g0, w7, g2 \rangle$$

This failure is similar to the one in the case above: if the delay of $w7 + g2$ is less than delay of $w6$, $g4$ will be disabled before it can fire. Note that the verification time is considerably longer due to the growth of unfolding prefix.



(a) Controller implementation



(b) Looped controller configuration for verification

Figure 4.11: A counterflow stage controller

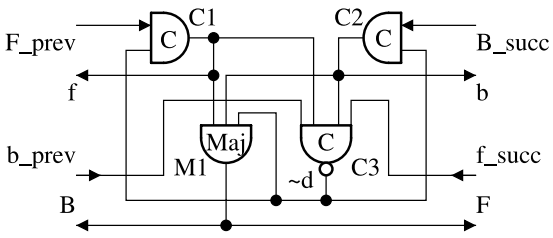


Figure 4.12: Revised counterflow stage controller

4.6 Verification of a counterflow data path controller

Let us consider a counterflow stage controller presented in [22] (Figure 4.11a). If the circuit is configured as shown in Figure 4.11b, where the inverters are used to emulate the surrounding pipeline stages, automated verification can be applied.

In this case, the verification completes in a negligible amount of time, and produces the following failure trace that leads to a potential hazard:

$$\langle F_{prev}+, f+, B+, F+, B_{succ}, b+, \sim d-, F_{prev}-, f-, B-, F-\rangle$$

By investigating the trace using the simulation feature in Workcraft, one can see that it corresponds to the following scenario. The previous stage controller issues a data token, causing the circuit to issue signals *B* and *F*. At the same time, the next stage controller sends a token in the opposite direction (note that this token is not an acknowledgement of signal *F* but rather a request for a borrowed token). Now the previous stage controller can reset the data token, which will cause the circuit to reset signals *F* and *B*. But the signal *F* may not have been latched yet into the C element *C1* of the next stage controller (which is emulated by the same circuit via an inverter loop in the test configuration), which will cause a hazard. A similar problem may occur with signal *B*.

It can be argued that it is a reasonable timing assumption that the next stage controller latches the value of *F* into the C-element faster than *F* resets to zero. This argument, however, does not take into account the delay of the combinational logic between the registers. This problem may not be critical for bundled-data implementations where the rising and falling transitions of *F* propagate through the matched delay with the same speed. However, in a dual-rail implementation the propagation time of data and spacer through combinational logic varies significantly and a wave of spacer may overtake the wave of data leading to a hazard. The hazards on outputs *F* and *B* can be avoided by explicitly acknowledging these outputs as shown in Figure 4.12 [107]. Input *f_succ* is connected to the output *f* of the next stage controller and acknowledges output *F* . Similarly, input *b_prev* is connected to output *b* of the previous stage controller and acknowledges output *B*. C-element *C3* blocks all changes on inputs *F_prev* and *B_succ* until both outputs *F* and *B* are acknowledged.

Benchmark	States	Net size (P/T)	Unfolding size (events/cutoffs)
reg2	$2.5*10^4$	183/124	368/29
reg4	$7.6*10^7$	337/220	2464/177
reg8	$7.1*10^{14}$	649/416	72192/4865
fifo5	$2.6*10^3$	97/58	86/1
fifo10	$1.2*10^6$	177/108	166/1
fifo15	$5.8*10^8$	257/158	246/1

(a) Benchmark statistics

Benchmark	Versify	zeta	Workcraft (PUNF+MPSAT)
reg2	n/a	0.47 sec	0.11 sec
reg4	388 sec	2.75 sec	6.33 sec
reg8	7246 sec	83.9 sec	48.38 sec
fifo5	8 sec	0.15 sec	0.02 sec
fifo10	130 sec	0.61 sec	1.02 sec
fifo15	634 sec	3.99 sec	2.4 sec

(b) Comparison of proposed method with existing tools

4.7 Performance and comparison statistics

The presented verification approach was tested on a set of benchmarks (see Table 4.1b) which included asynchronous multiport registers [86] and FIFO pipelines [74].

The results are compared with Versify [97] and zeta [75] tools. Note: the runtimes for Versify were taken from [97] (because of the technical problems running the old software) and thus cannot be compared directly with the results for zeta and Workcraft because the latter were obtained on a modern machine. The times for Versify are provided in order to highlight the rapid growth of the runtime due to exponential growth of the state space. It is possible to see that Versify and zeta runtimes grow considerably faster with the growth of the number of states that the size of the unfoldings and reachability analysis time, which in many cases grow linearly because the analysed circuits exhibit high degree of concurrency.

The tools used as the Petri net verification back-end were the unfolding-based Punf and MP-Sat [64, 12]. The benchmark results were obtained on a single-core Pentium 4 machine. More recent processors tend to be multi-core, which benefits the Petri net unfolding algorithm [55]. The runtime therefore can be significantly reduced on a multi-core system whilst computations for BDD-based techniques cannot be easily distributed between multiple processing units.

4.8 Conclusions

In this chapter a new method for the verification of asynchronous circuits was proposed. This method was previously published in [93]. It operates by translating a circuit specification (given in the form of a gate-level netlist) into a special class of Petri nets called a *circuit Petri net*. This net is then composed with a specification of the environment of the circuit given in the form of an STG using the parallel composition operation. The resulting net is checked for deadlocks as well as a number of reachability properties required to ensure that the circuit behaves correctly in the given environment.

Compared to the previously existing verification methods, the approach presented in this chapter exploits the flexibility and maturity of the existing Petri net verification tools. In particular, it allows to apply the state-of-the-art Petri net unfolding techniques to the verification of asynchronous circuits. Unfolding-based techniques are able to effectively exploit concurrency in order to build a highly compressed representation of the state space. This feature is especially useful in the context of asynchronous circuits as they are naturally concurrent. At the same time, the method is not bound to any single Petri net verification technique and therefore allows choosing the most appropriate verification back-end based on the nature of the circuit that is being verified.

The proposed method was successfully applied to detect and eliminate a problem in a previously published circuit (Section 4.6).

Chapter 5

Modelling, simulation and automated verification of the data path of asynchronous circuits

There has recently been an increase in research on design of self-timed data path logic and pipeline structures with much more sophistication in dynamic behaviour than simple Muller pipelines. However, the modelling, analysis and synthesis support is still very limited, mainly due to the lack of a formal model that could be used to adequately represents the asynchronous data path.

As a result, there are examples of circuit level solutions [33, 22] that have not been sufficiently analysed and the published circuits behave with certain undesirable effects. In particular, verification of counterflow data path controller using a method presented in Chapter 4 revealed a potentially hazardous behaviour (see Section 4.6). This example highlights the importance of a formal model for the asynchronous data path that would allow to verify potential hardware solutions against a set of strictly defined protocols.

Traditional models, such as *Petri nets* (PNs) [90] and *finite state machines* (FSMs), are abstract and are hard to mimic the behaviour of asynchronous data path with. The models which naturally capture the asynchronous data path, such as SDFS [110], have not yet been formally defined.

In this chapter the Static Data Flow Structure (SDFS) model is formally defined and three token game semantics on this model are introduced: atomic token, spread token and counterflow. These

semantics are compared and the advantages of each of them are studied. Atomic token model is intended as a formalisation of the original SDFS [110]. Spread token SDFS model addresses the drawbacks of the atomic token model and introduces a rudimentary early evaluation support. The counterflow semantics is capable of modelling preemption, early evaluation and speculation in asynchronous data path.

The goal of this chapter is to define a formal model and a verification method in order to assist the designers in analysing such structures early on in the design process.

5.1 The Static Data Flow Structure model

The SDFS is a high-level model for asynchronous data path that can be viewed as an equivalent to *register transfer level* (RTL) in synchronous design. The SDFS has been informally introduced in [110] concentrating on the structural and syntactical aspects of the model. However, the token game semantics (enabling and firing rules) is only defined by examples and is ambiguous in some cases.

This section focuses on the structure and the syntax of the SDFS model. Token game semantics is an independent issue as it is closely related to the architecture of the asynchronous data path. The most interesting token game semantics are studied separately in the following sections.

Definition 5.1. A static data flow structure (SDFS) is a directed graph $G = \langle V, E, D, M_0 \rangle$, where V is a set of *vertices* (or *nodes*), $E \subseteq V \times V$ is a set of *edges* denoting the flow relation, D is a semantic domain of *data values* and M_0 is an *initial marking* of the graph.

There is an edge between vertices $x \in V$ and $y \in V$ iff $(x,y) \in E$. There are two types of vertices with different semantics: *register nodes* (or simply *registers*) R and *combinational logic nodes* (or simply *logic*) L, $R \cup L = V$. The registers can contain *tokens*, thus defining the marking M of the SDFS. The tokens can be associated with data values from the semantic domain D. The marking of SDFS may evolve by *enabling* and subsequent *firing* of register nodes. The rules of enabling and firing are defined by the token game semantics and are discussed separately for each semantics.

Presets and postsets

The *preset* of a vertex $x \in V$ is defined as $\bullet x = \{y \mid (y,x) \in E\}$ and the *postset* as $x\bullet =$

$\{y \mid (x,y) \in E\}$.

Source and sink

Only registers can have empty presets and postsets. A register with empty preset is called a *source*, and with empty postset is called a *sink*. Note that source and sink nodes can represent inputs and outputs of a data path respectively, thus modelling a device-environment interface.

Path and cycle

A sequence of vertices $(z_0, z_1, ..., z_n)$ such that$(z_{i-1}, z_i) \in E$, $i = 1...n$ is called a *path* from $z_0 \in V$ (called a *start vertex*) to $z_n \in V$ (called an *end vertex*) and is denoted as $\delta(z_0, z_n)$. Note that there can be several paths from one vertex to another or no path at all. A *cycle* is a path whose start vertex is the same as end vertex. A path with no repeated vertices is called a *simple path*, and cycle with no repeated vertices aside from the start/end vertex is a *simple cycle*.

Deadlock and liveness

An SDFS reaches a *deadlock* state if no further firing can happen. If a deadlock state is not reachable the SDFS is called *deadlock-free*.

An SDFS is called *live* if all its registers can fire infinitely many times. In order to be live it is necessary for SDFS to have at least one token in every cycle. This leads to an important structural property of the SDFS model that any simple cycle must contain at least one register. Note that this condition may be not sufficient as liveness property also depends on token game semantics. For example, applying a token game semantics to SDFS model may further limit this requirement to at least three registers per simple cycle (similar to direct mapping from Petri nets [70]).

Projection

Projection of a path δ onto a set of vertices X is defined as $\delta \downarrow X = Set(\delta) \cap X$, where $Set(\delta)$ is the set of vertices in sequence δ.

R-preset and register R-postset

The *R-preset* of a vertex $x \in V$ is defined as $\star x = \{r \in R \mid \exists \delta(r,x) : \delta(r,x) \downarrow R = \{r,x\} \cap R\}$, i.e. a register r is in R-preset of a node x iff there exists a path $\delta(r,x)$ with no other

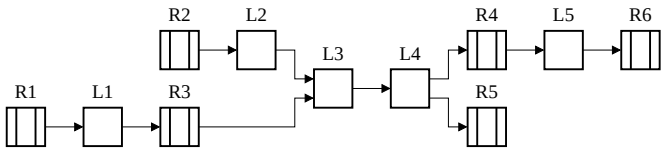


Figure 5.1: SDFS example

registers except r and x (if x is a register). Similarly, the *R-postset* is defined as $x\star = \{r \in R \mid \exists \delta(x,r) : \delta(x,r) \downarrow R = \{x,r\} \cap R\}$, i.e. a register r is in R-postset of a node x iff there exists a path $\delta(x,r)$ with no other registers except x (if x is a register) and r.

Graphical representation

Graphically, the combinational logic nodes are represented as boxes (□), the registers as boxes with two vertical lines (▯), and the edges are depicted by arrows (→). The tokens are usually drawn as filled cycles (•), however, this representation varies for different token game semantics (see Sections 5.3, 5.4 and 5.4).

For example, the SDFS fragment shown in Figure 5.1 consists of 11 nodes: 5 combinational logic nodes ($L1$, $L2$, $L3$, $L4$ and $L5$) and 6 registers ($R1$, $R2$, $R3$, $R4$, $R5$ and $R6$). Note that $R1$ and $R2$ are sources while $R5$ and $R6$ are sinks. The preset of node $L3$ is $\{L2, R3\}$ and it postset is $\{L4\}$; the R-preset of node $L3$ is $\{R2, R3\}$ and its R-postset is $\{R4, R5\}$.

In this section we have formally defined the structure and syntax of SDFS model using [110] as a guideline. The following sections introduce different token game semantics for the SDFS model.

5.2 Atomic token semantics

The atomic token semantic of the SDFS model, or simply *atomic token model*, is a formalisation of the intuitive token game which is presented in [110] on a set of simple examples.

Marking semantics

The *marking* in the atomic token model is defined as a function $M : R \to \{0,1\}$, i.e. a register can contain at most one token. The marking in this model represents data validity. The presence of a token in a register means it stores valid data. The absence of a token in a register represents invalid

data or spacer. Because a register can hold no more than one token, it is convenient to assume that the codomain of the function M is the Boolean domain.

A current marking in atomic token model can be viewed as a front of computation phase in a circuit, followed by a reset phase. Subsequent computation phases must not overlap, therefore for a marked register $r \in R$ all registers in its preset and postset must be unmarked, i.e.:

$$\forall r \in R, q \in {}^\star r \cup r^\star : M(r) \Rightarrow \overline{M(q)}.$$

Evaluation and reset of combinational logic nodes

The *evaluation state* of the atomic token model is a Boolean function $\Xi : L \to \{0,1\}$ which defines if a combinational logic node $l \in L$ has computed its output ($\Xi(l) = 1$) or has not computed it yet ($\Xi(l) = 0$). A node $l \in L$ is said to be in *reset state* if $\Xi(l) = 0$; it is said to be in *evaluated state* if $\Xi(l) = 1$. The switching of a combinational logic node form reset to evaluated state is called *evaluation transition*; its change from evaluated to reset state is called *reset transition*. Note, that words "state" and "transition" are often omitted in the text if it is clear from the context what is referred: the state of a node or its transition from one state to another.

Initially all combinational logic nodes are in reset states. A reset combinational logic node may evaluate iff all the combinational logic nodes in its preset are in evaluated states and all the registers in its preset are marked. This is the *evaluation condition*. Similarly, an evaluated combinational logic node may reset iff all the combinational logic nodes in its preset are in reset states and all the registers in its preset are unmarked. This is the *resetting condition*. For a combinational logic node $l \in L$ the evaluation condition $\xi_+(l)$ and resetting condition $\xi_-(l)$ can be formally expressed as:

$$\xi_+(l) = \bigwedge_{k \in {}^\bullet l \cap L} \Xi(k) \wedge \bigwedge_{q \in {}^\bullet l \cap R} M(q)$$

$$\xi_-(l) = \bigwedge_{k \in {}^\bullet l \cap L} \overline{\Xi(k)} \wedge \bigwedge_{q \in {}^\bullet l \cap R} \overline{M(q)}$$

In other words, a combinational logic node $l \in L$ may evaluate when $\xi_+(l) = 1$ and may reset when $\xi_-(l) = 1$. The evaluation and resetting conditions of atomic token SDFS are similar to

the firing conditions of phased logic [72], where a gate is enabled when all its input phases are opposite to the gate output phase; when an enabled gate fires, its outputs toggle to the opposite phase.

Enabling and disabling of registers

The *enabling state* of the atomic token model is a Boolean function $\Sigma : R \rightarrow \{0,1\}$ which defines if a register $r \in R$ is *disabled* ($\Sigma(r) = 0$) or *enabled* ($\Sigma(r) = 1$).

Initially all unmarked registers are disabled and all marked registers are enabled. A disabled and unmarked register becomes enabled iff all the combinational logic nodes in its preset are evaluated and all the registers in its preset are marked. This is a register *enabling condition.* Similarly, an enabled and marked register becomes disabled iff all the combinational logic nodes in its preset are reset and all the registers in its preset are unmarked. This is a register *disabling condition*. The enabling condition $\sigma_+(r)$ and disabling condition $\sigma_-(r)$ of a register $r \in R$ can be formally represented as follows:

$$\sigma_+(r) = \overline{M(r)} \wedge \bigwedge_{k \in {}^\bullet r \cap L} \Xi(k) \wedge \bigwedge_{q \in {}^\bullet r \cap R} M(q)$$

$$\sigma_-(r) = M(r) \wedge \bigwedge_{k \in {}^\bullet r \cap L} \overline{\Xi(k)} \wedge \bigwedge_{q \in {}^\bullet r \cap R} \overline{M(q)}$$

A register $r \in R$ becomes enabled when $\sigma_+(r) = 1$ and it becomes disabled when $\sigma_-(r) = 1$.

Propagation of tokens

In order to be marked with a token a register must be enabled first; and to be unmarked a register needs to get disabled. Therefore a register cycles through the following four phases: enabling, marking, disabling and unmarking, as shown by a register state graph in Figure 5.2. The state of each register is encoded by a vector $\langle M(r), \Sigma(r) \rangle$. The excited variables (the ones which may change in the current state) of this vector are denoted by '*' symbol on top right. In the initial state 00^*, which is outlined, a register is disabled and unmarked. This register may get enabled, which is denoted by the '*' symbol next to the $\Sigma(r)$ component of the vector. After being enabled it may be marked with a token, then get disabled and finally be unmarked, thus coming to the initial state.

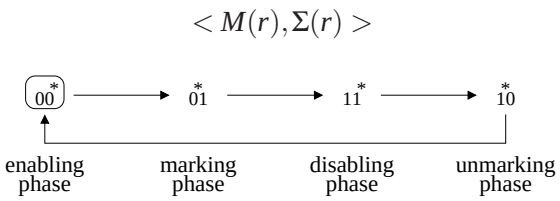


Figure 5.2: Behaviour of a register

To prevent overlapping of tokens from subsequent phases of computation, when propagating a token, the following two conditions have to be satisfied: i) a token can be removed from a disabled register iff all the registers in its R-postset are unmarked; ii) a token can be put into an enabled register iff all the registers in its R-preset are marked. Following these conditions, in a current marking, a set of registers R_- from which tokens can be potentially removed and a set of registers R_+ which can potentially receive tokens are defined as:

$$R_- = \left\{ r \in R \mid M(r) \wedge \overline{\Sigma(r)} \right\}, R_+ = \left\{ q \in R \mid \overline{M(q)} \wedge \Sigma(q) \right\}$$

Token propagation takes place when i) each register in R_- also belongs to R-preset of some register in R_+, i.e.: $\forall r \in R_-, q \in r\star \Rightarrow q \in R_+$; and ii) each register in R_+belongs to R-postset of some register in R_-, i.e. $\forall r \in R_+, q \in \star r \Rightarrow q \in R_-$. When these two conditions hold, the registers in R_- may fire in a single action, removing tokens from all registers of R_- and producing tokens in each register of R_+. The atomic nature of token propagation in this model is similar to firing in Petri nets, where places correspond to registers and transitions correspond to (possibly empty) combinational logic "clouds" between the registers.

This token game semantics works for simple examples but can be problematic for a more complex SDFS. For instance, consider the SDFS in Figure 5.3. At Step 1 only register $R1$ is enabled and has a token. It enables register $R2$ at Step 2 and the token propagates from $R1$ to $R2$ at Step 3 . Now the token in register $R2$ allows combinational logic node $L1$ to evaluate and enable the register $R3$.

Note that $R2$ still cannot fire and produce a token into $R3$, because there is register $R5$ in its R-postset which is still disabled. This results in a concurrency reduction, where the whole branch $\{R3, L2, R4\}$ waits for evaluation of the concurrent branch $\{L3\}$.

Another problem arises at Step 4, when the evaluation of the combinational logic node $L3$ leads to a deadlock. Indeed, after Step 4 the combinational logic node $L4$ cannot evaluate until

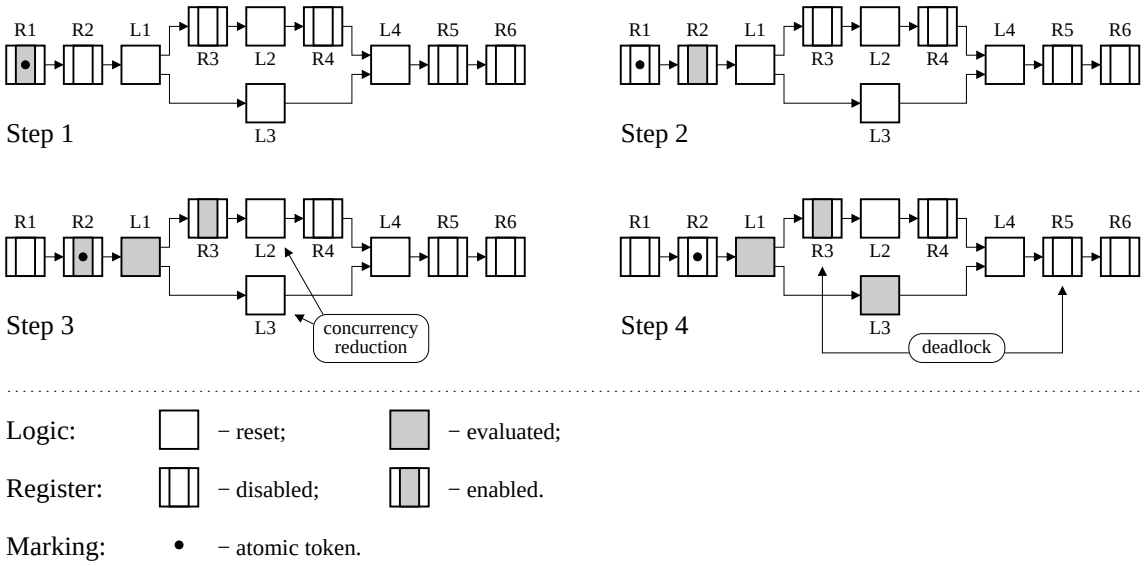


Figure 5.3: Atomic token SDFS example

a token propagates to the register $R3$ and then to $R4$. At the same time the register $R3$ can only receive a token when the combinational logic node $L4$ evaluates and enables register $R5$.

These concurrency reduction and deadlock problems can be avoided in two different ways. The easiest would be to introduce a set of constraints for *well-formed* SDFS. For example, a necessary constraint would be: if one of the concurrent branches contains a register, then all the other branches concurrent to it must also contain a register. However, this approach would significantly restrict the class of circuits the model can capture. A more practical approach is to define the token game rules which would naturally capture the pipeline-style behaviour of the asynchronous data path. For example, the firing can be split in two atomic actions: i) propagation of the tokens into the next-stage registers (can be associated with a request signal in a pipeline), and ii) removing the tokens from the previous stage registers (models an acknowledgement signal). Thus, a token can stretch over a chain of registers before being removed from the beginning of the chain. This token game semantics is called *spread token* and is formally defined in Section 5.3.

5.3 Spread token semantics

The spread token semantics of the SDFS model, or simply *spread token model*, is an extension of the atomic token semantics. It models asynchronous circuits of Muller pipeline architecture.

The spread token semantics does not capture preemption or token borrowing. However, it can be extended to model *token borrowing* as in low-latency structures with slack [35].

Marking, evaluation and enabling

The marking semantics, evaluation and reset of combinational logic nodes, and also enabling and disabling of registers in this model are exactly the same as in atomic token. The only difference is in the way tokens propagate from a register to a register. Also a concept of early evaluation will be introduced in this model, which has not been discussed in [110]. Therefore, in this section we concentrate on modelling the early evaluation and the new rules of token propagation. The rest of the terminology is adopted from Section 5.2.

Early evaluation

It is often sufficient to have only a subset of the inputs ready to evaluate a combinational logic node. This is called *early evaluation* and can be modelled by modifying the evaluation condition ξ_0 of the node. For example, a combinational logic node $l \in L$ which evaluates as soon as any of its inputs is ready, has the following evaluation condition:

$$\xi_+(l) = \bigvee_{k \in {}^\bullet l \cap L} \Xi(k) \vee \bigvee_{q \in {}^\bullet l \cap R} M(q).$$

Modification of the evaluation and resetting conditions is not limited to early evaluation. In fact, any *reasonable* expressions can be assigned to conditions $\xi_+(l)$ and $\xi_-(l)$ of a combinational logic node $l \in L$. For example, it is reasonable to assume that $\xi_+(l) \wedge \xi_-(l) \equiv 0$, i.e. evaluation and resetting conditions of a node are mutually exclusive to prevent a node from enabling and resetting at the same time. Also it is reasonable to assume that evaluation and resetting conditions depend on the marking and evaluation state of SDFS, i.e. they are not constant 1 or constant 0.

The concept of early evaluation can also be applied to enabling and disabling of a register. However, the same result can be achieved by splitting such a register into an early evaluation combinational logic node and the register itself. Therefore, the notion of early evaluation is restricted to combinational logic nodes.

Propagation of tokens

A token can be put into an enabled register iff all the registers in its R-preset are marked and all

the registers in its R-postset are unmarked. This is a *marking condition*. Similarly, a token can be removed from a disabled register iff all the registers in its R-preset are unmarked an all the registers in its R-postset are marked. This is *unmarking condition*. Formally this can be represented by assigning each register $r \in R$ a marking condition $m_+(r)$ and unmarking condition $m_-(r)$*:*

$$m_+(r) = \Sigma(r) \wedge \bigwedge_{q \in \star r} M(q) \wedge \bigwedge_{s \in r\star} \overline{M(s)}$$

$$m_-(r) = \overline{\Sigma(r)} \wedge \bigwedge_{q \in \star r} \overline{M(q)} \wedge \bigwedge_{s \in r\star} M(s)$$

A register $r \in R$ can be marked with a token when $m_+(r) = 1$ and can be unmarked when $m_-(r) = 1$.

Note that an unmarked source is always enabled because its R-preset is empty. Therefore, a new token can be put into an enabled source as soon as its R-postset is unmarked. Similarly, a token can be removed from a disabled sink as soon as its R-preset is unmarked (its R-postset is empty). These features of the source and sink registers are useful to model the communication with the environment which produces new tokens and consumes processed ones.

Consider the spread token model on a simple example of Figure 5.4. Enabled registers and evaluated combinational logic nodes are highlighted. Note that combinational logic node $L4$ is labelled with EE tag. This tag means the node exhibits early evaluation, i.e. for $L4$ to evaluate it is sufficient to have $R4$ marked or $L3$ evaluated: $\xi_+(L4) = M(R4) \vee \Xi(L3)$. Therefore, on Step 2, when token propagates to $R2$, combinational logic nodes $L3$ and $L4$ evaluate and register $R5$ becomes enabled. However, $R5$ cannot be marked with a token until all registers in its R-preset are marked. For this to happen two more steps are needed: at Step 3 the register $R3$ is marked and at Step 4 token propagates to $R4$. At Step 5 register $R5$ is finally marked. Similarly, the token cannot be removed from $R2$ until Step 6 when all registers in its R-postset are marked. Because of these restrictions the token spreads over four registers (at least four, as a tokens could still stay in $R1$ and $R2$) at Step 5. Finally, the tokens are removed one by one from the tail of the spread token, as shown at Steps 6-8.

The spread token model solves the concurrency reduction and deadlock problems of the atomic

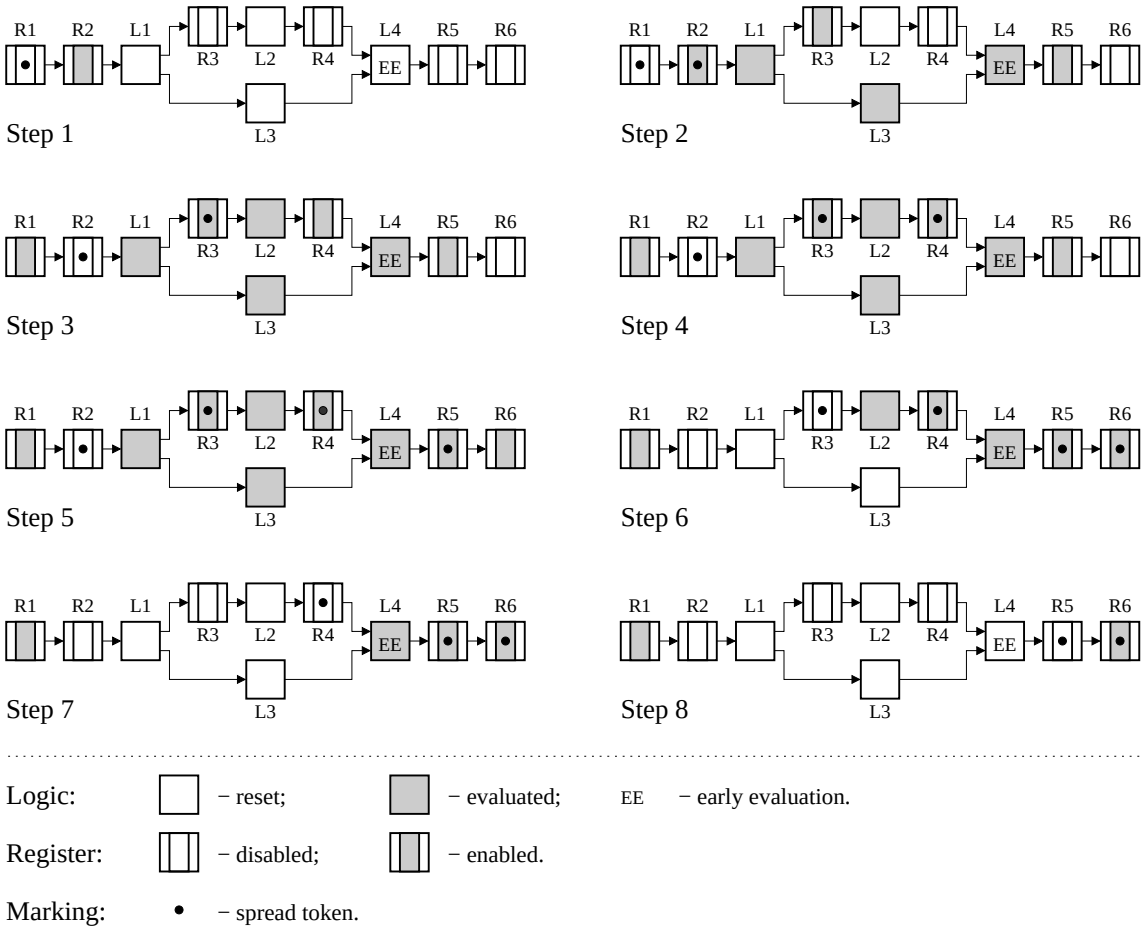


Figure 5.4: Spread token SDFS example

token semantics. It also has some rudimentary means to model early evaluation. However, in this model a register can only accept a token when all the registers in its R-preset are marked. This limits the early evaluation to one pipeline stage only and makes the model unusable for capturing preemption and speculation.

It would be natural to allow further propagation of a token into an enabled register without waiting for all the tokens in its R-preset, but there is a risk of mixing tokens from different computation cycles. In order to avoid this mixture, when a token propagates into an enabled register, all unmarked registers in its R-preset should be marked with a *negative token*. The next data token to arrive into a register with negative marking must be ignored as it carries old data. Therefore the data token and the negative token cancel each other. The described technique is called *token borrowing*. Different types of token borrowing and one of SDFS models implementing this technique are discussed in Section 5.4.

5.4 Counterflow semantics

The token borrowing techniques can be partitioned into two classes: *passive borrowing* and *active borrowing*. In the passive borrowing a special join block is responsible for counting the number of tokens borrowed from each of its inputs. The borrowing does not propagate further in the direction opposite to the token flow. The passive borrowing is introduced as a feature of the *change diagrams* model and is also modelled by unsafe (places can be marked with more than one token) Petri nets [127]. A model and an implementation of a join element capable of unbounded borrowing are presented in [35]. The main disadvantage of the passive borrowing is the lack of preemption mechanism in the unwanted branches, which may result in a higher power consumption and longer computation time.

The *active borrowing* is characterised by negative tokens which are able to propagate in the direction reverse to the data token flow. When a data token and a negative token collide, they are both eliminated. The major drawback of this technique is caused by the resolution of the conflicts when a data token and a negative token want to occupy the same register simultaneously. Usually, such conflicts result in arbitration which cause significant implementation overheads (increase in circuit size, power consumption and latency). On the positive side, preemption is captured

naturally by active borrowing.

Both passive and active borrowing can be defined as token game semantics for the SDFS model. In this thesis we model active borrowing only, as it is somewhat superior to passive borrowing and is advantageous for implementing the preemption mechanism. There are two SDFS model of active borrowing mechanism, namely *antitoken model* and *counterflow model*.

The antitoken semantics of the SDFS model, or simply antitoken model, is based on the idea of the two pipelines of opposite directions, one for data tokens and the other for negative tokens. Data tokens and negative tokens eliminate each other on collision. Similar idea is employed in *counterflow pipeline processor* (CFPP) [111] which allows instructions to move one way along a processing pipeline while results flow freely in the opposite direction; when collide instructions are executed on the corresponding data.

The main disadvantage of the antitoken model is that in order not to miss each other, data tokens and negative tokens must synchronise within each pipeline stage. This requires arbitration which is associated with metastability problems at the level of circuit implementation. The arbitration problem is avoided in counterflow semantics of SDFS model which is the main focus of this section. For more details on antitoken SDFS semantics the reader is referred to [107].

The counterflow semantics of SDFS model, or simply counterflow model, is based on the idea of OR-causality [127], which allows to avoid arbitration inherent in antitoken model. Data tokens and negative tokens are not distinguished in this model at the level of individual stages: the first to arrive propagates in both directions (as a data token forward and as a negative token backward), the second one is ignored. The idea of antitokens without arbitration is introduced in [33, 32] and is revisited with minor modifications in [22].

Marking semantics

In the counterflow SDFS model there are two types of tokens: *OR-tokens* and *AND-tokens*. The marking in the counterflow model is defined as $M = M^{OR} \times M^{AND}$, where the M^{OR} and M^{AND} are Boolean functions: $M^{OR} : R \to \{0,1\}$ is the *OR-marking* and $M^{AND} : R \to \{0,1\}$ is the *AND-marking*. The presence of an OR-token in a register means either that data has been received from its R-preset or that data is not needed anymore by its R-postset (e.g. due to early evaluation form another branch). An AND-token in a register means that data has been received from its R-preset

and has been consumed (or ignored, in case of early evaluation) by its R-postset. Graphically, an OR-token is depicted as a filled triangle (▲) while an AND-token as a filled box (■).

Evaluation and reset of combinational logic nodes

Forward evaluation state of SDFS is a Boolean function $\Xi^F : L \to \{0,1\}$ which defines if a combinational logic node $l \in L$ has computed its output ($\Xi^F(l) = 1$) or has not computed it yet ($\Xi^F(l) = 0$). A node $l \in L$ is said to be *forward evaluated* if $\Xi^F(l) = 1$ and *forward reset* if $\Xi^F(l) = 0$. Initially all combinational logic nodes are forward reset. A forward reset combinational logic node may forward evaluate iff all the combinational logic nodes in its preset are forward evaluated and all the registers in its preset have OR-tokens. Similarly, a forward evaluated combinational logic node may forward reset iff all the combinational logic nodes in its preset are forward reset and all the registers in its preset do not have OR-tokens. These are *forward evaluation condition* and *forward resetting condition* respectively.

Backward evaluation state of SDFS is a Boolean function $\Xi^B : L \to \{0,1\}$ which defines if the output of a combinational logic node $l \in L$ has been consumed and is not needed any longer ($\Xi^B(l) = 1$) or the output has not been received yet and is still awaited ($\Xi^B(l) = 0$). A combinational logic node may *backward evaluate* iff all the combinational logic nodes in its postset are backward evaluated and all the registers in its postset have OR-tokens. Similarly, a backward evaluated combinational logic node may *backward reset* iff all the combinational logic nodes in its postset are backward reset and all registers in its postset do not have OR-tokens. These are *backward evaluation condition* and *backward resetting condition*.

Formally, the forward evaluation condition $\xi_+^F(l)$ and the forward resetting condition $\xi_-^F(l)$ of a combinational logic node $l \in L$ can be expressed as:

$$\xi_+^F(l) = \bigwedge_{k \in \bullet l \cap L} \Xi^F(k) \wedge \bigwedge_{q \in \bullet l \cap R} M^{OR}(q)$$

$$\xi_-^F(l) = \bigwedge_{k \in \bullet l \cap L} \overline{\Xi^F(k)} \wedge \bigwedge_{q \in \bullet l \cap R} \overline{M^{OR}(q)}$$

Similarly, the backward evaluation condition $\xi_+^B(l)$ and the backward resetting condition $\xi_-^B(l)$ are:

$$\xi_+^B(l) = \bigwedge_{k \in l\bullet \cap L} \Xi^B(k) \wedge \bigwedge_{q \in l\bullet \cap R} M^{OR}(q)$$

$$\xi_-^B(l) = \bigwedge_{k \in l\bullet \cap L} \overline{\Xi^B(k)} \wedge \bigwedge_{q \in l\bullet \cap R} \overline{M^{OR}(q)}$$

A combinational logic node $l \in L$ may forward evaluate when $\xi_+^F(l) = 1$ and it may forward reset when $\xi_-^F(l) = 1$. Similarly, a combinational logic node $l \in L$ may backward evaluate when $\xi_+^B(l) = 1$ and it may backward reset when $\xi_-^B(l) = 1$.

The above conditions do not allow early forward (backward) evaluation because the change on all the node inputs (outputs) is required to change its forward (backward) state. By analogy with spread token model, the effect of early evaluation in counterflow semantics can be modelled by modifying the evaluation and resetting conditions of a combinational logic node, so, that a subset of node inputs (outputs) is sufficient to trigger its forward (backward) state.

Enabling and disabling of registers

Forward enabling state of SDFS is a Boolean function $\Sigma^F : R \to \{0,1\}$ which defines if a register $r \in R$ is *forward enabled* ($\Sigma^F(r) = 1$) or *forward disabled* ($\Sigma^F(r) = 0$). Similarly, *backward enabling state* of SDFS is a Boolean function $\Sigma^B : R \to \{0,1\}$ which defines if a register $r \in R$ is *backward enabled* ($\Sigma^B(r) = 1$) or *backward disabled* ($\Sigma^B(r) = 0$).

Initially all registers without AND-tokens are both forward disabled and backward disabled. All registers which are marked with AND-tokens are both forward enabled and backward enabled.

A register without an AND-token becomes forward enabled iff all the combinational logic nodes in its preset are forward evaluated and all the registers in its preset have OR-tokens. A register with an AND-token becomes forward disabled iff all the combinational logic nodes in its preset are forward reset and all the registers in its preset do not have OR-tokens. These are *forward enabling condition* and *forward disabling condition*. Note that a source without an AND-token becomes forward enabled and a source with AND-token becomes forward disabled (because its preset is empty).

A register without an AND-token becomes backward enabled iff all the combinational logic nodes in it postset are backward evaluated and all the registers in its postset have OR_tokens. A

register with an AND-token becomes backward disabled iff all the combinational logic nodes in its postset are backward reset and all the registers in its postset do not have OR-tokens. These are *backward enabling condition* and *backward disabling condition* respectively. Note that a sink without AND-token becomes backward enabled and a sink with AND-token becomes backward disabled (because its postset is empty).

Formally, the forward enabling condition $\sigma_+^F(r)$ and the forward disabling condition $\sigma_-^F(r)$ of a register $r \in R$ is defined as:

$$\sigma_+^F(r) = \overline{M^{AND}(r)} \wedge \bigwedge_{k \in {}^\bullet r \cap L} \Xi^F(k) \wedge \bigwedge_{q \in {}^\bullet r \cap R} M^{OR}(q)$$

$$\sigma_-^F(r) = M^{AND}(r) \wedge \bigwedge_{k \in {}^\bullet r \cap L} \overline{\Xi^F(k)} \wedge \bigwedge_{q \in {}^\bullet r \cap R} \overline{M^{OR}(q)}$$

The backward enabling condition $\sigma_+^B(r)$ and the backward disabling condition $\sigma_-^B(r)$ formally are:

$$\sigma_+^B(r) = \overline{M^{AND}(r)} \wedge \bigwedge_{k \in r^\bullet \cap L} \Xi^B(k) \wedge \bigwedge_{q \in r^\bullet \cap R} M^{OR}(q)$$

$$\sigma_-^B(r) = M^{AND}(r) \wedge \bigwedge_{k \in r^\bullet \cap L} \overline{\Xi^B(k)} \wedge \bigwedge_{q \in r^\bullet \cap R} \overline{M^{OR}(q)}$$

A register $r \in R$ becomes forward enabled when $\sigma_+^F(r) = 1$ and it becomes forward disabled when $\sigma_-^F(r) = 1$. Similarly, a register $r \in R$ becomes backward enabled when $\sigma_+^B(r) = 1$ and it becomes backward disabled when $\sigma_-^B(r) = 1$.

Propagation of tokens

A register can be marked with an OR-token iff it does not have an AND-token, it is either forward enabled or backward enabled and neither its R-preset nor its R-postset is marked with AND-token. An OR-token can be removed from a register iff it is marked with AND-token, it is either forward disabled or backward disabled and its R-preset and R-postset are both marked with AND-tokens. These are *OR-marking condition* $m_+^{OR}(r)$ and *OR-unmarking condition* $m_-^{OR}(r)$ of a register $r \in R$,

which are formally defined as:

$$m_+^{OR}(r) = \overline{M^{AND}(r)} \wedge \left(\Sigma^F(r) \vee \Sigma^B(r)\right) \wedge \bigwedge_{q \in \star r \cup r\star} \overline{M^{AND}(q)}$$

$$m_-^{OR}(r) = M^{AND}(r) \wedge \left(\overline{\Sigma^F(r)} \vee \overline{\Sigma^B(r)}\right) \wedge \bigwedge_{q \in \star r \cup r\star} M^{AND}(q)$$

A register can be marked with an AND-token iff it has an OR-token and it is both forward enabled and backward enabled and its R-preset and R-postset are marked with OR-tokens. An AND-token can be removed from a register iff it does not have an OR-token and it is both forward disabled and backward disabled and its R-preset and R-postset are not marked with OR-tokens. These are OR-marking and OR-unmarking conditions. These *AND-marking condition* $m_+^{AND}(r)$ and *AND-unmarking condition* $m_-^{AND}(r)$ are assigned to each register $r \in R$ as follows:

$$m_+^{AND}(r) = M^{OR}(r) \wedge \Sigma^F(r) \wedge \Sigma^B(r) \wedge \bigwedge_{q \in \star r \cup r\star} M^{OR}(q)$$

$$m_-^{AND}(r) = \overline{M^{OR}(r)} \wedge \overline{\Sigma^F(r)} \wedge \overline{\Sigma^B(r)} \wedge \bigwedge_{q \in \star r \cup r\star} \overline{M^{OR}(q)}$$

A register $r \in R$ can be marked with an OR-token when $m_+^{OR}(r) = 1$ and the OR-token can be removed when $m_-^{OR}(r) = 1$. Similarly, a register $r \in R$ can be marked with an AND-token when $m_+^{AND}(r) = 1$ and the AND-token can be removed when $m_-^{AND}(r) = 1$.

A counterflow register operation is represented by the state graph in Figure 5.5. Each state of the graph is encoded by a vector $\left\langle M^{AND}(r), M^{OR}(r), \Sigma^B(r), \Sigma^F(r)\right\rangle$. In the initial state 000^*0^*, which outlined by a box, a register is both forward and backward disabled and does not have tokens. This register may be forward and/or backward enabled, which is denoted by '*' symbol next to $\Sigma^B(r)$ and $\Sigma^B(r)$ variables. Changing any of the excited variables leads to the next state, where the variable $M^{OR}(r)$ becomes excited, i.e. the register may be marked with an OR-token, and so on.

Note the states where two variables are excited, e.g. the state 00^*0^*1. Changing one of the excited variables does not remove the excitement from the other one. Eventually both excited variables have to switch leading to the same state 0^*111. It is also possible for both excited variables to change simultaneously, which is depicted by dotted arcs.

$$< M^{AND}(r), M^{OR}(r), \Sigma^{B}(r), \Sigma^{F}(r) >$$

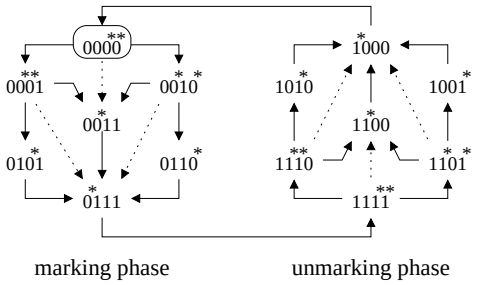


Figure 5.5: Behaviour of counterflow register

There are two distinctive phases in the operation of a counterflow register: *marking phase* and *unmarking phase*. At the marking phase a register gets enabled (forward and/or backward), then marked with OR-token and finally marked with AND-token. At the unmarking phase it is first disabled (forward and/or backward), then the OR-token leaves the register and finally the AND-token is removed.

Figure 5.6 illustrates the counterflow SDFS semantics on a simple example. Forward (backward) enabled registers and forward (backward) evaluated combinational logic nodes are highlighted on top (bottom). The combinational logic node $L4$ labelled with EE tag exhibits early forward evaluation: $\xi^{F}_{+}(L4) = M^{OR}(R4) \vee \Xi^{F}(L3)$, $\xi^{F}_{-}(L4) = \overline{M^{OR}(R4)} \wedge \overline{\Xi^{F}(L3)}$.

At Step 1 only register $R1$ has an OR-token, which forward enables register $R2$ (this models a request signal in the circuit). The OR-token propagates to $R2$ at Step 2 and backward enables register $R1$ (this models an acknowledgement signal). Also the combinational logic nodes $L1$, $L3$ and $L4$ evaluate at this step (note that $L4$ exhibits early evaluation). This allows forward enabling of registers $R3$ and $R5$. At Step 3 an AND-token is produced in register $R1$ because it is both forward enabled and backward enabled; as AND-token appears in $R1$ and it is a source, it becomes forward disabled. Also the OR-tokens propagate to forward enabled registers $R3$ and $R5$. Now register $R4$ becomes both forward enabled and backward enabled. As it does not have a token yet, first, an OR-token is generated in $R4$ at Step 4. After that, at Step 5, an AND-tokens appear registers $R3$ and $R4$ as they are both forward enabled and backward enabled. At Step 6 OR-token disappears from the forward disabled register $R2$, which leads to forward disabling of $R3$. OR-token leaves the forward disabled register $R3$ and register $R4$ becomes forward disabled at Step 7, therefore

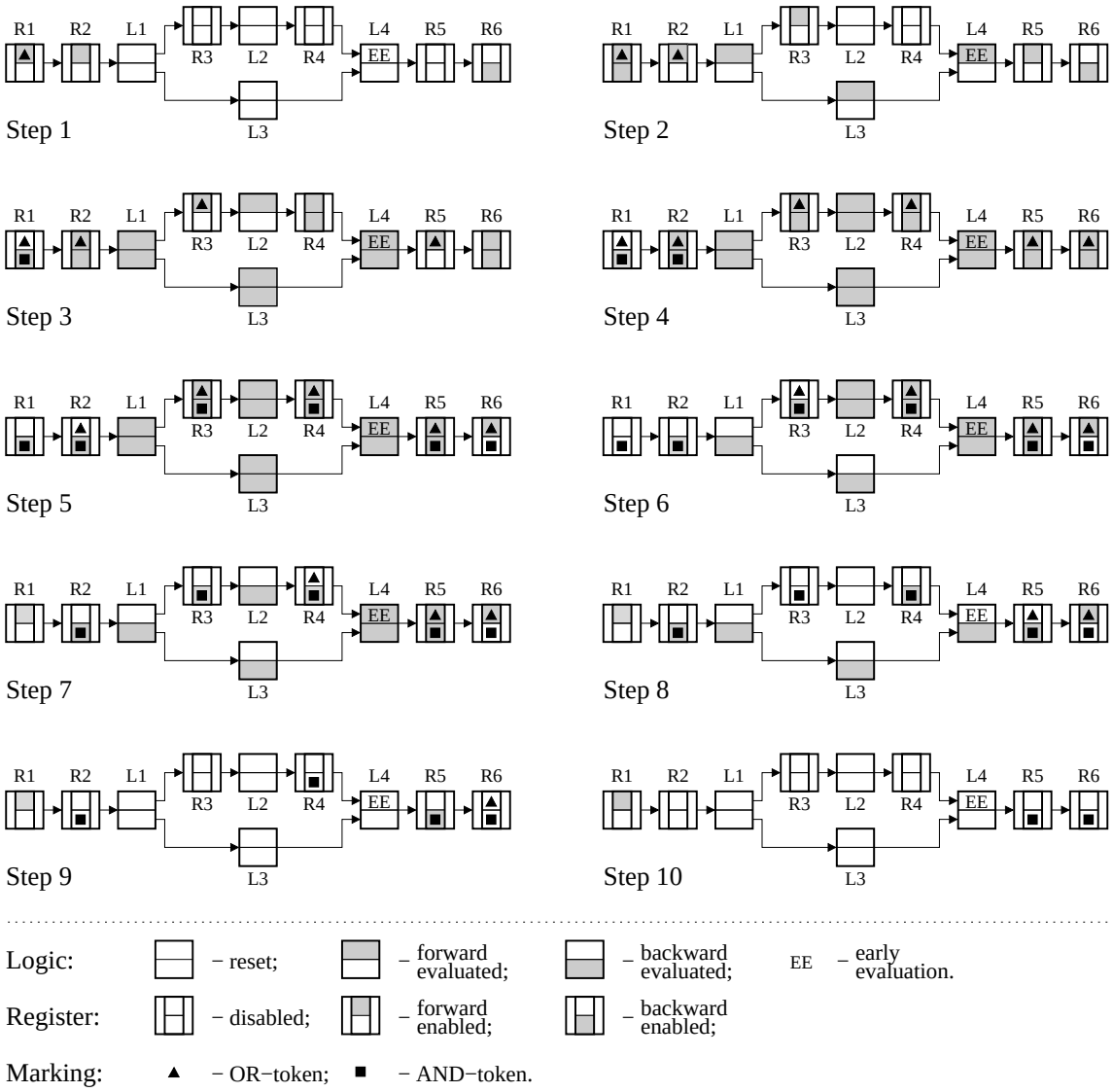


Figure 5.6: Counterflow SDFS example

OR-token is removed form $R4$ which forward disables $R5$ at Step 8. Now, at Step 9, OR-token leaves $R5$ and registers $R2$ and $R4$ become both forward disabled and backward disabled, therefore AND-tokens can be removed from them, as show at Step 10.

Note that at Step 4 it does not matter which register, $R3$ or $R5$, initiates the OR-token in $R4$ - the resultant marking is the same. Thus, the merge of the data tokens (moving in forward direction) and the negative tokens (moving in backward direction) is modelled by OR-causality instead of arbitration. This is the main advantage of the counterflow model over the antitoken model.

The major drawback of the counterflow model is the complex behaviour of its registers. It

is difficult to design a fully indicating and hazard-free controller for counterflow registers. Interesting implementations of such controller were proposed in [32, 22]. Due to the complexity of the counterflow protocol these implementations are several times larger than a Muller pipeline stage. This is particularly disadvantageous when no token borrowing is actually possible. For example, consider a long linear pipeline with a small section S having parallel branches, e.g. for speculative computation. The token borrowing is only possible within section S, but in order to satisfy the counterflow protocol the whole pipeline has to be implemented using large counterflow controllers.

A combination of counterflow pipeline (for the sections which require preemption) and Muller pipeline (for the rest of the circuit) is a promising way to build asynchronous data path. Such data path has all advantages of counterflow pipelines (no arbitration, preemption, early evaluation, speculation) for the price of moderate area increase compared to Muller pipeline. The hybrid data path can be modelled by combining spread token and counterflow semantics of SDFS model as is described in Section 5.5.

5.5 Hybrid semantics

The idea of combining a counterflow pipeline with a Muller pipeline originates from [107], where PN models and gate-level implementations for converters between different pipeline types were proposed. The subject of this section is to capture the behaviour of such hybrid pipeline in special SDFS model, which is a combination of spread token and counterflow models. The main idea for this model is that only those parts of data path which may exhibit preemption should be modelled by the counterflow semantics while the rest of the data path should have the spread token semantics. Such a syndication of the token game semantics is called a *hybrid* SDFS model. One of the ways to achieve this hybrid functionality is to introduce a pair of converters between the spread token SDFS nodes and the counterflow SDFS nodes. For this the set of SDFS registers R needs to be extended with a special kind of registers $C \subseteq R$, which have spread token type of interface on one side and counterflow interface on the other side.

A *spread token to counterflow* (ST2CF) converter behaves as a spread token register to its preset and as a counterflow register to its postset. Only nodes with spread token semantics are allowed

in the preset of an ST2CF converter and only nodes with counterflow semantics are allowed in its postset. The set of ST2CF converters is denoted as $C^{ST2CF} \subseteq C$.

A *counterflow to spread token* (CF2ST) converter appears as a counterflow register to its preset and as a spread token register to its postset. The preset of a CF2ST converter can only contain nodes with counterflow semantics, while its postset only allows nodes with spread token semantics. The set of ST2CF converters is denoted as $C^{CF2ST} \subseteq C$.

The behaviour of ST2CF and CF2ST converters is somewhat symmetrical. They are used in pairs forming structures of fork-join type. An ST2CF converter is used as fork interface from a part of the data path without early propagation to the part with several concurrent branches where preemption mechanism is employed. These concurrent branches are subsequently joined into a CF2SF converter which limits the early propagation and preemption to the fork-join part of the data path.

Marking semantics

The ST2CF and CF2ST converters should be able to accept three types of tokens: ordinary tokens (used in spread token model), OR-tokens and AND-tokens (used in counterflow model). Therefore the marking of the converters is defined as $M^C = M \times M^{OR} \times M^{AND}$, where function $M : C \to \{0,1\}$ is spread token marking, $M^{OR} : C \to \{0,1\}$ is OR-marking and $M^{AND} : C \to \{0,1\}$ is AND-marking.The semantics of these markings are the same as in spread token and counterflow models, respectively.

Enabling state

The hybrid enabling state for SDFS converters comprises of three components. The first component is enabling state $\Sigma : C \to \{0,1\}$ for the spread token part of all converters. The other two are $\Sigma^F : C \to \{0,1\}$ and $\Sigma^B : C \to \{0,1\}$ which are forward enabling and backward enabling states of the counterflow parts. The semantics of these enabling states are the same as for the registers of spread token model and counterflow model.

Operation of ST2CF converter

The enabling and disabling conditions for the spread token part of an ST2CF converter $c \in C^{ST2CF}$

are the same as for a spread token register:

$$\sigma_+(c) = \overline{M(c)} \wedge \bigwedge_{k \in {}^\bullet c \cap L} \Xi(k) \wedge \bigwedge_{q \in {}^\bullet c \cap R} M(q)$$

$$\sigma_-(c) = M(c) \wedge \bigwedge_{k \in {}^\bullet c \cap L} \overline{\Xi(k)} \wedge \bigwedge_{q \in {}^\bullet c \cap R} \overline{M(q)}$$

The spread token part of an ST2CF converter $c \in C^{ST2CF}$ becomes enabled when $\sigma_+(c) = 1$ and it becomes disabled when $\sigma_-(c) = 1$.

The forward enabling and forward disabling conditions for the counterflow part of an ST2CF converter $c \in C^{ST2CF}$ are similar to those of a counterflow register. The major simplification is because an ST2CF converter does not have any counterflow nodes in its preset and the marking of its spread token part is taken into account instead:

$$\sigma^F_+(c) = \overline{M^{AND}(c)} \wedge M(c);\ \sigma^F_-(c) = M^{AND}(c) \wedge \overline{M(c)}$$

The backward enabling and backward disabling conditions are the same as for a counterflow register:

$$\sigma^B_+(c) = \overline{M^{AND}(c)} \wedge \bigwedge_{k \in c^\bullet \cap L} \Xi^B(k) \wedge \bigwedge_{s \in c^\bullet \cap R} M^{OR}(s)$$

$$\sigma^B_-(c) = M^{AND}(c) \wedge \bigwedge_{k \in c^\bullet \cap L} \overline{\Xi^B(k)} \wedge \bigwedge_{s \in c^\bullet \cap R} \overline{M^{OR}(s)}$$

The counterflow part of an ST2CF converter $c \in C^{ST2CF}$ becomes forward enabled when $\sigma^F_+(c) = 1$ and it becomes forward disabled when $\sigma^F_-(c) = 1$. Similarly, it becomes backward enabled when $\sigma^B_+(c) = 1$ and it becomes backward disabled when $\sigma^B_-(c) = 1$.

Once the spread token part of an ST2CF converter is enabled, it may accept a spread token, providing all the spread token registers in its R-preset are marked and its counterflow part does not have an OR-token. When the spread token part becomes disabled it may lose the token. Formally, these marking conditions are:

$$m_+(c) = \overline{M^{OR}(c)} \wedge \Sigma(c) \wedge \bigwedge_{q \in \star c} M(q)$$

$$m_-(c) = M^{OR}(c) \wedge \overline{\Sigma(c)} \wedge \bigwedge_{q \in \star c} \overline{M(q)}$$

The spread token part of a converter $c \in C^{ST2CF}$ can be marked with a token when $m_+(c) = 1$ and can be unmarked when $m_-(c) = 1$.

The marking and unmarking conditions for the counterflow part of an ST2CF converter are identical to those of counterflow register. The only simplification is that there are no counterflow nodes in the preset of a ST2CF converter:

$$m_+^{OR}(c) = \overline{M^{AND}(c)} \wedge \left(\Sigma^F(c) \vee \Sigma^B(c)\right) \wedge \bigwedge_{q \in c\star} \overline{M^{AND}(q)}$$

$$m_-^{OR}(c) = M^{AND}(c) \wedge \left(\overline{\Sigma^F(c)} \vee \overline{\Sigma^B(c)}\right) \wedge \bigwedge_{q \in c\star} M^{AND}(q)$$

$$m_+^{AND}(c) = M^{OR}(c) \wedge \Sigma^F(c) \wedge \Sigma^B(c) \wedge \bigwedge_{q \in c\star} M^{OR}(q)$$

$$m_-^{AND}(c) = \overline{M^{OR}(c)} \wedge \overline{\Sigma^F(c)} \wedge \overline{\Sigma^B(c)} \wedge \bigwedge_{q \in c\star} \overline{M^{OR}(q)}$$

The counterflow part of a converter $c \in C^{ST2CF}$ can be marked with an OR-token when $m_+^{OR}(c) = 1$ and the OR-token can be removed when $m_-^{OR}(c) = 1$. Similarly, it can be marked with an AND-token when $m_+^{AND}(c) = 1$ and the AND-token can be removed when $m_-^{AND}(c) = 1$.

Operation of CF2ST converter

The forward enabling and forward disabling conditions for the counterflow part of a CF2ST converter $c \in C^{CF2ST}$ are identical to those of a counterflow register:

$$\sigma_+^F(c) = \overline{M^{AND}(c)} \wedge \bigwedge_{k \in \bullet c \cap L} \Xi^F(k) \wedge \bigwedge_{q \in \bullet c \cap R} M^{OR}(q)$$

$$\sigma_{-}^{F}(c) = M^{AND}(c) \wedge \bigwedge_{k \in {}^{\bullet}c^{\bullet} \cap L} \overline{\Xi^{F}(k)} \wedge \bigwedge_{q \in {}^{\bullet}c^{\bullet} \cap R} \overline{M^{OR}(q)}$$

For the backward enabling and backward disabling conditions there is a significant simplification compared to the counterflow registers. This is due to the fact that there is no counterflow nodes in the postset of a CF2ST converter and the marking of its spread token part is taken into account instead:

$$\sigma_{+}^{B}(c) = \overline{M^{AND}(c)} \wedge M(c);\ \sigma_{-}^{B}(c) = M^{AND}(c) \wedge \overline{M(c)}$$

The counterflow part of a CF2ST converter becomes forward enabled when $\sigma_{+}^{F}(c) = 1$ and it becomes forward disabled when $\sigma_{-}^{F}(c) = 1$. Similarly, it becomes backward enabled when $\sigma_{+}^{B}(c) = 1$ and it becomes backward disabled when $\sigma_{-}^{B}(c) = 1$.

The spread token part of a CF2ST converter $c \in C^{CF2ST}$ becomes enabled when there is an OR-token in its counterflow part; it becomes disabled when the OR-token leaves the converter. These enabling and disabling conditions can be formalised as:

$$\sigma_{+}(c) = \overline{M(c)} \wedge M^{OR}(c);\ \sigma_{-}(c) = M(c) \wedge \overline{M^{OR}(c)}$$

Marking and unmarking conditions of the counterflow part of a CF2ST converter are similar to those of a counterflow register. Formally, for a CF2ST converter $c \in C^{CF2ST}$ the OR-making/unmarking conditions and AND-marking/unmarking conditions are:

$$m_{+}^{OR}(c) = \overline{M^{AND}(c)} \wedge \left(\Sigma^{F}(c) \vee \Sigma^{B}(c)\right) \wedge \bigwedge_{q \in \star c} \overline{M^{AND}(q)}$$

$$m_{-}^{OR}(c) = M^{AND}(c) \wedge \left(\overline{\Sigma^{F}(c)} \vee \overline{\Sigma^{B}(c)}\right) \wedge \bigwedge_{q \in \star c} M^{AND}(q)$$

$$m_{+}^{AND}(c) = M^{OR}(c) \wedge \Sigma^{F}(c) \wedge \Sigma^{B}(c) \wedge \bigwedge_{q \in \star c} M^{OR}(q)$$

$$m_{-}^{AND}(c) = \overline{M^{OR}(c)} \wedge \overline{\Sigma^{F}(c)} \wedge \overline{\Sigma^{B}(c)} \wedge \bigwedge_{q \in \star c} \overline{M^{OR}(q)}$$

The counterflow part of a converter $c \in C^{CF2ST}$ can be marked with an OR-token when $m_{+}^{OR}(c) = 1$ and the OR-token can be removed when $m_{-}^{OR}(c) = 1$. Similarly, it can be marked with an AND-token when $m_{+}^{AND}(c) = 1$ and the AND-token can be removed when $m_{-}^{AND}(c) = 1$.

Finally, the marking and unmarking of the spread token part of a CF2ST converter $c \in C^{CF2ST}$ are determined by the following conditions:

$$m_{+}(c) = \Sigma(c) \wedge \bigwedge_{s \in c\star} \overline{M(s)};\ m_{-}(c) = \overline{\Sigma(c)} \wedge \bigwedge_{s \in c\star} M(s)$$

These conditions are derived from the marking and unmarking conditions for the spread token register, assuming there is no spread token register in the R-preset of a CF2ST controller. The spread token part of a converter $c \in C^{CF2ST}$ can be marked with a token when $m_{+}(c) = 1$ and can be unmarked when $m_{-}(c) = 1$.

Consider the operation of the hybrid SDFS model on a simple example shown in Figure 5.7. At Step 1 only ST2CF converter $R2$ is enabled and a token propagates into it as Step 2. This forward enables the counterflow part of the controller and it gets an OR-token at Step 3; the counterflow register $R3$ and the CF2ST converter $R5$ are forward enabled now. Also the tail of spread token is removed from disabled register $R1$ at this step. At Step 4 both forward enabled register $R3$ and forward enabled CF2ST converter $R5$ get marked with OR-tokens and backward enable the counterflow part of ST2CF converter $R2$. The OR-token in register $R3$ also forward enables register $R4$ and the OR-token in CF2ST converter $R5$ enables its spread token part. At Step 5 the counterflow part of ST2CF converter $R2$ is marked with AND-token because it has an OR-token and is both forward enabled and backward enabled. Also a token propagates to the spread token part of the CF2ST converter $R5$. At Step 6 a token is removed from the disabled spread token part of the ST2CF converter $R2$; also a token propagates from the CF2ST converter $R5$ to the register $R6$. The forward disabled counterflow part of the ST2CF converter $R2$ is freed of OR-token at Step 7, which forward disables the register $R3$. At Steps 8 and 9 OR-tokens first leave the register $R3$ and then the register $R4$, which forward disables the counterflow part of the

CF2ST converter $R5$. Now OR-token disappears from the forward disabled CF2ST converter $R5$, thus disabling its spread token part. Also the register $R4$ and the ST2CF converter $R2$ become backward disabled, see Step 10. Finally, the rest of the registers return to the initial state at Steps 11 and 12.

5.6 Verification of SDFS models

Direct verification of the SDFS models is a difficult task as there are no formal methods and no software tools to do this. It is however reasonable to reuse the variety of verification methods and model checking tools developed for Petri nets. In order to do this a conversion technique is required, which maps SDFS models into equivalent Petri nets.

An SDFS model with its token game semantics is a high level paradigm. At the low level this model can be viewed as a Petri net, or more precisely an STG, in which each state variable of the SDFS model is represented by an *elementary cycle*.

An elementary cycle models a state of a binary variable $x \in \{0,1\}$ by two places $x=0$ and $x=1$, which represent the value associated to variable x. There is at least one transition $x+$ and one transition $x-$ between places $x=0$ and $x=1$, such that $x+ \in (x=0)\bullet$, $x+ \in \bullet(x=1)$, $x- \in (x=1)\bullet$, $x- \in \bullet(x=0)$. Transition $x+$ determines the change of variable state from 0 to 1, while $x-$ represents the change of the state from 1 to 0. Transitions $x+$ and $x-$ may also connected to read-arcs which enable the transitions only when a certain condition is held.

Consider the mapping of spread token model into elementary cycles of PN. In this model a combinational logic node $l \in L$ is associated with a single evaluation state variable $\Xi(l)$ and a pair of evaluation condition $\xi_+(l)$ and resetting condition $\xi_-(l)$ (see Section 5.3 for details). At the PN level this is modelled as an elementary cycle $\Xi(l)$ shown in Figure 5.8(a). The read-arc connected to $\Xi(l)+$ allows this transitions to fire only when enabling condition $\xi_+(l)=1$ is held. Similarly, transition $\Xi(l)-$ becomes enabled only if its enabling condition $\xi_-(l)=1$ is held. Note that for readability of the figure the variable name $\Xi(l)$ is only shown in the middle of the elementary cycle; places and transitions associated with this variable are labelled in a shorthand notation. In particular, places $\Xi(l)=0$ and $\Xi(l)=1$ are labelled $'0'$ and $'1'$ while transitions $\Xi(l)+$ and $\Xi(l)-$ are labelled $'+'$ and $'-'$ respectively.

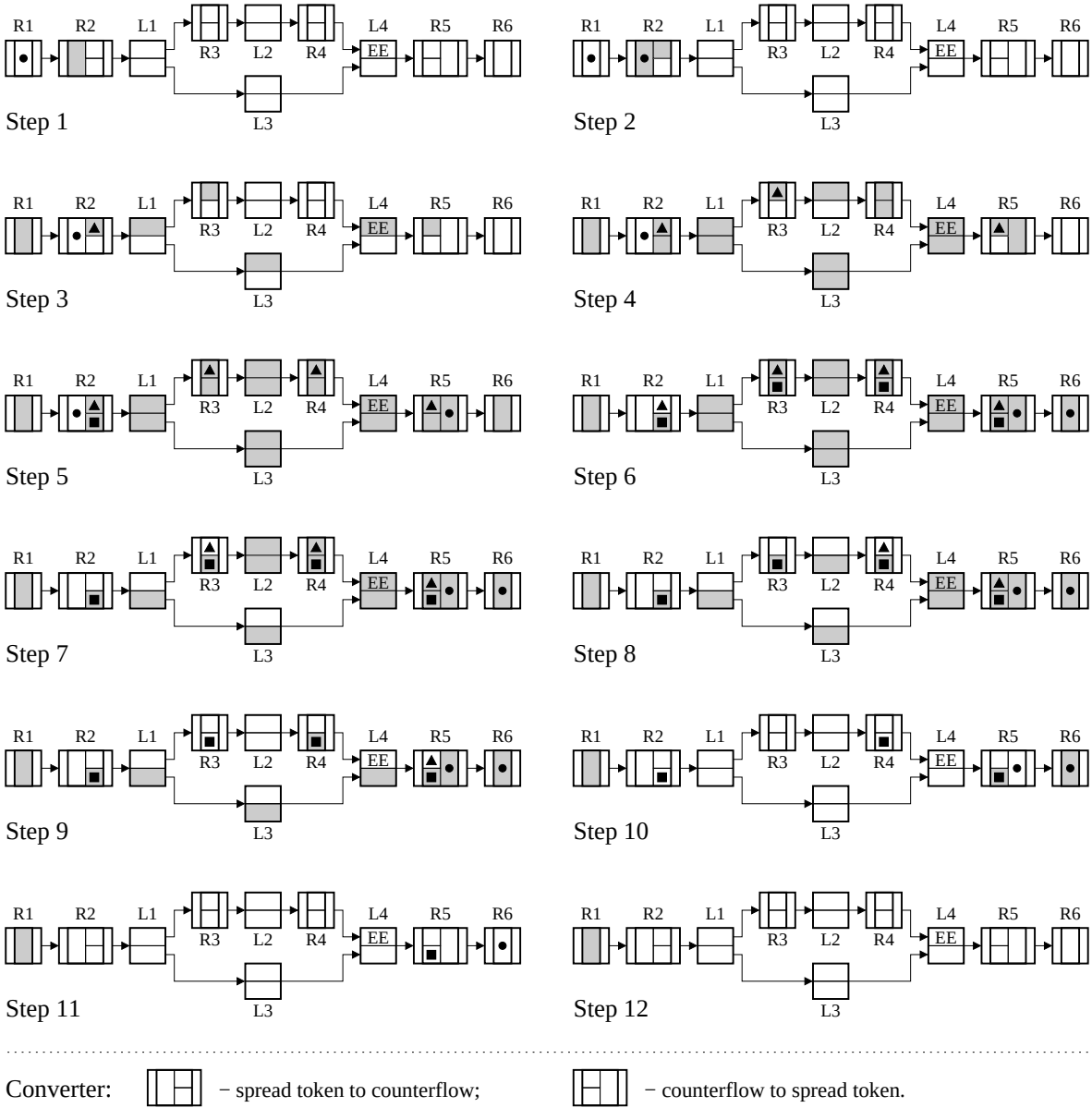
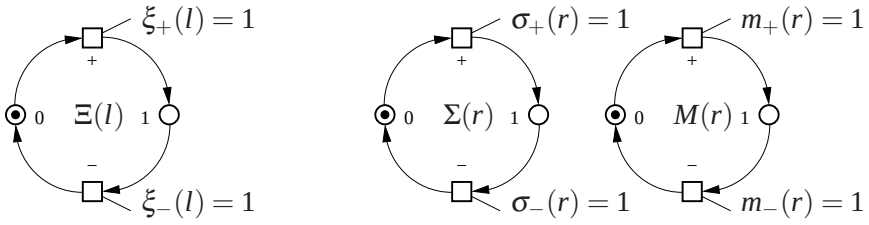


Figure 5.7: Combined spread token and counterflow SDFS example



(a) Logic node $l \in L$ (b) Register node $r \in R$

Figure 5.8: Underlying STG for spread token SDFS

Mapping of a spread token register into a PN is illustrated in Figure 5.8(b). There are two state variables associated with a register $r \in R$: enabling state $\Sigma(r)$ and marking $M(r)$. Therefore two elementary cycles are required to capture the register behaviour by a PN. Conditions $\sigma_+(r) = 1$ and $\sigma_-(r) = 1$ control transitions $\Sigma(r)+$ and $\Sigma(r)-$ respectively. The former denotes when the register is enabled and the later when it is disabled. Likewise, the change of register marking is defined by conditions $m_+(r) = 1$ and $m_-(r) = 1$, which enable transitions $M(r)+$ and $M(r)-$ respectively.

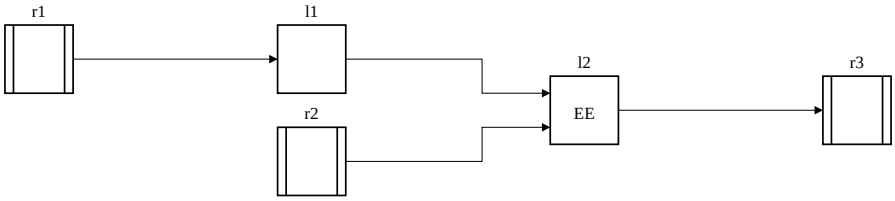
Usually the enabling conditions on the read-arcs are more complex than a single variable. Such conditions should be represented into a disjunctive normal form (DNF). Then each DNF clause is mapped into a separate transition of the elementary cycle and each variable of the clause is read by its own read-arc.

In order to illustrate how the enabling conditions are represented by means of read-arcs consider a simple spread token example shown in Figure 5.9(a). Note that the combinational logic node $l2$ is tagged with EE label, which means it can evaluate as soon as one of its inputs is ready. Let us concentrate on mapping of this node into an elementary cycle $\Xi(l2)$. The evaluation condition associated with this node is $\xi_+(l2) = \Xi(l1) \vee M(r2)$ while the resetting condition is $\xi_-(l2) = \overline{\Xi(l1)} \wedge \overline{M(r2)}$.

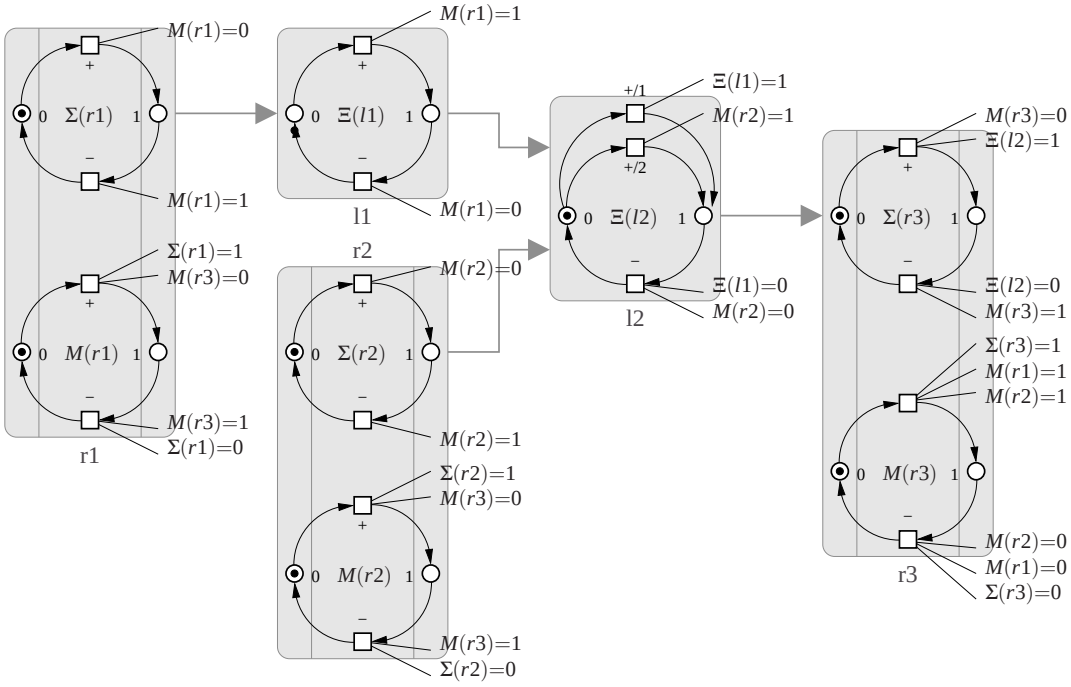
For the evaluation phase $\xi_+(l2) = 1$ implies $(\Xi(l1) = 1) \vee (M(r2) = 1)$. This expression has two DNF clauses, therefore transition $\Xi(l2)+$, which is controlled by the condition $\xi_-(l2)$, is split into a pair of transitions $\Xi(l2)+/1$ and $\Xi(l2)+/2$. Transition $\Xi(l2)+/1$ is enabled when place $\Xi(l1) = 1$ is marked and transition $\Xi(l2)+/2$ is enabled by a token in place $M(r2) = 1$, as shown in Figure 5.9(b). Firing either of these transitions changes the evaluation state of node $l2 \in L$, which models the early evaluation.

At the reset phase, $\xi_-(l1) = 1$ implies $(\Xi(l1) = 0) \wedge (M(r1) = 0)$. This expression has a single DNF clause and therefore both read-arcs, one from place $\Xi(l1) = 0$ and the other from place $M(r1) = 0$, are connected to the same transition $\Xi(l2)-$. This means that both places must be marked to allow the reset of node $l2 \in L$, i.e. no early reset is possible.

Elementary cycles for the rest of the nodes are built the same way. Note that the resultant STG is consistent by construction because the positive and negative transitions of each signal (or



(a) Static data flow structure



(b) Petri net

Figure 5.9: Mapping SDFS with spread token semantics into Petri net

variable) alternate in each the elementary cycle.

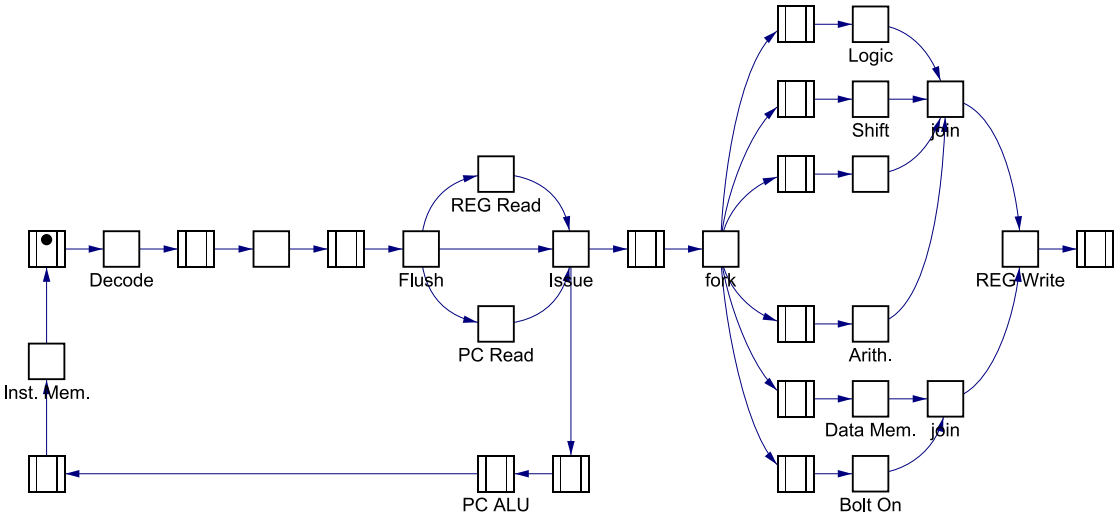
Consider the conversion of SDFS models into PNs on a more realistic benchmark, e.g. ARISC processor whose SDFS model is shown in Figure 5.10(a). This is a relatively small example which consists of 17 combinational logic nodes and 14 registers. However, its underlying PN is quite big even for a basic spread token semantics without early propagation, see Figure 5.10(b). The PN consists of 45 elementary cycles: 17 elementary cycles for combinational logic nodes and 28 elementary cycles to represent 14 registers. The names of places and transitions are hidden as they are not readable at this scale. It is still possible to see the correspondence of the elementary cycles to the original SDFS nodes - their relative layout is preserved.

Due to high concurrency this PN has more than 10^7 states and therefore cannot be verified by analysing the whole state space in reasonable time. For example, it took Petrify three hours before it ran out of memory. Instead, verification tools based on analysis of unfolding prefixes should be employed. The unfolding prefix for this PN has only 164 events and is built by Punf [64] in 18ms. Analysis of the resultant unfolding by MPSAT confirms that the model of the ARISC processor does not have deadlocks.

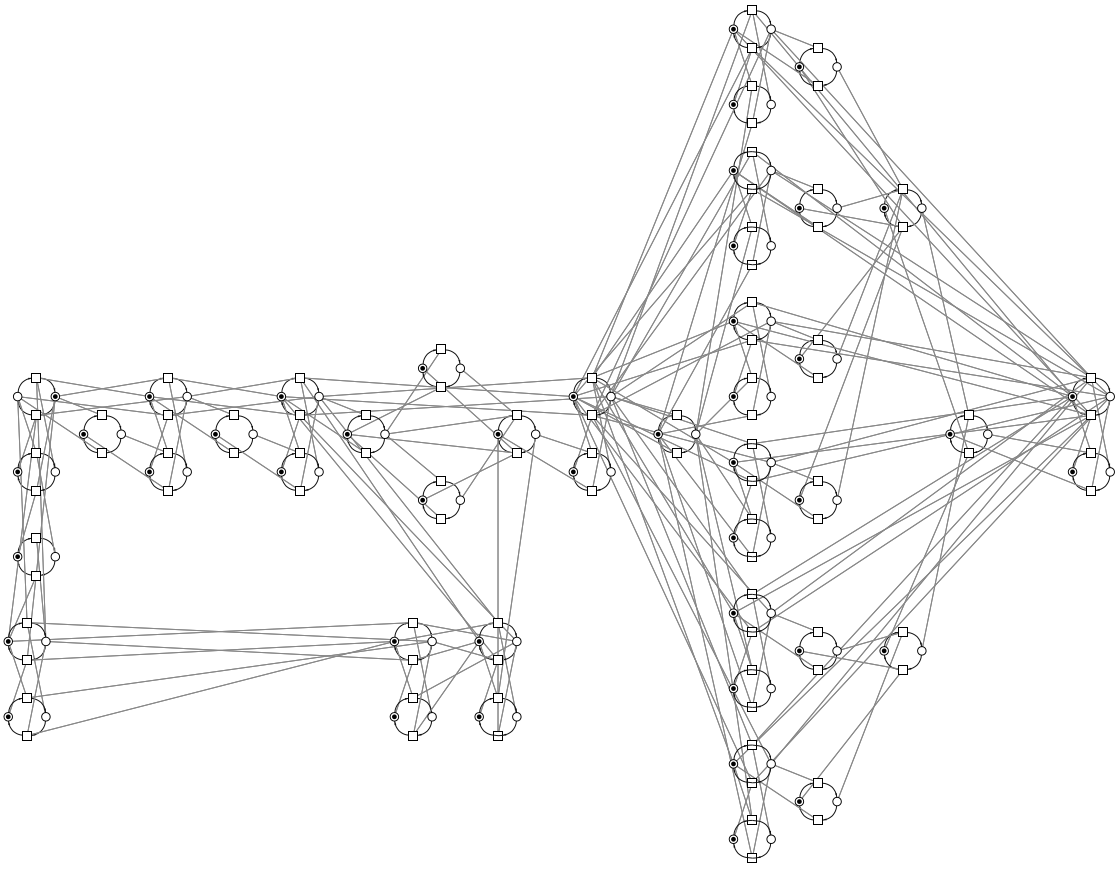
In this section a method for mapping of high-level spread token SDFS model into low-level PN has been presented. The same technique can be used to build underlying PNs for other SDFS models. The only difference is in the number of elementary cycles representing the state of SDFS nodes. For example, in counterflow model each combinational logic node $l \in L$ is associated with two state variables, the forward evaluation state $\Xi^F(l)$ and the backward evaluation state $\Xi^B(l)$ which are mapped into a pair of elementary cycles. A counterflow register $r \in R$ has four state variables: forward enabling $\Sigma^F(r)$, backward enabling $\Sigma^B(r)$, OR-marking $M^O(r)$ and AND-marking $M^A(r)$. Each of these variables is represented by its own elementary cycle. The transparent correspondence between SDFS and PNs allows to reuse model checking tools developed for PNs to verify SDFS specifications.

5.7 Comparison of SDFS token game semantics

All the token game semantics presented in this chapter have their advantages and drawbacks. In this section the models are informally compared in few aspects, which are summarised in Ta-



(a) SDFS model



(b) PN model

Figure 5.10: ARISC processor

Table 5.1: Comparison of SDFS token game semantics

Token game semantics	Model complexity	Model power	Early evaluation	Preemption mechanism	Conflict resolution	Control complexity
Atomic token	simple	limited	no	no	n/a	simple
Spread token	simple	good	partially	no	n/a	simple
Antitoken	complex	excellent	yes	yes	arbitration	complex
Counterflow	moderate	excellent	yes	yes	OR-causality	moderate
Hybrid	simple/moderate	excellent	yes	yes	OR-causality	simple/moderate

ble 5.1. In particular, the model complexity, model power, control complexity, support for early evaluation and preemption are compared.

The SDFS token game semantics can be classified as *basic* and *advanced* models. The former models only capture basic features of the asynchronous data path, while the later are able to capture more advanced concepts, such as preemption and speculation. Clearly, the atomic token and the spread token semantics belong to the class of basic models, while the antitoken and the counterflow are advanced models.

In the basic model category, both the atomic token and the spread token models have similar complexity. However, the atomic token semantics can only be applied to some class of well-formed SDFS, which limits its model power. The spread token semantics represents a much wider class of asynchronous data path circuits and has a rudimentary support for early evaluation (within one pipeline stage). Therefore, the spread token semantics a better choice for basic SDFS modelling.

In the category of advanced models the difference is mostly in the complexity of the semantics and the implementation of control logic. Both, antitoken and counterflow semantics capture early evaluation and preemption. However, the counterflow semantics has simpler token game rules. Also, the use of OR-causality (as opposed to arbitration in antitoken semantics) for the resolution of conflicts between tokens results in a simpler implementation for control logic. These advantages make counterflow semantics a better choice for modelling SDFS with early evaluation and preemption.

The hybrid token game semantics has the advantages of both, basic and advanced models. In this model the relatively complex counterflow semantics is only used in those parts of SDFS where preemption can be exploited to speed up the data path. In the rest of the SDFS simple

spread token semantics is employed. At the level of implementation this results in significant area decrease compared because the ordinary Muller pipeline stages are much smaller than counterflow pipeline controllers.

Verification of SDFS models is based on their conversion into schematic PNs, as has been described in Section 5.6. The verification tools which use the explicit state space representation of the underlying PN fail even on relatively small SDFS examples. The reason for this is a high level of concurrency in SDFS models, which leads to the state space explosion. The high level of concurrency does not cause a problem for unfolding-based verification tools because unfolding prefixes capture the concurrency in a very compact form, comparable to the size of original PNs. Choice becomes a problem for unfolding though, because each choice branch needs to be unfolded and stored explicitly. However, there is not much choice in the SDFS models. The only source of choice is early evaluation, which is usually limited to few nodes where concurrent branches synchronise. In our experiments, if no early evaluation was allowed, the unfolding time did not exceed few seconds even on relatively large SDFS examples containing few hundred nodes. If early evaluation was enabled, then benchmarks of up to a hundred counterflow SDFS nodes could be verified using unfolding-based tools. The benchmark results based on PUNF unfolder and MPSAT model checker [64] are presented in Table 5.2.

All the benchmarks in Table 5.2 have combinational logic nodes with early evaluation. In the *small* benchmark, which has 27 nodes only, the presence of early evaluation is not critical for the unfolder - it handles both spread token and counterflow semantics within a second. For the *average* benchmark, which has 70 nodes, the counterflow semantics becomes a problem - the unfolding prefix grows much larger than the PN and it takes nearly two minutes to build. The hybrid SDFS model becomes useful in this case. If the counterflow semantics is only applied to those 12 nodes which can exhibit preemption, then the unfolding size is much smaller and the computation time is just 4 seconds. The *large* benchmark, which consists of 524 nodes, is verified in 2 seconds under spread token semantics. However, if the counterflow semantics is applied, the computation time exceeds 38 minutes; if the hybrid semantics is used with 96 nodes exhibiting preemption, then the computation time is reduced to 8 minutes. Therefore, few hundred nodes is a practical limit for the size of SDFS models which can be verified by our method within acceptable

Table 5.2: Benchmark results

Benchmark	**Model semantics**	**SDFS size (basic / advanced)**	**PN size (place / transition)**	**Unfolding size (event / cutoff)**	**Computation time (sec)**
small	spread token	27 / 0	172 / 83	125 / 3	<1
	counterflow	0 / 27	484 / 193	524 / 18	<1
	hybrid	17 / 10	318 / 135	351 / 19	<1
average	spread token	70 / 0	452 / 205	2,063 / 92	1
	counterflow	0 / 70	1,080 / 463	20,933 / 858	117
	hybrid	58 / 12	602 / 261	7,668 / 443	4
large	spread token	524 / 0	3,352 / 1,520	6,570 / 192	2
	counterflow	0 / 524	9,324 / 3,448	144,574 / 6,444	2,319
	hybrid	428 / 96	4,632 / 1,976	83,476 / 7,484	492
ee2	spread token	58 / 0	436 / 188	297 / 9	<1
	counterflow	0 / 58	1,212 / 440	4,202 / 212	2
	hybrid	36 / 22	780 / 304	3,015 / 219	1
ee3	spread token	58 / 0	436 / 190	309 / 13	<1
	counterflow	0 / 58	1,212 / 442	11,516 / 742	15
	hybrid	25 / 33	952 / 364	9,652 / 793	8
ee4	spread token	58 / 0	436 / 192	321 / 17	<1
	counterflow	0 / 58	1,212 / 444	31,604 / 2,783	291
	hybrid	14 / 44	1,124 / 424	30,163 / 2,805	199
deadlock	hybrid	5 / 19	300 / 128	26,658 / 5,407	103

time.

Let us study the influence of early evaluation on the size of unfolding prefix and computation time using benchmarks *ee2*, *ee3* and *ee4*. These benchmarks are essentially the same SDFS, but with different number of early evaluating fork-join blocks - two, three and four early evaluating blocks, respectively. The early evaluating block in these benchmarks is such that any of its three inputs is sufficient to produce the output. For the spread token semantics the number of early evaluation blocks does not change the unfolding time or size much because there is no preemption in this model and the early evaluation is limited to a single pipeline stage. Contrary, for the counterflow semantics both the size of unfolding prefix and its computation time grow exponentially with the number of early evaluation blocks. This is due to the choices introduced by early evaluating combinational logic nodes.

In the last benchmark, called *deadlock*, the evaluating and resetting conditions of combinational logic nodes were modified to force a deadlock in the model. Verification of the model revealed a trace leading to a deadlock state.

5.8 SDFS with dynamic elements

Let us consider a following situation: data that comes into a section of the data path may need to be processed via two alternative computation paths, one significantly slower than the other. The decision which computation path to take is produced by the control path, which is external to the data path. In a fully static data flow model that has been presented in this chapter, both paths will have to start executing the computation simultaneously. Although the faster result can be output immediately by making the join element early propagative, in order to start the next wave of the computation the execution of the slower path still needs to be completed. If a more complex token game, such as counterflow, were used in the pipeline, then the execution of the slower path could be interrupted. However, in the modelled system only one path is enabled at a time, and the computation in the other branch should not start at all. Therefore, it is not always possible to model the expected behaviour using SDFS.

5.8.1 Dynamic elements

To resolve this limitation, it is necessary to introduce elements that would model the influence of the control path on the underlying data path. These elements are called *dynamic*, because they modify the otherwise static, or deterministic, execution flow of the model. To model the activity of the control signals, it is necessary to introduce a new class of tokens, called *control tokens*, that would represent the propagation of the control signals in a way similar to the propagation of data. As opposed to the data tokens that represent abstract data items in the SDFS model, the control tokens need to be associated with the actual data values. In the scope of this work only two possible values are used, depicted as ⓪-token and ①-token.

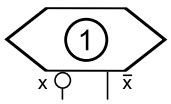


Figure 5.11: Graphical representation of a control node

5.8.2 Control

The control node acts similarly to the spread token SDFS register, with the exception that it propagates control tokens preserving their values. Note that the control is allowed to be connected only to the push/pop nodes or to another control node.

A control node is initially in a disabled state. It can be *enabled* iff all nodes in its preset are marked with a token. An enabled node can be *marked* with a ⓪-token if it is enabled, not yet marked and all nodes in its preset are marked with a ⓪-token, and, similarly, it can be marked with a ①-token iff all nodes in its preset are marked with a ①-token, thus achieving the propagation of the tokens while preserving their values. A marked control node can become *disabled* iff any of its preset nodes become unmarked, and the token can be removed from a disabled node iff none of its preset nodes hold a token.

If a control node has an empty preset, it is called an *external control* node. An external control node is always enabled and can be marked either with a ⓪-token or a ①-token in a free choice. A control node is not allowed to have an empty postset.

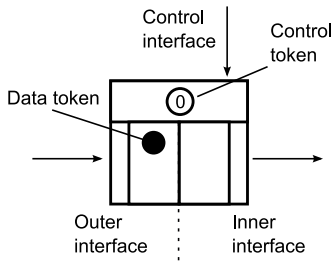


Figure 5.12: Graphical representation of the push and pop nodes

5.8.3 Push

Push is an element that, depending on the choice made by the control, either forwards the data token or destroys (acknowledges) it. Paired with pop, it can be used to select one of several possible paths of the data flow. The Push element is comprised of the three blocks: the *outer interface (OI)*, the *inner interface (II)*, and the *control interface (CI)* (Figure *5.12*). The outer and inner interfaces act as a pair of regular SDFS registers for the other SDFS nodes, i.e. they can be enabled, disabled, marked and unmarked; however the marking visible to its postset and preset nodes is different. If an SDFS node is in the push node's preset, it reads the marking of the *outer interface.* If an SDFS node is in the push node's postset, it reads the marking of the *inner interface.*

The transfer of tokens between the outer and inner interfaces is governed by a special set of rules, which are as follows. Note that *preset, postset, r-preset, r-postset* are defined for the Push and pop elements in the same way they are defined for regular SDFS elements [107].

The OI, II and CI are initially disabled and unmarked. The OI can become enabled iff all registers in the push's preset are marked and all logic nodes in the push's preset are evaluated. The OI can become marked with a token iff it is enabled, the II and CI are unmarked, the r-preset of the push is marked. OI can become disabled iff any of the registers in the push's preset becomes unmarked or any of the logic nodes in the push's preset becomes reset. The disabled OI can be unmarked iff the r-preset of the push is unmarked and the II is marked.

The II can become enabled iff the OI holds a token and the CI holds a ①-token. The enabled II can be marked iff the r-postset of the push is unmarked.

The CI behaves according to the similar set of rules as a control node, with the exception that it can only accept a token when the OI is marked, and can be unmarked when the OI is unmarked.

To summarise, the push element synchronises a data token on the outer interface with a control

token. If the control token is a ①-token, it forwards the data token by transferring it into its inner interface, and then allows the token to be removed from the outer interface. If the data token is a ⓪-token, it allows the token to be removed from the outer interface without transferring it into the inner interface.

5.8.4 Pop

Pop is an element that, depending on the choice made by the control, either forwards the data token or produces a dummy token. Paired with push, it can be used to select one of several possible execution paths. Its structure is similar to the push, but the marking rules are different and are as follows.

The OI, II and CI are initially disabled and unmarked. The OI can become enabled iff all registers in the pop's preset are marked and all logic nodes in the pop's preset are evaluated. The OI can become marked with a token iff it is enabled, the II is unmarked, the CI is holds a ①-token and the r-preset of the pop is marked. OI can become disabled iff any of the registers in the pop's preset becomes unmarked or any of the logic nodes in the pop's preset becomes reset. The disabled OI can be unmarked iff the r-preset of the pop is unmarked and the II is marked.

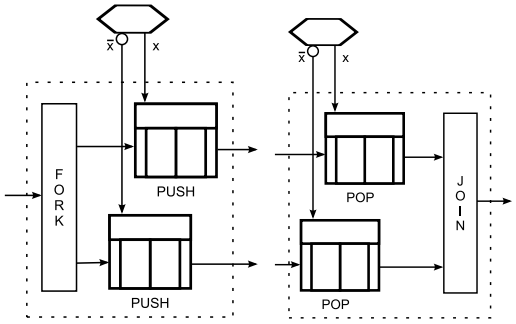
The II can become enabled if the OI holds a token and the CI holds a ①-token, or if the OI does not hold a token and CI holds a ⓪-token. The enabled II can be marked iff the r-postset of the pop is unmarked.

The CI behaves according to the similar set of rules as a control node, with the exception that in can only be marked when the II is unmarked, and can be unmarked when the II is marked.

To summarise, the pop element first receives a control token. If it is a ①-token, it then synchronises it with a data token on the outer interface and transfers it into the inner interface. If the token is a ⓪-token, it immediately produces a dummy data token on the inner interface.

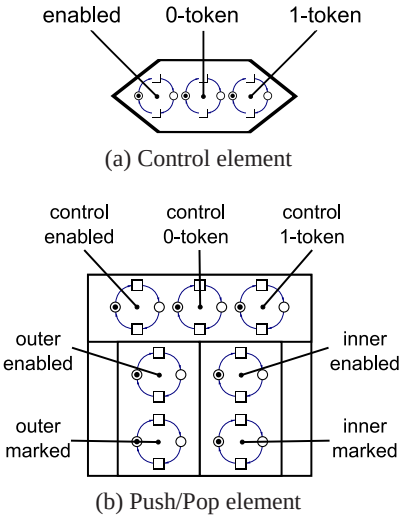
5.8.5 Mux and Demux

The multiplexer and demultiplexer are good examples of how the basic dynamic elements can be used. In Figure 5.13 (a), the demux is an element that, depending on the choice made by the control, forwards a data token from its input to one of its outputs. This is implemented using



(a) Demux (b) Mux

Figure 5.13: Implementation of the multiplexer and demultiplexer using dynamic components



(a) Control element

(b) Push/Pop element

Figure 5.14: Petri net mapping of the dynamic elements

push elements. Depending on the value of the control token, one of the push elements receives a ①-token and forwards the input token received via the fork element, and the other one receives a ⓪-token and blocks the token from entering its corresponding data path.

The mux (Figure 5.13 (b)) is an element that, depending on the choice made by the control, forwards a data token from one of its inputs to its output. Mux is implemented using two Pop elements. Depending on the value of the control token, one of the Pop elements receives a ①-token and forwards the input token to the join element, while the other one receives a ⓪-token and generates a dummy token that is also sent to the join element, where it is OR-ed with the actual data token resulting in the propagation of the data from the selected channel.

5.8.6 Mapping of the dynamic SDFS elements into Petri net fragments

To apply the verification method given in Section 4.4 to the dynamic elements, it is necessary to define the set of signals that are to be mapped into elementary cycles.

For the control node (Figure 5.14 (a)), 3 signals are necessary: the enabling state of the control node, and the presence of the control token. Because the control token can carry data, it has to be represented with more than one signal to encode the "value" of the token. The control tokens are allowed to only have 2 different values: 0 and 1, and thus two signals are enough to encode the value. The token presence signal can also be encoded using the same signals, similar to dual-rail encoding: the 00 value means "no token", 10 means "0-token present", 01 means "1-token present" and the value of 11 is not allowed. To build the firing rules that need to test only for the presence of a token (and do not care about its value) in the form of Boolean equations an OR construct is used. This approach also allows to extend the data domain if need arises simply by adding additional cycles.

For the push and pop nodes(Figure 5.14 (b)), the number of required signals is higher because they act as a 3-way node: they accept control tokens, data tokens and can generate (dummy) data tokens themselves. The signals for the outer and inner interfaces are the same as for the usual SDFS register: enabling and marking, and the signals for the control interface are the same as for the Control node: enabling and 2 signals for the control token value.

Once the Petri net cycles are constructed for each of the signals, they are ready to be interconnected using read arcs to impose the firing rules.

5.9 Conclusions

In this chapter a new token-based model (called a Static Data Flow Structure) that captures the behaviour of an asynchronous data path has been defined. The basic idea of an SDFS described in [110] has been formalised and extended using three different sets of token game rules: atomic token, spread token and counterflow. The rules controlling the behaviour of various elements in an SDFS pipeline (e.g., the marking and disabling of registers, propagation of tokens) have been formally defined and explained for each token semantic. The advantages and disadvantages of

each set of token game rules have been analysed. Additionally, a hybrid SDFS model, which allows combining the advantages of spread token and counterflow semantics, has been presented. An extension of SDFS model with dynamic elements that further extends the modelling power of SDFS had been defined.

The strict formalisation of the token game rules was used to implement an automated verification technique. SDFS models can be automatically translated into low-level Petri nets for subsequent verification and model checking by existing tools. The low level traces produces by those tools can be automatically re-interpreted in terms of the higher level SDFS model.

All of the SDFS models presented in this chapter have been implemented as plug-ins to Workcraft (Chapter 7) which were used to analyse the advantages and drawbacks of different SDFS token game rules on a set of benchmarks.

This chapter is based on a number of previously published papers [108, 109, 94].

Chapter 6

Interpreted Graph Models

Petri nets [90, 84] have historically been known as a formalism that is especially suitable for the modelling of concurrent and distributed systems. The value of Petri nets originates mainly from the fact that their clean and intuitive graphical notation is backed by a strict mathematical model of their behaviour. The graphical notation is very helpful during the manual system design and investigation. At the same time there exists a rich and mature formal theory of Petri nets as well as the numerous automated tools that are able to efficiently verify various behavioural properties of a given net. In particular, model checking [40] is an automated technique designed to either prove that a certain property (e.g., deadlock freeness, reachability of a certain marking, etc.) holds for a given net or to produce a trace demonstrating how the property is violated. This information is very useful for troubleshooting, and often allows to detect and fix errors early in the system design process.

On the other hand, Petri nets are often seen as a low-level formalism, much like an assembly language. The size of a Petri net required to describe the behaviour of a useful system can become so large that the designer is no longer able to comfortably perceive and manage it. To work around this problem, specialised higher level formalisms are often employed instead of Petri nets to de-

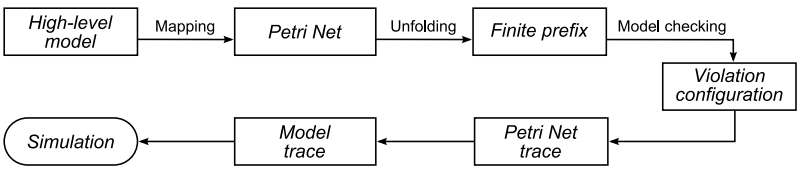


Figure 6.1: High level model verification workflow based on Petri nets

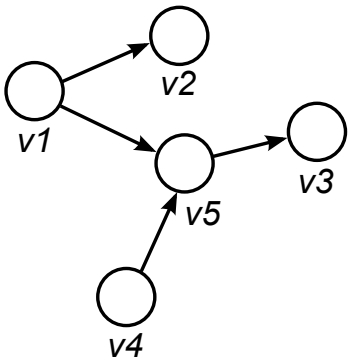


Figure 6.2: A directed graph

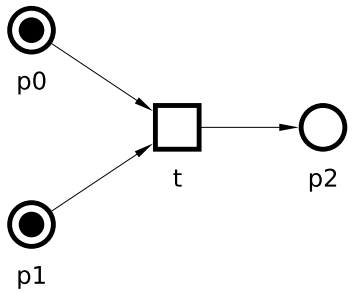
scribe concurrent systems. Such formalisms include, e.g., coloured Petri nets [60], Conditional Partial Order Graphs (CPOG) [80], networks of handshake components [47, 119], the Static Data Flow Structures (SDFS) model presented in Chapter 5 and many others. To be of practical use, the high level formalisms must be supported by an adequate set of analysis and verification methods. Development of specialised theory and tools for every formal model is often impractical — it may be more efficient to express a formalism in terms of another one (e.g., a Petri net) for which mature theory and tools have already been developed. Then, the result of the analysis (such as a violation trace) can be re-interpreted in terms of the original high-level model and presented to the designer (Figure 6.1). Naturally, Petri nets are a good choice for the target model, as their compositions are well understood, and efficient model checking tools for Petri nets are readily available.

A common feature of the high-level models mentioned above (and also of Petri nets) is the presence of an underlying static graph structure. Their semantics are defined using additional entities, such as tokens or node/arc states, which together form the overall state of the system. We jointly refer to such formalisms as *Interpreted Graph Models (IGM)*. The similarities in notation and expressive power allow a number of basic operations on these formalisms, such as visualisation and translation from one formalism into another, to be generalised. More complex operations on the models can also be used, such as interfacing one model type with another. This enables the designer to model subsystems using the most appropriate formalism, while still maintaining the ability to simulate and analyse the overall system.

$$\begin{cases} m_0(p_0) = 1 \\ m_0(p_1) = 1 \\ m_0(p_2) = 0 \end{cases}$$

$$\begin{cases} m'(p_0) = m(p_0) - 1 \\ m'(p_1) = m(p_1) - 1 \\ m'(p_2) = m(p_2) + 1 \end{cases}, m[t\rangle m'$$

(a) Mathematical interpretation (firing equation for transition t)



(b) Graphical interpretation

Figure 6.3: Different interpretations of a Petri net

6.1 Basic definitions

Definition 6.1. A *graph* is a pair $G = \langle V, E \rangle$ where V is a set of vertices and E is a set of two-element subsets of V that define edges representing connections between vertices. Often it is practical to consider the elements of E as ordered pairs ($E \subseteq V \times V$), then an edge $(a,b) \in E$ is said to be directed from a to b (and usually called an arc). A graph with directed edges is called a directed graph or a digraph (Figure 6.2).

An *Interpreted Graph Model (IGM)* is a pair $M = \langle G, I \rangle$ where G is a graph representing the *static structure* of the model and I is the *interpretation* of the elements of the graph. Note that I is not strictly defined and depends on the specific interpretation.

For instance, given a Petri net $N = \langle P, T, F, m_0 \rangle$, let $G = \langle P \cup T, F \rangle$ and $I = \langle P, T, m_0, \mathscr{M} \rangle$, where $\mathscr{M}$ represents the firing equations, then the Petri net N is also an IGM $M = \langle G, I \rangle$ with the Petri net token game interpretation (Figure 6.3a). Alternatively, let $G = \langle P \cup T, F \rangle$ and $I = \langle P, T, m_0, \mathscr{G} \rangle$ where $\mathscr{G}$ is the set of rules describing the Petri net graphical notation. Then $M = \langle G, I \rangle$ is an IGM with a graphical Petri net interpretation (Figure 6.3b). Similarly, a gate-level circuit model can be interpreted as a set of logic gates, a set of signals, simply a graph, or as information that can be used to produce an image.

The main idea behind the concept of an Interpreted Graph Model is to split the *structure* of the model from its *interpretation*. This allows to apply different interpretations to the same backing data structure, similar to how Petri nets may both be interpreted using the graphical notation or using the mathematical definitions such as the firing equations (Figure 6.3).

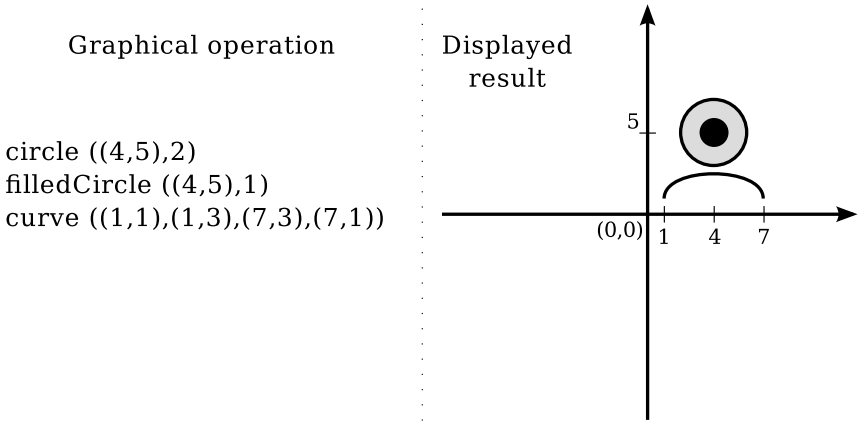


Figure 6.4: An example of a graphical operation

6.2 Graphical representation of Interpreted Graph Models

Models that have an underlying graph structure are generally rendered as a set of shapes that represent vertices and a set of lines or arrows that represent edges. For example, in Figure 6.2 a simple directed graph is drawn as a set of circles depicting vertices and a set of straight lines depicting edges. The lines end with arrows that represent the direction of the edges.

For the more complex models, the shapes of both vertices and edges may also depend on some additional state information. For example, a place of a Petri net is usually drawn as a circle with a number of smaller filled circles inside corresponding to the number of tokens in the place. Otherwise, however, Petri nets, as well as most other graph-based models, graphically look quite similar to the basic directed graph. In this section we will attempt to capture the similarities in the graphical presentation to construct a general-purpose graphical presentation algorithm.

The relative location of the graphical objects corresponding to the objects in the graph is determined either manually or using an automated layout tool (an example of the automated layout is shown in Figure C.8). When rendered on a computer screen, additional transform operations can be applied: the graphical objects may be translated, scaled or rotated to give an appropriate view to the user.

Let $\mathscr{G}$ be the set of graphical operations. The individual graphical operations can be seen as, for instance, sets of vector graphics commands that can be executed to produce an image (Figure 6.4). Let $\mathscr{D}(g)$ be a display operation[1] that executes a graphical operation $g \in \mathscr{G}$ and

[1] $\mathscr{D}$ is assumed to be an external operation, implemented by, e.g., a graphical toolkit. For example, it may be a

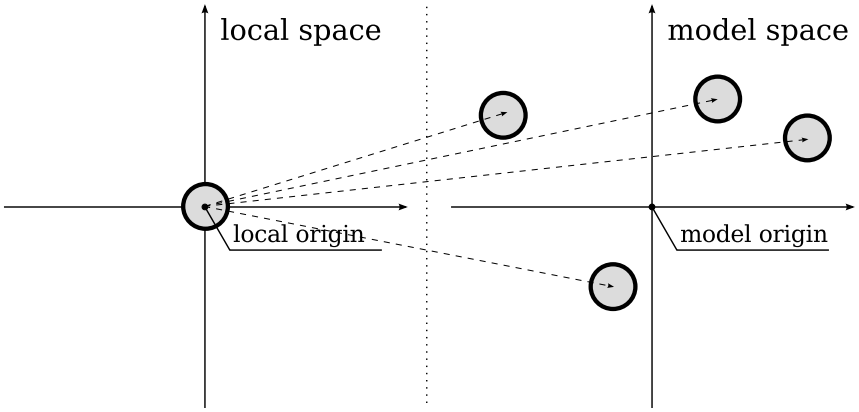


Figure 6.5: Combining a local space drawing function with a transformation

presents the result on the display. Then to display a graphical representation of a given graph $G = \langle V, E \rangle$ it is necessary to define such function $\gamma(G)$ that gives a graphical operation that can be used produce an image of G and evaluate $\mathscr{D}(\gamma(G))$.

Let $\varepsilon \in \mathscr{G}$ be an empty graphical operation that produces no image. Let $g_1 \circ g_2$ be a composition operation over $\mathscr{G}$ that produces a graphical operation that is the sequential execution of the operations g_1 and g_2. Let D be a function $D : V \cup E \to \mathscr{G}$ that associates a graphical operation with every object in the graph. Then

$$\gamma(G) = \begin{cases} \varepsilon, & \text{if } V \cup E \text{ is empty} \\ \bigcirc_{n \in V \cup E} D(n), & \text{otherwise} \end{cases} \qquad G = \langle V, E \rangle \tag{6.1}$$

It is usually natural that all objects of the same type are drawn using the same graphical operation. For instance, every vertex in a graph is drawn using a circle of the same size and in the same colours. However, if function D that associates the same graphical operation with every object were used in equation 6.1, then $\mathscr{D}(\gamma)$ would display all objects drawn on top of each other, which is obviously not the desired outcome. It is therefore practical to assume that there exists a function D_{local} that produces a set of graphical operations relative to a local coordinate space. The function D can then be derived from D_{local} and an *affine transformation* associated with every object. This way using the correct transformations the objects of the same type can be drawn using the same graphical operation but will assume the desired arrangement in the final image (Figure 6.5).

function that rasterises a sequence of vector graphics commands and paints the result in a window.

Let Δ be the set of all allowable affine transformations. Let $\mathscr{T}$ be a transformation function[2] $\mathscr{T} : \mathscr{G} \times \Delta \to \mathscr{G}$ that given a graphical operation $g \in \mathscr{G}$ and an affine transformation $\delta \in \Delta$ produces a graphical operation g' such that $\mathscr{D}(g')$ displays an image that is $\mathscr{D}(g)$ transformed by δ. Let X be a function $X : V \cup E \to \Delta$ that associates each object in the graph with an affine transformation. Equation 6.1 can then be rewritten as

$$\gamma(G) = \begin{cases} \varepsilon, & \text{if } V \cup E \text{ is empty} \\ \bigcirc\limits_{n \in V \cup E} \mathscr{T}(D_{local}(n), X(n)), & \text{otherwise} \end{cases} \qquad G = \langle V, E \rangle \tag{6.2}$$

which given an appropriate X will produce the correct graphical operation that can be used to generate an image of the graph G using $\mathscr{D}(\gamma(G))$.

So far in this section a graph $G = \langle V, E \rangle$ has been extended with an interpretation $I = \langle D_{local}, X \rangle$ which a graphical representation for this graph to be formally defined. Combining G and I into a single object we get an Interpreted Graph Model $M = \langle G, I \rangle$. Then

$$\gamma_{IGM}((G, I)) = \gamma(G) \tag{6.3}$$

$$\text{where } D_{local}, X \text{ are in } I$$

which gives a general-purpose graphical operation function for any IGM given an interpretation that defines D_{local} and X.

A pair $I = \langle D_{local}, X \rangle$ associated with a graph $G = \langle V, E \rangle$ where D_{local} is the drawing function $D_{local} : V \cup E \to \mathscr{G}$ and X is the transformation function $X : V \cup E \to \Delta$ is called a *graphical interpretation* of the graph G.

[2] Similar to $\mathscr{D}$, the implementation of the function $\mathscr{T}$ is provided by the graphical toolkit.

6.2.1 Building a graphical representation of a Petri net

Let N be a Petri net $N = \langle P, T, F, m_0 \rangle$ and D^{PN}_{local} be a function

$$D^{PN}_{local}(n) = \begin{cases} D^{T}_{local}, & \text{if } n \in T \\ D^{P}_{local}(m_0(n)), & \text{if } n \in P \\ D^{A}_{local}(n), & \text{if } n \in F \end{cases}$$

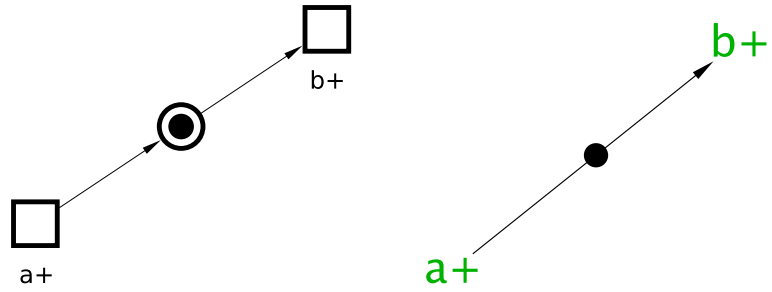
that defines a graphical operation for an object $n \in P \cup T \cup F$, where D^{T}_{local} is a graphical operation that draws a Petri net transition, D^{P}_{local} is a graphical operation that draws a Petri net place with the corresponding amount of tokens, D^{A}_{local} is a graphical operation that draws an arc. Let X_{PN} be a function $X_{PN} : P \cup T \cup F \to \Delta$ that associates each object in the Petri net with an affine transformation. Let $Map(N) = M$ where $M = \langle G, I \rangle, G = \langle P \cup T, F \rangle, I = \left\langle D^{PN}_{local}, X_{PN} \right\rangle$ be the function giving the IGM form of a Petri net N, such that I is a graphical interpretation. Then the equation 6.3 can be used to calculate the graphical representation γ_{PN} of the Petri net N:

$$\gamma_{PN}(N) = \gamma_{IGM}(Map(N))$$

To use this equation one needs to associate an affine transformation not only with places and transitions but also with arcs. However a more practical way is to derive the shape of a particular arc from the transformations of those objects that it connects. Then the arc will "follow" those objects even if their transformations change without the need to change the transformation associated with the arc. Let $(n_1, n_2) \in F$ be an arc connecting two objects n_1 and n_2. Let $\mathscr{A}(\delta_1, \delta_2)$ be a graphical operation[3] that draws an arc in such a way that it connects the objects having transformations δ_1 and δ_2. Let $X_{PN}(n) = \delta_0, n \in F$ where δ_0 is the identity transformation. Then $D^{A}_{local}((n_1, n_2)) = \mathscr{A}(X_{PN}(n_1), X_{PN}(n_2)), (n_1, n_2) \in F$ is the graphical representation function for arcs that does not require to explicitly define arc transformations.

To summarise, by converting a Petri net into an IGM form with a graphical interpretation, a generalised algorithm can be applied to produce its graphical representation. Any other IGM can

[3] This graphical operation may be implemented, for example, by assuming the origins of the local coordinate spaces of the two objects to be their centres, then using their corresponding transformations to calculate the positions of their centres in the global model coordinate space and then draw an arc connecting the centres.



(a) A place with one outgoing and one incoming arc

(b) The shorthand graphical notation used in the STG models

Figure 6.6: Graphical notation violating the one-to-one correspondence

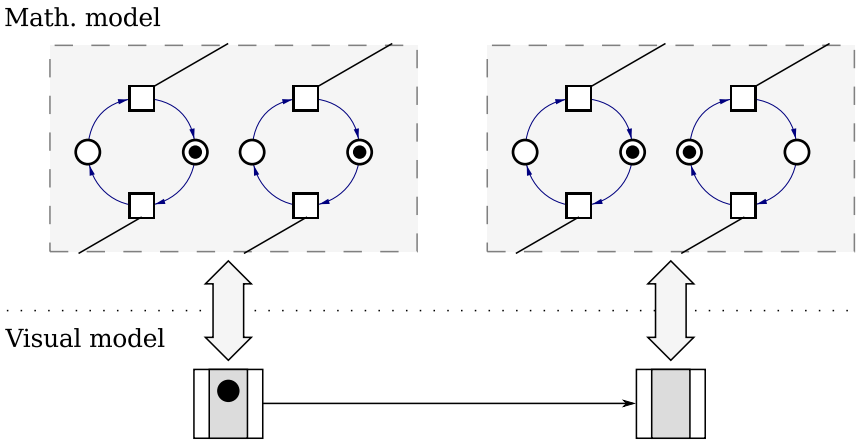


Figure 6.7: A Petri net model visualised using the SDFS graphical notation

similarly be extended with such an interpretation allowing it to be presented graphically. This feature is used in the Workcraft framework (Chapter 7) to provide a general-purpose graphical rendering implementation for the client models.

6.2.2 Using a separate visual model

Sometimes it may be practical to avoid the strict one-to-one mapping between the objects in a model and their graphical representations. For example, in Figure 6.6, a shorthand graphical notation used to represent the objects in an STG model is shown. Figure 6.6a shows a fragment of a Petri net containing a place with exactly one incoming and one outgoing arc. In Figure 6.6b the same fragment of the Petri net is shown using the shorthand STG notation (the place and its incident arcs are replaced with a single arc said to contain an *implicit place*). This notation is useful because such configuration of places is encountered very often in the STG models and

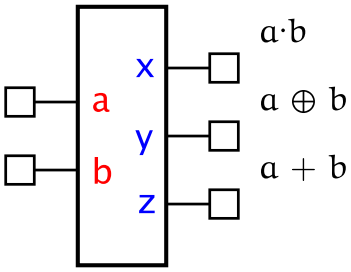


Figure 6.8: An example of the hierarchical arrangement of graph nodes

using the implicit place concept allows to reduce the visual complexity of the model's graphical representation.

Let M be an IGM $M = \langle G, I \rangle$. Let $I = \langle M_{visual} \rangle$, where M_{visual} is an IGM $M_{visual} = \langle G, I \rangle, I = \langle D_{local}, X \rangle$. Let $\gamma(M) = \gamma_{IGM}(M_{visual})$ be the graphical operation used to produce the image of the model M. Then it is said that M_{visual} is the *visual model* associated with the *mathematical model* M and I is called the *visual model interpretation* of the model M.

Using a visual model interpretation allows to define an arbitrary number of graphical representations of the same model. For example, the same Petri net can then be presented using the canonical graphical notation (Figure 6.6a) or using the STG notation (Figure 6.6b) depending on the context. A more complex example is shown in Figure 6.7. In this example, large fragments of a Petri net are mapped into the high level graphical objects (spread token SDFS registers).

6.2.3 Using a hierarchical structure

There are a number of models that are best represented in a hierarchical fashion: some objects are treated as *parents* of other objects. The transformation function is defined in such a way that the transformation of the parent object also affects its child objects. For example, in the gate-level circuit model a gate object acts as a parent for the set of its pins (Figure 6.8), which means that the transformations of the contacts are relative to the transformation of the gate object. Graphically, this results in contacts "following" the gate object when its transformation is changed (e.g., when the user is moving the gate object in the editor). The arrangement of the contacts relative to the parent gate can still be changed without affecting any other objects.

Let M be an IGM $M = \langle G, I \rangle, G = \langle V, E \rangle$. Let $I = \langle D_{local}, X, H \rangle$ where $\langle D_{local}, X \rangle$ is the graphical interpretation of G and P is the *hierarchy* function $H : V \cup E \to V \cup E$ that associates

each object of the graph G with a parent object. Let

$$X_H(n) = \begin{cases} X(n) & \text{if } H(n) = \emptyset \\ X(n) \bullet X_H(H(n)) & \text{otherwise} \end{cases}$$

be the hierarchical transformation function where $\bullet$ is the transformation concatenation operation $\bullet : \Delta \times \Delta \to \Delta$. Then

$$\gamma(M) = \gamma_{IGM}(\langle G, \langle D_{local}, X_H \rangle \rangle)$$

is the graphical representation of the hierarchical model M.

6.2.4 Redefining the display operation

So far in this section the display function $\mathscr{D}(g)$ was defined as a function that given a graphical operation g displays an image generated by the operation on the screen. By redefining this function, additional functionality can be obtained using the same graphical operation g. For example, instead of drawing the images on screen the graphical operations can be converted to a serial format such as EPS or SVG and stored to disk. This feature is exploited in the Workcraft framework (Chapter 7) to provide a graphics export function to any model that defines its graphical interpretation.

6.3 Logic networks

The ideas behind the verification methods for the gate-level circuits (Section 4.4) and the Static Data Flow Structures (Section 5.6) are quite similar. In both cases, the state of the analysed system is encoded using a set of binary signals. Their switching behaviour is captured using the *set* and *reset* functions associated with each signal. Those functions control when the signal may change its state based on the values of a set of other signals. Then a Signal Transition Graph is constructed in such a way that the state of each signal is encoded using a pair of complementary places, and transitions that transfer the token between those places are arranged in such a way that the token may only be transferred only when the *set* or *reset* conditions are met. The enabling

of these transitions is controlled using read arcs[4] that allow to read the state of other signals non-destructively. Verification is then carried out on this STG, which is called the *verification STG*.

For the gate-level circuit model the construction of the verification STG is straightforward. Each gate is associated with exactly one output signal, and the *set* and *reset* functions of that signal depend only on the set of gates directly connected to the inputs. For the SDFS model the construction of the STG is more complex, because the *set* and *reset* functions for a given node may depend on the state of nodes that are not connected to it directly (such as, e.g., the R-preset or a register). Additionally, the state of the SDFS nodes has to be described using more that one signal per node.

In the previous chapters, the algorithms that produce the verification STGs from the gate-level circuits or SDFS models were defined informally. In this section a formal framework for the STG-backed verification of high level models is proposed. Using the concept of an Interpreted Graph Model, the verification method can be applied to any model that defines a *logic network interpretation* for its graph structure.

Definition 6.2. Let S be a set of signals. Let $\mathscr{F}$ be a function that associates a signal $s \in S$ with a tuple $\langle I_s, f_s^{set}, f_s^{reset}, v_s^0 \rangle$, where $I_s \subset S$ is the set of input signals, f_s^{set} is the *set function* $f_s^{set} : \{0,1\}^{I_s} \to \{0,1\}$ of signal $s \in S$, f_s^{reset} is the *reset function* $f_s^{reset} : \{0,1\}^{I_s} \to \{0,1\}$ of signal $s \in S$, and $v_s^0 \in \{0,1\}$ is the initial value of the signal $s \in S$. The *value* v_s of a signal $s \in S$ may change from 0 to 1 at any time when f_s^{set} evaluates to 1, and from 1 to 0 at any time when f_s^{reset} evaluates to 1. Let I be a set of input signals $I \subset S$. Let O be a set of output signals $O \subset S$. Then a *logic network (LN)* is a tuple $L = \langle I, O, \mathscr{F} \rangle$.

Let L be a logic network $L = \langle In, Out, \mathscr{F} \rangle$. Then the function $\Gamma(L) = \langle P, T, F, m_0, \lambda, I, O, v_0 \rangle$ that constructs a verification STG from L is defined as follows.

Let $I = In$ and $O = Out$ respectively be the sets of input and output signals. Let $S = In \cup Out$. Let $v_0 = \{ v_s^0 \mid s \in S \}$ be the vector of initial signal values. Let $P = S \times \{0,1\}$ be the set of places of the required STG such that $(s,0) \in P$ represents the low value of the signal $s \in S$

[4]Unfortunately, the available verification tools do not recognise read arcs as a special class of arcs. Read arcs have to be emulated using two opposite arcs forming a loop. Although this technique is acceptable for verification, it is not as efficient as a true read arc would have been.

and $(s,1) \in P$ represents the high value. Let $DNF(f) = C$ be a disjunctive normal form of a Boolean function f, where C is a family of sets over P. The DNF function is constructed in such a way that its clauses are sets over P and $(s,1) \in P$ corresponds to the positive literal referring to signal s while $(s,0) \in P$ corresponds to the negative literal. Let $T_{set}(s) = s \times DNF(f_s^{set})$ be the set of rising transitions for the signal $s \in S$ labelled with DNF literals such that for every clause in the DNF there is a single transition. Similarly, let $T_{reset}(s) = s \times DNF(f_s^{reset})$ be the set of falling transitions for signal $s \in S$. Let $F_r = \{p,(z,c) \mid (z,c) \in T_{set}(s) \cup T_{reset}(s), p \in c, s \in S\}$ be the set of read arcs such that each transition t is connected with a set of places contained in the DNF clause that the transition is labelled with. Let $F_{pt+} = \{(s,0),t \mid t \in T_{set}(s), s \in S\}$ be the set of arcs connecting the places representing the low signal values to the rising transitions. Let $F_{pt-} = \{(s,1),t \mid t \in T_{reset}(s), s \in S\}$ be the set of arcs connecting the places representing the high signal values to the falling transitions. Let $F_{tp+} = \{t,(s,1) \mid t \in T_{set}(s), s \in S\}$ be the set of arcs connecting the rising transitions to the corresponding places representing the high signal values. Let $F_{tp-} = \{t,(s,0) \mid t \in T_{reset}(s), s \in S\}$ be the set of arcs connecting the rising transitions to the corresponding places representing the low signal values. Let $F_{loop} = F_r \cup \{t,p \mid (p,t) \in F_r\}$ be the set of arc loops emulating the read arcs. Let $T = \bigcup_{s \in S} T_{set}(s) \cup T_{reset}(s)$ be the set of transitions of the required STG. Let $F = F_{tp+} \cup F_{tp-} \cup F_{pt+} \cup F_{pt-} \cup F_{loop}$ be the set of arcs of the required STG. Let $m_0((s,v)) = v, (s,v) \in P$ be the initial marking of the required STG. Let

$$\lambda((s,c)) = \begin{cases} (s,-) & \text{if } t \in T_{reset} \\ (s,+) & \text{if } t \in T_{set} \end{cases}$$

be the labelling function of the required STG. All the elements of the required STG are now defined. By carefully controlling the names of the signals in S any violation trace produced by a verification tool for this STG can be converted into a trace for the source high level model.

Algorithm 2 is a possible implementation of the function Γ in an imperative programming language. This implementation works as follows. For every signal in the input logic network, a pair of complimentary places is created. A token is put into the place that corresponds to the initial state of the signal. Then the set and reset functions are converted into disjunctive normal form (DNF). For every clause in the DNF of the set function, a rising transition is created and connected to the

two places. For every literal in the DNF clause, a read arc connecting the transition with the place corresponding to the state of the signal represented by that literal is created. Falling transitions are constructed in a similar way using the DNF of the set function.

Figure 6.9 illustrates the method described in this section as applied to a gate-level circuit model. Figure 6.10 is the application of the same method to an SDFS model.

6.3.1 Using logic networks to verify multi-formalism models

Definition. Let M be an IGM $M = \langle G, I \rangle$ where $I = L$ is a logic network. I is called a *logic network interpretation* of the model M.

Let K be a set of models with a logic network interpretation. Let LN_k be the logic network for the model $k \in K$. Let $n_1 \parallel n$ be a composition operation producing an STG from two source STGs n_1 and n_2. Then

$$V = \Big\| _{k \in K} \Gamma(LN_k) \tag{6.4}$$

is the verification STG of the set K. The operation $\parallel$ may be defined in any appropriate way, for instance as the parallel composition of Petri nets [125]. Equation 6.4 is a very powerful compositional verification tool. It allows to co-verify and co-simulate an arbitrary set of models of various types, such as substituting a black box containing an STG specification for a part of a gate-level circuit, or using gate-level circuits to provide the environment for an SDFS model.

6.4 Conclusions

In this chapter a class of models called Interpreted Graph Models (IGM) has been defined. Such models use a static graph structure and an arbitrary number of *interpretations* of that structure. The separation of the structure from its interpretations allows generalised algorithms to be introduced. To access those algorithms, an IGM may be extended with additional interpretations without affecting the underlying static structure or the already existing interpretations. Two important algorithms that use this abstraction have been described (an algorithm for generating a graphical representation of an IGM and a verification algorithm that generates an STG that reflects the behaviour of the model).

The concept of an IGM allows to bridge the strictly mathematical objects comprising the various formal models used to describe concurrent systems with practical algorithms that can be implemented in a programming language. The Interpreted Graph Models serve as the fundamental abstraction in the CAD tool Workcraft described in the next chapter.

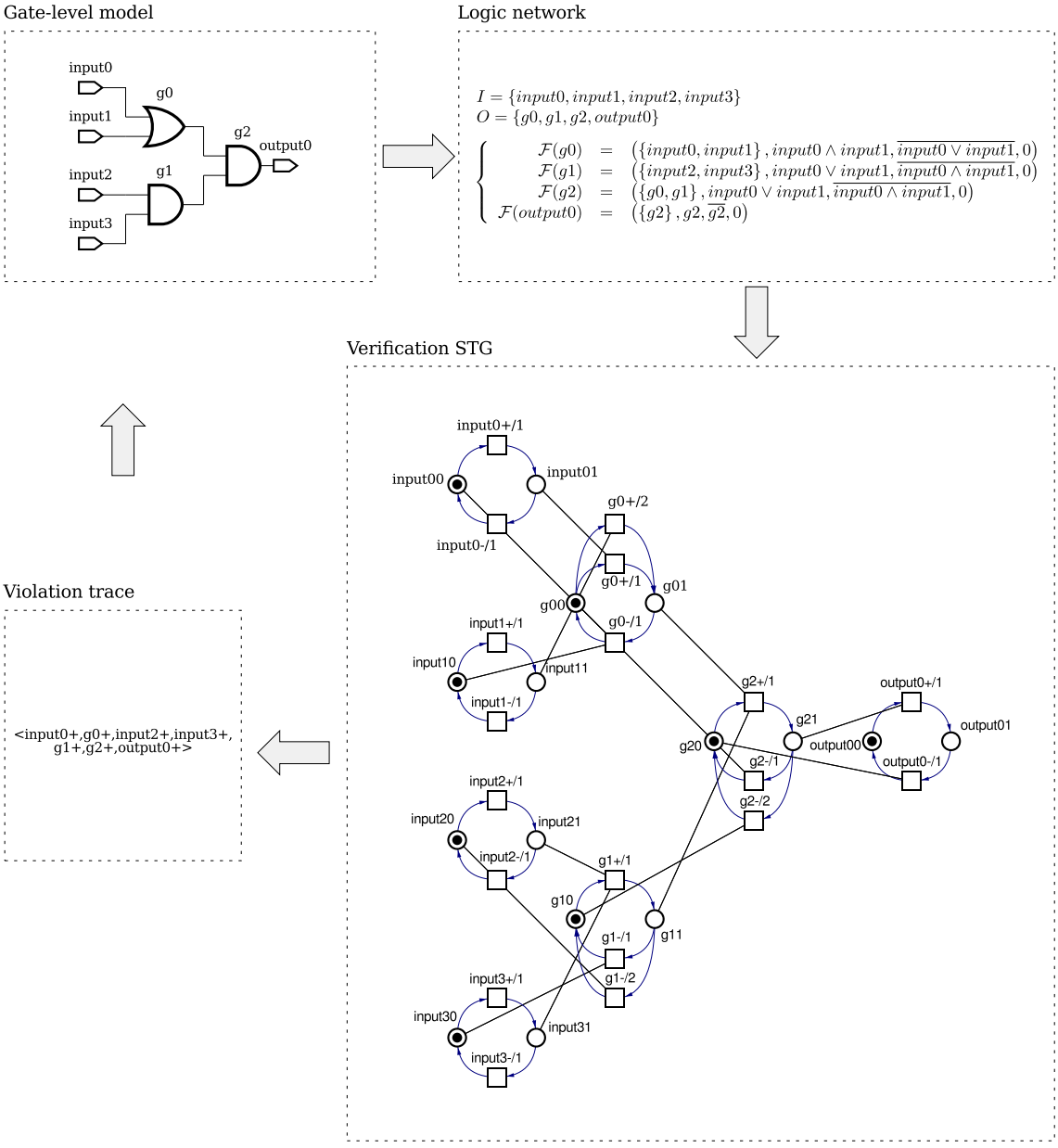


Figure 6.9: Verification of a gate-level model using a logic network

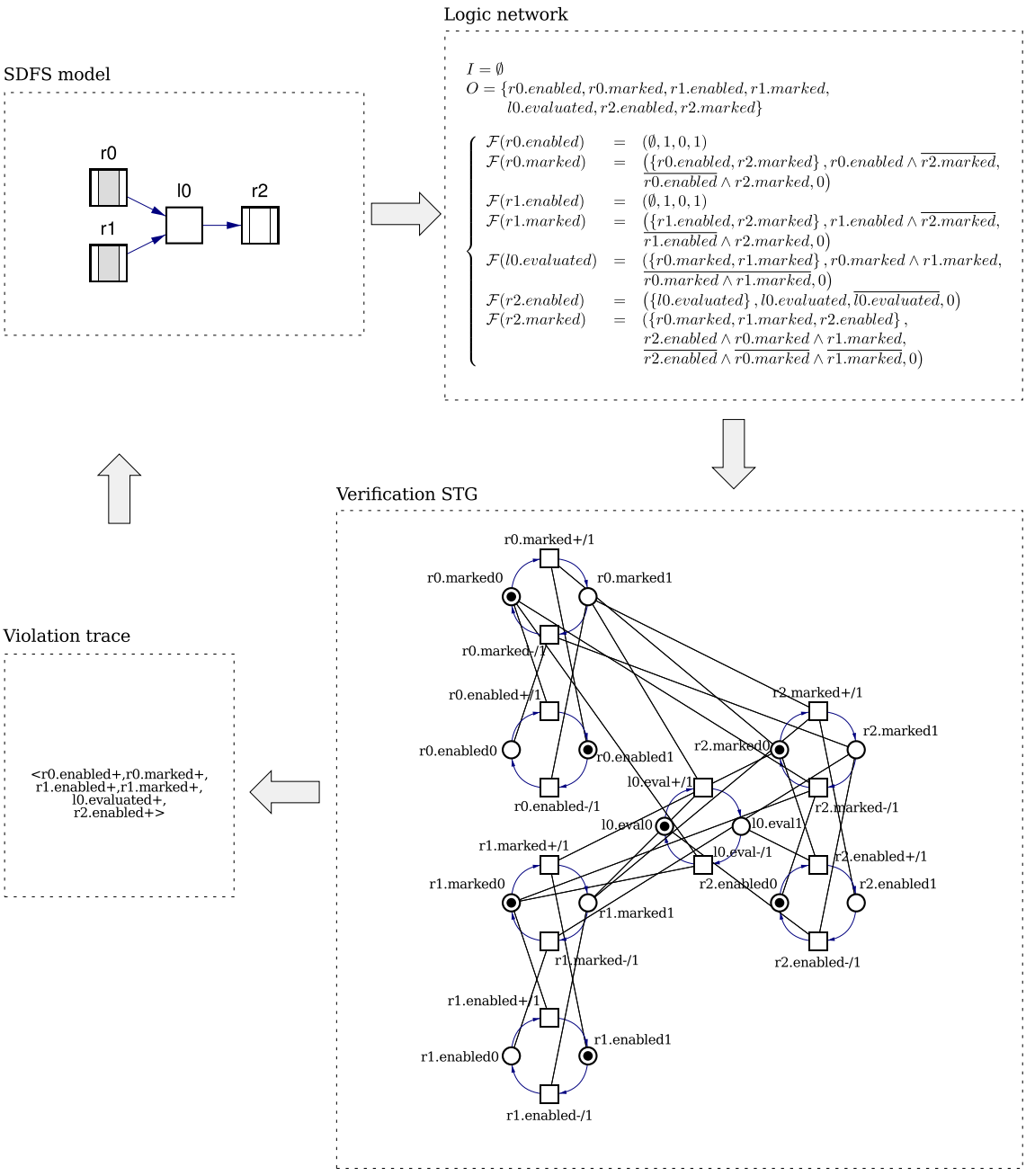


Figure 6.10: Verification of an SDFS model using a logic network

Algorithm 2 Generation of a verification STG from a logic network

```
let S = I ∪ O be the set of all signals in the source logic network
let V be the required STG, initially empty
for each signal s in S:
    create places s0, s1 in V
end for

for each signal s in S:
  set k = 0
  build a DNF set_DNF from f_s^set
   perform Boolean minimisation* of the set_DNF
   for each clause C in set_DNF:
     create a transition T_s^+k in V
     k = k+1
     add arcs (s0,T_s^+k) and (T_s^+k,s1)
     for each literal L in C:
        if (L is positive)
          find a place P in V
                      labelled L1
        else
          find a place P in V
                      labelled L0
        end if
        add a read-arc (T_s^+k,P)
     end for
   end for
   set k = 0
   build a DNF reset_DNF from f_s^reset
   perform Boolean minimisation of reset_DNF
   for each clause C in set_DNF:
     create a transition T_s^-k in Sys_dst
    k=k+1
     add arcs (s1,T_s^-k,T_s^-k-s0)
     for each literal L in C:
        if (L is positive)
          find a place P in V
                      labelled L1
        else
          find a place P in V
                      labelled L0
        end if
        add a read-arc T_j^-k-P
     end for
   end for
end for
```

Chapter 7

Workcraft: a framework for Interpreted Graph Models

This chapter introduces a computer aided design (CAD) tool called Workcraft. The tool is a software framework based on the Interpreted Graph Models (IGM) concept. Instead of binding to a particular set of supported models and analysis methods, Workcraft implements a number of fundamental features that can be inherited from the framework by the plug-ins that realise the concrete models and tools.

The plug-in driven architecture of the tool allows extending it with additional Interpreted Graph Models definitions, new interpretations of existing models and analysis/verification modules. By controlling the set of plug-ins that are included with Workcraft, the tool can be configured to serve as a specialised working environment for the design of specific types of concurrent systems. For instance, Appendix C contains a manual for using Workcraft as an environment for asynchronous circuit design based on the STG model.

7.1 Objectives

The primary design goal of the Workcraft framework is twofold. One target category of users are the researchers who would like to provide tool support for the new models, while the other category are those who wish to design, analyse and verify systems using the formalisms that have already been implemented. To appeal to the former category, a plug-in based architecture was

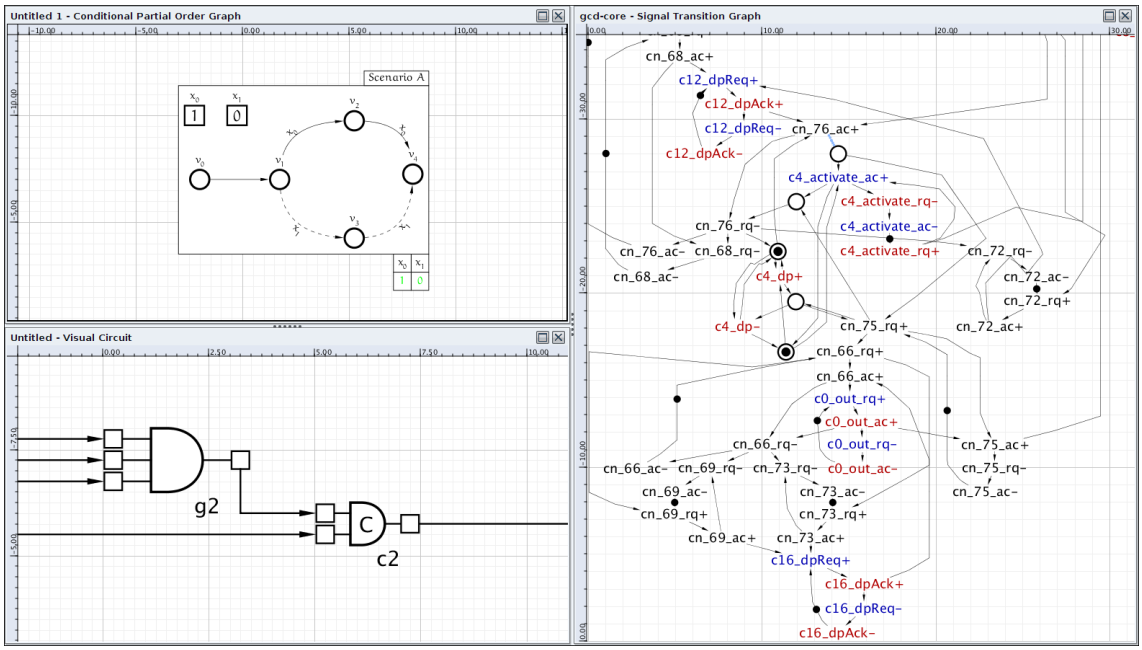


Figure 7.1: Working with three different model types simultaneously

designed, which allows new formalisms to be introduced with the minimum effort (Appendix B) — the benefits of the graphical presentation, automated serialisation and interfacing with external tools are inherited from the framework. In addition, some of the important algorithms (such as the Petri net-based verification) are generalised: by providing a model with a logic network interpretation (detailed in Section 6.3) the author of the model can use the verification functionality without worrying about implementing the Petri net generation and calling the external tools to carry out the verification.

7.1.1 Graphical user interface

One of the major features of Workcraft is the graphical user interface (GUI). Historically, automated graph layout tools such as Dot [7] have been used to produce the images of graph-based models using their graphical notation. This approach is not very efficient because most of the models are in fact dynamic. In Petri nets, for example, transitions transfer tokens between places according to the token game rules. To observe the *evolution* of the model state graphically, a series of static snapshots has to be produced and inspected. To remedy this shortcoming, tools such as PEP [11, 30, 112] that support the interactive simulation of Petri nets were developed. The tool

is able to highlight the currently enabled transitions, and the user can click them to cause them to fire and immediately see the consequences. This way, the user can investigate the behaviour of the net by triggering the different execution paths.

PEP tool, however, is closely tied to the Petri net model. Despite Petri nets being a very popular model, new models that aim to provide a more specific modelling solution are introduced relatively often (e.g., Static Data Flow Structures (Chapter 5), Conditional Partial Order Graphs [80]). To successfully apply these models in a practical design workflow an adequate tool support is extremely important. Designing and implementing a custom graphical environment to support functionality such as visual entry and interactive simulation for every new models is a task that usually requires significant effort.

Workcraft's GUI system is designed in such a way that it handles most of the routine tasks such as the creation of document windows, menus and configuration dialogues, managing the UI layout. Coupled with a generalised graphical presentation algorithm for Interpreted Graph Models, this enables rapid development of model plug-ins with support for advanced features such as visual entry and interactive simulation. For example, in Figure 7.1 a configuration of three editor windows arranged side-by-side is shown. The windows all contain different model types. The plug-ins that define these model types are unaware that such functionality is possible and only implement the drawing routines specific to the model type.

7.1.2 Tool integration

Most of the tools available in the academia, particularly in the context of asynchronous system design, are based on the command-line interface. This is justified because such tools are mostly designed to carry out one particular task (and do it well), but ultimately results in a fragmented state of the tool base because there exists a multitude of standalone tools but not a consistent development environment. From the point of view of a system designer, organising interaction between the tools may be rather cumbersome: every tool has its own set of command-line arguments and configuration parameters that are easy to forget, especially when a large number of tools is used in a single workflow.

Workcraft aims to improve this situation by introducing lightweight plug-ins that wrap the

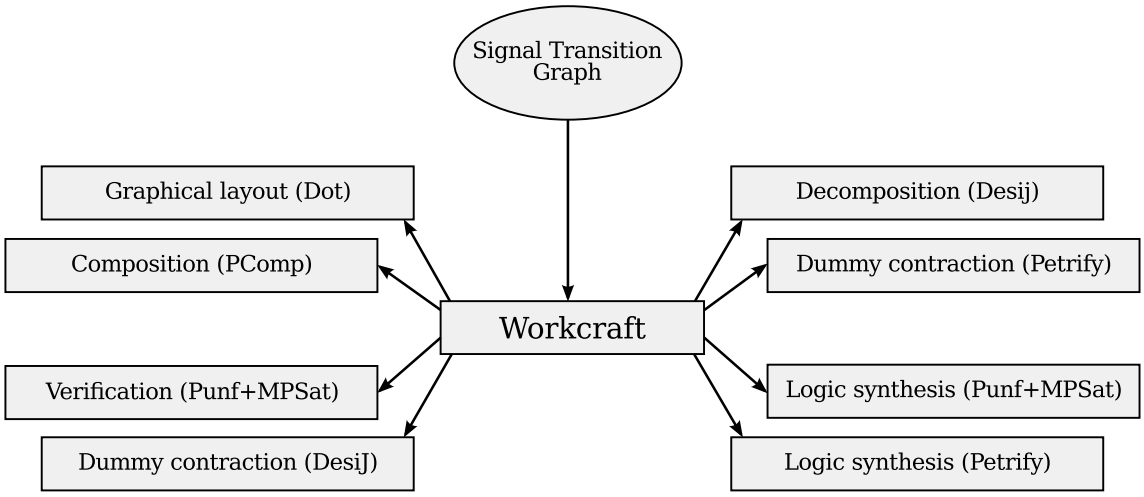


Figure 7.2: The tool integration aspect of Workcraft

command-line tools into organisational units called tasks (detailed in Sections A.3.3 and B.3.1). The tasks can be chained together to form sophisticated workflows by using simple and consistent APIs instead of calling the tools directly. Additionally, the tasks can be executed asynchronously without blocking the rest of the program.

Figure 7.2 illustrates the amount of tools that can be applied to one specific model type (STG). One of the goals of Workcraft is the integration of those tools in a consistent, user-friendly environment.

7.1.3 Formalism interoperation

Workcraft supports formalism interoperability using the STG composition operation (see Section 6.3). In this modelling approach different parts of the system can be specified using different formalisms. To produce a complete model of the system, the parts are individually converted into STGs, then merged to form an STG that describes the behaviour of the whole system which can then be used for verification. For example, it is often convenient to specify a circuit as a gate net-list and its environment as an STG. Then the verification result (i.e. the violation trace) is propagated back and presented to the user as a trace of the original model (Figures 6.9 and 6.10), rather than that of the STG to which the model was translated for verification.

7.2 Comparison with other tools

The tool closest to Workcraft with respect to the design philosophy is probably the OsMoSys framework [123] and its GUI shell called DrawNet [26].

OsMoSys/DrawNet is an environment specifically designed for multi-formalism modelling. The simulation and verification of the multi-formalism models are carried out using a process called *orchestration*, which involves simulating each fragment of the compound specification separately, using an algorithm specific to the formalism used to model that fragment. The simulation results are shared between the fragments using adapters called *bridge formalisms*. OsMoSys and DrawNet use a custom XML-based language called the Formalism Description Language (FDL) to add new formalisms to the framework. The language allows to define model classes, types of objects (nodes and arcs) allowable in those models, their properties and graphical representations. FDL supports object-oriented features such as inheritance and allows to formulate restrictions on the structure of the model (e.g., to disallow arcs between places and transitions).

Compared to OsMoSys/DrawNet, Workcraft does not place as much stress on the multi-formalism modelling paradigm. The multi-formalism approach is supported in Workcraft, but is not a fundamental part of the framework and is realised via plug-ins of the same level as the individual formalism plug-ins. This gives Workcraft more flexibility with respect to the modelling paradigms that can be used at the cost of additional development effort required to implement them. Workcraft also does not use a custom language to define formalisms. Instead all of the model logic is written in Java. Similarly, this allows much more freedom in customising the features of a particular model, but requires a slightly more complicated development process. For instance, the STG model uses the short-hand notation to display the objects of the model but maintains a standard Petri net in the background. Adding advanced editing features such as in-place editing of signal names (Section C.1.2) is also not possible in DrawNet without changing the internal code.

At the time of writing, the DrawNet project seemed to have been abandoned and neither its source code or binary distributions were available.

Pep tool [11, 30, 112] is a comprehensive and extensible framework that includes a set of utilities for verification of Petri nets. Pep tool supports a considerable number of models, including

process algebrae, high-level and low-level Petri nets; it can also export the models into a variety of formats (SPIN, INA etc.). The only models in Pep that support visual representation are high-level and low-level Petri nets; in particular, there is no support for circuits.

The Moebius framework [44] is a tool similar to OsMoSys in that it focuses on the multi-formalism modelling approach, however it does that differently. Instead of simulating the various formalisms individually, Moebius converts each block expressed in terms of a single formalism into an internal representation which allows composing them into a monolithic model that is used for simulation and verification. A comparable approach is used in Workcraft to enable multi-formalism modelling. In contrast to Moebius, Workcraft uses the STG model as the base model type that the other formalisms are translated into. This allows re-using the Petri net verification tools instead of maintaining its own verification code.

Visual STG Lab [59] is a tool for the visual editing of STGs. The tool is tightly integrated with Petrify [41] and is able to apply operations implemented in Petrify to the STG models designed using the GUI. The tool does not support any other model types or tools. The development of the tool is discontinued, and the existing version suffers from serious issues.

Overall, the main design decision that makes Workcraft different from the other similar tools is that it does not focus on algorithms for a particular model type, analysis tool or a modelling paradigm, but aims to provide a common environment, operating system of sorts, that helps to "glue" the existing tools together allowing to use them in a consistent manner. For example, Workcraft provides the visualisation and editing functionality for Petri nets, but does not have any internal verification routines. Instead, it relies on externals tools such as Punf [64] and MPSat [67] to carry out the verification. Then it is able to parse the verification output and present it to the user in a graphical manner. Similarly, Workcraft is interfaced with the tools such as Petrify, Dot and DesiJ and benefits from their algorithmic power while at the same time providing the tools with a user-friendly front-end.

7.3 Tool architecture

The Workcraft framework consists of three major parts (Figure 7.3). These parts are the plug-in manager that scans and categorises the plug-in classes, a set of services accessible to the plug-ins,

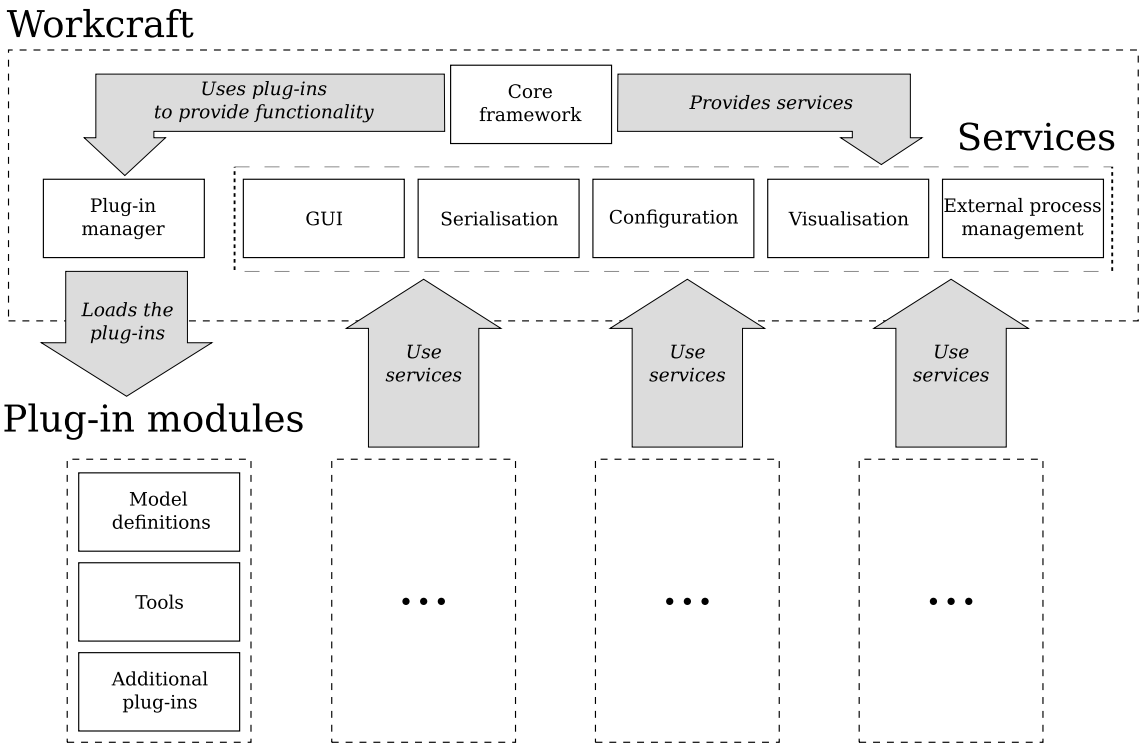


Figure 7.3: The architecture of Workcraft

and the core part of the framework that manages the start-up and shutdown processes, the GUI windows and also provides the scripting support for the command-line mode.

7.3.1 The framework core

The main responsibility of the framework core is to start the other systems that together form Workcraft. The start-up sequence works as follows. The configuration manager is the first component to be started. It loads the configuration files and allows other components to read their configuration variables in a centralised fashion. Then, the plug-in manager is initialised. It either reads the plug-in manifest (if it is present) or starts the plug-in reconfiguration process. When all of the plug-ins are loaded, the start-up scripts are executed. These scripts contain additional start-up logic that can be customised by the user. At this point all of the sub-systems are initialised. The framework core then decides what actions to take next by examining the command-line arguments. Workcraft can optionally be started in the command-line mode, in the script execution mode (a specified script file will be executed and the program will then quit), or, if no arguments are supplied, Workcraft start in the default graphical user interface mode (a detailed explanation

of the various modes of operation is given in Appendix A). When the program is shutting down, the configuration manager is informed so that it can save the current configuration to disk.

Configuration manager

The configuration manager is responsible for storing the configuration variables and provide the other components with a centralised way to access those variables. This allows the system components (including the plug-ins) to use the configuration interface without worrying about saving or loading their configuration parameters: the configuration manager loads the variables on start-up and automatically saves them to disk them on shutdown.

JavaScript engine

JavaScript is used as the scripting language in Workcraft. The scripting engine allows to execute script files or individual commands to further customise the functionality of the framework without having to go through the process of building a complete plug-in module. For example, Section A.2 describes a script that can be used to automatically produce vector graphics from the STG models without having to use the interface.

7.3.2 The plug-in manager

The plug-in manager is responsible for discovering the plug-in modules[1] and categorising the individual plug-ins. Its functionality is realised using the reflection mechanism of the Java language that allows the Workcraft run-time to dynamically load Java classes and inspect them to establish what interfaces they implement. Those classes that implement the *Module* interface defined by Workcraft (see Section B) are instantiated and initialised. During the initialisation each module is allowed to register the individual plug-in classes that implement some extended functionality. The nature of the functionality is defined by the Java interface that the plug-in implements (the set of plug-in interfaces is pre-defined). For each plug-in interface the plug-in manager maintains a list of registered plug-ins that implement it. When another part of the framework needs to know,

[1] A "plug-in module" is a related collections of plug-ins that together implement specific functionality such as a new model.

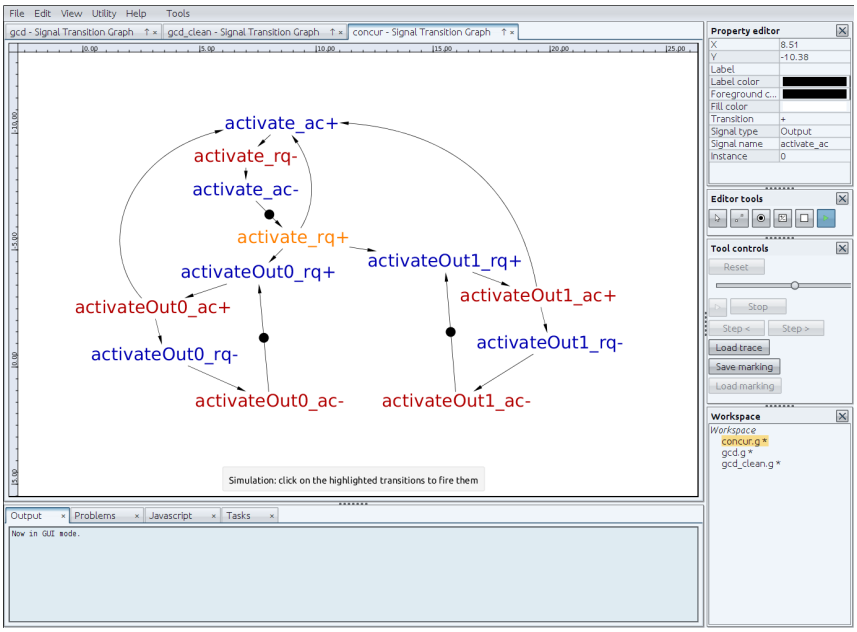


Figure 7.4: The graphical user interface of Workcraft

say, what tools are currently available for the current model it simply passes the corresponding interface to the plug-in manager which responds with the list of plug-ins.

Plug-in reconfiguration

Reconfiguration is an automated process during which the contents of the plug-in packages are analysed, and a list of all discovered compatible plug-ins is built and stored in a special file. During start-up, the plug-in manager uses this list to load the plug-ins instead of scanning the contents of the plug-ins directory every time, which greatly reduces the start-up time. Workcraft automatically reconfigures itself during the first start-up, however if any changes are made to the set of plug-ins in the future the reconfiguration must be triggered manually.

7.3.3 The graphical user interface

The graphical user interface (Figure 7.4) is fully managed by Workcraft, allowing the plug-in authors to focus on implementing the desired functionality of their tools and models without having to worry about things such as window creation and placement. The underlying window toolkit used by Workcraft is the Java Swing, which ensures compatibility and consistent look across all platforms. Workcraft supports a number of advanced GUI capabilities, including a multi-

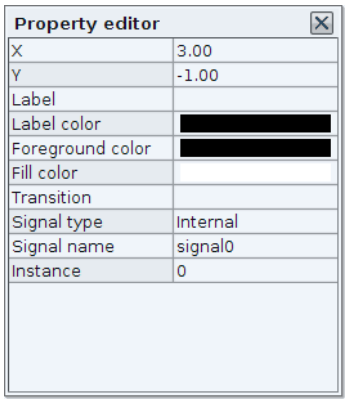


Figure 7.5: The property editor

Object source code

```
public class SignalTransition extends Transition

{

    public Type getSignalType() {
        return type;
    }

    public void setSignalType(Type type) {
        this.type = type;
    }

    public Direction getDirection() {
        return direction;
    }

    public void setDirection(Direction direction) {
        this.direction = direction;
    }

    ...
```

Automatically serialised object

```
 <node class="org.workcraft.plugins.stg.SignalTransition" ref="b-">
     <SignalTransition>


<property class="org.workcraft.plugins.stg.SignalTransition$Type"
    enum-class="org.workcraft.plugins.stg.SignalTransition$Type"
    name="signalType"
    value="INPUT"/>



<property class="org.workcraft.plugins.stg.SignalTransition$Direction"
    enum-class="org.workcraft.plugins.stg.SignalTransition$Direction"
    name="direction"
    value="MINUS"/>


            ...
```

Figure 7.6: An example of automated serialisation

document interface, the full-screen mode, the non-overlapping window docking system and the persistent window layout manager (the window layout configuration is saved to disk when the GUI is shutting down, and restored on the next start-up).

Most of the utility windows (e.g., the new model creation dialogue or the pages in the preferences window) are automatically constructed by Workcraft using the information provided by the plug-ins. Similarly, a graphical property editor (Figure 7.5) that provides support for the user-friendly editing of various property types (such as numerical values, strings and colours) can be used by the model plug-ins without having to explicitly specify the underlying GUI components.

A detailed explanation of Workcraft's GUI features is given in Appendix A.

7.3.4 Automated serialisation

The automated serialisation feature is very helpful to quickly get a working implementation of a new model in Workcraft. It uses the features of the Java language that allow it to inspect the objects contained in a given model to determine their types and extract the declaration of properties they contain. The object types and the values of the extracted properties are then recorded using an XML-based format (Figure 7.6). A set of frequently used property types (numbers, strings, colours, vectors, matrices etc.) is supported "out of the box" and models that use those property types to describe the state of their components can be saved and loaded as Workcraft documents without any additional effort from the author of the model plug-in. If needed, the set of the automatically managed property types can be extended with serialisation plug-ins. For advanced models that define their own serialisation format the automatic serialisation can be disabled.

7.3.5 Visualisation

Workcraft uses the generalised Interpreted Graph Model visualisation algorithm given in Section 6.2. Any model that defines the drawing and transformation functions for its node types can be used with the visual editor provided by Workcraft. Auxiliary editing operations, such as controlling the viewport via panning and zooming, selecting and moving individual nodes, choosing the nodes to be connected etc. are inherited from the framework and need not be implemented. Vector graphics export function that saves the model's graphical representation in the Scalable Vector Graphics format (SVG) can also be automatically applied to any model that defines the drawing functions mentioned above.

7.3.6 External process management

A mechanism for managing external processes (e.g., verification tools) is built into the Workcraft framework. Tool plug-ins relying on external programs can use this feature to avoid manually writing the code that starts and monitors the execution of programs. The task monitoring code is executed on a separate thread which allows executing time-consuming processes without blocking the reset of the program. Workcraft automatically places all external process tasks into the task manager interface. The task manager maintains the list of all running tasks and allows the user to

cancel individual tasks.

7.4 Availability

Workcraft supports all major OS platforms (Windows, Linux, Mac OS) and is freely available for academic use. The latest binary distribution can be downloaded from the tool web site [20]. Please see Appendix A for installation instructions and the user manual. Alternatively, Workcraft can be built using the source code. The building process is detailed in Appendix B.

7.5 Conclusions

In this chapter, a software framework called Workcraft was introduced. Goals pursued during the design and development of the tool were explained. The tool was compared to the previously existing similar tools. An overall architecture of the tool and the individual services accessible to the plug-ins were described.

Workcraft is based on the concept of Interpreted Graph Models that was explained in Chapter 6, which allowed a number of formal models to be implemented in a visually consistent and inter-operable fashion. These models include Signal Transition Graphs [126], Static Data Flow Structures (Chapter 5), Digital Circuits, Conditional Partial Order Graphs [80] and other. The tool has been successfully used in a number of practical applications (see Chapter 8).

Workcraft has been previously presented in [95, 92, 93].

Chapter 8

Use cases

The Workcraft framework based on the Interpreted Graph Model concept has been successfully used in a number practical applications, some of them employing complex interactions between several different model types. Several examples are presented in this chapter.

8.1 Verification of asynchronous circuits

The asynchronous circuit verification method described in Chapter 4 was applied to the following designs.

Verification of a counterflow controller A counterflow stage controller implementation published in [22] was verified and found to be hazardous. A detailed explanation of the verification process and the problem with the circuit that was discovered is given in Section 4.6.

Verification of the flat arbiter design Arbiters are special blocks controlling access to shared resources. They play a very important role in asynchronous circuit design an it is therefore critical to ensure their correct implementation. A method of constructing N-way arbiters was presented in [81]. The 'flat' arbitration method proposed in the paper is prone to threats such as formation of cycles, leading to deadlocks. The construction algorithm presented in the paper gives the correct implementations of N-way arbiters that are deadlock-free.

The circuit verification method presented in this thesis (and implemented in Workcraft) was

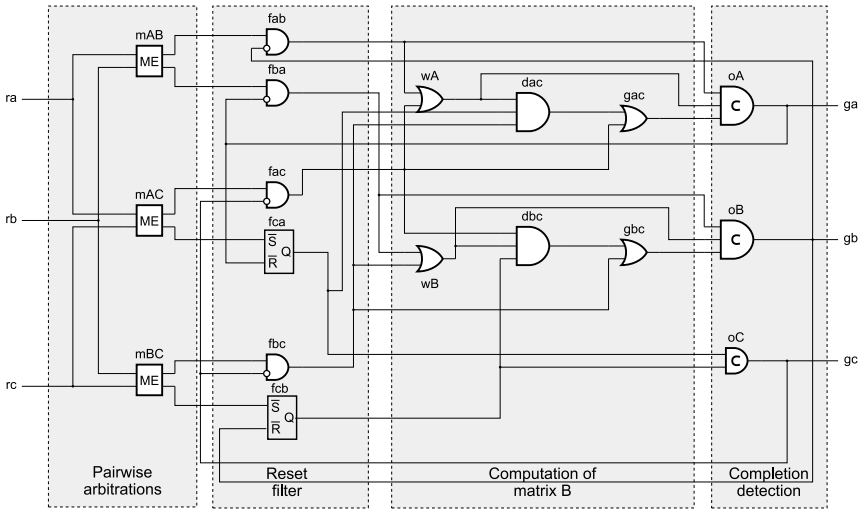


Figure 8.1: Implementation of a 3-way flat arbiter

applied to ensure that the resulting arbiter circuits are deadlock-free and conform to the environment specification.

Verification of a multi-resource arbiter A different class of arbiter circuits was studied in [50]. They are the general-purpose arbiters distributing M resources to N clients for the cases when the resources can be either active or passive participants of the arbitration. In the paper, the arbitration problem is first solved for the case of two active resources being offered to two clients (the implementation is shown in Figure 8.2). Then a general problem solution is provided.

The implementations of the arbiters were verified using Workcraft.

8.2 Static Data Flow Structures simulation and verification

Workcraft played a crucial role in the development of the SDFS model (Chapter 5). An essential property of the logic nodes in the model is the possibility of *early evaluation (EE)* — the situation where just a subset of inputs is sufficient to start producing the computation result. In such a case, all the other inputs are no longer required, and it is best to send a signal to terminate their computation in order to save power and time. There are several types of SDFS capable of expressing datapaths with EE, including spread token, and counterflow.

Systems with EE often have very intricate behaviour, and it is very easy to introduce subtle

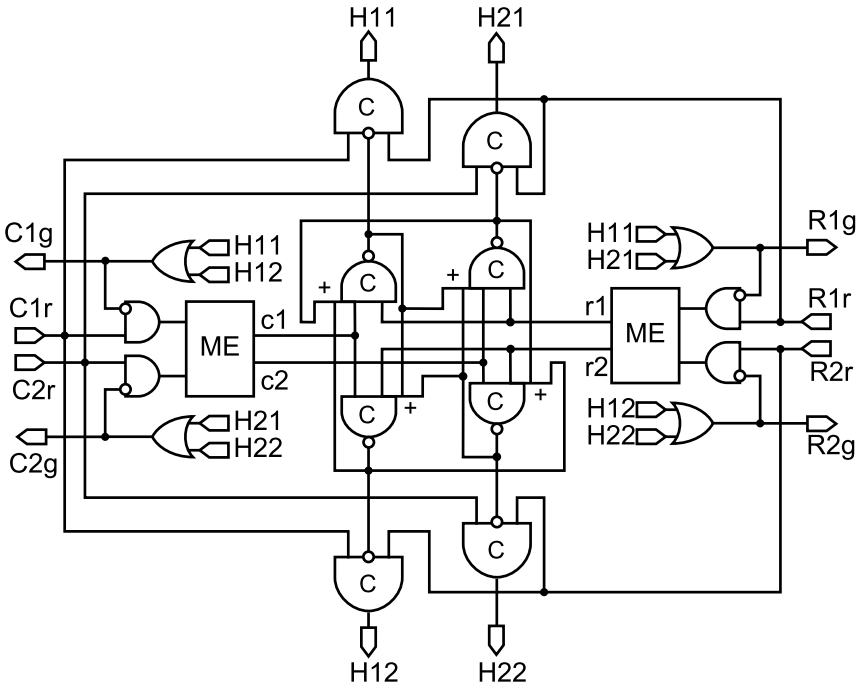
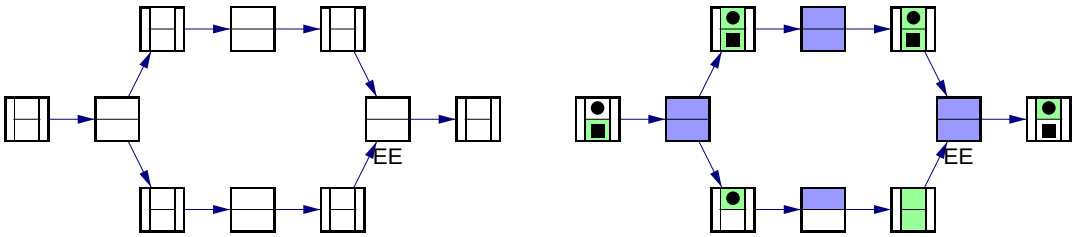


Figure 8.2: Implementation of a multi-resource arbiter



(a) Initial state (b) Deadlock state

Figure 8.3: Counterflow SDFS verification example

errors when designing them. For example, the shortest trace leading to a deadlock in a (rather small) Counterflow SDFS model in Figure 8.3 contains 29 steps. This problem would be rather hard to discover using the manual simulation, due to a very long and peculiar sequence of events that leads to the deadlock. In this example, Workcraft was able not only to detect a deadlock, but also to graphically reproduce, step-by-step, the problematic event trace. This has led to a better understanding of the limitations of the Counterflow SDFS model, and provided the motivation and essential ideas for further adjustment of the token game rules.

8.3 Asynchronous circuit synthesis based on Conditional Partial Order Graphs

Conditional Partial Order Graph [80] is a formalism for circuit specification that combines advantages of both Petri nets and Finite State Machines: it does not have the explicit notion of states (unlike Petri nets) and models the choice on the level of logic conditions (unlike FSMs). The specification size of a highly concurrent system with multiple combinational choice is often much smaller in the CPOG model than in a PN or FSM one.

CPOG support was implemented in Workcraft. The CPOG model appears to be the formalism with most links to other model types (Figure 8.5). Asynchronous circuits can be synthesised directly from CPOG specifications, and verified for speed-independence using the verification algorithm described in this thesis. A CPOG model can also be directly converted into a Petri net, and checked for properties such as deadlocks. Petrify tool can be used as an alternative method of synthesising the same specification, and its result can be compared with that of CPOG-based synthesis so that the user can choose the best one.

8.4 Modification of the workflow of Balsa asynchronous circuit synthesis system

The asynchronous controllers obtained by syntax-directed mapping (see Section 2.2.1) methods realised is systems such as Balsa [47] are usually not optimal, because the pre-designed implementations of the handshake components are required to implement their declared protocols fully and correctly in order to be reusable in all possible circuit configurations. However, it is often the case that a significant part of their functionality becomes redundant due to the peculiarities of the specific configuration, e.g. in many cases full handshaking between the components can be avoided.

This redundancy can be eliminated by replacing the manually designed gate-level implementation of the high level components with an equivalent STG (Figure 8.4). The individual component STGs are then composed together to form a complete system STG, which is optimised using Petrify [41]. An optimal gate-level implementation can then be automatically produced from the

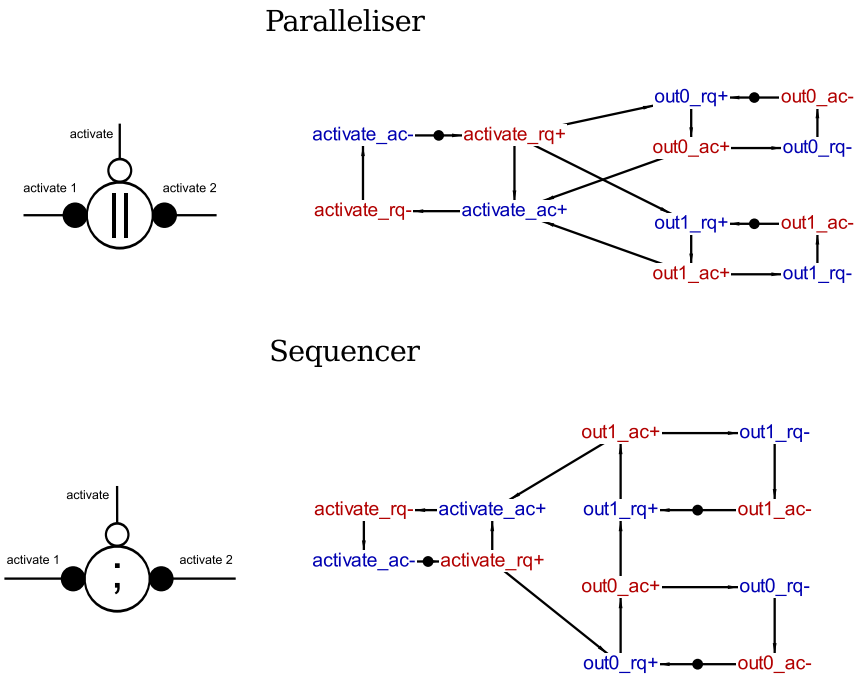


Figure 8.4: The STG specifications of handshake components

STG using tools such as Petrify [41], SIS [103] and MPSat [68]. Automatic synthesis becomes problematic when the size of the STG becomes large: modern synthesis tools can handle STGs of no more than 100 signals. The impact of this problem can be lessened by including STG decomposition tools [99] into the workflow that would break the large optimised STG down into several smaller STGs that are synthesisable in reasonable time. Alternatively, the decomposition step can be carried out on the level of the handshake circuits, dividing the circuit into smaller blocks of components. The whole process is illustrated in Figure 1.4.

For the purpose of implementation of this design flow the Workcraft framework was extended with a plug-in that introduces support for Breeze [28] handshake components. The handshake component model allows Workcraft's visual editing tools to be applied for the creation and editing of Breeze net-lists. The same plug-in also performs generation of the STG behaviour model from a given handshake circuit. The STG generation algorithm is designed to be highly customisable, with support of multiple handshake protocols and various STG implementations for each type of component.

8.5 A development environment based on the STG model

A configuration of Workcraft that makes it possible to use the tool as a feature-rich development environment based around the STG model is described in Appendix C. The tool is able to import and export STG models from the .g file format, automatically generate the graphical layout, perform logic synthesis using various tools (Petrify, MPSat and DesiJ), compose and decompose STG models.

8.6 Conclusions

In this chapter a number of practical applications of the Workcraft framework based on the Interpreted Graph Model concept were presented. These include the application of the asynchronous circuit verification method presented in the Chapter 4 of this thesis for the verification of a counterflow data path controller and two different types of arbiters; the simulation and verification of the SDFS model; the modification of the Balsa asynchronous circuit synthesis system and the application of Workcraft as an asynchronous circuit development environment based on the STG model.

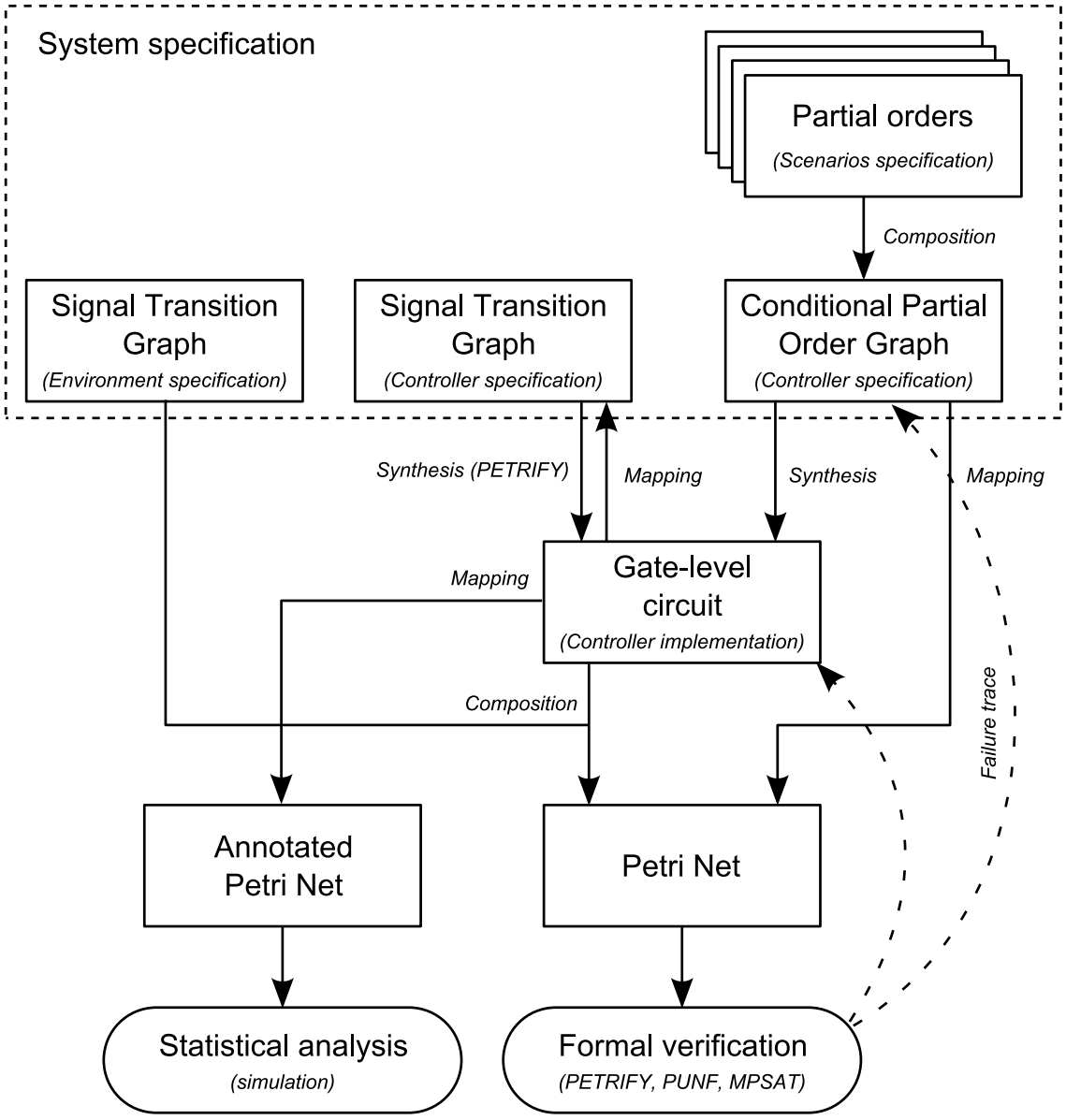


Figure 8.5: A complex model interoperability example

Chapter 9

Conclusions

In this thesis, several formal models and methods relevant to the design of asynchronous circuits were presented. The methods were implemented within a software framework called Workcraft which was also detailed.

9.1 Summary of the contribution

In chapter 2, the fundamental concepts behind the theory of asynchronous circuits concepts were given, such as delay models, operation modes, control, data protocols and the classification of circuits. The most widely used approaches to the design of asynchronous circuits were discussed including an overview of the tools implementing these techniques.

In chapter 3 a formal definition of the Petri net model that is often used in the thesis is given. Using two illustrative examples, the token game of a Petri net was explained. A number of problems characteristic to concurrent systems were highlighted, and it was shown that Petri nets are highly helpful in discovering such problems. Several properties of Petri nets relevant in the context of the thesis were defined.

In chapter 4, a method for verification of asynchronous circuits using Petri nets was proposed. The method checks a circuit given together with a specification of its environment for hazards and deadlocks. Among the advantages of the proposed method is that it uses the well-established Petri net tool base to solve the verification problem. This allows choosing the most efficient verification tool based on the structure of the original circuit. The performance of the new method was com-

pared to the previously existing verification techniques. A practical application example involving the verification of a previously published asynchronous data path controller circuit was given. The verification revealed critical problems with the controller which provided the motivation for the development of a formal model for asynchronous circuit data path.

In chapter 5, a new formal model of the data path in asynchronous circuits was detailed. The model, called the Static Data Flow Structure (SDFS) is comparable to the Register Transfer Level (RTL) used in the design of synchronous circuits. In contrast to the RTL model, it provides means to describe complex behaviours such as *preemption*, *early evaluation* and *speculation* that are useful in an asynchronous data path. Preemption is a technique which allows the destruction of data objects in a computation pipeline if the result of computation is no longer needed, reducing the power consumption. Early evaluation allows a circuit to compute the output using a subset of its inputs and preempting the inputs which are not needed. In speculation, all conflicting branches of computation run concurrently without waiting for the selecting condition; once the selecting condition is computed the unneeded branches are preempted. The proposed Petri net based verification technique is especially useful because of the complex nature of these features. A possible extension of the SDFS model that allowed to model the influence of the control path was investigated.

In chapter 6, a modelling abstraction called an Interpreted Graph Model (IGM) was introduced. This abstraction allows to separate the structure of a graph-based model from the interpretation of the objects that it contains. By associating different interpretations with the same underlying structure, various generalised algorithms can be applied. Two important algorithms were given in the chapter to illustrate the usability of the IGM concept. The first example is an algorithm that allows to produce a graphical representation of a graph-based model with minimal effort. The second example is the generalisation of the Petri net-based verification approach used in Chapters 4 and 5 that enables the application of the technique to other models, which is particularly useful for the multi-formalism modelling approach.

In chapter 7, a software framework called Workcraft was presented. Workcraft is designed to provide a consistent development environment based on various graph-like models. The framework is heavily based on the Interpreted Graph Model (IGM) concept which greatly facilitates

the introduction of new model types. The chapter explained the goals pursued during the design and development of the tool, compared the tool to other similar solutions and detailed its software architecture.

In chapter 8, a number of practical applications of Workcraft and its underlying IGM concept were presented, including the verification of several asynchronous circuit designs, debugging of the SDFS model, implementation of asynchronous circuit synthesis method based on the Conditional Partial Order Graph model and the modification of the workflow of Balsa asynchronous synthesis system.

The features and capabilities of the Workcraft framework are further detailed in the appendices. Appendix B explains how to introduce new models and tools into the framework from a programmer's point of view. Appendix A contains the overview of the graphical user interface of Workcraft from a user's point of view. Finally, Appendix C presents an example of using Workcraft as a development environment based on the STG model

9.2 Future work

The Workcraft framework that bases on the concept of Interpreted Graph Model has proven itself to be a useful tool in the context of asynchronous circuit design. However, there is still much work to be done before Workcraft meets its ultimate goal — to become a complete development environment for the design of asynchronous circuits.

In particular, the asynchronous circuit verification method proposed in this thesis can be improved by introducing means of detecting livelocks. There are rare cases of circuits that can be caused to be stuck in an in an infinite loop by the environment, repeating some actions but never achieving progress. The verification method described in this thesis is unable to detect such behaviour. The method could also be improved by adding support for relative timing assumptions, which would allow to exclude potential circuit failures that can never happen in practice due to the timing constraints.

To make the SDFS model more practical, a method for translating the abstract SDFS specifications into concrete asynchronous circuits has to be developed. Such method would be especially useful if realised in the Workcraft framework to complement the already existing methods for

verification and synthesis of asynchronous controllers.

To improve the modelling power of Workcraft, a hierarchical modelling solution could be implemented. In this paradigm a system would be composed using modular blocks in such a way that the designer could control the observed level of detalisation of the sub-blocks. For example, a model of a CPU on the highest level of abstraction would consist of large blocks such as the ALU, the microcontroller, the register banks etc. Using the hierarchical modelling method, a designer would be able to "descend" into one of the high level sub-blocks to explore and change its specification. The specification of the sub-blocks could be expressed using different formal models, such as, e.g., the CPOG model for the microcontroller, the SDFS model for the data paths in the ALU, the STG model for an external communication unit or even manually designed blocks for components such as the arbiters in the data bus controllers. Because each of these models can be translated into fragments of asynchronous logic, the whole model could be compiled into a monolithic gate-level implementation.

In order to support such modelling approach, a meta-model needs to be implemented. This model would be an IGM in itself, and its nodes would be the high level blocks containing the specifications expressed using the lower level formalisms. Several levels of abstraction could be introduced, where the specifications of the sub-blocks would also be meta-models. Additionally, support for a library of standard elements (e.g., arbiters, mutexes, registers) needs to be implemented. Using the library, the designer would be able to instantiate the pre-designed blocks to build a complex model instead of assembling them manually.

Appendix A

Workcraft user manual

This appendix contains a general manual for Workcraft. It explains the steps required to install, configure and run the tool and gives an introduction on using the two operating modes: the command-line and the GUI.

A.1 Installation and system requirements

Latest Workcraft distributions are available from its web site [20].The distributions currently do not include an automatic installer. To install Workcraft, the files from the distribution archive need to be extracted manually into the preferred directory.

There are no strict system requirements to run Workcraft, the only requirement is that the system has a compatible Java Runtime Environment. Performance of the tool was found to be acceptable on any modern machine, including those based on the slower Intel Atom processors.

A.1.1 Setting up the Java Runtime Environment

Workcraft requires a properly configured Java Runtime Environment (JRE) version 6 or higher to run. The standard JRE [8] is provided by Sun Microsystems and is available for Windows, Linux, and Mac OS. OpenJDK [10], the open-source Java Development Kit, also includes a compatible JRE.

Workcraft is regularly tested only against Sun's proprietary JDK, and may have performance issues when run using OpenJDK. It is therefore recommended to switch to Sun's JRE if any unex-

```
/                       — Workcraft distribution root
    /plugins            — Directory containing the
                          plug-in packages
    /config
        config.xml      — Configuration variables
        plugins.xml     — The plug-in manifest
        uilayout.xml    — Layout of the UI elements
    workcraft.js        — Windows startup script
        OR
    workcraft.sh        — Mac/Linux startup script
```

Figure A.1: The Workcraft distribution structure

pected behaviour or performance problems arise.

In Ubuntu (and derived operating systems) the Sun's proprietary JDK is available through the package sun-java6-jdk, which may be installed either using the Synaptics package manager or by running the following command:

```
sudo apt-get install sun-java6-sdk
```

For better performance in GNU/Linux operating systems it is also recommended to turn off the desktop effects managers (e.g. Compiz).

A.1.2 Distribution structure

Figure A.1 shows the structure of the distribution. The directory called "plugins" is of particular interest to the user: it contains the plug-in packages that provide the implementation of various Interpreted Graph Models and the supporting tools. By managing the contents of this directory, Workcraft may be configured to provide the necessary functionality. The plug-in management process is straightforward: the plug-in packages (in the form of *jar* files) may simply be added to or removed from this directory.

The directory called "config" contains three files. The "config.xml" file contains various user-defined configuration parameters such as visual preferences, external tool paths, etc. This file is usually updated from the GUI, however it is stored in a human-readable XML format and may be edited manually if some variables need to be changed without starting the GUI. The "plugins.xml" file contains the list of plug-ins found during the reconfiguration process (see Section A.1.3). It should never be changed manually. Finally, the "uilayout.xml" file contains the layout parameters

of the user interface elements. If it is removed, the UI will reset to the default configuration. This may be useful for troubleshooting and can also be done from the GUI (*Utility* → *Reset UI layout* in the main menu, see Figure A.4).

A.1.3 Plug-in reconfiguration

If any changes are made to the contents of the "plugins" directory, Workcraft must be *reconfigured*. Reconfiguration is an automated process during which the contents of the plug-in packages are analysed, and a list of all discovered compatible plug-ins is built and stored in the "config/plugins.xml" file. During startup, Workcraft uses this list to load the plug-ins instead of scanning the contents of the directory every time, which greatly reduces the start-up time. Reconfiguration must be triggered manually, either by starting Workcraft from the command-line with the "-reconfigure" argument, or using the GUI (*Utility* → *Reconfigure plugins* in the main menu, see Figure A.4).

A.1.4 Launching Workcraft

In the Windows distribution, the start-up script is called workcraft.js and can be run either by double-clicking on it in the Windows Explorer window or by typing "workcraft" in the command-line window (the current directory should be the directory extracted from the distribution archive). In the Linux distribution, the script is called workcraft.sh and can be similarly run either from the command-line ("./workcraft.sh") or from a graphical file manager.

When creating an application launcher in a desktop environment (also called a "shortcut" in Windows), it is necessary to ensure that the working directory is set to the root of the Workcraft distribution. It is possible to set the working directory in the shortcut properties tab in Windows, however in some Linux desktop environments (e.g., GNOME) application launchers do not have such a parameter. To work around this limitation, the start-up command must change the working directory before launching Workcraft. One way to achieve this is as follows:

```
bash -c "cd [Workcraft distribution directory] && ./workcraft.sh"
```

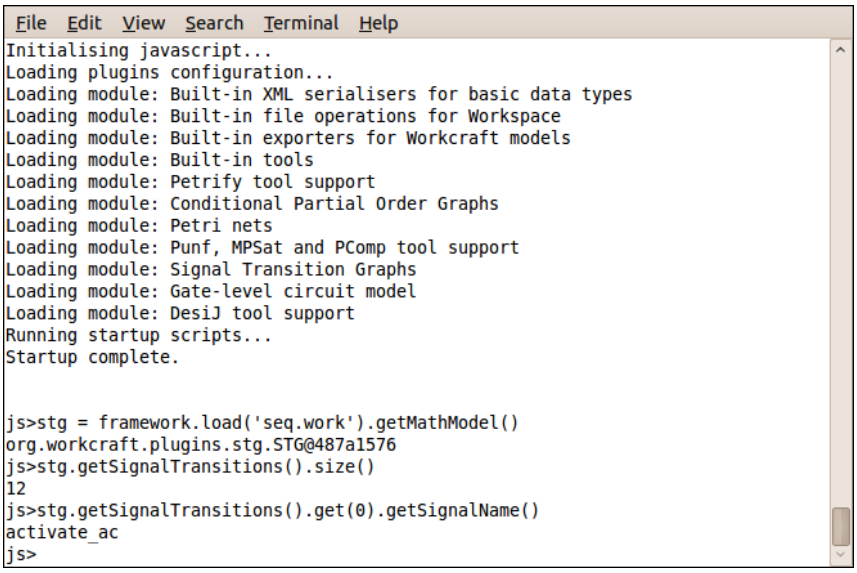


Figure A.2: Workcraft running in the interactive command-line mode

A.2 Command-line mode

Workcraft supports two different modes of operation: the command-line mode and the GUI mode. The command-line mode is implemented using a JavaScript interpreter and may be used either in the interactive mode or in the batch mode. The interactive mode allows to execute single JavaScript statements and immediately see their results (Figure A.2). The batch mode is used to execute a set of script files without user interaction.

Workcraft is started in the interactive command-line mode using the “-nogui” argument:

```
./workcraft.sh -nogui
```

Alternatively, “exec:filename” argument is used to run a script file without interaction:

```
./workcraft.sh -nogui -exec:gtosvg.js seq.g
```

The command-line mode allows to use Workcraft for processing Interpreted Graph Models as a part of a larger task. In Figure A.3, an example script is given that produces an SVG image of a Signal Transition Graph given in the form of a .g file. The script works as follows. First, the STG model is imported from a .g file using the DotGImporter class. Because the .g file does not define any visual layout information for the model, a visual model must be created explicitly. When that is done, the dot layout plug-in (implemented by the class DotLayout) is applied to the model. Finally, the model is exported to an .svg file using the SVGExporter class.

This script may be run as a standalone command as shown above. Apart from producing the SVG files, this command can be used, for example, as a part of a shell script to generate a PostScript image of the STG (using the Inkscape editor):

```
./workcraft.sh -nogui -exec:gtosvg.js $1
inkscape $1.svg --export-eps=$1.eps --export-text-to-path
```

Workcraft uses the Rhino engine to execute JavaScript. Because Rhino is implemented in Java, it allows to use the Java objects directly from JavaScript and therefore no special objects are needed to organise the interaction of the script with the Workcraft's core objects. A useful tutorial on using the Rhino JavaScript implementation to interact with Java programs is available in [13].

A.3 GUI mode

The GUI mode is the default mode used by Workcraft. In this mode, the interaction with the Interpreted Graph Models is done via the visual editing interface and the interactive tools. The GUI mode also provide facilities for managing larger projects (Workspace).

A.3.1 User interface overview

The default GUI configuration is shown in Figure A.4. This configuration is used when Workcraft is started for the first time, or when the GUI layout is reset as explained in Section A.1.2. All of the interface windows can be re-arranged by the user, and the layout configuration will automatically be saved and restored during the next start-up of the program.

The user interface of Workcraft consists of eight main elements as shown in Figure A.4.

The main menu (1) is composed of the "File" and "Edit" menus that provide the general file- and editing-related operations, the "View" menus that controls the visibility of various GUI elements and the "Tools" menu that contains the automatically selected set of tools that are applicable to the current model.

The editor tabs (2) allow to switch between the individual editor windows. The editor windows (4) provide a graphical view of the current model and the interface of the selected tool. These windows are used for navigating the model and support scaling (using the mouse wheel) and panning (holding the middle mouse button and dragging) operations to control the viewport. The same

```
importPackage(org.workcraft.util);
importPackage(org.workcraft.plugins.interop);
importPackage(org.workcraft.plugins.layout);
importPackage(org.workcraft.workspace);

if (args.length != 1)
{
    println ('.g␣file␣name␣missing,␣aborting');
}
else
{
    println ('Converting␣' + args[0] + '␣to␣' + args[0] + '.svg');

    stgImporter = new org.workcraft.plugins.interop.DotGImporter();
    svgExporter = new org.workcraft.plugins.interop.SVGExporter();
    dotLayout = new org.workcraft.plugins.layout.DotLayout(framework);

    modelEntry =
        org.workcraft.util.Import.importFromFile(stgImporter, args[0]);

    visualModel =
        modelEntry.getDescriptor().getVisualModelDescriptor().create(modelEntry.getModel
            ());

    modelEntry.setModel(visualModel);

    workspaceEntry = new org.workcraft.workspace.WorkspaceEntry(null);
    workspaceEntry.setModelEntry(modelEntry);
    dotLayout.run(workspaceEntry);

    org.workcraft.util.Export.exportToFile(svgExporter, visualModel, args[0] + '.svg');

    println ('Done!');
}

done();
```

Figure A.3: A script for automated generation of SVG images from .g files

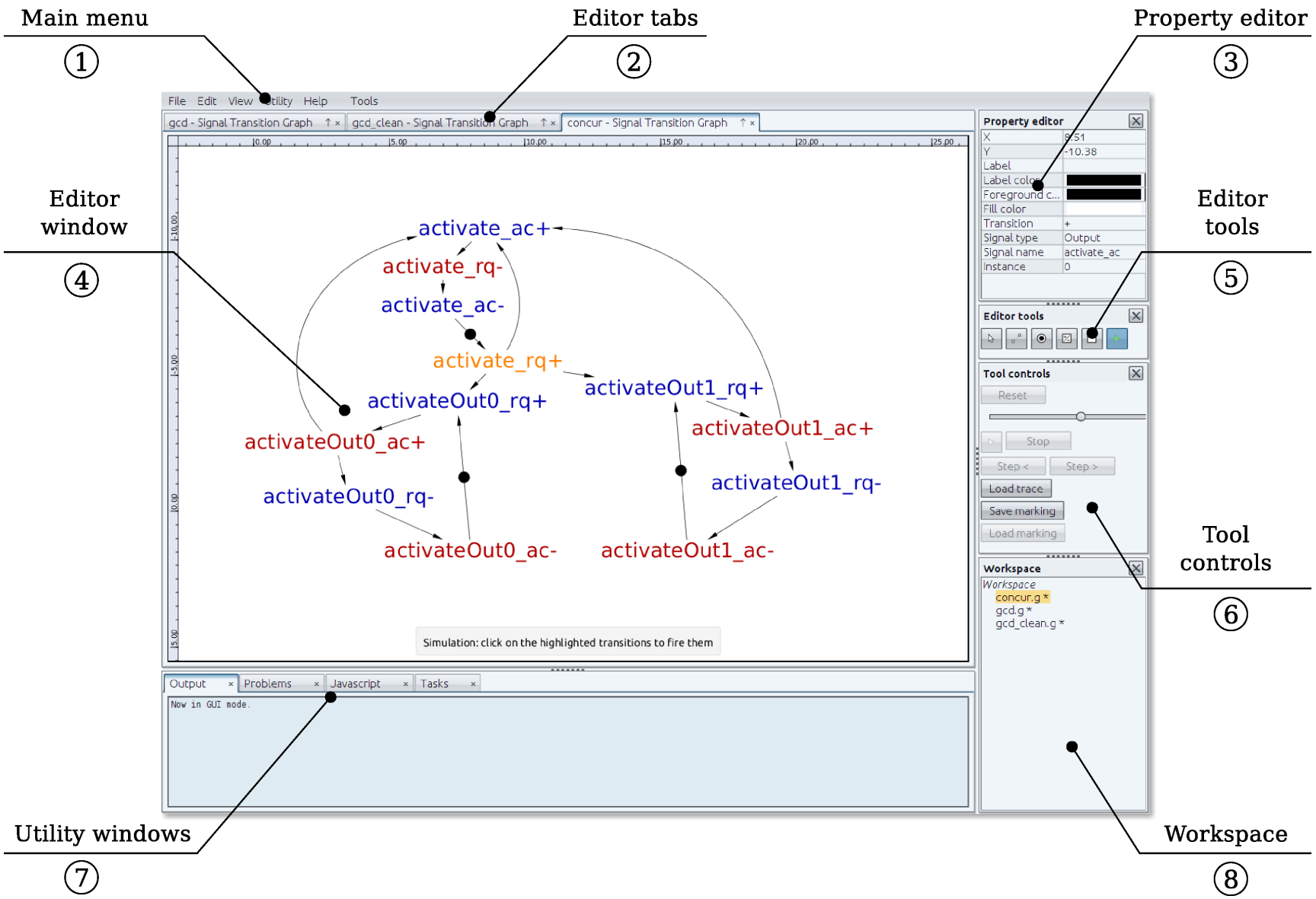


Figure A.4: The main window of Workcraft

model representation is usually used for the interactive simulation, however this functionality may be changed by the implementation of a particular model.

Editor tools (5) are used to switch between working modes, such as creating a particular type of node, creating connections between nodes, simulating the model etc. The tool buttons optionally define hotkeys that allow to quickly switch between the tools using the keyboard. The hotkey associated with a tool is displayed when the mouse pointer hovers over the tool button for a certain amount of time.

The property editor (3) displays the properties of the currently selected node and provides the user with the means to change them. The property editor is used, for example, to change the node label or its colour, to set the number of tokens in a Petri net place, set the type and the logical function of a circuit gate.

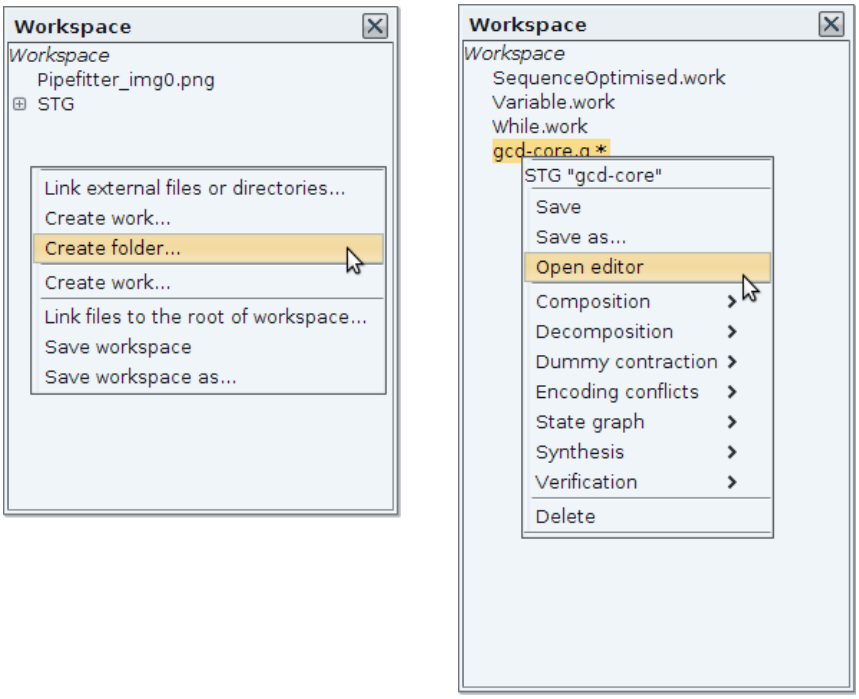
The tool controls window (6) contains the control elements defined by the active tool. In the figure, the simulation tool is the active tool, and this window contains buttons that allow to step the simulation forward and backward, to save and load simulation traces etc.

The utility area (7) has four tabs: the output, which is used to display various information during normal execution of the program, the problems window that displays errors which might have occurred during the execution, the JavaScript window that allows to execute scripts and the tasks window that allows to control the progress of currently executed tasks.

The workspace window (8) shows the files that are contained in the current workspace (see Section A.3.2).

A.3.2 Workspace

Workcraft uses the concept of *workspace* to make managing collections of related files easier. A workspace is very similar to what is usually called a *project* in an integrated development environments (IDE) such as Eclipse or Visual Studio. More specifically, a workspace is a directory in the file system that contains files and directories that are shown in the Workcraft's workspace window (Figure A.5), allowing to perform actions on those files using the interface of Workcraft. Additionally, a workspace stores a set of *mount points* that are directories external to the workspace, but are treated as a part of the workspace by Workcraft. This feature allows to share files between



(a) Workspace operations (b) Workspace entry operations

Figure A.5: The workspace window and its context menus

several workspaces.

When Workcraft is started, a temporary workspace is created. This workspace is stored in the system's standard location for temporary files. All files that are opened or created will automatically be added to this workspace. Additional external directories may be added to the workspace either using the main menu (*File→Link files to the root of workspace*) or using the context menu that is brought up by right-clicking on the blank space in the workspace window (Figure A.5a). The current workspace may be saved to a user-specified location using either of those menus.

A context menu for workspace entries is brought up by right-clicking on a particular entry. The contents of this menu depend on the type of the selected entry. For example, in Figure A.5b a context menu for a Signal Transition Graph entry (stored in a .g file) is shown. This menu contains the same set of tools (applicable to a Signal Transition Graph model) as does the "Tools" sub-menu of the main menu. However, it is not necessary to open an editor window for the model to access the tools using the workspace interface.

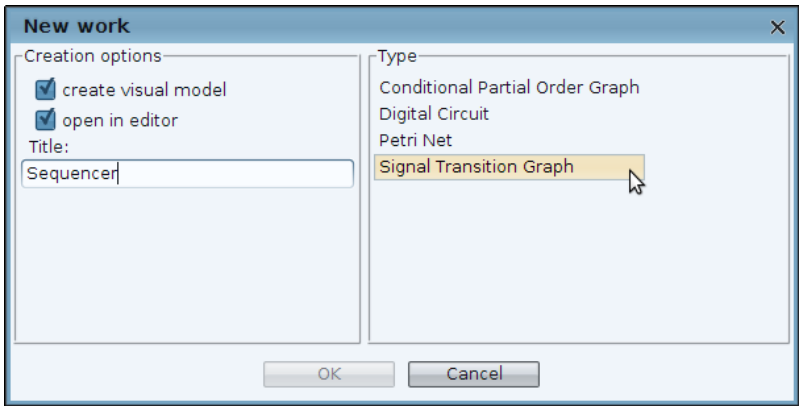


Figure A.6: The “New work” dialogue

A.3.3 Working with models

Creating models

New models are created using the “New work” dialogue (Figure A.6). This dialogue is accessible either from the main menu (*File→Create work*), by using the keyboard shortcut (*Ctrl+N*) or from the workspace context menu. To create a new model, its type and the optional title should be specified. By default, the model is created with the corresponding visual model, but this can be disabled by unchecking the check-box labelled “Create visual model”. Omitting the visual model may be useful if the new model is not supposed to be edited manually, but rather using tools or scripts. Some model types may also lack support for visual editing. If the “Open in editor” check-box is checked, an editor window will automatically be opened for the new model.

Import and export

The import operation creates a Workcraft model from a given file and adds it to the workspace. The set of supported file types is defined by the set of currently loaded import plug-ins. The model import dialogue (Figure A.7) is accessible from the main menu (*File→Import*). It is possible to import multiple files at once using the dialogue by shift-clicking on the additional files to add them to the selection. It is also possible to filter the displayed files using the “Files of type” combo-box, showing only those that are supported by the chosen import plug-in.

Export is the reverse operation, i.e. it creates a file of a certain type from a Workcraft model. It is similarly accessible from the main menu (*File→Export*). Export operations are also defined

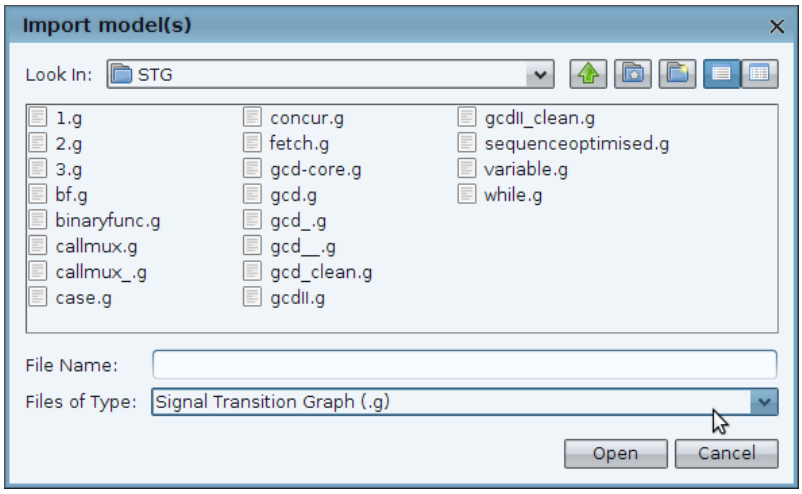


Figure A.7: The model import dialogue

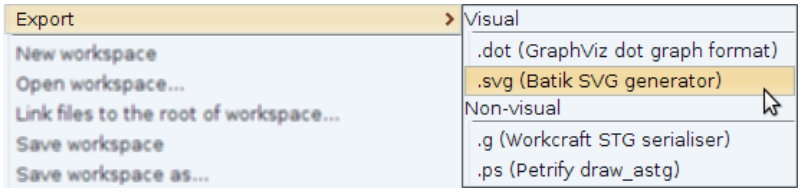


Figure A.8: The model export sub-menu

by the plug-ins, and it is possible to have an arbitrary number of target file formats for a given model type. For instance, the graphical representation of model may be written as an image file. In Figure A.8, several possible export targets for a Signal Transition Graph model are shown.

Editing

A new visual editor window can be created by right-clicking on a model entry in the workspace window and choosing “Open editor”. The number of simultaneously open editor windows is not limited by Workcraft. Additional editor windows will be attached to the primary editor window using a tab-panel interface (see Figure A.4, items 2 and 4). The currently active editor window is highlighted with a black border. Focus can be shifted between the editor windows by clicking on

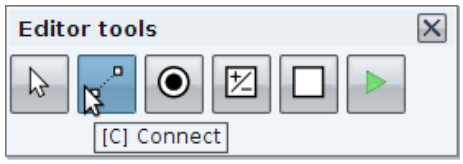


Figure A.9: The editor tools window with a hotkey tool-tip

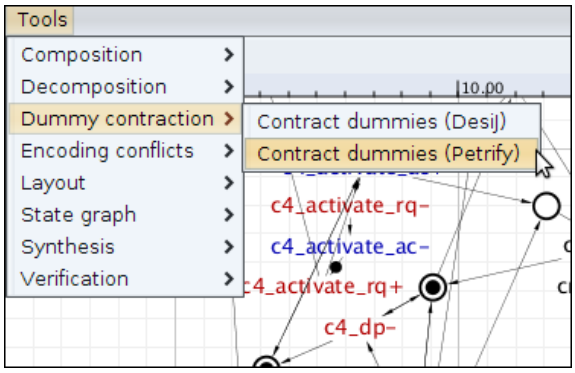


Figure A.10: The set of tools applicable for the current model (an STG)

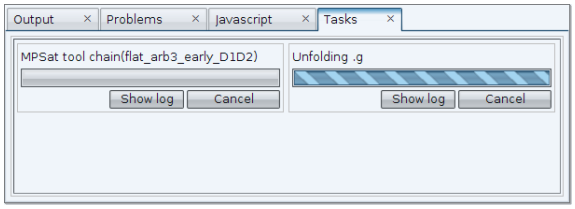
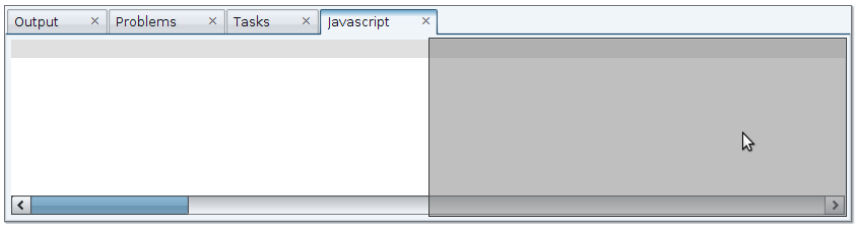


Figure A.11: The tasks window

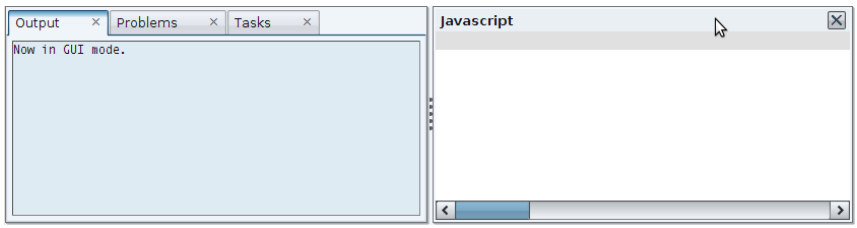
their contents.

The contents of the toolbox (Figure A.4, item 5) and the “Tools” sub-menu (Figure A.4, item 1) depend on the type of the model that the currently active editor window holds. When another editor window is made active, the tools are updated accordingly. The editor tools can be switched either by clicking on the tool icon with a mouse, or by pressing the corresponding hotkey. The hotkeys are shown when the mouse cursor hovers above the tool button for a small period of time (Figure A.9).

The set of editor tools and the way they interact with the model via the editor window is completely defined by the model plug-in and the user should refer to the documentation of a particular model plug-in for reference. The only conventions are that the mouse wheel controls the zoom level of the editor viewport, holding the right (or middle) mouse button and moving the mouse pans the view. These operations are also accessible from the keyboard: + and - keys control the zoom level and *Ctrl+arrow keys* controls the panning.



(a) Window being dragged to another docking location



(b) Window moved to the new docking location

Figure A.12: Changing the interface layout

Applying tools and controlling asynchronous processes

The analysis tools can be invoked either from the main menu as shown in Figure A.10 or from the workspace window as shown in Figure A.5b. The set of applicable tools is defined by the currently loaded plug-ins. The tools usually present the user with a dialogue for interaction, although this is not required. If the operation performed by the tool potentially takes a considerable a considerable amount of time, the tool may choose to start its computationally intensive process asynchronously, without blocking the rest of the user interface. When the tasks completes, the tool will present the user with the results.

The progress of such tasks can be monitored using the “Tasks” window shown in Figure A.11. Tasks that are no longer needed or take unexpectedly long to complete may be cancelled from this window.

A.3.4 Changing the user interface layout

The layout of the user interface in Workcraft is defined using the relation called docking. Every window, regardless of its type, is assigned a docking region relative to some other window (except for the root window that is invisible to the user). There are two types of regions: the side regions and the central region. If the window is docked to another window’s central region, the two windows will share the same space on the screen, and will be placed into a tabbed window

container to allow switching between them. An example of such docking arrangement is the utility area (Figure A.4, item 7). If the window is docked to another window's side region, they will be arranged in a side-by-side fashion. The divider between them can be dragged to distribute the screen space as required.

Docking of any window can be changed by "dragging" its header to the desired location. As the mouse cursor is moved over the regions of other windows, the position that the window would take if the button were released is shown using a grey shade. For side-to-side arrangements, a half of the target window (top, bottom, left or right) is shaded. For the tabbed pane arrangement, the whole window area is shaded. Figure A.12 show the window titled "Javascript" being moved to another docking location. In subfigure A.12a, the window is being dragged and the grey docking location preview is seen. In subfigure A.12b, the new docking location is accepted and the window is docked.

The layout is persistent and is restored each time Workcraft is started. It can be reset back to the default arrangement using the main menu (*Utility→Reset UI layout*).

Workcraft has eight standard utility windows. They are "Output", "Problems", "Javascript", "Workspace", "Property editor", "Editor tools", "Tasks" and "Tool controls". These windows may be hidden at any time either by clicking on the close button (located in the window header panel) or by using the main menu (*View→Windows*). A hidden utility window may be shown again by clicking its name in the *View→Windows* menu. Those windows that are currently shown will have a checked box near their name.

A.3.5 Changing the look and feel of the interface

Workcraft uses the Java Swing library for its user interface. This library is designed to achieve a consistent look across all platforms. The look of the Swing UI elements is defined by a "look and feel" package that may be changed on the fly. The look and feel used by Workcraft may be changed using the main menu (*View→Look and feel*). An advanced look and feel package called Substance [15] is included with the Workcraft distribution. This package provides a large selection of colour schemes and styles for the UI elements. Additionally, it honours the DPI setting of the monitor correctly (as opposed to the standard Swing look and feel) which may be critical for very

high resolution displays. To achieve the best performance, however, the default Swing look and feel is the best choice and it is recommended to use it on the slower systems.

Appendix B

Extending Workcraft

This appendix explains how to build a Workcraft distribution from the source code and how to extend Workcraft with additional Interpreted Graph Model classes and tools.

B.1 Building Workcraft

Before building Workcraft from the source code, it is necessary to make sure that the Java Development Kit (JDK) is properly set up. This can be checked by trying to run the binary Workcraft distribution and the command-line Java tools: *java*, *javac*. If Workcraft does not work correctly or some of the Java tools are missing, the JDK should be reinstalled. The JDK implementations that are known to compile and run Workcraft correctly are the Sun Microsystems standard JDK [8] and OpenJDK [10]. Most of the development of the main code base is done using the Sun's JDK. When running with OpenJDK, Workcraft may have some small (but not fatal) issues.

Workcraft uses the Bazaar version control system [3] to manage the source code base and the Launchpad collaboration platform [9] to publish the development branches and to track issues. This chapter assumes that the reader is familiar with Bazaar and has it installed (Bazaar web site [3] contains very good documentation and tutorials). Workcraft web site [20] also contains a quick introduction to getting started with Bazaar.

B.1.1 Creating a code branch

This documentation is written against the version 2.0.1 of Workcraft. The following command will get this code branch:

```
bzr branch lp:~workcraft/workcraft/2.0.1
```

The following command will get the main development branch:

```
bzr branch lp:workcraft
```

Please note that this branch contains the latest development code and may be unstable. The plug-in API discussed in this chapter is also likely to change significantly over time. It is therefore recommended to use the version 2.0.1 to follow this document, or to refer to the Workcraft web site for the up-to-date documentation.

B.1.2 Building with Maven

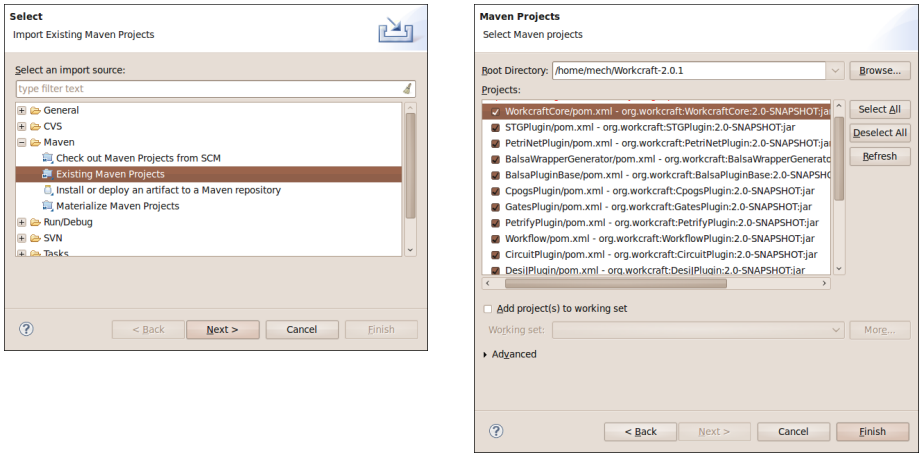
Workcraft uses Maven [1] as its build system — a Maven installation is therefore required to build Workcraft. With Maven properly installed and the path to its executable present in the PATH environment variable, Workcraft can be built using the following command (provided that the current directory is the root of the code branch):

```
mvn clean package
```

Note that Maven can take quite a long time to build Workcraft for the first time. This is because Maven depends heavily on plug-ins to perform the various build steps, and during the first build the set of the plug-ins that are required to build Workcraft (such as, e.g., the JavaCC parser generator) will be downloaded from the central Maven repository on the web. In addition to the plug-ins, Maven will need to download some of the supporting libraries (e.g., the DesiJ library) from the repository located on Workcraft's web site. Having a working Internet connection is therefore critical during the first build.

Maven will cache all the plug-ins and dependencies locally. This means that all the subsequent builds will be performed much faster and will no longer require Internet access.

The result of the Maven build will be the four distribution archives. The projects that contain the distributions have names starting with "WorkcraftDistr". Projects that have "Full" in their



(a) Selecting import source (Maven project) (b) Selecting projects

Figure B.1: Eclipse project import

name will include all the plug-ins available in the code branch, while projects that are called "Basic" will only include the STG and Petri net model support.

B.1.3 Building Workcraft using the Eclipse integrated development environment (IDE)

Using an IDE makes managing a large project such as Workcraft easier. The Maven build system is supported by most of the Java IDEs, natively (such as NetBeans) or through a plug-in (Eclipse). Bazaar version control system, however, is not supported well enough by some IDEs (Bazaar support status in various IDEs is listed in [2]). This is not critical because Bazaar has its own GUI interface implementations (QBzr, TortoiseBzr etc.) and its command-line interface is simple enough, however support for operations such as version control aware file renaming directly from an IDE is helpful.

The Eclipse IDE [16] provides good support for both Maven build system and Bazaar version control system and is recommended for Workcraft development. The rest of this section assumes that the user's Eclipse installation has the m2eclipse plug-in [6] installed for Maven integration. Installing the Bazaar integration plug-ins (BzrEclipse or QBzrEclipse) is optional.

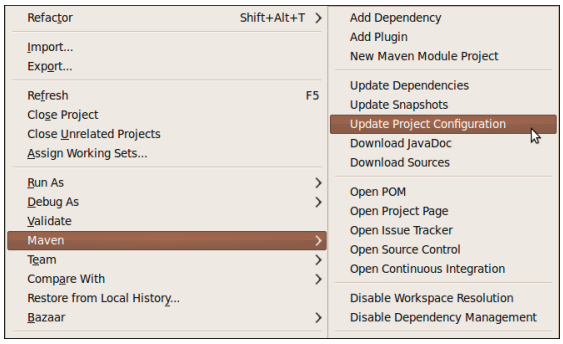


Figure B.2: Updating Eclipse project configuration

Importing the Workcraft projects into an Eclipse workspace

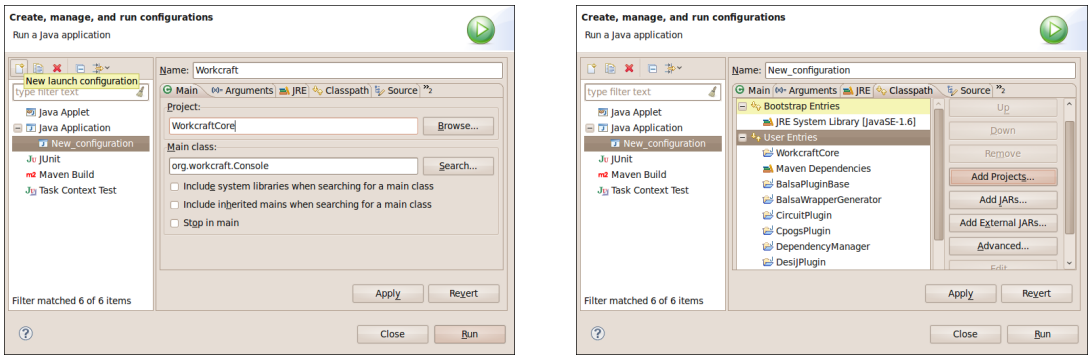
Workcraft projects can be imported to Eclipse directly from the code branch created by Bazaar using the Import window (Figure B.1a). The Workcraft source code package consists of a number of separate projects (Figure B.1b). If there are other projects already in the Eclipse workspace, all Workcraft projects can be added to a new working set ("Add projects to working set" option) for easier organisation, however it may be more convenient to create a new empty Eclipse workspace specifically for Workcraft.

Fixing the missing type errors

When the Workcraft Maven projects are imported into an Eclipse workspace and built for the first time, Eclipse will not be able to find some parsing related types and the red error icons will appear next to the projects that use those types (WorkcraftCore and STGPlugin). This is because some Java classes are generated by JavaCC from the grammar definition files, and they are created only when the first build completes. To fix this issue, Eclipse project configuration must be updated by Maven following the first build. This operation is available in the project context menu (Figure B.2) accessible by right-clicking on a project in the Package Explorer window. At this point all Workcraft projects should be able to be built without errors.

Creating a run configuration

To start Workcraft from Eclipse, it is necessary to create a run configuration for the project that contains the main executable class of Workcraft. The run configurations window is accessible



(a) Choosing the start-up project and the main class

(b) Configuring the classpath

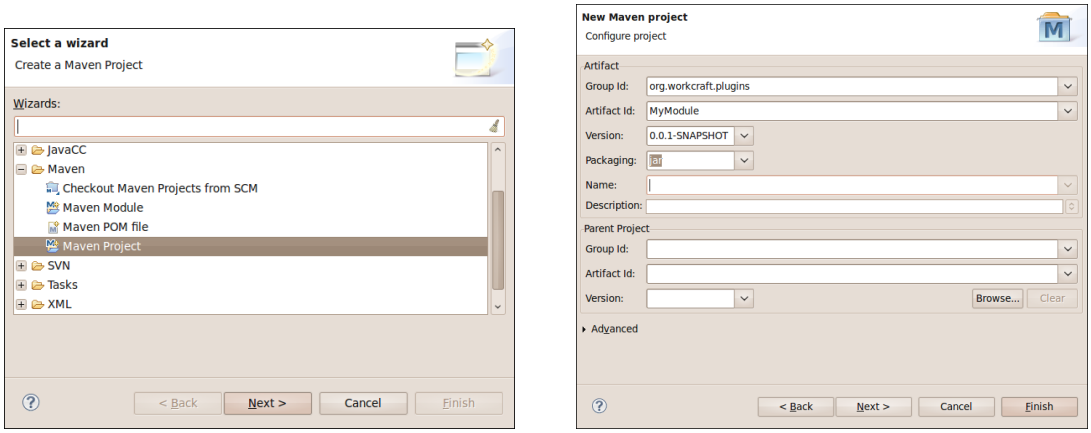
Figure B.3: The run configuration

from the main menu (*Run→Run configurations...*). A new run configuration can be created by double-clicking on "Java Application". As shown in Figure B.3a, the name of the start-up project is "WorkcraftCore" and the name of the main class is "org.workcraft.Console". All other projects (except the project called "Tests" that contains the unit tests and is not required at runtime) should be added into the classpath of the run configuration (Figure B.3b). This step is required because Workcraft searches its classpath to locate compatible plug-in classes, and most of Workcraft's functionality is contained in the plug-ins. At this point Workcraft can be started by clicking on the "Run" button in the run configurations window. For subsequent runs it is not necessary to use the run configuration window — a shortcut "Run" button (a green button with a white triangle) is available in the toolbar.

B.2 Creating a Workcraft module project in Eclipse

During the plug-in reconfiguration process, Workcraft scans the classpath to find all classes that implement the *org.workcraft.Module* interface. All discovered modules are initialised via their *init* method during the plug-in initialisation phase. Modules can use the *Framework* interface passed to this method to register individual plug-ins such as tools or models. A *Workcraft module* is therefore an organisational unit that represents a related collection of plug-ins that implement the extended functionality. Modules can be added and removed from the classpath to achieve a particular configuration of Workcraft (see Section A.1.3).

The module interface is defined as follows:



(a) Selecting the project type

(b) Configuring the Maven project

Figure B.4: Creating a new Maven project

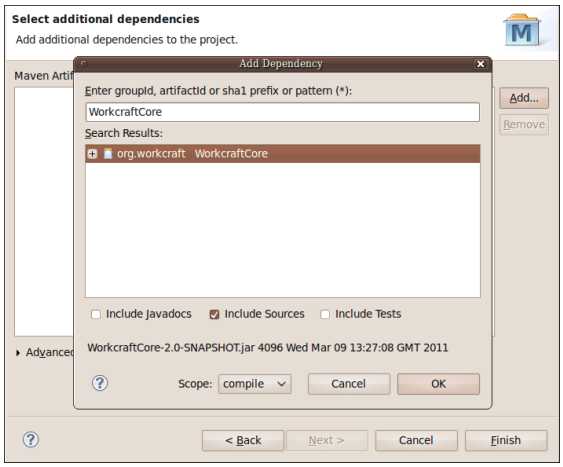


Figure B.5: Setting the project dependencies

```
public interface Module {

    public String getDescription();
    public void init(Framework framework);
}
```

The *getDescription* method returns a human-readable description of the module, and the *init* method is called to initialise the module as explained above.

B.2.1 Creating a new Maven project

Although it does not matter where the module is located as long as it is present on the classpath, it is recommended to create a separate project for each module for better organisation. A new Maven project can be created in Eclipse using the "New project" window, accessible from the main menu (*File→New project*) or using the keyboard shortcut (*Ctrl+N*). The type of the project should be "Maven project" as shown in Figure B.4a. The project should be created as a simple project ("Create a simple project (skip archetype selection)" option in the following configuration dialogue). The artifact details should be filled with the appropriate values of the Artifact ID and version, as shown in Figure B.4b).

In the final stage of the project configuration the project dependencies must be configured. A dependency on the "WorkcraftCore" project must be present in all projects (Figure B.5). Additional dependencies are optional.

When the project has been created, it is necessary to ensure that it uses the correct Java language version. This can be done in the project property window (right-click on the project name, then *Properties)*, section "Java compiler". The compiler compliance level should be set to 1.6 or higher. It is often enough to uncheck the "Use project specific settings" option — Eclipse will then use the compliance level corresponding to the currently installed JDK version.

B.2.2 Creating a Workcraft module

The rest of this section explains how to create a simple module that registers a tool. Because the tool will be applied to Signal Transition Graph (STG) models, dependencies on the Petri net and STG plug-ins must be set when the project is created. If the project has already been created

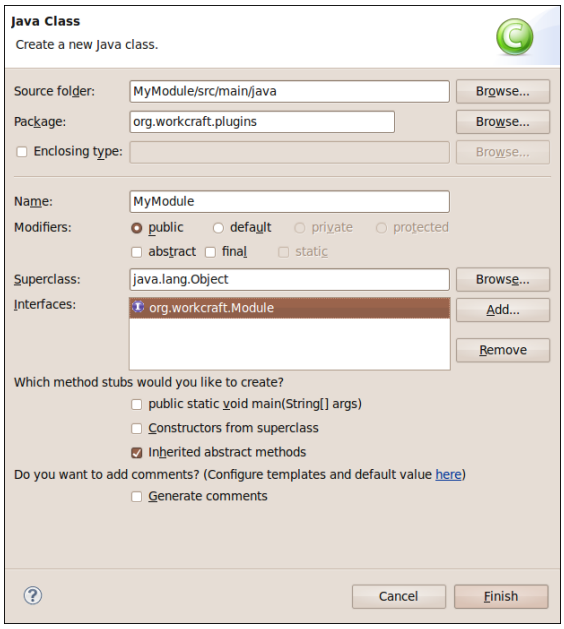


Figure B.6: Creating a Workcraft module class

without those dependencies, they can be added by right-licking the project name in the Package Explorer and selecting *Maven→Open POM* from the context menu. Current dependencies are listed in the "Dependencies" tab in the POM editor window, and more dependencies can be added using the "Add..." button.

A new class can be added to the project using the main menu (*File→New→Class*) or the project's context menu (*New→Class*). By convention, the package name of a module class must start with "org.workcraft.plugins", otherwise it will be ignored by Workcraft. The class must implement the *org.workcraft.Module* interface (Figure B.6). An example implementation of a module class is shown in Figure B.7.

As can be seen from the code, during its initialisation the module registers a tool class *NodeCounter*. The implementation of the tool itself is explained in the next section.

B.3 Adding tools

The tool classes can be created using the Eclipse UI in the same way as the module class, except that they must implement the *org.workcraft.Tool* interface instead of *org.workcraft.Module*. The implementation of the NodeCounter tool is shown in Figure B.8. The tool counts the different

```
package org.workcraft.plugins;

import org.workcraft.Framework;
import org.workcraft.Module;

public class MyModule implements Module {

    public String getDescription() {
        return "My module";
    }

    public void init(Framework framework) {
        framework.getPluginManager().registerTool(NodeCounter.class);
    }
}
```

Figure B.7: A basic module implementation

types of objects in an STG model: places, dummy transitions and signal transitions.

The method *isApplicableTo* defined in the *Tool* interface determines whether the tool is compatible with a given model. The NodeCounter tool accepts a model if its underlying mathematical model is an STG (and simply ignores the visual model if one is present). The method *getSection* is used to organise tools in the Tools menu (Figure A.4) by semantic categories. The *NodeCounter* tool tells the framework to put it in the "Statistics" category. The method *getDisplayName* defines the name of the tool that is shown to the user. Finally, the method *run* is called when the user chooses to apply the tool to the current model. Since it is guaranteed that the tool will not be passed a model that failed the *isApplicableTo* check, the tool can assume that the mathematical model that is passed to the method *run* is an STG and obtain a correspoding interface using a type-cast operation. The method then builds a string containing the result and displays it using the standard Swing message window.

B.3.1 Using asynchronous tasks

The *NodeCounter* tool is very simple: it does not perform any complex computations or calls to the external tools. In this case it is acceptable for the tool to perform all its work directly in the *run* method. If the tool potentially takes long to complete, however, this may be unacceptable. The tool's *run* method is called on the same thread as the GUI and hence the whole application is blocked until the method returns. Workcraft framework provides facilities for executing longer

tasks asynchronously. Using this functionality, the tool needs only to request the framework to queue the task and then immediately return from its *run* method. The framework will then automatically start the task on a separate thread, without affecting the rest of the application. When the task finishes, the framework will notify the progress monitor provided by the tool. The progress monitor can verify the result and react accordingly.

To be able to queue tasks, the tool needs to obtain a reference to the *TaskManager* interface provided by the framework. This is done by passing a reference to the required interface to the tool's constructor. As the tools are instantiated by the framework when needed, the constructor argument must be included to the *registerTool* call in the module's *init* method along with the tool class (Figure B.9).

An example asynchronous tool implementation is shown in Figure B.10. The tool registers as compatible with any model (because it only serves a demonstrative purpose) and in its *run* method it queues a new task instance. The class that defines the task to be executed is called *MyTask* and is shown in Figure B.11. The *Task* interface has a type parameter that defines the type of the value that the task is expected to produce upon successful completion. In the example the return type is *Nothing*, a special type meaning that no return value is actually expected. The tool also passes a progress monitor (*MyTaskProgressMonitor*) that is responsible for handling the task's progress updates and completion.

In its *run* method the task emulates a sequence of a hundred of computationally expensive steps. Each step takes a random amount of time to complete (emulated by the *Thread.sleep* call). In every step the task reports the progress to the task monitor that is passed as argument to the *run* method. The task also checks if the monitor reports the cancel request (usually initiated by the user via GUI).

The *run* method must return an object of type *Result*. This is a parametrised type that encapsulates the return value and the outcome of the task. It can be optionally constructed without an actual return value (e.g., if the calculations were terminated prematurely), but the outcome must always be specified. The outcome of the task may be one of the following: finished correctly, failed or cancelled. In the *MyTask* example, the task does not need to return any value so it only returns the outcome.

The implementation of the progress monitor used to handle the completion of *MyTask* is shown in Figure B.12. The method *finished* of the progress monitor is called when the queued task completes (i.e., returns from its *run* method). In this example, the progress monitor simply checks if the task has successfully completed and, if that is the case, shows a simple message window. Note that it uses the *SwingUtilities.invokeLater* method to execute the GUI code (as opposed to the simple tool example shown in Figure B.8). This is because the progress monitor code is executed on a separate thread to avoid blocking the GUI and all the GUI-related code must be executed on the Swing event dispatch thread. The *invokeLater* method will call the code passed to it on the event dispatch thread at the first opportunity.

The progress of the task can be observed using the Tasks window (Figure A.11). Tasks can be terminated using the "Cancel" button in a particular task's box in the Tasks window.

B.3.2 Interfacing with external tools

Workcraft provides several convenience classes for interfacing with external tools. The class *ExternalProcessTask* allows starting external processes and provides support for operations such as stopping the process and reading its standard output. To create a Workcraft tool that relies on an external command-line based tool, a small modification can be made to the *AsyncTool* example (Figure B.10). Instead of queueing a custom task, an instance of *ExternalToolTask* should be queued as follows:

```
taskManager.queue(new ExternalProcessTask(new String[] {"echo", "Hello␣world!"}, "."),
    "External␣tool␣test", new ExternalTaskProgressMonitor());
```

This will queue a task that starts the tool "echo" with the parameter "Hello world!". "echo" is the standard tool in most operating systems that simply repeats whatever line was passed to it as an argument to its standard output. The completion of the *ExternalProcessTask* is handled in the same fashion as that of any other task, except that the task result type is fixed to *ExternalTaskResult.* A progress monitor implementation called *ExternalTaskProgressMonitor* can be produced by modifying the *MyTaskProgressMonitor* slightly: the type parameter should be changed from *Nothing* to *ExternalProcessResult,* and the line that generates the message window should be changed as follows:

```
JOptionPane.showMessageDialog(null, new String(result.getReturnValue().getOutput()));
```

This will construct a string from the standard output of the external process, which is in this case “Hello world!”.

B.4 Adding models

Adding support for new model types in Workcraft is very similar to adding new tools for existing models, although the classes that define a model may seem more complex than the tool classes. A model is added to Workcraft by implementing the *ModelDescriptor* interface and registering it with the plug-in manager during the module initialisation:

```
framework.getPluginManager().registerModel(MyModelDescriptor.class);
```

An example *ModelDescriptor* implementation is shown in Figure B.13. This interface exposes three operations to the Workcraft framework. The *getDisplayName* method returns a human-readable name of the model that will be displayed, e.g., in the model creation dialogue. The *createMathModel* method generates a new instance of the mathematical model. The *getVisualModelDescriptor* optionally returns a visual model descriptor that implements additional operations that define the visual model. If this method returns *null*, the framework assumes that the model does not support visual editing and will not be able to create editor windows.

An example *VisualModelDescriptor* implementation is shown in Figure A.2. It consists of only two methods: the *createVisualModel* method returns a new visual model instance given a reference to the mathematical model which it should represent. The type of the mathematical model that will be passed to this method is guaranteed to be the same as the type returned by the *createMathModel* method in the corresponding model descriptor. The second method, *createTools*, defines the set of graph editor tools that will be used to interact with the visual model. In the example, the visual model descriptor defines two standard tools: the selection tool and the connection tool (these tools are described in Section A.3.3).

To return the new instances of the mathematical and the visual models, the model descriptors create new instances of the *MyModel* and *MyVisualModel* classes correspondingly. These classes implement the model logic and store the node graph. Workcraft provides two helper classes,

AbstractMathModel and *AbstractVisualModel* that implement the general functionality of an Interpreted Graph Model. To produce a working instance of *MyModel* and *MyVisualModel*, it is enough to declare these classes as extending the *AbstractMathModel* and *AbstractVisualModel* respectively and leave the methods such as *connect* and *validateConnection* empty. At this point, Workcraft will be able to create models of the type "My model" through the new model dialogue and to create editor windows for these models. The models will be empty at this time, however, because no node types have yet been defined.

B.4.1 Adding a node type

To add a new node type to the model, two steps are required. First, the new node class should be implemented both for the mathematical and the visual model. The implementation of a node in the mathematical model is trivial (it may even be empty, but in this example we assume that it has an integer field called *myProperty*), but the implementation of a node in the visual model is more complicated. To help with this task Workcraft provides the base class called *VisualComponent* that implements most of the logic required for a visual graph node. The only methods that have to be implemented by the user are *draw* that produces the graphical representation of the node, *hitTestInLocalSpace* that tests whether a given point is inside the node's visible shape (this method is used to check whether a mouse pointer is inside the node), *getBoundingBoxInLocalSpace* that returns a rough approximation of the node's shape in the form of an axis-aligned rectangle (the bounding box is used during the first pass of mouse pointer/node hit detection to quickly reject most of the nodes before calling the potentially expensive *hitTestInLocalSpace*) and *getMathReferences* that returns a list of all nodes in the mathematical model that the visual node refers to.

Figure B.15 shows an example implementation of a visual node, the class called *MyVisualNode*.

The second step is to add a graph editor tool that will allow the nodes of the new type to be created. Workcraft provides a generalised implementation for this class of tools called *NodeGeneratorTool*. To create an instance of a *NodeGeneratorTool*, an implementation of the *NodeGenerator* interface for the node type is required. An example implementation is shown in Figure B.16. The method *getIcon* returns an icon that will be drawn on the graph editor tool button. In the example,

the icon is created from an SVG file located in the project's "resources" directory. The method *getLabel* returns the text that will be displayed in the button's tool-tip. The method *getText* returns the text that will be shown in the graph editor window when the tool is activated. The method *getHotKeyCode* returns the code of the key that is used to quickly activate the tool using the keyboard (in this case, it is the "N" key). The method *generate* is responsible for the instantiation of a new node at the given location. In the example, the node generator delegates this task to the model. The method *createNode* in the type *MyVisualModel* is defined as follows:

```
public void createNode (Point2D position) {
    // create a new backing node in the math. model
    MyNode node = new MyNode();
    mathModel.add(node);

    // create the visual node corresponding to the math. node
    MyVisualNode node = new MyVisualNode(node);
    node.setPosition(position);
    // add the node to the graph
    add(node);
}
```

Finally, the tool is added to the list of tools returned by *MyVisualModelDescriptor*:

```
tools.add(new NodeGeneratorTool(new MyNodeGenerator()));
```

B.4.2 Implementing the connection methods

The two methods that are used by the graph editor to create arcs connecting the nodes in the model are the *connect* and *validateConnection* methods in the type *MyVisualModel*. The *validateConnection* method is called when the user has selected a node in the connection mode and hovers the mouse cursor above some other node. This method should do nothing if the connection between these nodes is allowed, and throw an *InvalidConnectionException* otherwise. An example implementation is as follows:

```
@Override
public void validateConnection(Node first, Node second) throws InvalidConnectionException
    {
    if (!(first instanceof MyVisualNode && second instanceof MyVisualNode))
        throw new InvalidConnectionException ("Unexpected node types");
    for (Connection con : getConnections(first))
            if (con.getSecond() == second)
                throw new InvalidConnectionException ("Arc already exists");
}
```

This implementation first checks that both nodes are of the correct type (*MyVisualNode*) and then ensures that an arc between the nodes does not yet exist.

The method that is called to create a connection between the two nodes (guaranteed to have passed the *validateConnection* check) is called *connect*. This method should create a connection between the two visual nodes and the corresponding connection in the mathematical model. To implement it, a method in the mathematical model (*MyModel*) that would create a connection between the mathematical nodes is required:

```
public MathConnection connect (MathNode first, MathNode second) {
    MathConnection result = new MathConnection(first, second);
    add (result);
    return result;
}
```

Using this method, the *connect* method in the visual model (*MyVisualModel*) can be implemented as follows:

```
@Override public void connect(Node first, Node second) throws InvalidConnectionException
    {
    MyVisualNode firstVisualNode = (MyVisualNode)first;
    MyVisualNode secondVisualNode = (MyVisualNode)second;
```

```
    MathConnection con = mathModel.connect(firstVisualNode.getReferencedNode(),
        secondVisualNode.getReferencedNode());
    VisualConnection vcon = new VisualConnection(con, firstVisualNode, secondVisualNode
        );
    add(vcon);
}
```

This implementation uses the default connection classes provided by Workcraft (*MathConnection* and *VisualConnection*) that model a directed arc.

B.4.3 Defining editable properties

Workcraft provides a user-friendly property editor interface (Figure A.4, item 3) that allows changing the values of the node properties such as, e.g., the number of tokens in a Petri net place. To determine what properties should be displayed in the property editor, Workcraft requests a list of property descriptors from the model implementation via the *getProperties* method. The default implementation of this method provided by the *AbstractVisualModel* and *AbstractMathModel* types is empty, so it needs to be overridden in the following way (in the type *MyModel*):

```
@Override
public Properties getProperties(Node node) {
    if (node instanceof MyNode) {
        return Properties.Set.of(new MyPropertyDescriptor((MyNode)node));
    }
    return Properties.Set.empty();
}
```

With this implementation, the model checks the type of the node whose properties are being requested. *MyModel* defines only one type of node: *MyNode*, and therefore it returns an empty property descriptor list if the node is of any other type. If the node is of the type *MyNode*, the model adds an implementation of the *PropertyDescriptor* interface (*MyPropertyDescriptor*, shown in Figure B.17) that defines how the property should be presented to the user. The methods are mostly self-explanatory. The *setValue* and *getValue* methods handle the exchange of the property values

between the node and the GUI controls that are used to display and edit them. The *getChoice* method allows defining a set of values that the user will be able to choose from instead of being allowed to edit the value directly. The *getType* returns the type of the property that is used to determine how the property value is displayed and edited.

To determine the complete set of properties displayed by the property editor, Workcraft uses the following algorithm. Given a visual node, it first requests a list of properties defined for this type of node from the visual model. Then, for each mathematical node from the list of nodes referred to by the visual node, Workcraft requests its list of properties from the mathematical model. All those property lists are finally merged.

To keep the UI up-to-date, the framework must be notified when the property is changed. This is done in the property setter method as follows:

```
public void setMyProperty(int myProperty) {
    this.myProperty = myProperty;
    sendNotification(new PropertyChangedEvent(this, "myProperty"));
}
```

B.4.4 Using the automatic serialisation

Workcraft provides an automatic serialisation facility for all models. The example model defined in this section will be able to be serialised (i.e., saved to disk) without any additional code. To support deserialisation (loading from the files on disk), however, models must define a special constructor that accepts the graph data loaded from the disk. Workcraft's *AbstractMathModel* and *AbstractVisualModel* classes implement those constructors, and it is enough to call those constructors from the *MyModel* and *MyVisualModel* types to implement the deserialisation support:

```
// for the type MyModel
public MyModel(MathGroup root) {
    super(root);
}

...
```

```
//for the type MyVisualModel
public MyVisualModel (MyModel model, VisualGroup root) {
    super(model, root);
    this.mathModel = model;
}
```

At this point, Workcraft will be able to save and load the models of type “My model”.

```
package org.workcraft.plugins;

import javax.swing.JOptionPane;

import org.workcraft.Tool;
import org.workcraft.plugins.stg.STGModel;
import org.workcraft.workspace.WorkspaceEntry;

public class NodeCounter implements Tool {

    public boolean isApplicableTo(WorkspaceEntry we) {
        if (we.getModelEntry().getMathModel() instanceof STGModel)
            return true;
        else
            return false;
    }

    public String getSection() {
        return "Statistics";
    }

    public String getDisplayName() {
        return "Count␣nodes";
    }

    public void run(WorkspaceEntry we) {
        STGModel stg = (STGModel)we.getModelEntry().getMathModel();

        StringBuilder result = new StringBuilder();

        result.append("STG␣statistics:\n");
        result.append("Number␣of␣places:␣" + stg.getPlaces().size() + "\n");
        result.append("Number␣of␣signal␣transitions:␣" + stg.getSignalTransitions().size() + "\n");
        result.append("Number␣of␣dummy␣transitions:␣" + stg.getDummyTransitions().size() + "\n");

        JOptionPane.showMessageDialog(null, result.toString());
    }
}
```

Figure B.8: A simple tool implementation

```
package org.workcraft.plugins;

import org.workcraft.Framework;
import org.workcraft.Module;

public class MyModule implements Module {

    public String getDescription() {
        return "My module";
    }

    public void init(Framework framework) {
        framework.getPluginManager().registerTool(NodeCounter.class);
        // the task manager will be passed to the AsyncTool's contstructor
        // whenever an instance of this tool is created
        framework.getPluginManager().registerTool(AsyncTool.class, framework.getTaskManager());
    }
}
```

Figure B.9: Registering a tool with a constructor parameter

```
package org.workcraft.plugins;

import org.workcraft.Tool;
import org.workcraft.tasks.TaskManager;
import org.workcraft.workspace.WorkspaceEntry;

public class AsyncTool implements Tool {

    private final TaskManager taskManager;

    public AsyncTool(TaskManager taskManager) {
        this.taskManager = taskManager;
    }

    @Override
    public boolean isApplicableTo(WorkspaceEntry we) {
        return true;
    }

    @Override
    public String getSection() {
        return "General";
    }

    @Override
    public String getDisplayName() {
        return "Asyncrhonous tool test";
    }

    @Override
    public void run(WorkspaceEntry we) {
        taskManager.queue(new MyTask(), "Testing my task", new MyTaskProgressMonitor());
    }
}
```

Figure B.10: A tool using the asynchronous tasks functionality

```
package org.workcraft.plugins;

import org.workcraft.Nothing;
import org.workcraft.tasks.ProgressMonitor;
import org.workcraft.tasks.Result;
import org.workcraft.tasks.Result.Outcome;
import org.workcraft.tasks.Task;

public class MyTask implements Task<Nothing> {

    @Override
    public Result<Nothing> run(ProgressMonitor<? super Nothing> monitor) {
        for (int i=0; i < 100; i++) {
            try {
                if (monitor.isCancelRequested()) {
                    return new Result<Nothing>(Outcome.CANCELLED);
                }

                // emulate a long calculation step
                Thread.sleep((int)(Math.random()*100+20));
            } catch (InterruptedException e) {
                return new Result<Nothing>(Outcome.FAILED);
            }
            monitor.progressUpdate(i/99.0);
        }
        return new Result<Nothing>(Outcome.FINISHED);
    }
}
```

Figure B.11: An asynchronous task implementation

```
package org.workcraft.plugins;

import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;

import org.workcraft.Nothing;
import org.workcraft.tasks.DummyProgressMonitor;
import org.workcraft.tasks.Result;
import org.workcraft.tasks.Result.Outcome;

public class MyTaskProgressMonitor extends DummyProgressMonitor<Nothing> {

    @Override
    public void finished(Result<? extends Nothing> result, final String description) {
        if (result.getOutcome() == Outcome.FINISHED )
        {
            SwingUtilities.invokeLater(new Runnable() {
                @Override
                public void run() {
                    JOptionPane.showMessageDialog(null, "Task " + description + " finished!");
                }
            });
        }
    }
}
```

Figure B.12: A progress monitor implementation

```
package org.workcraft.plugins;

import org.workcraft.dom.ModelDescriptor;
import org.workcraft.dom.VisualModelDescriptor;
import org.workcraft.dom.math.MathModel;

public class MyModelDescriptor implements ModelDescriptor {

    @Override
    public String getDisplayName() {
        return "My model";
    }

    @Override
    public MathModel createMathModel() {
        return new MyModel();
    }

    @Override
    public VisualModelDescriptor getVisualModelDescriptor() {
        return new MyVisualModelDescriptor();
    }
}
```

Figure B.13: An example *ModelDescriptor* implementation

```
package org.workcraft.plugins;

import java.util.LinkedList;
import java.util.List;

import org.workcraft.dom.VisualModelDescriptor;
import org.workcraft.dom.math.MathModel;
import org.workcraft.dom.visual.VisualModel;
import org.workcraft.exceptions.VisualModelInstantiationException;
import org.workcraft.gui.graph.tools.ConnectionTool;
import org.workcraft.gui.graph.tools.GraphEditorTool;
import org.workcraft.gui.graph.tools.SelectionTool;

public class MyVisualModelDescriptor implements VisualModelDescriptor {

    @Override
    public VisualModel create(MathModel mathModel)
            throws VisualModelInstantiationException {
        return new MyVisualModel((MyModel)mathModel);
    }

    @Override
    public Iterable<GraphEditorTool> createTools() {
        List<GraphEditorTool> tools = new LinkedList<GraphEditorTool>();

        tools.add(new SelectionTool());
        tools.add(new ConnectionTool());

        return tools;
    }
}
```

Figure B.14: An example *VisualModelDescriptor* implementation

```
package org.workcraft.plugins;

/* imports omitted */

public class MyVisualNode extends VisualComponent {
    final Path2D shape;
    final MyNode node;

    public MyVisualNode(MyNode node) {
        // the backing node in the math. model
        this.node = node;

        // a simple diamond shape
        shape = new Path2D.Float();
        shape.moveTo(-0.5, 0);
        shape.lineTo(0.0, 1.0);
        shape.lineTo(0.5, 0);
        shape.lineTo(0, -1);
        shape.closePath();
    }

    @Override
    public void draw(DrawRequest r) {
        Graphics2D graphics = r.getGraphics();

        // draw a filled shape first
        graphics.setColor(Coloriser.colorise(Color.LIGHT_GRAY, r.getDecoration().getColorisation()));
        graphics.fill(shape);

        // now draw an outline
        graphics.setStroke(new BasicStroke(0.1f));
        graphics.setColor(Coloriser.colorise(Color.BLACK, r.getDecoration().getColorisation()));
        graphics.draw(shape);

        // draw the value of "my property"
        String text = "" + node.getMyProperty();
        Font font = new Font("Sans-serif", Font.PLAIN, 1);
        graphics.setFont(font);

        // calculate the text bounds to center the text on the node
        Rectangle2D stringBounds = font.getStringBounds(text, graphics.getFontRenderContext());
        graphics.drawString(text, (float)(-0.5*stringBounds.getWidth()), (float)(-0.5f*-stringBounds.getHeight()));
    }

    @Override
    public boolean hitTestInLocalSpace(Point2D pointInLocalSpace) {
        return shape.contains(pointInLocalSpace);
    }

    @Override
    public Rectangle2D getBoundingBoxInLocalSpace() {
        return shape.getBounds2D();
    }

    @Override
    public Collection<? extends MathNode> getMathReferences() {
        return Collections.singletonList(node);
    }
}
```

Figure B.15: A visual node implementation

```
class MyNodeGenerator implements NodeGenerator {
    @Override
    public Icon getIcon() {
        return GUI.createIconFromSVG("mynode.svg");
    }

    @Override
    public String getLabel() {
        return "Create␣my␣node";
    }

    @Override
    public String getText() {
        return "Click␣to␣create␣a␣node";
    }

    @Override
    public void generate(VisualModel model, Point2D where)
            throws NodeCreationException {
        ((MyVisualModel)model).createNode(where);
    }

    @Override
    public int getHotKeyCode() {
        return KeyEvent.VK_N;
    }
}
```

Figure B.16: An example *NodeGenerator* implementation

```
package org.workcraft.plugins;

import java.lang.reflect.InvocationTargetException;
import java.util.Map;

import org.workcraft.gui.propertyeditor.PropertyDescriptor;

public class MyPropertyDescriptor implements PropertyDescriptor {
    private final MyNode node;

    public MyPropertyDescriptor(MyNode node)
    {
        this.node = node;
    }

    @Override
    public boolean isWritable() {
        return true;
    }

    @Override
    public Object getValue() throws InvocationTargetException {
        return node.getMyProperty();
    }

    @Override
    public void setValue(Object value) throws InvocationTargetException {
        node.setMyProperty((Integer)value);
    }

    @Override
    public Map<Object, String> getChoice() {
        return null;
    }

    @Override
    public String getName() {
        return "My property";
    }

    @Override
    public Class<?> getType() {
        return int.class;
    }
}
```

Figure B.17: An example *PropertyDescriptor* implementation

Appendix C

Working with Signal Transition Graphs

A set of plug-ins that together provide a rich environment for system design based on the Signal Transition Graph (STG) model is included with the standard Workcraft distribution. Besides providing the support for visual entry and simulation of Signal Transition Graphs, these plug-ins implement a number of advanced operations such as verification, encoding conflict resolution, logic synthesis and other.

This appendix documents how to use Workcraft to design STG models and how to apply tools to these models. Before reading this chapter, please see the Appendix A for the general overview of the user interface of Workcraft.

C.1 Using the STG editor interface

The functionality of the visual STG editor is provided by the set of editor tools shown in Figure C.1. A particular tool is activated either by clicking on its icon in the “Editor tools” window or by pressing the corresponding hotkey on the keyboard. Once activated, the tool handles all

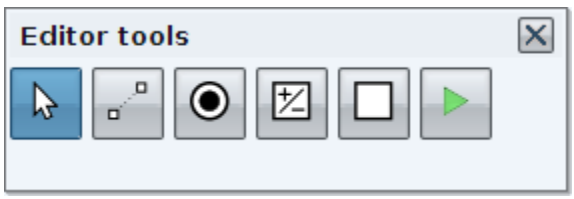


Figure C.1: The STG editor tools

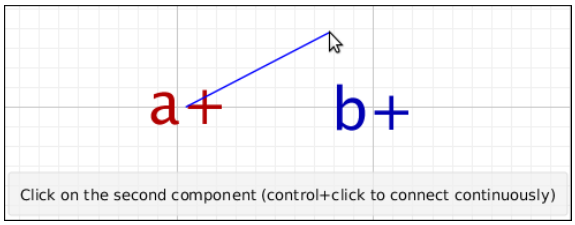


Figure C.2: Creating a connection

user input to the editor window (e.g., the mouse clicks and key presses) to implement a certain operation. The functionality of the tools for editing the STG model is explained below.

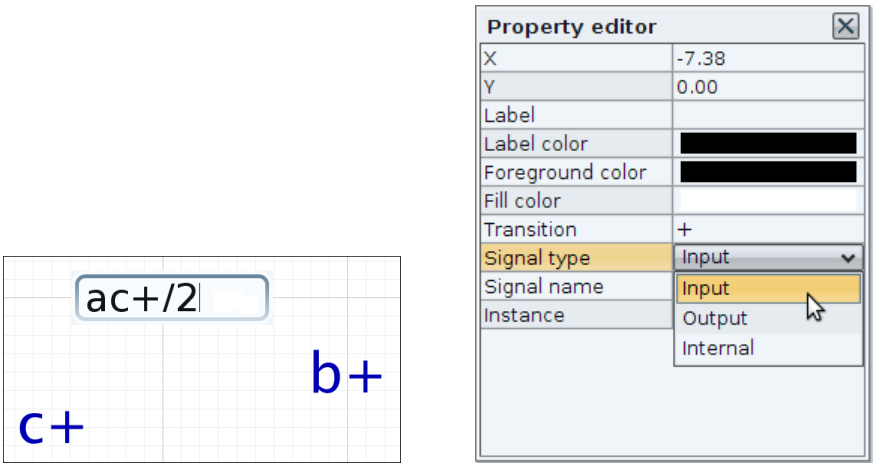
C.1.1 STG editor tools

Selection tool (hotkey: *S*) This tool is used to select, delete, move and group nodes. Single nodes are selected by clicking on them. Multiple nodes are selected by clicking on an empty space, holding the mouse button and dragging the cursor to draw a selection box. Selected nodes may be deleted by pressing the *Delete* key on the keyboard. Nodes are moved by clicking on a selected node and holding the left mouse button while moving the cursor.

Selected nodes can be grouped together by pressing *Ctrl+G*. Grouped nodes are treated as a single node for the purpose of selection and transform operations. A group of nodes can be broken apart by selecting it and pressing *Ctrl+U*.

If a single node is selected, the property editor window (Figure A.4, item 3) will show the list of properties defined for that node. Properties can be changed by clicking on their corresponding values in the property editor window. The method of specifying the value depends on the type of the property. For instance, numerical properties can be changed by simply typing in the new value, but for the colour properties a special colour chooser window is used.

Connection tool (hotkey: *C*) The connection tool is used to create directed arcs. When this tool is active, two nodes can be connected by clicking on the first node and then clicking on the second node. During the choosing of the second node, a visual connection hint line is displayed (Figure C.2). The colour of this line depends on what is located under the cursor at the moment. If there is no node, the line is blue. If there is a node that can be connected with the first node, the line is green. If there is a node under the cursor, but the types of node are such that the connection



(a) In-place signal transition editing

(b) Signal transition properties

Figure C.3: Editing signal transitions

between them is invalid (e.g., two places) the line is red and the reason why such a connection cannot be created is displayed in the bottom part of the editor window.

Arcs created using the connection tool can be removed using the selection tool.

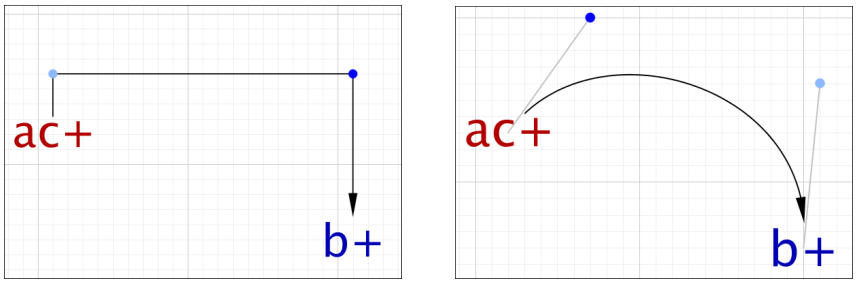
Place tool (hotkey: *P*) This tool is used to create places. When this tool is active, a new place will be created under the mouse cursor when the user clicks anywhere in the editor window.

Signal transition and Dummy transition tools (hotkey: *T*) These tools are used similarly to the Place tool to create transitions. Both tools share the same hotkey and it may be pressed repeatedly to cycle between them.

Simulation tool (hotkey: *M*) This tool activates the simulation mode. Detailed explanation of the simulation functionality is given in Section C.2.

C.1.2 Assigning signal names and types

When a new signal transition is created, the signal name and direction can be assigned to it in two ways. One way is to use in-place editing feature that is activated by double-clicking on a signal transition in the editor window. A text box will appear with the current name and direction of the signal transition (Figure C.3a). The new name and/or direction can be typed directly into this text box (pressing *Enter* accepts the change). The second way is to use the property editor



(a) Polyline (b) Bezier curve

Figure C.4: Arcs drawn using different shapes

window (Figure C.3b). The type of the signal (input, output or internal) can also be changed using the property editor.

C.1.3 Placing tokens

The number of tokens in a place (or in an implicit place) can be set using the property editor window. For implicit places, the properties of the arc that holds them contain the corresponding property.

There is also a shortcut to place and remove single tokens (places in an STG will most often contain at most one token). This is done by double-clicking on a place or an arc with an implicit place.

C.1.4 Changing arc shapes

When new arcs are created, they have a simple straight line shape. Sometimes it is useful to give some arcs a more complex shape. Workcraft supports two modes of controlling the arc shapes: polylines and Bezier curves. Polyline is the default mode, and the arc shape in this mode is controlled by a series of anchor points (Figure C.4a). The graphical representation of the arc is constructed from the straight line segments connecting the anchor points. In the Bezier mode, the arc is drawn using a cubic Bezier curve. The shape of the curve is controlled by the two "handles" as shown in Figure C.4b.

The shape editing mode can be selected in the property editor. The anchor points can be edited using the Selection tool. When an arc is selected, its anchor points (or the handles of the Bezier curve) are shown. They can be moved or deleted in the same way as moving nodes. In the Polyline

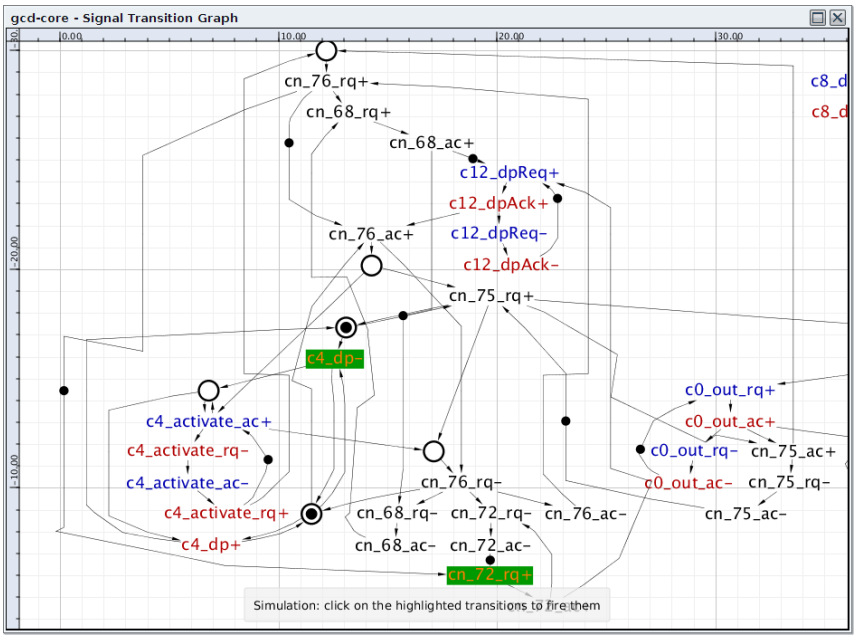


Figure C.5: Editor window in simulation mode

mode, additional control points are created by holding *Ctrl* and clicking on the line segment.

C.2 Simulation

Simulation mode is activated using the Simulation tool. In this mode, the editor window highlights the currently enabled transitions (Figure C.5). An enabled transition can be fired simply by clicking on it. The Simulation tool control window (Figure C.6) maintains the history of transition firing events. It is possible to restore the state of the STG to any point in history by double-clicking

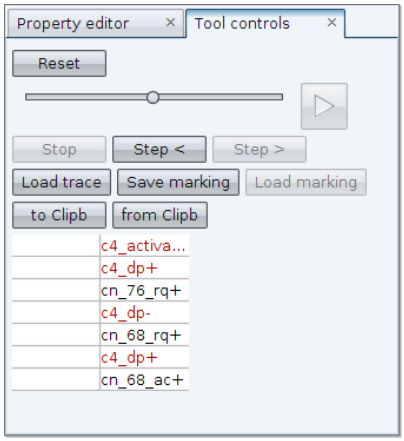


Figure C.6: Simulation tool controls

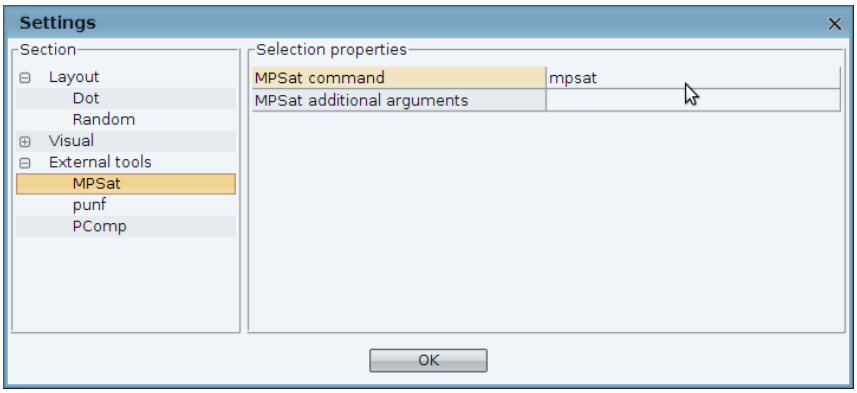


Figure C.7: The settings window

on the transition name.

The “Reset” button resets the marking to the initial state (i.e., the marking that the STG had when the Simulation tool was activated). The “Step back” and “Step forward” buttons allow to move through a trace (or the simulation history) one event at a time. “Load trace”, “Save trace”, “Load marking” and “Save marking” buttons are self-explanatory: they allow managing files storing the traces and markings. The buttons “To clipboard” and “From clipboard” allow correspondingly saving and restoring the trace in the form of a comma-separated list of signal transition to and from the system’s clipboard.

C.3 Using tools

The STG model implementation in Workcraft uses several external tools to provide support for a number of advanced operations. These tools must be accessible to Workcraft for those operations to work correctly. The commands used to start the external tools can be configured using the “External tools” section in the Settings window (Figure C.7). This window can be brought up using the main menu (*Edit→Preferences*).

All tools are accessible from the “Tools” sub-menu of the main menu (Figure A.4, item 1).

C.3.1 Visual layout

Tools→Layout

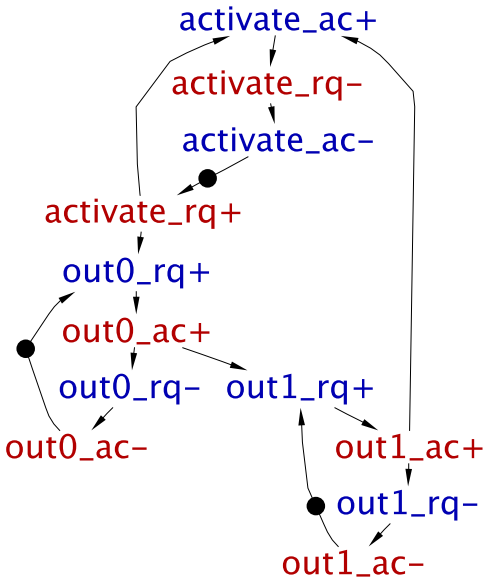
Workcraft can use the Dot tool [7] to automatically produce the graphical layout for models

```
.outputs activate_ac out0_rq out1_rq
.inputs activate_rq out0_ac out1_ac

.graph
activate_ac+ activate_rq-
activate_ac- activate_rq+
activate_rq+ activate_ac+ out0_rq+
activate_rq- activate_ac-
out0_ac+ out0_rq- out1_rq+
out0_ac- out0_rq+
out0_rq+ out0_ac+
out0_rq- out0_ac-
out1_ac+ activate_ac+ out1_rq-
out1_ac- out1_rq+
out1_rq+ out1_ac+
out1_rq- out1_ac-

.marking { <activate_ac-,activate_rq+>
  <out0_ac-,out0_rq+> <out1_ac-,out1_rq+>
}
.end
```

(a) Source STG file



(b) Generated graphical representation

Figure C.8: Automatic STG layout generation using Dot

that lack one. Signal Transition Graph models are usually stored in the .g file format that does not contain any information about the arrangement of the nodes in the graphical STG representation. Workcraft will apply the Dot-based layout tool automatically when a non-visual model is attempted to be edited using the visual editor, for example when a .g file is imported and opened in the editor.

The main advantage of using Workcraft to work with visual representation of the STG models is that the layout information obtained from Dot is used only to initialise the visual model. The user can use the automatically produced layout as something to start with, and then modify parts of the layout manually. This contrasts with tools such as *draw_astg* in the Petrify package, that also use Dot to calculate the layout but can only produce static images of the graph.

By default, Workcraft does not import the complex arc shapes produced by Dot and treats all arcs as straight lines instead. This behaviour can be changed by setting the "Import connection shapes" option in the Settings window (section *Layout→Dot*).

C.3.2 Parallel composition

Tools→Composition→Parallel composition

Parallel composition [125] is an operation that builds a composite STG from a set of input

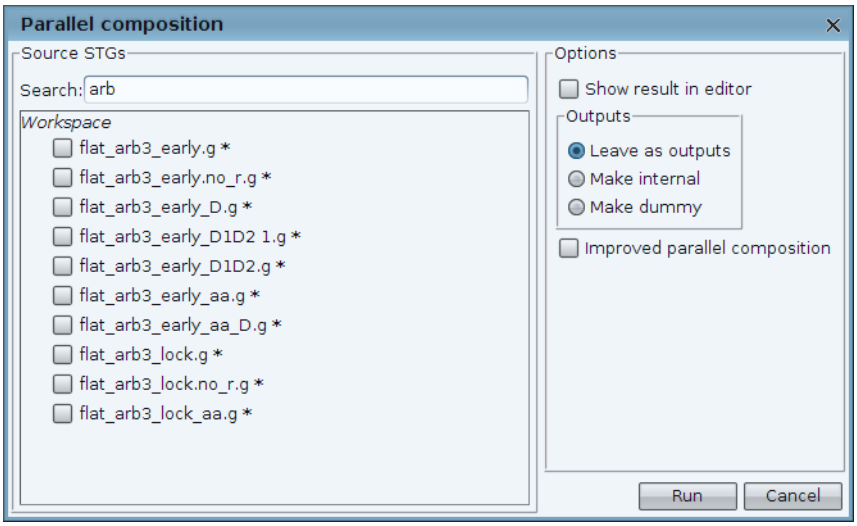


Figure C.9: STG selection for parallel composition

STGs by merging the signal transitions having the same label. This operation is used, e.g., to construct a closed system from a circuit Petri net and the environment specification (see Section 4.3.1). Workcraft can perform the parallel composition of an arbitrary input set of STGs using the PComp tool [12]. The Parallel composition window (Figure C.9) allows choosing the set of input STGs from the Workspace. The source of the STGs can be both Workcraft STG models or .g files present in the Workspace. The "Search" text box allows to filter the list of displayed STG sources by entering the partial name.

C.3.3 Decomposition

Tools→Decomposition

The decomposition operation [125] splits an STG into several components. STG decomposition is particularly useful for synthesis of large circuits, where synthesising the whole circuit at once is computationally infeasible. Synthesising a set of smaller circuits is significantly easier, and it is often the case that their composition gives a better implementation that one large circuit obtained from the original STG [125]. Decomposition is also useful to detach library elements (such as arbiters) to avoid the expensive synthesis process for the circuits that already have a well-known implementation.

Workcraft supports STG decomposition using the DesiJ tool [99]. DesiJ is tightly integrated

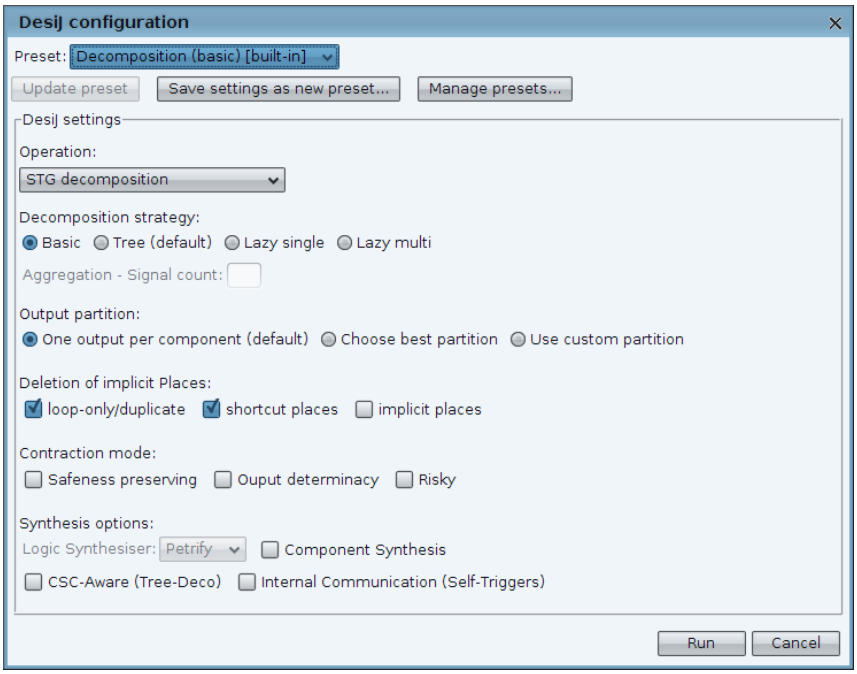


Figure C.10: DesiJ configuration window

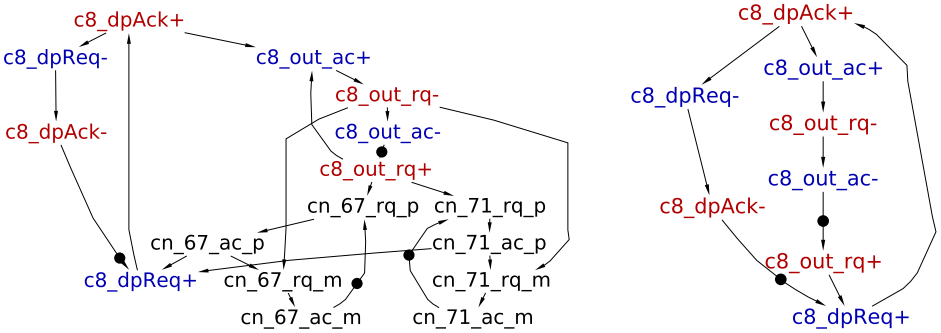
with Workcraft: it is used as an internal library and not a stand-alone tool. DesiJ can be run using the default parameters (*Tools→Decomposition→Standard decomposition*) or with a customised set of parameters (*Tools→Decomposition→Customised function*).

The parameters in the DesiJ configuration window (Figure C.10) mirror the command-line arguments of the stand-alone version of DesiJ. A detailed explanation of those parameters is given in [100]. The configuration window allows saving the current values of the parameters in a named *preset*. The presets are persistent across program runs — they are stored in the configuration directory of Workcraft.

C.3.4 Dummy contraction

Tools→Dummy contraction

Dummy transitions is a special class of transitions defined in the Signal Transition Graph model. These transitions do not reflect any physical events in the modelled system and are used as a design aid. The dummy contraction operation attempts to remove the dummy transitions from the model while preserving the behaviour of the signal transitions, as shown in Figure C.11. Subfigure C.11a is the original STG where some of the transitions are dummy transitions. Sub-



(a) Original STG with dummy transitions

(b) STG with dummy transitions removed

Figure C.11: Dummy contraction example

figure C.11b is an STG that has the same observed behaviour, but contains no dummies. In this example dummy contraction was performed by Petrify (*Tools→Dummy contraction→Contract dummies (Petrify)*). Petrify uses state space exploration techniques to produce the STG without dummy transitions, which often results in very good solutions but may be very slow for larger STGs. Workcraft supports an alternative (structural) dummy contraction method using DesiJ (*Tools→Dummy contraction→Contract dummies (DesiJ)*). This method does not suffer from the state space explosion problem, however it cannot guarantee that all dummy transitions will be removed.

When a dummy contraction tool is applied, the resulting STG will appear in the Workspace window alongside the original STG with the suffix "_contracted".

C.3.5 CSC conflict resolution

Tools→Encoding conflicts→Resolve CSC conflicts

The Complete State Coding (CSC) condition means that there are no semantically different STG states (markings) that share the same binary encoding of signal states. This condition is a necessary condition for successful synthesis of an asynchronous circuit from the STG (see Section 2.2.2 and Figure 2.5). Workcraft uses MPSat for CSC conflict resolution.

When this tool is applied, the resulting STG will appear in the Workspace window alongside the original STG with the suffix "_resolved".

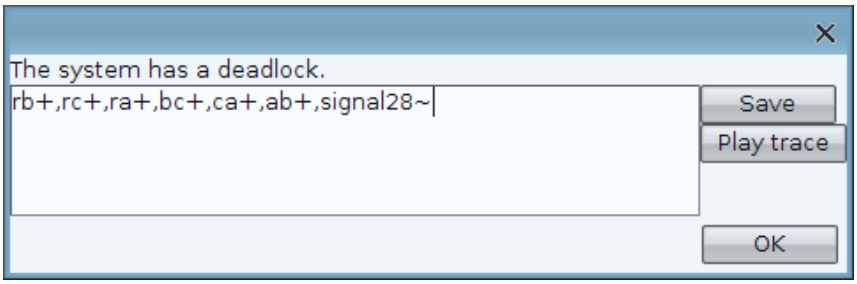


Figure C.12: Failure trace report

C.3.6 Deadlock detection

Tools→Verification→Check for deadlocks

Workcraft uses the unfolding-based tools Punf and MPSat to detect deadlock states (see Definition 3.7) in Signal Transition Graphs. If a reachable deadlock state is found, Workcraft shows a report window containing that trace. Optionally, the trace can be loaded into the simulation tool which helps to examine the particular sequence of events that leads into the problematic state (Figure C.12).

C.3.7 Reachability analysis

Tools→Verification→Check custom property

Workcraft provides a user-friendly interface to the MPSat tool chain for verification of custom properties (Figure C.13). This mode is useful to specify reachability-like properties (e.g., output persistence, consistency, variations of the deadlock property, etc.) using a language called Reach [66]. The configuration of the MPSat parameters and the property specification can be saved using the preset system similar to the DesiJ tool interface.

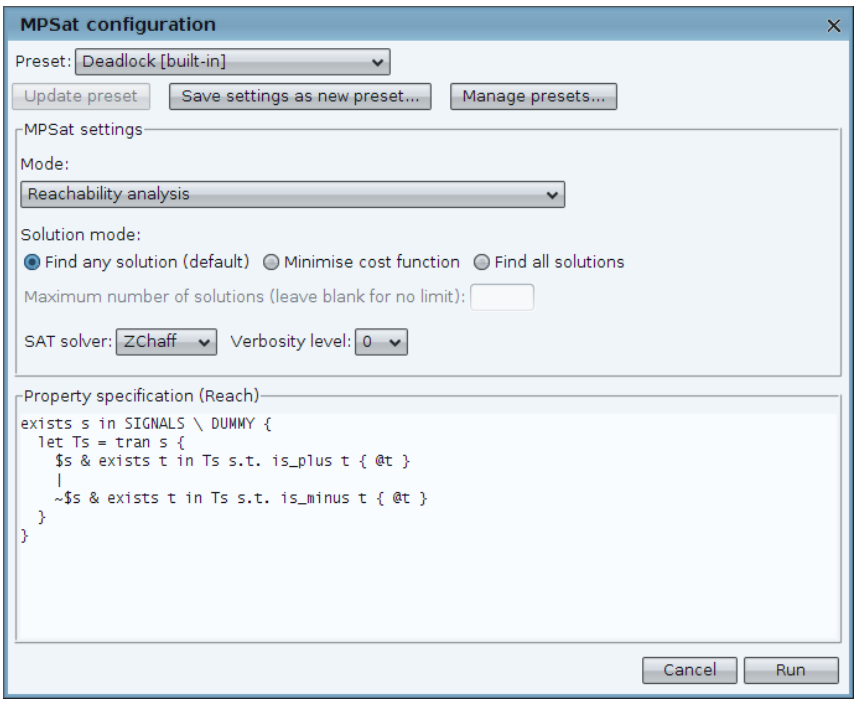


Figure C.13: MPSat configuration interface

Bibliography

[1] Apache Maven — http://maven.apache.org/.

[2] Bazaar IDE Integration — http://wiki.bazaar.canonical.com/ideintegration.

[3] Bazaar version control system — http://bazaar.canonical.com/.

[4] CPN Tools — http://wiki.daimi.au.dk/cpntools/.

[5] Design/CPN Online — http://www.daimi.au.dk/designCPN/.

[6] Eclipse Maven Integration — http://www.eclipse.org/m2e/.

[7] Graph visualisation tools — http://www.graphviz.org.

[8] Java SE downloads — http://java.sun.com/javase/downloads/.

[9] Launchpad collaboration platform — https://launchpad.net/.

[10] OpenJDK — http://openjdk.java.net/.

[11] PEP Tool — http://theoretica.informatik.uni-oldenburg.de/ pep/.

[12] Punf and MPSat tools — http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/tools/mpsat/.

[13] Scripting Java — http://www.mozilla.org/rhino/ScriptingJava.html.

[14] SPIN — http://spinroot.com/.

[15] The Substance project — http://java.net/projects/substance/.

[16] The Eclipse Foundation — http://www.eclipse.org/.

[17] VeriMap tool — http://async.org.uk/screen/verimap/.

[18] VeriSyn: asynchronous high-level synthesis tool — http://async.org.uk/besst/verisyn/.

[19] Versify: speed-independent asynchronous circuit verification tool — http://research.ac.upc.edu/vlsi/versify/.

[20] Workcraft — http://workcraft.org/.

[21] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[22] Manoj Ampalam and Montek Singh. Counterflow pipelining: architectural support for preemption in asynchronous systems using anti-tokens. In *Proc. International Conference Computer-Aided Design (ICCAD)*, November 2006.

[23] Sreekaanth Isloor Anthony and T. Anthony Marsland. The deadlock problem: An overview. *IEEE Computer*, 13:58–78, 1980.

[24] J. C. M. Baeten, editor. *Applications of Process Algebra*. Cambridge Press, 2005.

[25] K.R Baker and A.J. Currie. Multiple objective optimization in a behavioral synthesis system. In *in Proc. Inst. Elect. Eng.*, volume 140, pages 253–260, 1993.

[26] A. Baravalle, G. Franceschinis, M. Gribaudo, V. Lanfranchi, M. Iacono, N. Mazzocca, and V. Vittorini. *DrawNET Xe: GUI and Formalism Definition Language*.

[27] A. Bardsley, L. Tarazona, and D. Edwards. *Teak: A Token-Flow Implementation for the Balsa Language*. 2009.

[28] Andrew Bardsley. *Implementing Balsa handshake circuits*. PhD thesis, Dept. of Computer Science, University of Manchester, 2000.

[29] Jan A. Bergstra, Jan Willem Klop, and J. V. Tucker. Algebraic tools for system construction, 1984.

[30] E. Best, J. Esparza, B. Grahlmann, S. Melzer, S. Romer, and F. Wallner. The PEP verification system. Tool presentation In the FEmSys Conference., 1997.

[31] I. Blunno and L. Lavagno. Designing an asynchronous microcontroller using Pipefitter. In *Proc. International Conference Computer Design (ICCD)*, pages 488 – 493, 2002.

[32] Charles Brej. *Early output logic and anti-tokens*. PhD thesis, Dept. of Computer Science, University of Manchester, 2005.

[33] Charles Brej and Jim Garside. Early output logic using anti-tokens. In *Proc. International Workshop on Logic Synthesis*, pages 302–309. ACM Press, May 2003.

[34] Frank Burns, Delong Shang, Albert Koelmans, and Alex Yakovlev. An asynchronous synthesis toolset using Verilog. In *Proceedings of the conference on Design, automation and test in Europe (Volume 1)*, DATE '04, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Alex Bystrov, Danil Sokolov, and Alex Yakovlev. Low-latency control structures with slack. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 164–173. IEEE Computer Society Press, May 2003.

[36] Josep Carmona, Jordi Cortadella, and Michael Kishinevsky. Genet: a tool for the synthesis and mining of Petri Nets. pages 181–185, 2009.

[37] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous systems*. PhD thesis, Stanford University, 1984.

[38] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[39] Wesley A Clark. *Macromodular computer systems. In AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume volume 30. Academic Press, 1967.

[40] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[41] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[42] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[43] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical report UUCS-97-013. Technical report, University of Utah, 1997.

[44] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Moebius Framework and its implementation. Technical Report 10, Piscataway, NJ, USA, 2002.

[45] E.W. Dijkstra. Technical Report EWD-123. Technical report, Technological University, Eindhoven, The Netherlands, 1965.

[46] A. V. Dinh Duc, Jean-Baptiste Rigaud, Amine Rezzag, Antoine Sirianni, Joo Fragoso, Laurent Resquet, and Marc Renaudin. TAST CAD Tools. Tutorial given at the International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'02), April 2002.

[47] D. A. Edwards and A. Bardsley. Balsa: an asynchronous hardware synthesis language. *The Computer Journal*, 45 (1):12–18, jan 2002.

[48] E. G. Friedman, editor. *Clock Distribution Networks in VLSI Circuits and Systems*. IEEE Press, 1995.

[49] Gerald C. Gannod and Sunil Gupta. An automated tool for analyzing Petri Nets using SPIN. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 404, Washington, DC, USA, 2001. IEEE Computer Society.

[50] Stanislavs Golubcovs, Delong Shang, Fei Xia, Andrey Mokhov, and Alex Yakovlev. Modular approach to multi-resource arbiter design. In *Proceedings of the 2009 15th IEEE Symposium on Asynchronous Circuits and Systems (async 2009)*, pages 107–116, Washington, DC, USA, 2009. IEEE Computer Society.

[51] J. Grabowski. On the analysis of switching circuits by means of Petri nets. *Elektronische Informations-verarbeitung und Kybernetik*, 14:611– 617, 1978.

[52] Bernd Grahlmann, Carola Pohl, and Sercon Mainz. Profiting from Spin in PEP. 1998.

[53] T Grotker, S Liao, G. Martin, and Swan S. *System Design with SystemC*. Springer, 2002.

[54] Naohiro Hamada, Yuki Shiga, Takao Konishi, Hiroshi Saito, Tomohiro Yoneda, Chris Myers, and Takashi Nanya. A behavioral synthesis system for asynchronous circuits with bundled-data implementation. *Information and Media Technologies*, 4(2):211–226, 2009.

[55] Keijo Heljanko, Victor Khomenko, and Maciej Koutny. Parallelisation of the Petri Net unfolding algorithm. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 371–385, 2002.

[56] C. A. R. Hoare. Communicating sequential processes, 2004.

[57] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[58] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[59] Visual STG Lab http://vstgl.sourceforge.net/.

[60] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Springer-Verlag, 1997.

[61] M.B. Josephs, S.M. Nowick, and C.H. Van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234 –242, February 1999.

[62] Ali Khalili, Amir Jalaly Bidgoly, and Mohammad Abdollahi Azgomi. PDETool: A multi-formalism modeling tool for discrete-event systems based on SDES description. In *PETRI NETS '09: Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, pages 343–352, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] V. Khomenko. Computing shortest violation traces in model checking based on Petri Net unfoldings and SAT (technical report CS-TR-84). Technical report, School of Computing Science, Newcastle University, 2004.

[64] Victor Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, School of Computing Science, 2003.

[65] Victor Khomenko. Efficient automatic resolution of encoding conflicts using STG unfoldings. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 137–146, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Victor Khomenko. A Usable Reachability Analyser. In Alex Yakovlev, editor, *Proc. of 21st UK Asynchronous Forum'2009, University of Bristol*, 2009.

[67] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state encoding conflicts in stg unfoldings using sat. *Fundam. Inf.*, 62(2):221–241, 2004.

[68] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Logic synthesis for asynchronous circuits based on stg unfoldings and incremental sat. *Fundam. Inf.*, 70(1):49–73, 2005.

[69] M. A. Kishinevsky, A. Y. Kondratyev, A.R. Taubin, and V.I. Varshavsky. On self-timed behavior verification. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[70] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. Series in Parallel Computing. Wiley-Interscience, John Wiley& Sons, Inc., 1994.

[71] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[72] Daniel H. Linder and James C. Harden. Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers*, 45:1031–1044, September 1996.

[73] Alain J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed Computing*, 1(4):226–234, 1986.

[74] Alain J. Martin. Self-timed FIFO: an exercise in compiling programs into VLSI circuits. *HDL description to guaranteed correct circuit design, North-Holland*, 1986.

[75] Koichi Masukura, Moniru Tomisaka, and Tomohiro Yoneda. Verification of asynchronous circuits based on zero-suppressed BDDs. *Systems and Computers in Japan*, 32:43–54, 2001.

[76] E. J. McCluskey. Minimization of Boolean functions. *Bell Syst. Tech.*, 35:1417–1444, 1956.

[77] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie-Mellon University Pittsburgh Dept. of Computer Science, 1992.

[78] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Comput.*, 51(5):541–552, 2002.

[79] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. pages 272–277, 1993.

[80] Andrey Mokhov. *Conditional Partial Order Graphs*. PhD thesis, University of Newcastle upon Tyne, School of Electrical, Electronic and Computer Engineering, 2009.

[81] Andrey Mokhov, Victor Khomenko, and Alex Yakovlev. Flat arbiters. *Application of Concurrency to System Design, International Conference on*, 0:99–108, 2009.

[82] U. Montanari and F. Rossi. Contextual nets. *Acta Informacia*, 32(6):545–596, 1995.

[83] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[84] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[85] Chris J. Myers. *Asynchronous circuit design*. Wiley-Interscience, John Wiley& Sons, Inc., July 2001.

[86] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. Titac: Design of a quasi-delay-insensitive microprocessor. *IEEE Des. Test*, 11(2):50–63, 1994.

[87] Volnei A. Pedroni. *Circuit Design with VHDL*. The MIT Press, 2004.

[88] A. Peeters, F. te Beest, M. de Wit, and W. Mallon. Click elements: An implementation style for data-driven compilation. pages 3 –14, May 2010.

[89] Marco A. Pena and Jordi Cortadella. Combining Process Algebras and Petri Nets for the specification and synthesis of asynchronous circuits. In *ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 222, Washington, DC, USA, 1996. IEEE Computer Society.

[90] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. English translation: Technical Report RADC-TR-65–377, Vol.1, 1966, New York: Griffiss Air Force Base,.

[91] Luis A. Plana, Doug Edwards, Sam Taylor, Luis A. Tarazona, and Andrew Bardsley. Performance-driven syntax-directed synthesis of asynchronous processors. pages 43–47, 2007.

[92] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft âĂŞ a framework for interpreted graph models. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 333–342. Springer Berlin / Heidelberg, 2009.

[93] Ivan Poliakov, Andrey Mokhov, Ashur Rafiev, Danil Sokolov, and Alex Yakovlev. Automated verification of asynchronous circuits using circuit petri nets. volume 0, pages 161–170, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[94] Ivan Poliakov, Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. High-level model verification within workcraft framework. In *19th UK Asynchronous Forum*, 2007.

[95] Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: A static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency - ICATPN 2007*, 2007.

[96] Jan M. Rabaey and Alberto Sangiovanni-Vincentelli. System-on-a-Chip - A Platform Perspective. University of California.

[97] Oriol Roig. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Universitat Politecnica de Catalunya, 1997.

[98] Leonid Rosenblum and Alex Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[99] M. Schaefer, D. Wist, and R. Wollowski. DESIJ–enabling decomposition-based synthesis of complex asynchronous controllers. pages 186 –190, jul. 2009.

[100] Mark Schaefer. DesiJ - A Tool for STG Decomposition. Technical report 2007-11. Technical report, Institute of Computer Science, University of Augsburg, 2007.

[101] Karsten Schmidt. LoLA: A low level analyser. 1825:465–474, 2000.

[102] Charles L. Seitz. *Introduction to VLSI systems. Chapter 7: System timing.* Addison-Wesley, 1980.

[103] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[104] Delong Shang, Frank Burns, Albert M. Koelmans, Alex Yakovlev, and F. Xia. Asynchronous system synthesis based on direct mapping using VHDL and Petri nets. *IEE Proceedings, Computers and Digital Techniques*, 151(3):209–220, May 2004.

[105] Alexandre Smirnov, Alexander Taubin, Mark Karpovsky, and Leonid Rozenblyum. Gate transfer level synthesis as an automated approach to fine-grain pipelining. In *in Workshop on Token Based Computing (ToBaCo)*, 2004.

[106] Danil Sokolov. *Automated synthesis of asynchronous circuits using direct mapping for control and data paths*. PhD thesis, Microelectronic System Design Group, School of EECE, University of Newcastle upon Tyne, 2006.

[107] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Asynchronous data path models. In *7th International Conference on Application of Concurrency to System Design*, 2007.

[108] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Asynchronous data path models. In *International Conference Application of Concurrency to System Design*, July 2007.

[109] Danil Sokolov, Ivan Poliakov, and Alexandre Yakovlev. Analysis of static data flow structures. *Fundam. Inform.*, 88(4):581–610, 2008.

[110] Jens Sparsø and Steve Furber. *Principles of asynchronous circuit design: a system perspective*. Kluwer Academic Publishers, 2001.

[111] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[112] C. Stehno. PEP Version 2.0. Tool demonstration In the ICATPN conference, 2001.

[113] S. Tam, D.L. Limaye, and U.N Desai. Clock Generation and Distribution for the 130-nm Itanium 2 Processor with 6-MB On-Die L3 Cache. *IEEE Journal of Solid-State Circuits*, 39, 2004.

[114] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language*. Springer.

[115] Sunan Tugsinavisut, Roger Su, and Peter A. Beerel. High-level synthesis for highly concurrent hardware systems. *Application of Concurrency to System Design, International Conference on*, 0:79–90, 2006.

[116] Jan Tijmen Udding. *Classification and composition of delay-insensitive circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[117] Antti Valmari. The state explosion problem, 1998.

[118] C.H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. pages 223–233, 1999.

[119] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. pages 384–389, 1991.

[120] Kees van Hee, Olivia Oanea, Reinier Post, Lou Somers, and Jan Martijn v an der Werf. Yasper: a tool for workflow modeling and analysis. *Application of Concurrency to System Design, International Conference on*, 0:279–282, 2006.

[121] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tzirlin. *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1990.

[122] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3:1–8, 1988. 10.1007/BF01788562.

[123] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis. The osmosys approach to multi-formalism modeling of systems. *Software and Systems Modeling*, 3:68–81, 2004. 10.1007/s10270-003-0039-5.

[124] Walter Vogler, Alexei L. Semenov, and Alexandre Yakovlev. Unfolding and finite prefix for nets with read arcs. In *International Conference on Concurrency Theory*, pages 501–516, 1998.

[125] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In Jordi Cortadella, Alex Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 152–190. Springer-Verlag, 2002.

[126] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified Signal Transition Graph model for asynchronous control circuit synthesis. In *ICCAD'92*, 1992.

[127] Alex Yakovlev, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Marta Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, 9:189–233, 1996.

[128] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. Zero-Suppressed BDDs: For CTL symbolic model checking of Petri Nets. In *In Proc. of International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume LNCS 1166, pages 435–449. Springer, 1996.

[129] Tomohiro Yoneda, Atsushi Matsumoto, Manabu Kato, and Chris Myers. High level synthesis of timed asynchronous circuits. Washington, DC, USA, 2005.

[130] Armin Zimmermann. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications*. Springer, 2008.

Index